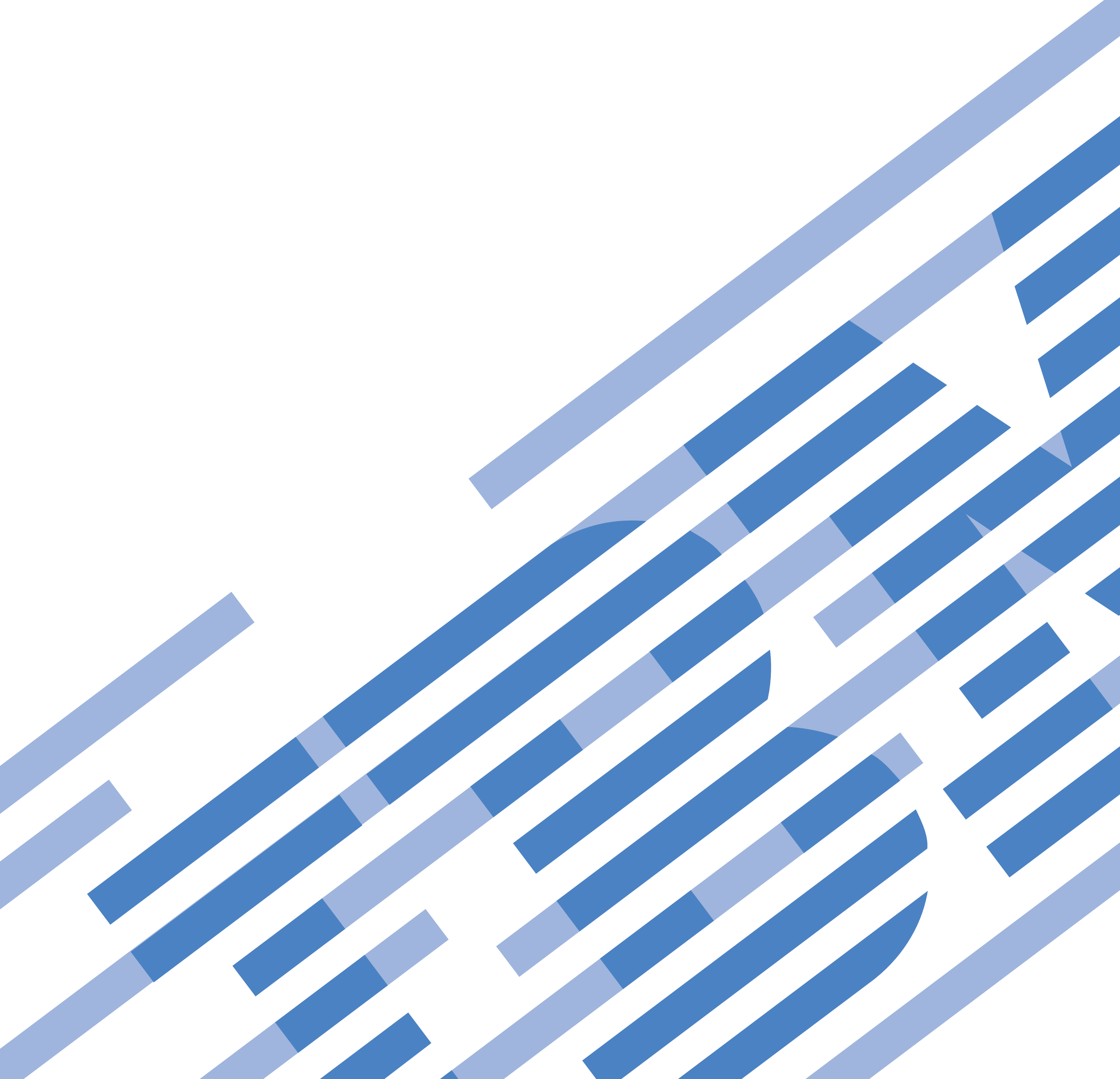
IBM

IBM i

プログラミング
IBM Developer Kit for Java

7.1



IBM

IBM i

プログラミング
IBM Developer Kit for Java

7.1

ご注意!

本書および本書で紹介する製品をご使用になる前に、 563 ページの『特記事項』に記載されている情報をお読みください。

本書は、IBM Developer Kit for Java (製品番号 5761-JV1) のバージョン 6、リリース 1、モディフィケーション 0 に適用されます。また、改訂版で断りがない限り、それ以降のすべてのリリースおよびモディフィケーションに適用されます。このバージョンは、すべての RISC モデルで稼働するとは限りません。また CISC モデルでは稼働しません。

お客様の環境によっては、資料中の円記号がバックスラッシュと表示されたり、バックスラッシュが円記号と表示されたりする場合があります。

原典： IBM i
Programming
IBM Developer Kit for Java
7.1

発行： 日本アイ・ビー・エム株式会社

担当： トランスレーション・サービス・センター

第1刷 2010.4

© Copyright IBM Corporation 1998, 2010.

目次

IBM Developer Kit for Java 1
IBM i 7.1 の新機能 1
IBM Developer Kit for Java の PDF ファイル . . . 2
Java をインストールおよび構成する 3
  IBM i サーバーへの Java のインストール . . . 3
  Hello World Java プログラムを初めて実行する . . 9
  HelloWorld Java プログラムを作成、コンパイル、および実行する 9
  ネットワーク・ドライブをサーバーにマップする 11
  Java ソース・ファイルを作成および編集する . . 11
Java を使用するための IBM i サーバーのカスタマイズ 12
  Java クラスパス 12
  Java システム・プロパティー 14
  国際化対応 21
リリース間の互換性 28
Java プログラムからのデータベース・アクセス . . 29
  Java JDBC ドライバーを使用して IBM i データベースにアクセスする 29
  DB2 SQLJ サポートを使用するデータベースへのアクセス 175
  Java SQL ルーチン 186
Java と他のプログラム言語 207
  ネイティブ・メソッドおよび Java ネイティブ・インターフェース 208
  Java 呼び出し API. 220
  java.lang.Runtime.exec() を使用する 225
  プロセス間通信 228
Java プラットフォーム 234
  Java アプレットおよびアプリケーション . . . 234
  Java 仮想マシン 235
  Java JAR とクラス・ファイル 236
  Java スレッド 237
  Java Development Kit 238
高度なトピック 239
  Java クラス、パッケージ、およびディレクトリー 239
  IFS の Java 関連ファイル 240
  統合ファイル・システム内の Java ファイル権限 241
  バッチ・ジョブで Java を実行する 241
GUI を使用しないホスト上で Java アプリケーションを実行する 242
  Native Abstract Windowing Toolkit 242
Java セキュリティー 250
  IBM i 7.1 での借用権限に関する変更点 . . . 250
  Java セキュリティー・モデル 264
  Java Cryptography Extension. 264
  Java Secure Socket Extension 267
  Java Authentication and Authorization Service . . 337
  IBM Java Generic Security Service (JGSS) . . . 367
Java プログラムのパフォーマンスのチューニング 401
  Java ガーベッジ・コレクション 401
  Java ネイティブ・メソッド呼び出しのパフォーマンスに関する考慮事項. 402
  Java の例外処理のパフォーマンスの考慮事項 402
  Java プロファイルのパフォーマンス測定ツール 402
  Java パフォーマンス・データを収集する . . . 403
Java コマンドおよびツール. 403
  Java ツールおよびユーティリティー. 404
  Java でサポートされる CL コマンド 409
IBM i での Java プログラムのデバッグ 409
  System i Debugger を使用して Java プログラムをデバッグする. 410
  Java Platform Debugger Architecture 419
  メモリー・リークを検出する 420
  「JVM ダンプの生成」コマンドの使用 421
Java コード例 421
Java プログラムのトラブルシューティング . . . 555
  制限 555
  Java の問題分析用のジョブ・ログを検索する 555
  Java の問題分析用のデータを収集する 556
  プログラム一時修正を適用する 557
  IBM i での Java サポートの利用. 557
関連情報 558
  Java Naming and Directory Interface 559
  JavaMail 559
  Java 印刷サービス. 559

付録. 特記事項 563
プログラミング・インターフェース情報 565
商標 565
使用条件 565

© Copyright IBM Corp. 1998, 2010

IBM Developer Kit for Java

IBM Developer Kit for Java™ は、IBM® i 環境で使用するために最適化されています。Java プログラミング・インターフェースとユーザー・インターフェースの互換性を利用することによって、開発者が独自の IBM i アプリケーションを開発できるようにしています。

IBM Developer Kit for Java を使用すると、ご使用の IBM i サーバー上で Java プログラムを作成および実行することができます。 IBM Developer Kit for Java は Sun Microsystems, Inc. の Java Technology と互換性のある製品なので、Sun Microsystems, Inc. の Java Development Kit (JDK) の資料を十分理解していることが前提となります。 Sun Microsystems, Inc. と IBM の情報を利用しやすくするために、Sun Microsystems, Inc. の情報へのリンクが用意されています。

Sun Microsystems, Inc. の Java Development Kit 関連資料へのリンクが何らかの理由で機能しない場合は、必要な情報について、Sun Microsystems, Inc. の HTML の参照資料を利用してください。この情報は、WWW の The Source for Java Technology java.sun.com にあります。

注: 法律上の重要な情報については、 560 ページの『コードに関するライセンス情報および特記事項』をお読みください。

IBM i 7.1 の新機能

IBM Developer Kit for Java のトピック・コレクションで新しく追加された点や大幅に変更された点について説明します。

以下の変更が IBM i 7.1 での IBM Developer Kit for Java に加えられました。

- IBM Developer Kit for Java のライセンス・プログラムは、5761-JV1 です。

  ライセンス・プログラムは、IBM i 6.1 の場合と同じです。ただし、IBM i 6.1 と IBM i 7.1 の違いである Java Developer Kit (Classic Java とも言われる) は、IBM i 7.1 ではサポートされなくなりました。 Classic Java サポートは、IBM Technology for Java で置き換えられました。

  引き続き Classic Java を使用しているお客様は、IBM Technology for Java にアップグレードする前に 4 ページの『IBM Technology for Java Virtual Machine の使用に関する考慮事項』 を参照する必要があります。
- IBM i 7.1 でサポートされる 5761-JV1 のオプションを反映して、以下のトピックが更新されました。
  - 6 ページの『複数の Java Development Kit (JDK) のサポート』
  - 3 ページの『IBM i サーバーへの Java のインストール』
  - 15 ページの『Java システム・プロパティーのリスト』
- PASE for i は、スタック実行の無効化保護を強制するようになりました。

© Copyright IBM Corp. 1998, 2010

システム・セキュリティーを向上させるために、PASE for i プログラムのデフォルト動作が変更され、プロセスのメモリー域 (スタック、ヒープ、および共有メモリー) から実行される命令はブロックされます。 IBM Technology for Java JIT によって生成されたコードがメモリー域に作成されます。 JNI_CreateJavaVM() API を呼び出す PASE for i プログラムは、PASE for i IBM i 7.1 の新機能の指示に従って、メモリー域からプログラムを実行できるようにする必要があるものとして、プログラムにマークを付ける必要があります。

- IBM Technology for Java に関する追加資料については、IBM Center for Java Technology Developer の Web サイトを参照してください。

IBM Technology for Java JVM は、AIX® バージョンの IBM Center for Java Technology Developer Kit に基づいています。 IBM Center for Java Technology Developer Kits には、サポートされているすべてのプラットフォームに適用される共通資料が含まれています。その共通資料には、プラットフォームの違いについて記載したセクションがあります。 IBM i プラットフォーム資料が存在しない場合は、AIX バージョンの資料を使用する必要があります。詳しくは、IBM Center for Java Technology Developer 診断ガイド (英語) を参照してください。

新規情報または変更情報の見分け方

技術上の変更が加えられたことを識別するのに役立つように、以下の情報が使用されています。

- » は、新情報や変更情報の先頭を示すマークです。
- « は、新情報や変更情報の終了を示すマークです。

今回のリリースの新規情報または変更情報に関するその他の情報は、プログラム資料説明書を参照してください。

IBM Developer Kit for Java の PDF ファイル

この情報の PDF ファイルを表示または印刷できます。

本資料の PDF バージョンを表示またはダウンロードするには、IBM Developer Kit for Java を選択します。

PDF ファイルの保存

表示または印刷のために PDF をワークステーションに保存するには、以下のようにします。

1. ご使用のブラウザーで PDF リンクを右クリックする。
2. PDF をローカルに保存するオプションをクリックする。
3. PDF を保存したいディレクトリーに進む。
4. 「**保存**」をクリックする。

Adobe® Reader のダウンロード

これらの PDF を表示または印刷するには、Adobe Reader がご使用のシステムにインストールされている必要があります。このアプリケーションは、 Adobe Web サイト (www.adobe.com/products/acrobat/readstep.html) から無償でダウンロードできます。

Java をインストールおよび構成する

IBM i サーバーで Java を初めて使用する場合は、以下の手順に従ってインストールと構成を行い、簡単な Hello World Java プログラムを実行して使い方に慣れてください。

1 ページの『IBM i 7.1 の新機能』
IBM Developer Kit for Java のトピック・コレクションで新しく追加された点や大幅に変更された点について説明します。

12 ページの『Java を使用するための IBM i サーバーのカスタマイズ』
ご使用のサーバーに Java をインストールした後、サーバーをカスタマイズすることができます。

8 ページの『Java パッケージをダウンロードしてインストールする』
IBM i プラットフォーム上でより効果的に Java パッケージをダウンロード、インストール、および使用するために、以下の情報を使用してください。

28 ページの『リリース間の互換性』
このトピックでは、Java アプリケーションを以前のリリースから最新リリースに移行する場合の考慮事項について説明します。

IBM i サーバーへの Java のインストール

IBM Developer Kit for Java をインストールすると、ご使用のシステムで Java プログラムを作成したり実行したりすることができます。IBM Developer Kit for Java に含まれる Java 仮想マシン (JVM) は、IBM Technology for Java Virtual Machine であり、32 ビット・バージョンと 64 ビット・バージョンの両方で使用可能です。

IBM Technology for Java Virtual Machine はライセンス・プログラム 5761-JV1 に組み込まれています。ライセンス・プログラム 5761-JV1 は、システム CD に同梱されています。IBM Technology for Java オプションにアクセスするには、以下のステップを実行します。

1. 「ライセンス・プログラムの処理 (GO LICPGM)」コマンドを入力し、オプション 10 (表示) を選択します。
2. このライセンス・プログラムがリストされていない場合は、以下のステップを実行してください。
   a. コマンド行で、GO LICPGM コマンドを入力します。
   b. オプション 11 (ライセンス・プログラムの導入) を選択します。
   c. ライセンス・プログラム (LP) 5761-JV1 *BASE についてオプション 1 (導入) を選択し、インストールするオプションを選択します。
3. 最新の Java PTF グループをロードします。このステップはオプションですが、推奨されています。詳しくは、 557 ページの『プログラム一時修正を適用する』を参照してください。
4. JAVA_HOME 環境変数を、使用する Java Development Kit のホーム・ディレクトリーに設定します。コマンド行から、以下のいずれかのコマンドを入力します。
   a. ADDENVVAR ENVVAR(JAVA_HOME) VALUE('/QOpenSys/QIBM/ProdData/JavaVM/jdk14/64bit')
   b. ADDENVVAR ENVVAR(JAVA_HOME) VALUE('/QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit')
   c. ADDENVVAR ENVVAR(JAVA_HOME) VALUE('/QOpenSys/QIBM/ProdData/JavaVM/jdk50/64bit')
   d. ADDENVVAR ENVVAR(JAVA_HOME) VALUE('/QOpenSys/QIBM/ProdData/JavaVM/jdk60/32bit')
   e. ADDENVVAR ENVVAR(JAVA_HOME) VALUE('/QOpenSys/QIBM/ProdData/JavaVM/jdk60/64bit')

現在どの JVM を使用しているかがはっきり分からない場合は、次の方法で確認できます。結果に IBM J9 VM と表示されていれば、IBM Technology for Java を使用していることになります。

- JVM が収容されているジョブのジョブ・ログを調べます。この中に、どの JVM を使用しているかを示すメッセージがあります。
- アプリケーションの実行に使用している Java コマンドの一部として、`-showversion` を追加します。1 行、追加の行が表示され、そこに使用している JVM が示されます。
- qsh または qp2term から、`java -version` を実行します。

関連概念

12 ページの『Java を使用するための IBM i サーバーのカスタマイズ』
ご使用のサーバーに Java をインストールした後、サーバーをカスタマイズすることができます。

関連タスク

9 ページの『Hello World Java プログラムを初めて実行する』
このトピックは、Java プログラムを初めて実行する場合に役立ちます。

関連情報

ライセンス・プログラムのリリースとサイズ

IBM Technology for Java Virtual Machine の使用に関する考慮事項

IBM Technology for Java Virtual Machine を使用する時に、以下の考慮事項に注意を払ってください。

Java ネイティブ・インターフェースに関する考慮事項

Java ネイティブ・インターフェース (JNI) 関数を使用する Integrated Language Environment (ILE) プログラムがある場合は、これらのプログラムを Teraspace ストレージを使用可能にしてコンパイルする必要があります。 Teraspace ストレージはデフォルトで使用可能になっていないため、多くの場合は再コンパイルが必要になります。これが必要なのは、Teraspace ストレージのトップにマップされ、Teraspace ストレージのポインターが戻される Java オブジェクトが PASE for i ストレージの中にあるためです。また、GetxxxArrayRegion などの JNI 関数には、データが置かれているバッファーへのパラメーターがあります。PASE for i の JNI 関数がデータを Teraspace ストレージにコピーできるようにするためには、このポインターは Teraspace ストレージを指していなければなりません。 Teraspace ストレージを使用可能にしてプログラムをコンパイルしていない場合、エスケープ・メッセージ `MCH4443` (ターゲット・プログラム `LOADLIB` で無効なストレージ・モデル) が戻されます。

借用権限

Java プログラムの借用権限は、IBM Technology for Java Virtual Machine ではサポートされていません。

診断メッセージおよびファイル

ILE ネイティブ・メソッドで問題が発生した場合は、ジョブ・ログにメッセージが書き込まれます。IBM Technology for Java Virtual Machine または PASE for i ネイティブ・メソッドで問題が発生した場合は、診断ファイルが IFS にダンプされます。これらの「コア・ファイル」にはいくつかのタイプがあり、`core.*.dmp`、`javacore.*.txt`、`Snap*.trc`、および `heapdump.*.phd` などがあります。ファイルのサイズは、数十 KB から数百 MB まで広い範囲に及びます。ほとんどの場合、問題が深刻であるほど大きなファイルが生成されます。サイズの大きなファイルは、気付かないうちにいつの間にか大量の IFS スペースを消費してしまうことがあります。これらのファイルは、スペースを消費するとはいえ、デバッグ目的では有用です。可能なら、これらのファイルは根本的な問題が解決されるまで保存しておいてください。

詳細は、Java Diagnostics Guide の『Advanced control of dump agents』を参照してください。

移行に関する考慮事項

IBM i 6.1 で存在したデフォルトの 64 ビット仮想マシンだった Classic JVM から 32 ビット・バージョンの IBM Technology for Java へ移行する際には、32 ビット環境を使用する場合には制約が存在することを考慮に入れてください。例えば、アドレス可能なメモリーの量が大幅に少なくなります。32 ビット・モードでは、Java オブジェクトのヒープを 3 G バイトより大きくすることができません。また、実行するスレッドの数も約 1000 スレッドに制限されます。アプリケーションが、1000 スレッドを超すスレッドまたは 3 G バイトより大きな Java オブジェクトのヒープを必要とする場合は、64 ビット・バージョンの IBM Technology for Java を使用してください。詳しくは、 6 ページの『複数の Java Development Kit (JDK) のサポート』を参照してください。

表 1 は、Java Developer Kit のレベル (Classic Java とも言われる) と、推奨される IBM Technology for Java の置き換えを示しています。

注: Java Developer Kit 1.4 または 5.0 から移行する場合は、Java SE 6 をお勧めします。

表 1. Classic Java のレベルと、推奨される IBM Technology for Java の置き換え

使用中の Classic Java のバージョン	可能な IBM Technology for Java の置き換えに含まれる内容
Java Developer Kit 1.4 (5761-JV1 オプション 6)	Java SE 6 32 ビット (5761-JV1 オプション 11) Java SE 6 64 ビット (5761-JV1 オプション 12) J2SE 5.0 32 ビット (5761-JV1 オプション 8) J2SE 5.0 64 ビット (5761-JV1 オプション 9) J2SE 1.4 64 ビット (5761-JV1 オプション 13)
Java Developer Kit 5.0 (5761-JV1 オプション 7)	Java SE 6 32 ビット (5761-JV1 オプション 11) Java SE 6 64 ビット (5761-JV1 オプション 12) J2SE 5.0 32 ビット (5761-JV1 オプション 8) J2SE 5.0 64 ビット (5761-JV1 オプション 9)
Java Developer Kit 6 (5761-JV1 オプション 10)	Java SE 6 32 ビット (5761-JV1 オプション 11) Java SE 6 64 ビット (5761-JV1 オプション 12)

関連概念

28 ページの『リリース間の互換性』

このトピックでは、Java アプリケーションを以前のリリースから最新リリースに移行する場合の考慮事項について説明します。

「ライセンス・プログラムの復元」コマンドを使ってライセンス・プログラムをインストールする

サーバーが新規の場合、「ライセンス・プログラムのインストール」画面でリストされているプログラムは、LICPGM インストール・システムによってサポートされています。時折、使用可能になった新しいプログラムが、サーバー上のライセンス・プログラムとしてリストされないことがあります。インストールしたいプログラムでこの状況になったときは、インストールするために「ライセンス・プログラムの復元 (RSTLICPGM)」コマンドを使用する必要があります。

「ライセンス・プログラムの復元」コマンドを使ってライセンス・プログラムをインストールする方法は、以下のとおりです

1. ライセンス・プログラムが含まれているテープまたは CD-ROM を、適切なドライブに入れる。

2. IBM i コマンド行で次のように入力する。

   RSTLICPGM

   その後、Enter キーを押します。

   「ライセンス・プログラムの復元 *(RSTLICPGM)*」画面が表示されます。
3. 「プロダクト *(Product)*」フィールドに、インストールしたいライセンス・プログラムの ID 番号を入力する。
4. 「装置 *(Device)*」フィールドで、インストール装置を指定する。

   注: 磁気テープ・ドライブからインストールする場合は、装置 ID は常に **TAP*xx*** という形式になります。ここで ***xx*** は **01** のような番号です。
5. ライセンス・プログラムの復元画面の他のパラメーターはデフォルトの設定のままにする。 Enter キーを押します。
6. さらにパラメーターが表示される。これもデフォルトの設定のままにします。 Enter キーを押します。プログラムのインストールが開始されます。

ライセンス・プログラムのインストールが完了すると、「ライセンス・プログラムの復元」画面が再び表示されます。

複数の Java Development Kit (JDK) のサポート

IBM i プラットフォームでは、複数のバージョンの Java Development Kit (JDK) と Java 2 Platform, Standard Edition がサポートされています。

注: この資料で (文脈によりますが) JDK という語は、サポートされているバージョンの JDK または Java 2 Platform, Standard Edition (J2SE) を指します。 通常、JDK が現れる状況では、特定のバージョンとリリース番号が参照されます。

ご使用の IBM i では、複数の JDK の同時使用がサポートされていますが、これは複数の Java 仮想マシンを通した場合だけです。単一の Java 仮想マシンは、 1 つの指定された JDK を実行します。1 つのジョブについて実行できる Java 仮想マシンは、1 つだけです。

使用しているまたは使用したい JDK を見つけてから、インストールする調整オプションを選択します。一度に複数の JDK をインストールするには、 3 ページの『IBM i サーバーへの Java のインストール』を参照してください。

IBM Technology for Java を使用している場合は、JAVA_HOME 環境変数を設定することによって、どの 5761-JV1 オプション (つまりどの JDK/ビット・モード) を実行するかを選択します。Java 仮想マシンが稼働状態になってからは、JAVA_HOME 環境変数を変更しても効果はありません。

以下の表は、このリリースでサポートされているオプションのリストです。

5761-JV1 オプション	JAVA_HOME	java.version
オプション 8 - IBM Technology for Java 5.0 32 ビット	/QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit	1.5
オプション 9 - IBM Technology for Java 5.0 64 ビット	/QOpenSys/QIBM/ProdData/JavaVM/jdk50/64bit	1.5
オプション 11 - IBM Technology for Java 6 32 ビット	/QOpenSys/QIBM/ProdData/JavaVM/jdk60/32bit	1.6

5761-JV1 オプション	JAVA_HOME	java.version
オプション 12 - IBM Technology for Java 6 64 ビット	/QOpenSys/QIBM/ProdData/JavaVM/jdk60/64bit	1.6
オプション 13 - IBM Technology for Java 1.4.2 64 ビット	/QOpenSys/QIBM/ProdData/JavaVM/jdk14/64bit	1.4

この複数の JDK の環境で選択されるデフォルトの JDK は、どの 5761-JV1 オプションがインストールされているかによって異なります。以下の表は、その例です。IBM Technology for Java JDK は、JAVA_HOME 環境変数を設定することによって、または使用する JDK に位置する Java ツールまたはユーティリティーへの絶対パスを設定することによってアクセス可能です。

インストール	入力	結果
サポートされているオプションがすべてインストールされる	`java QIBMHello`	6 の 32 ビットを使用
オプション 13 (1.4)	`Java QIBMHello`	1.4 を使用
オプション 13 (1.4) およびオプション 9 (5.0 64 ビット)	`java QIBMHello`	5.0 の 64 ビットを使用

注: JDK を 1 つだけインストールしている場合は、 その JDK がデフォルトになります。複数の JDK をインストールしている場合は、優先順位は以下のとおりです。

1. オプション 11 - IBM Technology for Java 6 32 ビット
2. オプション 12 - IBM Technology for Java 6 64 ビット
3. オプション 8 - IBM Technology for Java 5.0 32 ビット
4. オプション 9 - IBM Technology for Java 5.0 64 ビット
5. オプション 13 - IBM Technology for Java 1.4.2 64 ビット

Java の拡張機能をインストールする

拡張機能は、コア・プラットフォームの機能を拡張するために使用できる Java クラスのパッケージです。拡張機能は、1 つまたは複数の ZIP ファイルまたは JAR ファイルのパッケージであり、拡張クラス・ローダーによって Java 仮想マシンにロードされます。

拡張機能のメカニズムにより、Java 仮想マシンでは、システム・クラスを使用するのと同じ方法で拡張クラスを使用することができます。また、この拡張機能のメカニズムでは、まだ Java 2 Platform, Standard Edition (J2SE) にインストールされていない拡張機能を、指定した URL から取得できるようになっています。

拡張機能のためのいくつかの JAR ファイルは、IBM i と共に出荷されます。これらの拡張機能のいずれかをインストールするときには、以下のコマンドを入力してください。

```
ADDLNK OBJ('/QIBM/ProdData/Java400/ext/extensionToInstall.jar')
     NEWLNK('/QIBM/UserData/Java400/ext/extensionToInstall.jar')
     LNKTYPE(*SYMBOLIC)
```

ここで、

`extensionToInstall.jar`

は、インストールしたい拡張機能を含む ZIP または JAR ファイルの名前です。

注: /QIBM/UserData/Java400/ext ディレクトリーに拡張機能のための JAR ファイルが格納されている可能性がありますが、これは IBM によって提供されたものではありません。

/QIBM/UserData/Java400/ext ディレクトリー内の拡張機能へのリンクの作成またはファイルの追加を行うと、*サーバー上で実行されているそれぞれの Java 仮想マシン* について、拡張クラス・ローダーによって検索されるファイルのリストが変更されます。サーバー上の他の Java 仮想マシンの拡張クラス・ローダーに影響を与えたくないが、拡張機能へのリンクを作成したり、IBM がサーバーと共に出荷したものではない拡張機能をインストールしたい場合は、以下のステップに従ってください。

1. 拡張機能をインストールするためのディレクトリーを作成します。 IBM i コマンド行から「ディレクトリーの作成 (MKDIR)」コマンドを使用するか、または Qshell インタープリターから `mkdir` コマンドを使用します。
2. 作成したディレクトリーに拡張機能の JAR ファイルを置きます。
3. 新しいディレクトリーを java.ext.dirs プロパティーに追加します。新しいディレクトリーを java.ext.dirs プロパティーに追加するには、IBM i コマンド行から JAVA コマンドの PROP フィールドを使用します。

新しいディレクトリーの名前が "/home/username/ext" で、拡張機能ファイルの名前が extensionToInstall.jar、Java プログラムの名前が Hello の場合は、入力するコマンドは次のようになります。

```
MKDIR DIR('/home/username/ext')

CPY OBJ('/productA/extensionToInstall.jar') TODIR('/home/username/ext') または
FTP (ファイル転送プロトコル) を使って、ファイルを /home/username/ext にコピーします。

JAVA Hello PROP((java.ext.dirs '/home/username/ext'))
```

Java パッケージをダウンロードしてインストールする

IBM i プラットフォーム上でより効果的に Java パッケージをダウンロード、インストール、および使用するために、以下の情報を使用してください。

グラフィカル・ユーザー・インターフェースを使用するパッケージ

グラフィカル・ユーザー・インターフェース (GUI) と共に使用する Java プログラムでは、グラフィカル表示装置能力のある表示装置を使用する必要があります。たとえば、パーソナル・コンピューター、テクニカル・ワークステーション、またはネットワーク・コンピューターなどが使用できます。 Native Abstract Windowing Toolkit (NAWT) を使用して、Java アプリケーションおよびサーブレットに Java 2 Platform, Standard Edition (J2SE) Abstract Windowing Toolkit (AWT) グラフィックス機能の全機能を提供することができます。詳しくは、Native Abstract Windowing Toolkit (NAWT)を参照してください。

大文字小文字の区別および統合ファイル・システム

統合ファイル・システムは、大文字小文字の区別をするファイル・システムと、ファイル名とは関係していないファイル・システムの両方を提供します。 QOpenSys は、統合ファイル・システムにある大文字小文字の区別をするファイル・システムの例です。ルート '/' は、大文字小文字を区別しないファイル・システムの例です。詳しくは、統合ファイル・システムを参照してください。

JAR またはクラスは大文字小文字を区別しないファイル・システム上に配置されますが、Java は引き続き大文字小文字を区別する言語です。 `wrklnk '/home/Hello.class'` と `wrklnk '/home/hello.class'` は同一の結果を生成しますが、`JAVA CLASS(Hello)` と `JAVA CLASS(hello)` は別々のクラスを呼び出します。

ZIP ファイルの処理

ZIP ファイルには、JAR ファイルと同様に、一連の Java クラスが格納されています。 ZIP ファイルは JAR ファイルと同様に扱われます。

Java 拡張フレームワーク

J2SE では、拡張機能は、コア・プラットフォームの機能を拡張するために使用できる Java クラスのパッケージです。拡張機能またはアプリケーションは、1 つまたは複数の JAR ファイルにあります。拡張機能のメカニズムにより、Java 仮想マシンでは、システム・クラスを使用するのと同じ方法で拡張クラスを使用することができます。拡張機能機構はさらに、拡張機能がまだ J2SE または Java 2 Runtime Environment, Standard Edition にインストールされていないときに、指定した URL から拡張機能を検索することを可能にします。

拡張のインストールについては、 7 ページの『Java の拡張機能をインストールする』を参照してください。

Hello World Java プログラムを初めて実行する

このトピックは、Java プログラムを初めて実行する場合に役立ちます。

Hello World Java プログラムを起動して実行するには、次の 2 つの方法があります。

1. IBM Developer Kit for Java に付属している Hello World Java プログラムを実行する。

   付属のプログラムを実行する方法は次のとおりです。

   a. 「ライセンス・プログラムの処理 (GO LICPGM)」コマンドを入力し、IBM Developer Kit for Java がインストールされていることを確認する。その後、オプション 10 (導入済みライセンス・プログラムの表示) を選択する。ライセンス・プログラム 5761-JV1 *BASE と、少なくとも 1 つのオプションがインストール済みとしてリストされていることを確認してください。

   b. IBM i メイン・メニューのコマンド行に、`java QIBMHello` と入力する。Enter キーを押すと、Hello World Java プログラムが実行されます。

   c. IBM Developer Kit for Java が正しくインストールされていれば、Java シェル画面に `QIBMHello` と表示される。 F3 (終了) または F12 (終了) を押すと、コマンド入力画面に戻ります。

   d. Hello World クラスが実行されない場合は、インストールが正常に完了したことを確認するか、 557 ページの『IBM i での Java サポートの利用』のサービス情報を参照する。

2. また、ユーザー独自の Hello Java プログラムも実行することができます。ユーザー独自の Hello Java プログラムの作成方法については、『HelloWorld Java プログラムを作成、コンパイル、および実行する』を参照してください。

HelloWorld Java プログラムを作成、コンパイル、および実行する

単純な Hello World Java プログラムを作成することは、IBM Developer Kit for Java を十分理解する上での第一歩となります。

独自の Hello World Java プログラムを作成し、コンパイルして実行する手順は、次のとおりです。

1. ネットワーク・ドライブをシステムに割り当てる。
2. Java アプリケーション用に、サーバー上にディレクトリーを作成する。

   a. コマンド入力行で、次のように入力する。

   ```
   CRTDIR DIR('/mydir')
   ```

   ここで *mydir* は作成するディレクトリーの名前です。

Enter キーを押す。

3. 統合ファイル・システム内に、ASCII テキスト・ファイルとしてソース・ファイルを作成する。 Java アプリケーションのコーディングには、統合開発環境 (IDE) 製品を使用することも、Windows® のメモ帳のようなテキスト・エディターを使用することもできます。
   a. テキスト・ファイルの名前を `HelloWorld.java` にする。
   b. ファイルに次のソース・コードが含まれていることを確認する。

```
class HelloWorld {
     public static void main (String args[]) {
          System.out.println("Hello World");
     }
}
```

4. ソース・ファイルをコンパイルする。
   a. 「Qshell の開始 (STRQSH)」コマンドを入力して、Qshell インタープリターを開始する。
   b. 「ディレクトリーの変更 (cd)」コマンドを使用して、現行ディレクトリーを `HelloWorld.java` ファイルが入っている統合ファイル・システムのディレクトリーに変更する。
   c. `javac` に続けて、ディスクに保管したファイルの名前を入力する。たとえば、`javac HelloWorld.java` と入力する。
5. 統合ファイル・システムのクラス・ファイル上のファイル権限を設定する。
6. クラス・ファイルを実行する。
   a. Java クラスパスが正しく設定されていることを確認する。
   b. Qshell コマンド入力行で、`java` に続けて `HelloWorld` と入力し、Java 仮想マシンで `HelloWorld.class` を実行する。たとえば、`java HelloWorld` と入力します。「Java プログラムの実行 (RUNJVA)」コマンドを使用して、ご使用のシステムで `HelloWorld.class` を実行することも可能です。 RUNJVA CLASS(HelloWorld)
   c. すべてが正しく入力されていれば、画面に `"Hello World"` と表示されます。 Qshell 環境で実行している場合は、シェル・プロンプト (デフォルトでは $) が表示され、Qshell が次のコマンドを受ける準備ができたことを示します。
   d. F3 (Exit (終了)) または F12 (Disconnect (切断)) を押して、コマンド入力画面に戻る。

システム上でタスクを実行するためのグラフィカル・ユーザー・インターフェースである System i® Navigatorを使用して、Java アプリケーションのコンパイルと実行を簡単に行うこともできます。

11 ページの『ネットワーク・ドライブをサーバーにマップする』
ネットワーク・ドライブを割り当てるには、以下のステップを実行してください。

11 ページの『Java ソース・ファイルを作成および編集する』
Java ソース・ファイルを作成および編集する方法は多数あり、IBM i Access for Windows を使用する方法、ワークステーション上で行う方法、EDTF を使用する方法、および SEU を使用する方法があります。

12 ページの『Java クラスパス』
Java 仮想マシンは、実行時にクラスを検索するために、Java クラスパスを使用します。また、Java のコマンドとツールも、クラスパスを使ってクラスの位置を判別します。 デフォルトのシステム・クラスパス、CLASSPATH 環境変数、および classpath コマンド・パラメーターはすべて、クラスを探すときにどのディレクトリーを検索するかを指定するために使用されます。

241 ページの『統合ファイル・システム内の Java ファイル権限』
Java プログラムを実行およびデバッグするには、クラス・ファイル、JAR ファイル、および ZIP ファイルに読み取り権限 (*R) が必要です。 ディレクトリーには読み取りおよび実行の権限 (*RX) が必要です。

Java プログラムの実行 (RUNJVA) コマンド

System i Navigatorについて

ネットワーク・ドライブをサーバーにマップする

ネットワーク・ドライブを割り当てるには、以下のステップを実行してください。

1. サーバーおよびワークステーションに IBM i Access for Windows がインストールされていることを確認してください。IBM i Access for Windows をインストールおよび構成する方法について詳しくは、『IBM i Access for Windows のインストールおよびセットアップ』を参照してください。ネットワーク・ドライブを割り当てる前に、サーバー用に接続を構成する必要があります。
2. Windows エクスプローラを開く。
   a. Windows タスクバーの「**スタート**」ボタンの上で右マウス・ボタン・クリックする。
   b. メニューの中の「**エクスプローラ**」をクリックする。
3. 「**ツール**」メニューから「**ネットワーク ドライブの割り当て**」を選択する。
4. サーバーへの接続に使用したいドライブを選択する。
5. サーバーへのパス名を入力する。たとえば、`\\MYSERVER` とします。ここで *MYSERVER* はサーバーの名前です。
6. 「**ログオン時に再接続**」が選択されていない場合、ボックスをチェック・マークを入れる。
7. 「**OK**」をクリックして完了する。

割り当てたドライブが、Windows エクスプローラの「**すべてのフォルダ**」セクションに表示されます。

Java ソース・ファイルを作成および編集する

Java ソース・ファイルを作成および編集する方法は多数あり、IBM i Access for Windows を使用する方法、ワークステーション上で行う方法、EDTF を使用する方法、および SEU を使用する方法があります。

IBM i Access for Windows を使用する方法

Java ソース・ファイルは、統合ファイル・システムにある、ASCII テキスト・ファイルです。

Java ソース・ファイルは IBM i Access for Windows やワークステーション・ベースのエディターを使用して作成および編集することができます。

ワークステーション上で編集する

Java ソース・ファイルをワークステーション上で作成することができます。その場合は、作成したファイルをファイル転送プロトコル (FTP) を使って統合ファイル・システムに転送します。

ワークステーション上で Java ソース・ファイルを作成し、編集する方法は次のとおりです。

1. 任意のエディターを使って、ワークステーション上で ASCII ファイルを作成する。
2. サーバーに FTP 接続する。
3. ASCII 形式が維持されるように、ソース・ファイルをバイナリー・ファイルとして統合ファイル・システムのディレクトリーに転送する。

EDTF を使用する

任意のファイル・システムからファイルを編集するには、「ファイル編集 (EDTF)」CL コマンドを使用できます。これは、ストリーム・ファイルやデータベース・ファイルを編集するための原始ステートメント入力ユーティリティー (SEU) と類似のエディターです。詳細は、「ファイル編集 (EDTF)」CL コマンドを参照してください。

EDTF コマンドを使用して新規のストリーム・ファイルを作成する場合は、拡張 2 進化 10 進コード (EBCDIC) のコード化文字セット ID (CCSID) でファイルがタグ付けされます。 Java ファイルは、ASCII CCSID でタグ付けする必要があります。 Qshell ユーティリティー touch を使用して ASCII CCSID により空のストリーム・ファイルを作成し、EDTF コマンドを使用してファイルを編集できます。たとえば、ASCII CCSID 819 により空のストリーム・ファイル /tmp/Test.java を作成する場合は、次のコマンドを使用します。

```
QSH CMD('touch -C 819 /tmp/Test.java')
```

原始ステートメント入力ユーティリティーを使用する

原始ステートメント入力ユーティリティー (SEU) を使うと、Java ソース・ファイルをテキスト・ファイルとして作成することができます。

SEU を使って Java ソース・ファイルをテキスト・ファイルとして作成する方法は次のとおりです。

1. SEU を使ってソース・ファイル・メンバーを作成する。
2. 「ストリーム・ファイルへのコピー (CPYTOSTMF)」コマンドを使用して、ソース・ファイル・メンバーを統合ファイル・システム・ストリーム・ファイルにコピーする。このとき、データは ASCII コードに変換されます。

ソース・コードを変更する必要がある場合は、SEU を使ってデータベース・メンバーを変更し、ファイルを再びコピーしてください。

ファイルの保管については、240 ページの『IFS の Java 関連ファイル』を参照してください。

Java を使用するための IBM i サーバーのカスタマイズ

ご使用のサーバーに Java をインストールした後、サーバーをカスタマイズすることができます。

Java クラスパス

Java 仮想マシンは、実行時にクラスを検索するために、Java クラスパスを使用します。また、Java のコマンドとツールも、クラスパスを使ってクラスの位置を判別します。デフォルトのシステム・クラスパス、CLASSPATH 環境変数、および classpath コマンド・パラメーターはすべて、クラスを探すときにどのディレクトリーを検索するかを指定するために使用されます。

ロードされる拡張のクラスパスが java.ext.dirs プロパティーによって決定されます。詳しくは、7 ページの『Java の拡張機能をインストールする』を参照してください。

デフォルトのブートストラップ・クラスパスはシステムによって定義されており、変更できません。サーバーでは、IBM Developer Kit for Java、および他のシステム・クラスに属するクラスをどこで検索するかを、デフォルトのブートストラップ・クラスパスによって指定します。

| java.endorsed.dirs プロパティーは、JAR ファイルをブートストラップ・クラスパスに追加することによっ
| て、承認されたバージョンの Java クラスをオーバーライドする標準的な方法です。詳しくは、Endorsed
| Standards Override Mechanism を参照してください。

これら以外のクラスをシステム上で検出するには、CLASSPATH 環境変数または classpath パラメーターを使用して、検索対象のクラスパスを指定します。ツールやコマンドで classpath パラメーターを使用すると、CLASSPATH 環境変数で指定されている値は無効になります。

CLASSPATH 環境変数の設定には、「環境変数の処理 (WRKENVVAR)」コマンドを使用します。WRKENVVAR の画面から、CLASSPATH 環境変数の追加や変更を行うことができます。 CLASSPATH 環境変数を追加する場合は「環境変数の追加 (ADDENVVAR)」コマンドを、CLASSPATH 環境変数を変更する場合は「環境変数の変更 (CHGENVVAR)」コマンドを使用します。

CLASSPATH 環境変数の値はパス名のリストであり、コロン (:) によって分けられています。これは、特定のクラスを探すために検索されます。パス名は、0 または複数の一連のディレクトリー名です。これらのディレクトリー名の後には、ディレクトリーの名前、ZIP ファイル、または JAR ファイル (統合ファイル・システムで検索する) が続きます。パス名のコンポーネントはスラッシュ (/) 文字によって分けられています。ピリオド (.) を使用して、現行作業ディレクトリーを示します。

Qshell インタープリターで使用可能なエクスポート・ユーティリティーを使うと、Qshell 環境で CLASSPATH 変数を設定できます。

これらのコマンドは、CLASSPATH 変数をユーザーの Qshell 環境に追加し、それを値 `".:/myclasses.zip:/Product/classes"` に設定します。

- 次に、Qshell 環境で CLASSPATH 変数を設定するコマンドを示します。

```
export -s CLASSPATH=.:/myclasses.zip:/Product/classes
```

- 次に、コマンド行から CLASSPATH 変数を設定するコマンドを示します。

```
ADDENVVAR ENVVAR(CLASSPATH) VALUE(".:/myclasses.zip:/Product/classes")
```

J2SE は最初にブートストラップ・クラスパスを検索してから、次に拡張ディレクトリーを検索し、その後クラスパスを検索します。上記の例のコードでの J2SE の検索順序は次のようになります。

1. sun.boot.class.path プロパティーのブートストラップ・クラスパス
2. java.ext.dirs プロパティーの拡張ディレクトリー
3. 現行作業ディレクトリー
4. 「ルート」(/) ファイル・システムにある myclasses.zip ファイル
5. 「ルート」(/) ファイル・システムにある、プロダクト・ディレクトリーにあるクラス・ディレクトリー

Java ツールおよびコマンドの中には、パス名のリストを指定できるクラスパス・パラメーターを含むものがあります。パラメーターの構文は CLASSPATH 環境変数の構文と同じです。以下のリストは、クラスパス・パラメーターを指定できるツールおよびコマンドの一部を示しています。

- Qshell の `java` コマンド
- `javac` ツール
- `javah` ツール
- `javap` ツール
- `javadoc` ツール
- `rmic` ツール

- 「Java プログラムの実行 (RUNJVA)」コマンド

これらのコマンドの詳細については、 403 ページの『Java コマンドおよびツール』を参照してください。これらのコマンドやツールで classpath パラメーターを使用すると、CLASSPATH 環境変数は無視されます。

CLASSPATH 環境変数をオーバーライドするには、java.class.path プロパティーを使用します。他のプロパティーと同様に、java.class.path プロパティーを変更するには、SystemDefault.properties ファイルを使用します。 SystemDefault.properties ファイルの値は、CLASSPATH 環境変数をオーバーライドします。SystemDefault.properties ファイルについては、 15 ページの『SystemDefault.properties ファイル』を参照してください。

さらに -Xbootclasspath オプションおよび java.endorsed.dirs プロパティーは、クラスの検索時にシステムがどのディレクトリーを検索するかに影響します。 -Xbootclasspath/a:*path* を使用すると、デフォルトのブートストラップ・クラスパスの後に *path* が付加され、/p:*path* と指定すると、デフォルトのブートストラップ・クラスパスの前に *path* が付加され、:*path* と指定すると、ブートストラップ・クラスパスは *path* によって置き換えられます。java.endorsed.dirs プロパティー用に指定されたディレクトリーに配置された JAR ファイルは、ブートストラップ・クラスパスの前に付加されます。

注: -Xbootclasspath を指定すると、システム・クラスが見つからなかったり、システム・クラスが誤ってユーザー定義クラスで置き換えられた場合に結果が保証されないので、注意が必要です。このため、ユーザー指定のクラスパスの前にデフォルトのシステム・クラスパスが検索されるように指定することをお勧めします。

Java プログラムのランタイム環境を決定する方法については、『Java システム・プロパティー』を参照してください。

詳しくは、プログラムおよび CL コマンド API または統合ファイル・システムを参照してください。

Java システム・プロパティー

Java システム・プロパティーにより、Java プログラムを実行する環境が決まります。 Java システム・プロパティーは、IBM i のシステム値や環境変数と似ています。

Java 仮想マシン (JVM) のインスタンスを開始すると、その JVM に影響するシステム・プロパティーの値が設定されます。

Java システム・プロパティーのデフォルト値を使用するか、以下の方法でそれらの値を指定できます。

- Java プログラムを開始するときにコマンド行 (または Java Native Interface (JNI) 呼び出し API) にパラメーターを追加する。
- QIBM_JAVA_PROPERTIES_FILE ジョブ・レベル環境変数を使用して特定のプロパティー・ファイルを指し示す。以下に例を示します。

```
ADDENVVAR ENVVAR(QIBM_JAVA_PROPERTIES_FILE)
  VALUE(/QIBM/userdata/java400/mySystem.properties)
```

- user.home ディレクトリーに作成する SystemDefault.properties ファイルを作成する。
- /QIBM/userdata/java400/SystemDefault.properties ファイルを使用する。

IBM i および JVM が、以下の優先順序で Java システム・プロパティーの値を決定します。

1. コマンド行または JNI 呼び出し API
2. QIBM_JAVA_PROPERTIES_FILE 環境変数

3. user.home SystemDefault.properties ファイル
4. /QIBM/UserData/Java400/SystemDefault.properties
5. デフォルト・システム・プロパティーの値

SystemDefault.properties ファイル

SystemDefault.properties ファイルは、Java 環境のデフォルト・プロパティーを指定できる、標準の Java プロパティー・ファイルです。

このファイルを使用して、JVM プロパティーと JVM オプションの両方で送信することができます。従来は、JVM プロパティーのみがサポートされていました。 JVM オプションも許可するためには、ファイルの最初の行に "#AllowOptions" を含める必要があります。含めない場合は、すべてが JVM プロパティーとして扱われます。

ホーム・ディレクトリーにある SystemDefault.properties ファイルは、/QIBM/UserData/Java400 ディレクトリーにある SystemDefault.properties よりも優先されます。

/YourUserHome/SystemDefault.properties ファイルで設定するプロパティーは、以下の特定の Java 仮想マシンにのみ影響します。

- 別の user.home プロパティーを指定せずに開始する JVM
- プロパティー user.home = /YourUserHome/ を指定して他のユーザーが開始する JVM

例: SystemDefault.properties ファイル

以下の例では、いくつかの Java プロパティーおよびオプションを設定しています。

```
#AllowOptions
#Comments start with pound sign
prop1=12345
-Dprop2
-Dprop3=abcd
-Xmx200m
prop4=value
-Xnojit
```

上記の Java プロパティーおよびオプションは、次のように JVM に影響します。

- 4 つのプロパティー、prop1、prop2、prop3、および prop4 があります。
- 最大のヒープ・サイズは、200 MB です。
- JIT は使用されません。

#AllowOptions の行が上記の例から除去されると、JVM には 6 つのプロパティー、prop1、-Dprop2、-Dprop3、-Xms200m、prop4、および -Xnojit が含まれることになります。

Java システム・プロパティーのリスト

Java システム・プロパティーにより、Java プログラムのランタイム環境が決まります。 Java システム・プロパティーは、IBM i のシステム値や環境変数と似ています。

Java 仮想マシン (JVM) を開始すると、JVM のそのインスタンスのシステム・プロパティーが設定されます。 Java システム・プロパティーの値の指定方法について詳しくは、以下のページを参照してください。

- 14 ページの『Java システム・プロパティー』
- 『SystemDefault.properties ファイル』

Java システム・プロパティーについて詳しくは、 280 ページの『JSSE for 1.4 Java システム・プロパティー』、 297 ページの『JSSE for 1.5 Java システム・プロパティー』、および 319 ページの『JSSE for 6 Java システム・プロパティー』を参照してください。

以下の表は、サポートされる IBM Technology for Java (5761-JV1) オプションのための Java システム・プロパティーのリストです。この表では、各プロパティーごとに、プロパティーの名前と、適用されるデフォルト値または簡略説明をリストしています。この表には、Java 2 Platform, Standard Edition (J2SE) のバージョンによって値が異なるシステム・プロパティーが示されています。デフォルト値がリストされている列に、種々の J2SE のバージョンが示されていない場合は、サポートされているすべてのバージョンの J2SE でそのデフォルト値が使用されます。

注: すべてのプロパティーがリストされているわけではありません。 IBM i に対して一意的に設定されたプロパティーのみがリストされています。

Java プロパティー	デフォルト値
file.encoding	デフォルトは、ジョブのデフォルトの言語 ID および国別 ID に基づいて設定されます。 コード化文字セット ID (CCSID) を、対応する ISO ASCII CCSID にマップします。また、`file.encoding` 値の集合を、その ISO ASCII (CCSID) を表す Java 値に設定します。 `file.encoding` 値は、JVM 始動時に指定する必要があり、実行時に変更してはなりません。デフォルトが選択される方法、および `file.encoding` に指定可能な値とそれに最も近い CCSID の関係を示した表については、 23 ページの『file.encoding の値と IBM i CCSID』を参照してください。
i5os.crypto.device	使用する暗号化装置を指定します。このプロパティーが設定されていない場合は、デフォルト装置の CRP01 が使用されます。
i5os.crypto.keystore	使用する CCA 鍵ストア・ファイルを指定します。このプロパティーが設定されていない場合は、暗号化装置の記述内で指定されている鍵ストア・ファイルが使用されます。
java.compiler	IBM Technology for Java のコンパイラー・レベル。このプロパティーは、出力の目的だけに使用されます。
java.ext.dirs	J2SE 1.4 64 ビット: • `/QOpenSys/QIBM/ProdData/JavaVM/jdk14/64bit/jre/lib/ext` • `/QIBM/UserData/Java400/ext` J2SE 5.0 32 ビット: • `/QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/jre/lib/ext` • `/QIBM/UserData/Java400/ext` J2SE 5.0 64 ビット: • `/QOpenSys/QIBM/ProdData/JavaVM/jdk50/64bit/jre/lib/ext` • `/QIBM/UserData/Java400/ext` Java SE 6 32 ビット: (デフォルト) • `/QOpenSys/QIBM/ProdData/JavaVM/jdk60/32bit/jre/lib/ext` • `/QIBM/UserData/Java400/ext` Java SE 6 64 ビット: • `/QOpenSys/QIBM/ProdData/JavaVM/jdk60/64bit/jre/lib/ext` • `/QIBM/UserData/Java400/ext`

Java プロパティー	デフォルト値
java.home	J2SE 1.4 64 ビット: /QOpenSys/QIBM/ProdData/JavaVM/jdk14/64bit J2SE 5.0 32 ビット: /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit J2SE 5.0 64 ビット: /QOpenSys/QIBM/ProdData/JavaVM/jdk50/64bit Java SE 6 32 ビット: /QOpenSys/QIBM/ProdData/JavaVM/jdk60/32bit Java SE 6 64 ビット: /QOpenSys/QIBM/ProdData/JavaVM/jdk60/64bit このプロパティーは、出力の目的だけに使用されます。詳細は 6 ページの『複数の Java Development Kit (JDK) のサポート』を参照してください。
java.library.path	このプロパティーは、アプリケーション用のネイティブ・メソッド・ライブラリー、および内部 JVM ネイティブ・ライブラリーを位置指定するために使用されます。デフォルト値は、2 つのリスト (IBM i ライブラリー・リストおよび LIBPATH 環境変数用に指定されたパス) の連結から取得されます。詳しくは、216 ページの『ネイティブ・メソッド・ライブラリーの管理』を参照してください。
java.net.preferIPv4Stack	• false (no's) - デフォルト値 • true デュアル・スタック・マシンでは、優先されるプロトコル・スタック (IPv4 または IPv6) と、優先されるアドレス・ファミリー・タイプ (inet4 または inet6) を設定するためのシステム・プロパティーが用意されています。デュアル・スタック・マシンの IPv6 ソケットは、IPv4 と IPv6 の両方のピアと対話できるので、デフォルトでは IPv6 スタックが優先されます。この設定は、このプロパティーを使用して変更できます。 詳しくは、「Networking IPv6 User Guide」を参照してください。
java.net.preferIPv6Addresses	• true • false (no's) (デフォルト値) オペレーティング・システムで IPv6 が使用可能でも、デフォルト設定では、IPv6 アドレスよりも IPv4 マップ・アドレスが優先されます。このプロパティーによって、IPv6 (true) と IPv4 (false) のどちらのアドレスを使用するかが制御されます。 詳しくは、「Networking IPv6 User Guide」を参照してください。
java.policy	J2SE 1.4: /QIBM/ProdData/OS400/Java400/jdk/lib/security/java.policy J2SE 5.0: /QIBM/ProdData/ Java400/jdk15/lib/security/java.policy Java SE 6: /QIBM/ProdData/Java400/jdk6/lib/security/java.policy (デフォルト値)
java.use.policy	true
java.vendor	IBM Corporation
java.vendor.url	http://www.ibm.com

Java プロパティー	デフォルト値
java.version	• 1.4.2 • 1.5.0 • 1.6.0 (デフォルト値) このプロパティーは、出力の目的だけに使用されます。従来は、このプロパティーは JDK の選択用に使用されました。現在では、JDK バージョンは、JAVA_HOME 環境変数の値によって判別されます。
java.vm.name	IBM J9 VM
java.vm.specification.name	Java 仮想マシンの仕様
java.vm.specification.vendor	Sun Microsystems, Inc.
java.vm.specification.version	1.0
java.vm.vendor	IBM Corporation
java.vm.version	• J2SE 1.4: 2.3 • J2SE 5.0: 2.3 • Java SE 6: 2.4
os.arch	PowerPC®
os.name	OS/400®
os.version	V7R1M0 (デフォルト値) IBM i のリリース・レベルを、Retrieve Product Information アプリケーション・プログラミング・インターフェース (API) から取得します。
os400.certificateContainer	Java Secure Sockets Layer (SSL) サポートで、開始済みの Java プログラムおよび指定済みのプロパティーの証明書コンテナーとして指定したものが使用されるようにします。 `os400.secureApplication` システム・プロパティーを指定すると、このプロパティーは無視されます。例えば、`-Dos400.certificateContainer=/home/username/mykeyfile.kdb` と入力するか、または統合ファイル・システム内の他の鍵ファイルを入力してください。
os400.certificateLabel	このシステム・プロパティーは、`os400.certificateContainer` システム・プロパティーと一緒に指定できます。このプロパティーを指定すると、指定されているコンテナー中のどの証明書が、Secure Sockets Layer (SSL) で使用されるか選択できます。たとえば、`-Dos400.certificateLabel=myCert` (`myCert` は、証明書の作成時かインポート時にディジタル証明書マネージャー (DCM) によってその証明書に割り当てられるラベル名) と入力してください。
os400.child.stdio.convert	Java での stdin、stdout、および stderr に関するデータ変換を制御します。 ASCII データと Extended Binary Coded Decimal Interchange Code (EBCDIC) 間のデータ変換は、Java 仮想マシンにデフォルトで存在します。このプロパティーを使用してこれらの変換をオンにしたりオフにしたりする場合、その設定は、このプロセスが `Runtime.exec()` を使用して開始する子プロセスに対してのみ作用します。この場合、実行されるコマンドは、Java に基づくコマンドです。 子プロセスでは、このプロパティー値が `os400.stdio.convert` のデフォルト値になります。 20 ページの『os400.stdio.convert および os400.child.stdio.convert システム・プロパティーの値』を参照してください。

Java プロパティー	デフォルト値
os400.class.path.security.check	20 (デフォルト値) 有効値: • 0 セキュリティー検査なし • 10 RUNJVA CHKPATH(*IGNORE) と同等 • 20 RUNJVA CHKPATH(*WARN) と同等 • 30 RUNJVA CHKPATH(*SECURE) と同等
os400.class.path.tools	0 (デフォルト値) 有効値: • 0: `java.class.path` プロパティーに Sun ツールを含めない • 1: `java.class.path` プロパティーの前に J2SE の固有のツール・ファイルを付加する tools.jar へのパス: • J2SE 1.4 64 ビット: `/QOpenSys/QIBM/ProdData/JavaVM/jdk14/64bit/lib` • J2SE 5.0 32 ビット: `/QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/lib` • J2SE 5.0 64 ビット: `/QOpenSys/QIBM/ProdData/JavaVM/jdk50/64bit/lib` • Java SE 6 32 ビット (デフォルト): `/QOpenSys/QIBM/ProdData/JavaVM/jdk60/32bit/lib` • Java SE 6 64 ビット: `/QOpenSys/QIBM/ProdData/JavaVM/jdk60/64bit/lib`
os400.display.properties	この値が 'true' に設定されている場合、 Java 仮想マシンのプロパティーのすべてが標準値に設定されます。その他の値は認識されません。
os400.file.create.auth、 os400.dir.create.auth	これらのプロパティーは、ファイルやディレクトリーに割り当てる権限を指定します。このプロパティーに何も値を指定しない場合や、サポートされていない値を指定した場合、権限は共通権限 *NONE になります。 `os400.file.create.auth=RWX` または `os400.dir.create.auth=RWX` (R= 読み取り、W= 書き込み、X= 実行) を指定できます。これらの権限は、どのような組み合わせでも有効です。
os400.job.file.encoding	このプロパティーは、出力の目的だけに使用されます。これにより、JVM が稼働中の IBM i ジョブのジョブ CCSID に等価の文字エンコードがリストされます。
os400.secureApplication	このシステム・プロパティー (`os400.secureApplication`) の使用中に開始される Java プログラムと、登録済みの保護アプリケーション名を関連付けます。ディジタル証明書マネージャー (DCM) を使用すると、登録済みの保護アプリケーション名を参照できます。
os400.security.properties	どの `java.security` ファイルを使用するかに関する全制御を許可します。このプロパティーを指定すると、J2SE は、J2SE 特有の `java.security` デフォルト・ファイルを含め、その他のどの `java.security` ファイルも使用しません。

Java プロパティー	デフォルト値
os400.stderr	stderr をファイルまたはソケットにマッピングできます。 21 ページの『os400.stdin、os400.stdout、および os400.stderr システム・プロパティー値』を参照してください。
os400.stdin	stdin をファイルまたはソケットにマッピングできます。 21 ページの『os400.stdin、os400.stdout、および os400.stderr システム・プロパティー値』を参照してください。
os400.stdin.allowed	stdin を使用できる (1) か、使用できない (0) かを指定します。デフォルト値は、対話式ジョブの場合は 1、バッチ・ジョブの場合は 0 です。
os400.stdio.convert	Java での stdin、stdout、および stderr に関するデータ変換を制御できます。デフォルトでは、 Java 仮想マシンでは ASCII データと EBCDIC データの間のデータ変換が行われます。このプロパティーを指定して、この種の変換をオン/オフにすることができます。この指定は、現行の Java プログラムに反映されます。『os400.stdio.convert および os400.child.stdio.convert システム・プロパティーの値』を参照してください。 `Runtime.exec()` メソッドを使用して開始される Java プログラムについては、os400.child.stdio.convert を参照してください。
os400.stdout	stdout をファイルまたはソケットにマッピングできます。 デフォルト値を参照してください。
os400.xrun.option	このプロパティーは、`java` コマンドの `-Xrun` オプションの代わりに使用して、JVM 始動時にエージェント・プログラムを実行することができます。
os400.vm.inputargs	このプロパティーは、出力の目的だけに使用されます。これは、JVM が入力として受け取った引数を表示します。このプロパティーは、JVM 始動時に指定されたものをデバッグするために役立ちます。
user.timezone	• JVM は、現行ジョブの QTIMZON 値を使用することによって、このプロパティーの値を選択します。このオブジェクトの「Alternate Name」フィールドの名前は、このプロパティーのために使用される値です。「Alternate Name」フィールドの値は長さが 3 文字以上でなければならず、それより短い場合は使用されません。 • QTIMZON オブジェクトの「Alternate Name」フィールドに指定された値の長さが 3 文字に満たない場合、JVM は現行のシステム・オフセットに基づいて一致する GMT 値の検出を試みます。例えば、「Alternate Name」フィールドが空で、オフセットが -5 の QTIMZON オブジェクトの場合は、設定は `user.timezone=GMT-5` となります。 • それでも値が検出されない場合は、JVM はデフォルトで user.timezone を協定世界時 (UTC) に設定します。 詳細は、WebSphere® Software Information Center の『サポートされるタイム・ゾーンの値』を参照してください。

関連概念

12 ページの『Java を使用するための IBM i サーバーのカスタマイズ』
ご使用のサーバーに Java をインストールした後、サーバーをカスタマイズすることができます。

os400.stdio.convert および os400.child.stdio.convert システム・プロパティーの値:

以下の表は、`os400.stdio.convert` システム・プロパティーと `os400.child.stdio.convert` システム・プロパティーのシステム値を示しています。

表 2. **os400.stdio.convert** のシステム値

値	説明
Y (デフォルト)	読み取りまたは書き込み中の、 `file.encoding` 値とジョブ CCSID との間のすべての stdio 変換。
N	読み取りまたは書き込み中に stdio 変換は実行されません。

表 3. **os400.child.stdio.convert** のシステム値

値	説明
N (デフォルト)	読み取りまたは書き込み中に stdio 変換は実行されません。
Y	読み取りまたは書き込み中の、 `file.encoding` 値とジョブ CCSID との間のすべての stdio 変換。

os400.stdin、os400.stdout、および os400.stderr システム・プロパティー値:

以下の表は、os400.stdin、os400.stdout、および os400.stderr システム・プロパティーのシステム値を示しています。

値	例の名前	説明	例
ファイル	SomeFileName	SomeFileName は、現行ディレクトリーに対する絶対パスか相対パスです。	file:/QIBM/UserData/ Java400/Output.file
ポート	HostName	ポート・アドレス	port:myhost:2000
ポート	TCPAddress	ポート・アドレス	port:1.1.11.111:2000

国際化対応

国際化 Java プログラムを作成することによって、Java プログラムを世界の特定の地域用にカスタマイズすることができます。時間帯、ロケール、および文字エンコード方式を使用することにより、Java プログラムが正しい時刻、場所、および言語を反映するようにできます。

IBM i グローバリゼーション

Sun Microsystems, Inc. による「Internationalization」

時間帯構成

時間帯に依存する Java プログラムがある場合は、その Java プログラムが正しい時間を使用するようにシステム上で時間帯を構成する必要があります。

時間帯を構成する最も簡単な方法は、QTIMZON システム値を IBM i で提供されているいずれかの *TIMZON オブジェクトに設定することです。Java 仮想マシン (JVM) が地方時を正しく判断するためには、QUTCOFFSET システム値と `user.timezone` Java システム・プロパティーの両方が正しく設定されている必要があります。QTIMZON システム値を設定すると、これらの両方が設定されます。TIMZON オブジェクトには、使用する Java `user.timezone` の値を指定する代替ロング・ネームが含まれているため、QTIMZON 値は、該当する代替名が含まれているものを選択する必要があります。例えば、TIMZON オブジェクト QN0600CST2 には、代替名 America/Chicago が含まれており、米国中部時間帯の正しい時刻サポートを提供します。

| **注:** QTIMZON システム値によって設定される user.timezone システム・プロパティー設定は、コマンド
| 行または SystemDefault.properties ファイル内で、明示的に user.timezone 値を指定することによって
| オーバーライドできます。これによって、各 Java ジョブは固有の user.timezone 値を持ち、複数の
| 時間帯を同じシステム上でサポートすることができます。

IBM i システム値: QTIMZON

「時間帯記述処理 (WRKTIMZON)」CL コマンド

Sun Microsystems, Inc. による TimeZone Javadoc の参照情報

Java 文字のエンコード

Java プログラムは、他のフォーマットのデータを変換して、アプリケーションが多様な国際文字セットの情報を転送および使用できるようにします。

Java 仮想マシン (JVM) は、内部では常に Unicode 形式でデータを扱います。ただし、JVM が外部とやり取りするすべてのデータは、file.encoding プロパティーと一致したフォーマットになっています。 JVM が読み取るデータは file.encoding から Unicode に変換され、JVM から送信されるデータは Unicode から file.encoding へ変換されます。

Java プログラムのデータ・ファイルは、統合ファイル・システム (IFS) に保管されています。統合ファイル・システムの中のファイルは、コード化文字セット識別コード (CCSID) でタグ付けされており、これによってファイル内に含まれているデータの文字エンコード方式を識別します。

Java プログラムが読み取るデータは、file.encoding と一致する文字エンコード方式であることが要求されます。 Java プログラムがファイルに書き込むデータは、file.encoding と一致する文字エンコード方式で書き込まれます。このことは、javac コマンドが処理する Java のソース言語ファイル (.java ファイル) や、java.net パッケージを使用して伝送制御プロトコル/インターネット・プロトコル (TCP/IP) ソケットを介して送受信されるデータにも当てはまります。

System.in、System.out、および System.err で読み書きされるデータの処理方法は、stdin、stdout、および stderr に割り当てられた他のソースで読み書きされるデータの処理方法とは異なります。 stdin、stdout、stderr は通常、IBM i サーバーの EBCDIC 装置に接続されているので、データは JVM によって通常の file.encoding の文字コード方式から IBM i ジョブの CCSID と一致する CCSID に変換されます。System.in、System.out、System.err のいずれかがファイルやソケットにリダイレクトされ、stdin、stdout、stderr のいずれにも送信されない場合、この付加的な変換は実行されず、データは file.encoding と一致するエンコード方式のままになります。

Java プログラムで、file.encoding 以外のエンコード方式を使ってデータを読み書きする必要がある場合は、プログラムで Java の IO クラス (java.io.InputStreamReader、java.io.FIleReader、java.io.OutputStreamReader、および java.io.FileWriter) を使用することができます。 Java クラスを使用すれば、JVM が現在使用しているデフォルトの file.encoding プロパティーよりも優先される file.encoding 値を指定することができます。

DB2® データベースとの間で受け渡されるデータは JDBC API を介して IBM i データベースの CCSID との間で双方向に変換されます。

Java ネイティブ・インターフェースを介して他のプログラムとの間で転送されるデータは、変換されません。

グローバリゼーション

Sun Microsystems, Inc. による「Internationalization」

file.encoding の値と IBM i CCSID:

次の表は、`file.encoding` に指定可能な値と、それに最も近い IBM i コード化文字セット識別コード (CCSID) の関係を示したものです。

`file.encoding` サポートについて詳しくは、Sun Microsystems, Inc. が提供する「Supported encodings」 を参照してください。

file.encoding	CCSID	説明
ASCII	367	情報交換用米国標準コード
Big5	950	8 ビット ASCII 中国語 (繁体字) BIG-5
Big5_HKSCS	950	Big5_HKSCS
Big5_Solaris	950	Solaris zh_TW.BIG5 ロケール用の 7 つの追加の繁体字マッピングを含む Big5
CNS11643	964	中国語 (繁体字) の中国文字セット
Cp037	037	IBM EBCDIC 米国、カナダ、オランダ、...
Cp273	273	IBM EBCDIC ドイツ、オーストリア
Cp277	277	IBM EBCDIC デンマーク、ノルウェー
Cp278	278	IBM EBCDIC フィンランド、スウェーデン
Cp280	280	IBM EBCDIC イタリア
Cp284	284	IBM EBCDIC ラテンアメリカ・スペイン語
Cp285	285	IBM EBCDIC 英国
Cp297	297	IBM EBCDIC フランス
Cp420	420	IBM EBCDIC アラビア語
Cp424	424	IBM EBCDIC ヘブライ語
Cp437	437	8 ビット ASCII US PC
Cp500	500	IBM EBCDIC 国際
Cp737	737	8 ビット ASCII ギリシャ語 MS-DOS
Cp775	775	8 ビット ASCII バルト語 MS-DOS
Cp838	838	IBM EBCDIC タイ語
Cp850	850	8 ビット ASCII Latin-1 多国語
Cp852	852	8 ビット ASCII Latin-2
Cp855	855	8 ビット ASCII キリル文字使用言語
Cp856	0	8 ビット ASCII ヘブライ語
Cp857	857	8 ビット ASCII Latin-5
Cp860	860	8 ビット ASCII ポルトガル語
Cp861	861	8 ビット ASCII アイスランド語
Cp862	862	8 ビット ASCII ヘブライ語
Cp863	863	8 ビット ASCII カナダ
Cp864	864	8 ビット ASCII アラビア語
Cp865	865	8 ビット ASCII デンマーク、ノルウェー
Cp866	866	8 ビット ASCII キリル文字使用言語
Cp868	868	8 ビット ASCII ウルドゥー語
Cp869	869	8 ビット ASCII ギリシャ語

file.encoding	CCSID	説明
Cp870	870	IBM EBCDIC Latin-2
Cp871	871	IBM EBCDIC アイスランド
Cp874	874	8 ビット ASCII タイ語
Cp875	875	IBM EBCDIC ギリシャ語
Cp918	918	IBM EBCDIC ウルドゥー語
Cp921	921	8 ビット ASCII バルト語
Cp922	922	8 ビット ASCII エストニア語
Cp930	930	IBM EBCDIC 日本語拡張カタカナ
Cp933	933	IBM EBCDIC 韓国語
Cp935	935	IBM EBCDIC 中国語 (簡体字)
Cp937	937	IBM EBCDIC 中国語 (繁体字)
Cp939	939	IBM EBCDIC 日本語拡張ローマ字
Cp942	942	8 ビット ASCII 日本語
Cp942C	942	Cp942 の変種
Cp943	943	日本語オープン環境用混合 PC データ
Cp943C	943	日本語オープン環境用混合 PC データ
Cp948	948	8 ビット ASCII IBM 中国語 (繁体字)
Cp949	944	8 ビット ASCII 韓国語 KSC5601
Cp949C	949	Cp949 の変種
Cp950	950	8 ビット ASCII 中国語 (繁体字) BIG-5
Cp964	964	EUC 中国語 (繁体字)
Cp970	970	EUC 韓国語
Cp1006	1006	ISO 8 ビット ウルドゥー語
Cp1025	1025	IBM EBCDIC キリル文字
Cp1026	1026	IBM EBCDIC トルコ語
Cp1046	1046	8 ビット ASCII アラビア語
Cp1097	1097	IBM EBCDIC ペルシア語
Cp1098	1098	8 ビット ASCII ペルシア語
Cp1112	1112	IBM EBCDIC バルト語
Cp1122	1122	IBM EBCDIC エストニア語
Cp1123	1123	IBM EBCDIC ウクライナ
Cp1124	0	ISO 8 ビット ウクライナ
Cp1140	1140	ユーロ文字を含む Cp037 の変種
Cp1141	1141	ユーロ文字を含む Cp273 の変種
Cp1142	1142	ユーロ文字を含む Cp277 の変種
Cp1143	1143	ユーロ文字を含む Cp278 の変種
Cp1144	1144	ユーロ文字を含む Cp280 の変種
Cp1145	1145	ユーロ文字を含む Cp284 の変種
Cp1146	1146	ユーロ文字を含む Cp285 の変種
Cp1147	1147	ユーロ文字を含む Cp297 の変種
Cp1148	1148	ユーロ文字を含む Cp500 の変種

file.encoding	CCSID	説明
Cp1149	1149	ユーロ文字を含む Cp871 の変種
Cp1250	1250	MS-Win Latin-2
Cp1251	1251	MS-Win キリル文字使用言語
Cp1252	1252	MS-Win Latin-1
Cp1253	1253	MS-Win ギリシャ語
Cp1254	1254	MS-Win トルコ語
Cp1255	1255	MS-Win ヘブライ語
Cp1256	1256	MS-Win アラビア語
Cp1257	1257	MS-Win バルト語
Cp1258	1251	MS-Win ロシア語
Cp1381	1381	8 ビット ASCII 中国語 (簡体字) GB
Cp1383	1383	EUC 中国語 (簡体字)
Cp33722	33722	EUC 日本語
EUC_CN	1383	EUC 中国語 (簡体字)
EUC_JP	5050	EUC 日本語
EUC_JP_LINUX	0	JISX 0201、0208、EUC エンコードの日本語
EUC_KR	970	EUC 韓国語
EUC_TW	964	EUC 中国語 (繁体字)
GB2312	1381	8 ビット ASCII 中国語 (簡体字) GB
GB18030	1392	中国語 (簡体字)、PRC 標準
GBK	1386	8 ビット ASCII 9 中国語 (新簡体字)
ISCII91	806	インド語文字の ISCII91 エンコード
ISO2022CN	965	ISO 2022 CN、中国語 (Unicode への変換のみ)
ISO2022_CN_CNS	965	ISO 2022 CN 形式の CNS11643、中国語 (繁体字) (Unicode への変換のみ)
ISO2022_CN_GB	1383	ISO 2022 CN 形式の GB2312、中国語 (簡体字) (Unicode からの変換のみ)
ISO2022CN_CNS	965	7 ビット ASCII 中国語 (繁体字)
ISO2022CN_GB	1383	7 ビット ASCII 中国語 (簡体字)
ISO2022JP	5054	7 ビット ASCII 日本語
ISO2022KR	25546	7 ビット ASCII 韓国語
ISO8859_1	819	ISO 8859-1 Latin Alphabet No. 1
ISO8859_2	912	ISO 8859-2 ISO Latin-2
ISO8859_3	0	ISO 8859-3 ISO Latin-3
ISO8859_4	914	ISO 8859-4 ISO Latin-4
ISO8859_5	915	ISO 8859-5 ISO Latin-5
ISO8859_6	1089	ISO 8859-6 ISO Latin-6 (アラビア語)
ISO8859_7	813	ISO 8859-7 ISO Latin-7 (ギリシャ語/ラテン語)
ISO8859_8	916	ISO 8859-8 ISO Latin-8 (ヘブライ語)
ISO8859_9	920	ISO 8859-9 ISO Latin-9 (ECMA-128、トルコ語)
ISO8859_13	0	Latin Alphabet No. 7
ISO8859_15	923	ISO8859_15
ISO8859_15_FDIS	923	ISO 8859-15、Latin alphabet No. 9

file.encoding	CCSID	説明
ISO-8859-15	923	ISO 8859-15、Latin Alphabet No. 9
JIS0201	897	日本工業規格 X0201
JIS0208	5052	日本工業規格 X0208
JIS0212	0	日本工業規格 X0212
JISAutoDetect	0	Shift-JIS、EUC-JP、ISO 2022 JP を検出し、変換する (Unicode への変換のみ)
Johab	0	韓国構成ハングル・エンコード (全)
K018_R	878	キリル語
KSC5601	949	8 ビット ASCII 韓国語
MacArabic	1256	Macintosh アラビア語
MacCentralEurope	1282	Macintosh Latin-2
MacCroatian	1284	Macintosh クロアチア語
MacCyrillic	1283	Macintosh キリル文字
MacDingbat	0	Macintosh Dingbat
MacGreek	1280	Macintosh ギリシャ語
MacHebrew	1255	Macintosh ヘブライ語
MacIceland	1286	Macintosh アイスランド語
MacRoman	0	Macintosh Roman
MacRomania	1285	Macintosh ルーマニア
MacSymbol	0	Macintosh シンボル
MacThai	0	Macintosh タイ
MacTurkish	1281	Macintosh トルコ語
MacUkraine	1283	Macintosh ウクライナ
MS874	874	MS-Win タイ
MS932	943	Windows 日本語
MS936	936	Windows 中国語 (簡体字)
MS949	949	Windows 韓国語
MS950	950	Windows 中国語 (繁体字)
MS950_HKSCS	NA	中国香港特別行政区拡張を含む WindowsWindows 中国語 (繁体字)
SJIS	932	8 ビット ASCII 日本語
TIS620	874	タイ工業規格 620
US-ASCII	367	情報交換用米国標準コード
UTF8	1208	UTF-8
UTF-16	1200	16 ビット UCS 変換フォーマット、オプションのバイト・オーダー・マークによって示されるバイト・オーダー
UTF-16BE	1200	16 ビット Unicode 変換フォーマット、ビッグ・エンディアン・バイト・オーダー
UTF-16LE	1200	16 ビット Unicode 変換フォーマット、リトル・エンディアン・バイト・オーダー
UTF-8	1208	8 ビット UCS 変換フォーマット
Unicode	13488	UNICODE、UCS-2
UnicodeBig	13488	Unicode と同じ

file.encoding	CCSID	説明
UnicodeBigUnmarked		Unicode (バイト・オーダー・マークなし)
UnicodeLittle		Unicode (リトル・エンディアン・バイト・オーダー)
UnicodeLittleUnmarked		UnicodeLittle (バイト・オーダー・マークなし)

デフォルト値については、file.encoding のデフォルト値を参照してください。

file.encoding のデフォルト値:

この表は、Java 仮想マシンの起動時に `file.encoding` 値が PASE for i コード化文字セット ID (CCSID) に基づいてどのように設定されるかを示しています。

注: PASE for i CCSID は、ジョブの言語 ID および国別 ID に基づいて設定されます。 PASE for i が使用する CCSID を決定する方法について詳しくは、『IBM PASE for i ロケール』を参照してください。

PASE for i CCSID	`file.encoding` のデフォルト	説明
813	ISO8859_7	Latin-7 (ギリシャ語/ラテン語)
819	ISO8859_1	Latin-1
874	TIS620	タイ語
912	ISO8859_2	Latin-2 (チェコ語/チェコ共和国、クロアチア語/クロアチア、ハンガリー語/ハンガリー、ポーランド語/ポーランド)
915	ISO8859_5	キリル文字 8 ビット (ブルガリア)
916	ISO8859_8	ヘブライ語 (イスラエル)
920	ISO8859_9	Latin-5 (トルコ拡張)
921	Cp921	バルト語 8 ビット (リトアニア語/リトアニア、ラトビア語/ラトビア)
922	Cp922	エストニア ISO-8
923	ISO8859_15	Latin-9
1046	Cp1046	Windows アラビア語
1089	ISO8859_6	アラビア語
1208	UTF-8	8 ビット UCS 変換フォーマット
1252	Cp1252	Windows Latin-1

例: 国際化 Java プログラムを作成する

特定の地域に Java プログラムをカスタマイズする必要がある場合は、Java ロケールを使用して、国際化 Java プログラムを作成できます。

Java ロケール。

国際化 Java プログラムの作成には、以下のようないくつかのタスクが関係します。

1. ロケール依存のコードとデータを分離する。たとえば、プログラム内のストリング、日付、数値などです。
2. Locale クラスを使って、ロケールを設定または取得する。

3. デフォルト・ロケールを使用したくない場合は、日付と数値をフォーマットしてロケールを指定する。
4. ストリングとその他のロケール依存データを処理するためのリソース・バンドルを作成する。

以下の例を参照してください。国際化 Java プログラムを作成するために必要なタスクを完了するための方法が参照できます。

- 424 ページの『例: java.util.DateFormat クラスを使用して日付を国際化する』
- 424 ページの『例: java.util.NumberFormat クラスを使用して数値表示を国際化する』
- 425 ページの『例: java.util.ResourceBundle クラスを使用してロケール固有データを国際化する』

グローバリゼーション

Sun Microsystems, Inc. による「Internationalization」

リリース間の互換性

このトピックでは、Java アプリケーションを以前のリリースから最新リリースに移行する場合の考慮事項について説明します。

Java アプリケーションを現行リリースで実行する際には、以下の互換性の問題を考慮する必要があります。

- IBM Technology for Java は、PASE for i からの JVMTI インターフェースのみをサポートしています。その結果、JVMTI エージェントを PASE for i に移植する必要があります。
- PASE for i ネイティブ・メソッドを使用する際には、ネイティブ・コードのアーキテクチャーが、JVM のアーキテクチャーに一致する必要があります。つまり、オブジェクト・バイナリーは、32 ビット JVM の場合は 32 ビット・バイナリーとして、64 ビット JVM の場合は 64 ビット・バイナリーとしてコンパイルする必要があります。これはユーザー提供の JVMTI エージェントなどのエージェントにも当てはまります。
- PASE for i コードで例外が発生した際には、Classic JVM とは異なる動作が予期されます。 Classic JVM であれば、エラーを検出して処理し、適切な Java 例外に変換できました。しかし、IBM Technology for Java の場合は、これによって通常、JVM が終了します。
- Java システム・プロパティーの `java.version` は、IBM Technology for Java JVM の入力プロパティーとしては認識されません。従来のリリースでは、使用する JDK を判別するための入力として `java.version` Java システム・プロパティーを反映する Classic JVM が使用可能でした。 IBM i 7.1 からは、IBM Technology for Java が唯一使用可能な JVM となり、使用する JDK を判別するためには、環境変数 JAVA_HOME を指定する必要があります。
- Classic JVM では、Java メソッド `System.getenv()` によって、適切な ILE 環境変数の値が戻されました。 IBM Technology for Java では、代わりに PASE for i 環境変数が戻されます。これによって、ユーザーが ILE ネイティブ・メソッドで環境変数を設定し、後で `System.getenv()` を呼び出してこれを取り出すことを想定する場合に問題が発生する可能性があります。一般に、ユーザーは、ILE と PASE for i がそれぞれ異なる環境変数セットを持っていることに注意する必要があります。
- 直接処理のサポートは、IBM i 6.1 で停止されました。 IBM i 7.1 でも Java プログラム・コマンドは引き続きサポートされていますが、従来のリリースを対象として使用する場合に限られています。詳しくは、IBM i 6.1 の『リリース間の互換性』のセクションを参照してください。

関連概念

1 ページの『IBM i 7.1 の新機能』
IBM Developer Kit for Java のトピック・コレクションで新しく追加された点や大幅に変更された点について説明します。

Java プログラムからのデータベース・アクセス

Java プログラムはいくつかの方法でデータベース・ファイルにアクセスできます。

Java JDBC ドライバーを使用して IBM i データベースにアクセスする

「ネイティブ」ドライバーとも呼ばれる Java JDBC ドライバーは、IBM i データベース・ファイルへのプログラマチック・アクセスを提供します。 Java Database Connectivity (JDBC) API を使用すれば、Java 言語で作成されたアプリケーションは、 組み込まれた構造化照会言語 (SQL) を使って JDBC データベースの機能にアクセスしたり、 SQL ステートメントを実行したり、結果を検索したり、変更をデータベースに戻したりできます。また、JDBC API を使用して、分散した異機種混合環境内の複数のデータ・ソースと対話できます。

JDBC API のベースである SQL99 コマンド言語インターフェース (CLI) は、ODBC の基本となるものです。 JDBC は、Java プログラム言語から、SQL 標準で定義されている抽象および概念への、自然な使いやすいマッピングを提供します。

Sun Microsystems, Inc. による JDBC 資料

Native JDBC Driver FAQs

JDBC 4.0 API Specification

JDBC 入門

IBM i 上の Java に付属している Java Database Connectivity (JDBC) ドライバーのことを、IBM Developer Kit for Java の JDBC ドライバーと呼びます。このドライバーは、一般にネイティブ JDBC ドライバーとも呼ばれます。

どの JDBC ドライバーが必要にかなうかを選択する場合、以下の提案を考慮してください。

- データベースが置かれているサーバーで直接実行するプログラムは、パフォーマンス上の理由で、ネイティブ JDBC ドライバーを使用すべきです。これには、ほとんどのサーブレットおよび JavaServer Pages (JSP) ソリューション、およびシステムでローカルに実行するように作成されているアプリケーションが含まれます。
- リモート IBM i サーバーに接続しなければならないプログラムは、IBM Toolbox for Java JDBC クラスを使用します。この IBM Toolbox for Java JDBC ドライバーは JDBC の堅固なインプリメンテーションであり、IBM Toolbox for Java の一部として提供されています。 IBM Toolbox for Java JDBC ドライバーは Pure Java であるため、クライアント用にセットアップするのが容易であり、サーバーのセットアップがほとんど必要ありません。
- IBM iサーバー 上で実行され、リモートにある非 IBM i データベースへの接続を必要とするプログラムは、ネイティブ JDBC ドライバーを使用し、そのリモート・サーバーに対してDistributed Relational Database Architecture™ (DRDA®) 接続をセットアップします。

JDBC ドライバーのタイプ:

このトピックでは、Java Database Connectivity (JDBC) ドライバーのタイプを定義します。ドライバーのタイプは、データベースに接続するために使用されるテクノロジーを分類するために使用されます。 JDBC ドライバーのベンダーは、製品の動作方法を記述するためにこれらのタイプを使用します。一部のアプリケーションには、一部の JDBC ドライバーのタイプの方が、他のタイプよりも向いています。

タイプ 1

タイプ 1 のドライバーは、「ブリッジ」ドライバーです。これらのドライバーは、データベースと通信するために Open Database Connectivity (ODBC) などの別のテクノロジーを使用します。これは利点となります。多くのリレーショナル・データベース管理システム (RDBMS) プラットフォーム用の ODBC ドライバーが存在するからです。 JDBC ドライバーから ODBC 機能を呼び出すために、Java Native Interface (JNI) が使用されます。

タイプ 1 のドライバーで JDBC を使用するには、その前にブリッジ・ドライバーがインストールおよび構成されている必要があります。これは、実動アプリケーションにとって重大な欠点となり得ます。アプレットはネイティブ・コードをロードできないので、タイプ 1 のドライバーをアプレットで使用することはできません。

タイプ 2

タイプ 2 のドライバーは、データベース・システムと通信するためにネイティブ API を使用します。データベース操作を実行する API 関数を呼び出すために、Java ネイティブ・メソッドが使用されます。タイプ 2 のドライバーは、一般にタイプ 1 のドライバーよりも高速です。

タイプ 2 のドライバーが機能するためには、ネイティブ・バイナリー・コードがインストールおよび構成されている必要があります。タイプ 2 のドライバーも JNI を使用します。アプレットはネイティブ・コードをロードできないので、タイプ 2 のドライバーをアプレットで使用することはできません。タイプ 2 の JDBC ドライバーでは、何らかのデータベース管理システム (DBMS) ネットワーキング・ソフトウェアがインストールされていなければならない場合があります。

Developer Kit for Java の JDBC ドライバーは、タイプ 2 の JDBC ドライバーです。

タイプ 3

これらのドライバーは、サーバーと通信するためにネットワーク・プロトコルとミドルウェアを使用します。次いでサーバーは、プロトコルを DBMS 固有の DBMS 関数呼び出しに変換します。

タイプ 3 の JDBC ドライバーは、クライアント上にネイティブ・バイナリー・コードを必要としないので、最も柔軟な JDBC ソリューションです。タイプ 3 のドライバーは、クライアント側でのインストールを必要としません。

タイプ 4

タイプ 4 のドライバーは、DBMS ベンダー・ネットワーク・プロトコルを実装するために Java を使用します。通常、プロトコルはメーカー独自仕様なので、一般に DBMS のベンダーはタイプ 4 の JDBC ドライバーの唯一の提供元です。

タイプ 4 のドライバーは、すべて Java ドライバーです。つまり、クライアント側でインストールや構成が行われないという意味です。ただし、タイプ 4 のドライバーは、基礎プロトコルがセキュリティーやネットワーク接続性などの問題をうまく処理できない場合、一部のアプリケーションには向いていません。

IBMToolbox for Java JDBC ドライバーは、タイプ 4 の JDBC ドライバーであり、これは API が Pure Java ネットワーキング・プロトコル・ドライバーであることを示しています。

JDBC の要件:

このトピックでは、コア JDBC および Java Transaction API (JTA) にアクセスするために必要な要件を示します。

JDBC アプリケーションを作成および展開する前に、特定の JAR ファイルをクラスパスに組み込むことが必要になる場合があります。

コア JDBC

ローカル・データベースへのコア Java Database Connectivity (JDBC) アクセスの場合、要件はありません。すべてのサポートが組み込まれ、プリインストールされ、構成済みとなっています。

JDBC の準拠

ネイティブ JDBC ドライバーは、すべての関連した JDBC 仕様に準拠しています。 JDBC ドライバーの準拠レベルは、IBM i のリリースとは無関係ですが、使用する JDK のリリースに依存しています。各種 JDK のネイティブ JDBC ドライバーの準拠レベルは、以下のリストのとおりです。

バージョン	JDBC ドライバーの準拠レベル
JDK 1.4 および後続のバージョン	これらの JDK バージョンは JDBC 3.0 に準拠しています。
J2SE 6 および後続のバージョン	JDBC 4.0

JDBC チュートリアル:

以下は、Java Database Connectivity (JDBC) プログラムを作成し、ネイティブ JDBC ドライバーが組み込まれた IBM i 上で、そのプログラムを実行する方法についてのチュートリアルです。このチュートリアルは、プログラムで JDBC を実行するために必要な基本的なステップを示すように設計されています。

例では、テーブルを作成してデータを挿入します。プログラムは、そのデータをデータベースから取り出して画面に表示するための照会を処理します。

サンプル・プログラムの実行

サンプル・プログラムを実行するには、以下のステップを実行してください。

1. プログラムをワークステーションにコピーする。
   a. 例をコピーして、ワークステーション上のファイルにペーストする。
   b. 提供されている共通クラスと同じ名前を使用し、.java 拡張子を付けてファイルを保管する。この場合、ローカル・ワークステーション上で、ファイルに BasicJDBC.java という名前を付ける必要があります。
2. ファイルをワークステーションからサーバーに転送する。コマンド・プロンプトから、以下のコマンドを入力します。

```
ftp <server name>
<Enter your user ID>
<Enter your password>
cd /home/cujo
put BasicJDBC.java
quit
```

これらのコマンドが機能するには、ファイルを書き込むディレクトリーが存在していなければなりません。この例では、/home/cujo が書き込み先の場所ですが、任意の場所を使用できます。

注: 上記の FTP コマンドは、サーバーがどのようにセットアップされているかによって異なる場合がありますが、類似のコマンドとなります。ファイルを統合ファイル・システム内に転送する限り、どのような方法でファイルをサーバーに転送しても構いません。

3. 実行時にそれらの Java コマンドがファイルを発見できるように、そのファイルが置かれているディレクトリーを示すクラスパスが設定されていることを確認する。CL コマンド行から WRKENVVAR を使用して、ユーザー・プロファイルに設定されている環境変数を調べることができます。
   - CLASSPATH という名前の環境変数を見つけたら、そこにリストされている一連のディレクトリーの中に .java ファイルを置くようにするか、場所が指定されていない場合は追加する。
   - CLASSPATH 環境変数がない場合は、追加する必要がある。これを行うには、以下のコマンドを使用します。

     ```
     ADDENVVAR ENVVAR(CLASSPATH)
     VALUE('/home/cujo:/QIBM/ProdData/Java400/jdk15/lib/tools.jar')
     ```

   注: Java コードを CL コマンドからコンパイルするには、tools.jar ファイルを組み込む必要があります。この JAR ファイルには、javac コマンドが入っています。

4. Java ファイルをクラス・ファイルにコンパイルする。 CL コマンド行から、以下のコマンドを入力します。

   ```
   JAVA CLASS(com.sun.tools.javac.Main) PARM(My_Program_Name.java)
   java BasicJDBC
   ```

   以下のようにして、Java ファイルを QSH からコンパイルすることもできます。

   ```
   cd /home/cujo
   javac BasicJDBC.java
   ```

   QSH は、自動的に tools.jar ファイルが検出されるようにします。その結果、このファイルをクラスパスに追加する必要はなくなります。現行ディレクトリーもクラスパス内にあります。ディレクトリーの変更 (cd) コマンドを発行することによっても、BasicJDBC.java ファイルを検出できます。

   注: ファイルをワークステーション上でコンパイルし、FTP を使用してバイナリー・モードでクラス・ファイルをサーバーに送信するともできます。これは、どのプラットフォーム上でも実行できるという Java の機能の一例です。

   CL コマンド行または QSH から以下のコマンドを使用して、プログラムを実行する。

   ```
   java BasicJDBC
   ```

   出力は以下のようになります。

   ```
   ----------------------
   | 1 | Frank Johnson  |
   |                    |
   | 2 | Neil Schwartz  |
   |                    |
   | 3 | Ben Rodman     |
   |                    |
   ```

```
| 4 | Dan Gloore      |
----------------------
There were 4 rows returned.
Output is complete.
Java program completed.
```

IBM Toolbox for Java JDBC driver Web サイト

Sun Microsystems, Inc. の JDBC のページ

例: JDBC:

以下に、BasicJDBC プログラムの使用法の例を示します。このプログラムでは、IBM Developer Kit for Java のネイティブ JDBC ドライバーを使用して簡単な表を作成し、その表のデータを表示する照会を処理します。

注: サンプル・コードをご使用の場合は、560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
//////////////////////////////////////////////////////////////////////////////////
//
// BasicJDBC example.  This program uses the native JDBC driver for the
// Developer Kit for Java to build a simple table and process a query
// that displays the data in that table.
//
// Command syntax:
//    BasicJDBC
//
//////////////////////////////////////////////////////////////////////////////////
//
// This source is an example of the native JDBC driver.
// IBM grants you a nonexclusive license to use this as an example
// from which you can generate similar function tailored to
// your own specific needs.
//
// This sample code is provided by IBM for illustrative purposes
// only. These examples have not been thoroughly tested under all
// conditions. IBM, therefore, cannot guarantee or imply
// reliability, serviceability, or function of these programs.
//
// All programs contained herein are provided to you "AS IS"
// without any warranties of any kind.  The implied warranties of
// merchantability and fitness for a particular purpose are
// expressly disclaimed.
//
// IBM Developer Kit for Java
// (C) Copyright IBM Corp. 2001
// All rights reserved.
// US Government Users Restricted Rights -
// Use, duplication, or disclosure restricted
// by GSA ADP Schedule Contract with IBM Corp.
//
//////////////////////////////////////////////////////////////////////////////////

// Include any Java classes that are to be used. In this application,
// many classes from the java.sql package are used and the
// java.util.Properties class is also used as part of obtaining
// a connection to the database.
import java.sql.*;
import java.util.Properties;

// Create a public class to encapsulate the program.
public class BasicJDBC {
```

```
// The connection is a private variable of the object.
private Connection connection = null;

// Any class that is to be an 'entry point' for running
// a program must have a main method. The main method
// is where processing begins when the program is called.
public static void main(java.lang.String[] args) {

    // Create an object of type BasicJDBC. This
    // is fundamental to object-oriented programming. Once
    // an object is created, call various methods on
    // that object to accomplish work.
    // In this case, calling the constructor for the object
    // creates a database connection that the other
    // methods use to do work against the database.
    BasicJDBC test = new BasicJDBC();

    // Call the rebuildTable method. This method ensures that
    // the table used in this program exists and looks
    // correct. The return value is a boolean for
    // whether or not rebuilding the table completed
    // successfully. If it did no, display a message
    // and exit the program.
    if (!test.rebuildTable()) {
        System.out.println("Failure occurred while setting up " +
                           " for running the test.");
        System.out.println("Test will not continue.");
        System.exit(0);
    }

    // The run query method is called next. This method
    // processes an SQL select statement against the table that
    // was created in the rebuildTable method. The output of
    // that query is output to standard out for you to view.
    test.runQuery();

    // Finally, the cleanup method is called. This method
    // ensures that the database connection that the object has
    // been hanging on to is closed.
    test.cleanup();
}


/**
This is the constructor for the basic JDBC test. It creates a database
connection that is stored in an instance variable to be used in later
method calls.
**/
public BasicJDBC() {

    // One way to create a database connection is to pass a URL
    // and a java Properties object to the DriverManager. The following
    // code constructs a Properties object that has your user ID and
    // password. These pieces of information are used for connecting
    // to the database.
    Properties properties = new Properties ();
    properties.put("user", "cujo");
    properties.put("password", "newtiger");

    // Use a try/catch block to catch all exceptions that can come out of the
    // following code.
    try {
        // The DriverManager must be aware that there is a JDBC driver available
        // to handle a user connection request. The following line causes the
        // native JDBC driver to be loaded and registered with the DriverManager.
        Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
```

```
            // Create the database Connection object that this program uses in all
            // the other method calls that are made. The following code specifies
            // that a connection is to be established to the local database and that
            // that connection should conform to the properties that were set up
            // previously (that is, it should use the user ID and password specified).
            connection = DriverManager.getConnection("jdbc:db2:*local", properties);

        } catch (Exception e) {
            // If any of the lines in the try/catch block fail, control transfers to
            // the following line of code. A robust application tries to handle the
            // problem or provide more details to you. In this program, the error
            // message from the exception is displayed and the application allows
            // the program to return.
            System.out.println("Caught exception: " + e.getMessage());
        }
    }


    /**
    Ensures that the qgpl.basicjdbc table looks you want it to at the start of
    the test.

    @returns boolean    Returns true if the table was rebuild successfully;
                        returns false if any failure occurred.
    **/
    public boolean rebuildTable() {
        // Wrap all the functionality in a try/catch block so an attempt is
        // made to handle any errors that may happen within this method.
        try {

            // Statement objects are used to process SQL statements against the
            // database. The Connection object is used to create a Statement
            // object.
            Statement s = connection.createStatement();

            try {
                // Build the test table from scratch. Process an update statement
                // that attempts to delete the table if it currently exists.
                s.executeUpdate("drop table qgpl.basicjdbc");
            } catch (SQLException e) {
                // Do not perform anything if an exception occurred. Assume
                // that the problem is that the table that was dropped does not
                // exist and that it can be created next.
            }

            // Use the statement object to create our table.
            s.executeUpdate("create table qgpl.basicjdbc(id int, name char(15))");

            // Use the statement object to populate our table with some data.
            s.executeUpdate("insert into qgpl.basicjdbc values(1, 'Frank Johnson')");
            s.executeUpdate("insert into qgpl.basicjdbc values(2, 'Neil Schwartz')");
            s.executeUpdate("insert into qgpl.basicjdbc values(3, 'Ben Rodman')");
            s.executeUpdate("insert into qgpl.basicjdbc values(4, 'Dan Gloore')");

            // Close the SQL statement to tell the database that it is no longer
            // needed.
            s.close();

            // If the entire method processed successfully, return true. At this point,
            // the table has been created or refreshed correctly.
            return true;

        } catch (SQLException sqle) {
            // If any of our SQL statements failed (other than the drop of the table
            // that was handled in the inner try/catch block), the error message is
            // displayed and false is returned to the caller, indicating that the table
            // may not be complete.
```

```
        System.out.println("Error in rebuildTable: " + sqle.getMessage());
        return false;
    }
}



/**
Runs a query against the demonstration table and the results are displayed to
standard out.
**/
public void runQuery() {
    // Wrap all the functionality in a try/catch block so an attempts is
    // made to handle any errors that might happen within this
    // method.
    try {
        // Create a Statement object.
        Statement s = connection.createStatement();

        // Use the statement object to run an SQL query. Queries return
        // ResultSet objects that are used to look at the data the query
        // provides.
        ResultSet rs = s.executeQuery("select * from qgpl.basicjdbc");

        // Display the top of our 'table' and initialize the counter for the
        // number of rows returned.
        System.out.println("--------------------");
        int i = 0;

        // The ResultSet next method is used to process the rows of a
        // ResultSet. The next method must be called once before the
        // first data is available for viewing. As long as next returns
        // true, there is another row of data that can be used.
        while (rs.next()) {

            // Obtain both columns in the table for each row and write a row to
            // our on-screen table with the data. Then, increment the count
            // of rows that have been processed.
            System.out.println("| " + rs.getInt(1) + " | " + rs.getString(2) + "|");
            i++;
        }

        // Place a border at the bottom on the table and display the number of rows
        // as output.
        System.out.println("--------------------");
        System.out.println("There were " + i + " rows returned.");
        System.out.println("Output is complete.");

    } catch (SQLException e) {
        // Display more information about any SQL exceptions that are
        // generated as output.
        System.out.println("SQLException exception: ");
        System.out.println("Message:....." + e.getMessage());
        System.out.println("SQLState:...." + e.getSQLState());
        System.out.println("Vendor Code:." + e.getErrorCode());
        e.printStackTrace();
    }
}


/**
The following method ensures that any JDBC resources that are still
allocated are freed.
**/
public void cleanup() {
    try {
        if (connection != null)
```

```
                connection.close();
            } catch (Exception e) {
                System.out.println("Caught exception: ");
                e.printStackTrace();
            }
        }
    }
```

Java 用 JNDI の例のセットアップ:

DataSource は JNDI (Java Naming and Directory Interface) と協調して動作します。 Java Database Connectivity (JDBC) がデータベースの抽象化層であるのと同じように、JNDI はディレクトリー・サービスの Java 抽象化層です。

ほとんどの場合、JNDI は LDAP (Lightweight Directory Access Protocol) と共に使用されますが、CORBA Object Services (COS)、Java Remote Method Invocation (RMI) レジストリー、またはベースのファイル・システムと共に使用されることもあります。この多様化された使用法は、一般的な JNDI 要求を、特定のディレクトリー・サービスの要求に転換する、各種のディレクトリー・サービス・プロバイダーの方式によって実現されます。

注: RMI を使用することは、複雑な作業になる可能性があるということを銘記しておいてください。 RMI をソリューションとして選ぶ前に、これを選ぶと及ぶ影響について必ず理解しておいてください。

RMI の評価を開始するのに適した場所は、Java Remote Method Invocation (RMI) です。

この DataSource のサンプルは、JNDI ファイル・システム・サービス・プロバイダーを使用するように設計されています。提供された例を実行する場合は、JNDI サービス・プロバイダーが適切な場所になければなりません。

ファイル・システム・サービス・プロバイダーを使用する環境をセットアップするには、以下の手順に従ってください。

1. Sun Microsystems の JNDI サイトから、ファイル・システム JNDI サポートをダウンロードする。
2. (FTP または他のメカニズムを使って、) fscontext.jar および providerutil.jar をシステムに転送し、/QIBM/UserData/Java400/ext に配置する。ここは拡張機能ディレクトリーで、ここに配置された JAR ファイルはアプリケーションの実行時に自動的に検索されます (クラスパスに追加する必要はありません)。

JNDI のサービス・プロバイダーがサポートされたら、アプリケーションのコンテキスト情報をセットアップする必要があります。これは、SystemDefault.properties ファイル内に必要な情報を書き込むことによって行えます。デフォルト・プロパティーを指定できる場所はシステム上にいくつかありますが、最善の方法は、ユーザーのホーム・ディレクトリー (/home/) に SystemDefault.properties という名前のテキスト・ファイルを作成することです。

ファイルを作成するには、以下の行を使用するか、既存のファイルに以下の行を追加します。

```
# Needed env settings for JNDI.
java.naming.factory.initial=com.sun.jndi.fscontext.RefFSContextFactory
java.naming.provider.url=file:/DataSources/jdbc
```

これらの行は、JNDI 要求をファイル・システム・サービス・プロバイダーが処理し、/DataSource/jdbc が JNDI を使用するタスクのルートであることを指定しています。この場所は変更することもできますが、ユ

ーザーが指定したディレクトリーは必ず存在していなければなりません。ユーザーが指定した場所に、例で使用する DataSource がバインドおよび展開されます。

Connections

Connection オブジェクトは、Java Database Connectivity (JDBC) 内のデータ・ソースへの接続を表しています。 SQL ステートメントを処理するために作成された Statement オブジェクトは、Connection オブジェクトを介してデータベースに接続します。アプリケーション・プログラムは、一度に複数の接続を持つことができます。これらの Connection オブジェクトは、すべて同じデータベースに接続することも、別々のデータベースに接続することもできます。

JDBC 内で接続を取得するには、2 つの方法があります。

- DriverManager クラスを通して取得する。
- DataSources を使用して取得する。

アプリケーションの移植性と保守容易性を高めることができるため、接続を取得するためには DataSource を使用するほうが便利です。これはまた、アプリケーションが接続およびステートメント・プーリング、および分散トランザクションを容易に使用することができるようにします。

関連概念

さまざまなタイプの Statement オブジェクトを作成し、データベースを操作する。
Statement オブジェクトは、静的 SQL ステートメントの処理と、それによって生成される結果の取得に使用されます。 一度にオープンできるのは、各 Statement オブジェクトにつき 1 つの ResultSet だけです。 SQL ステートメントを処理するすべてのステートメント・メソッドは、すでにオープンされている ResultSet があると、暗黙的にステートメントの現行の ResultSet をクローズします。

データベースに対するトランザクションを制御する。
トランザクションは作業の論理単位です。 作業の論理単位を完了するには、データベースに対していくつかのアクションを実行しなければならない場合があります。

データベースに関するメタデータを取得する。
DatabaseMetaData インターフェースは、IBM Developer Kit for Java JDBC ドライバーによってインプリメントされ、基礎となるデータ・ソースに関する情報を提供します。これは、提供されているデータ・ソースとの対話方法を決定するため、主にアプリケーション・サーバーとツールによって使用されます。 アプリケーションは、DatabaseMetaData メソッドを使用してもデータ・ソースの情報を入手することができますが、こちらはそれほど一般的ではありません。

Java DriverManager クラス:

DriverManager は、Java2 Plaform, Standard Edition (J2SE) および Java SE Development Kit (JDK) 内の静的クラスです。DriverManager は、アプリケーションで使用可能な Java Database Connectivity (JDBC) ドライバーのセットを管理します。

アプリケーションは、必要があれば、複数の JDBC ドライバーを同時に利用することができます。各アプリケーションは Uniform Resource Locator (URL) を使用して、JDBC ドライバーを指定します。特定の JDBC ドライバーの URL を DriverManager に渡すことにより、アプリケーションは DriverManager に対して、どの種類の JDBC 接続をアプリケーションに戻すべきかを通知します。

これが完了するまでは、DriverManager が接続を提供することのできる、利用可能な JDBC ドライバーを認知していなければなりません。 Class.forName メソッドへの呼び出しを行うことによって、メソッドの中に渡されたストリング名に基づいて実行中の Java 仮想マシン (JVM) にクラスがロードされます。ネイティブ JDBC ドライバーをロードするために class.forName メソッドを使用する例を以下に示します。

例: ネイティブ JDBC ドライバーをロードする

```
// Load the native JDBC driver into the DriverManager to make it
// available for getConnection requests.

Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
```

JDBC ドライバーは、ドライバーのインプリメンテーション・クラスがロードされると、自動的に自分自身について DriverManager に通知するように設計されています。前述のコードの行が一度処理されると、そのネイティブ JDBC ドライバーは DriverManager と共に動作できるようになります。以下のコードは、ネイティブ JDBC URL を使って、Connection オブジェクトを要求しています。

例: Connection オブジェクトを要求する

```
// Get a connection that uses the native JDBC driver.

Connection c = DriverManager.getConnection("jdbc:db2:*local");
```

最も単純な JDBC URL の形式は、コロンで区切られた 3 つの値のリストです。リストの最初の値はプロトコルを示しており、JDBC URL では常に `jdbc` になります。 2 番目の値はサブプロトコルで、ネイティブ JDBC ドライバーを指定するために `db2` または `db2iSeries` を使用しています。 3 番目の値は、指定したシステムへの接続を確立するためのシステム名です。ローカル・データベースに接続するために、2 つの特殊値があります。それは、*LOCAL と localhostです (どちらも大文字小文字を区別しません)。特定のシステム名は次のようにも指定できます。

```
Connection c =
  DriverManager.getConnection("jdbc:db2:rchasmop");
```

これは、rchasmop システムへの接続を作成します。システムがリモート・システムへの接続を試行する場合 (たとえば、分散リレーショナル・データベース体系) は、リレーショナル・データベース・ディレクトリーにあるシステム名を使用する必要があります。

注: 指定されない場合、サインインに使用されているユーザー ID とパスワードが、データベースへの接続の確立にも使用されます。

注: IBM DB2 JDBC Universal ドライバーも db2 サブプロトコルを使用します。ネイティブ JDBC ドライバーが URL を処理するようにするためには、アプリケーションで jdbc:db2:xxxx の URL ではなく jdbc:db2iSeries:xxxx の URL を使用する必要があります。アプリケーションで、db2 サブプロトコルを含む URL をネイティブ・ドライバーが受け入れないようにしたい場合は、そのアプリケーションで com.ibm.db2.jdbc.app.DB2Driver ではなく、com.ibm.db2.jdbc.app.DB2iSeriesDriver クラスをロードする必要があります。このクラスをロードすると、ネイティブ・ドライバーは db2 サブプロトコルを含む URL を処理しなくなります。

プロパティー

DriverManager.getConnection メソッドは DriverManager 上で Connection オブジェクトを取得する唯一のメソッドで、前述の単一のストリング URL を取ります。 DriverManager.getConnection メソッドの別のバージョンでは、ユーザー ID とパスワードを取ります。このバージョンの例は次のとおりです。

例: ユーザー ID とパスワードを取る DriverManager.getConnection メソッド

```
// Get a connection that uses the native JDBC driver.

Connection c = DriverManager.getConnection("jdbc:db2:*local", "cujo", "newtiger");
```

このコードは、誰がこのアプリケーションを実行しているかにかかわりなく、ユーザー cujo、パスワード newtiger としてローカル・データベースに接続することを想定しています。 DriverManager.getConnection メソッドの別のバージョンでは、さらにカスタマイズを行うため、java.util.Properties オブジェクトを取ります。以下に、この例を示します。

例: java.util.Properties オブジェクトを取る DriverManager.getConnection メソッド

```
// Get a connection that uses the native JDBC driver.

Properties prop = new java.util.Properties();
prop.put("user", "cujo");
prop.put("password","newtiger");
Connection c = DriverManager.getConnection("jdbc:db2:*local", prop);
```

このコードは、前述のバージョンと機能的には同等ですが、ユーザー ID とパスワードをパラメーターとして渡しています。

ネイティブ JDBC ドライバーの接続プロパティーの完全なリストは、接続プロパティーを参照してください。

URL プロパティー

プロパティーを指定する別の方法は、それらのプロパティーを URL オブジェクトのリストに格納することです。リスト内のそれぞれのプロパティーはセミコロンで区切られ、リストはプロパティー名 = プロパティー値という形式になっている必要があります。これは単なるショートカットであり、処理される方法には違いはありません。次の例のように記述されます。

例: URL プロパティーを指定する

```
// Get a connection that uses the native JDBC driver.

Connection c = DriverManager.getConnection("jdbc:db2:*local;user=cujo;password=newtiger");
```

このコードも、前述の例と機能的には同等です。

プロパティー値が properties オブジェクトと URL オブジェクトの両方で指定された場合は、URL バージョンの指定が properties オブジェクトよりも優先されます。以下に、この例を示します。

例: URL プロパティー

```
// Get a connection that uses the native JDBC driver.
Properties prop = new java.util.Properties();
prop.put("user", "someone");
prop.put("password","something");
Connection c = DriverManager.getConnection("jdbc:db2:*local;user=cujo;password=newtiger",
prop);
```

この例では、Properties オブジェクトで指定されたユーザー ID とパスワードではなく、URL ストリングで指定されたユーザー ID とパスワードが使用されます。結果として、前述のコードと機能的に完全に同等になります。

例: 無効なユーザー ID とパスワード:

以下は、SQL 命名モードでの Connection プロパティーの使用法の例です。

注: サンプル・コードをご使用の場合は、560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
//////////////////////////////////////////////////////////////////////////////////
//
// InvalidConnect example.
//
// This program uses the Connection property in SQL naming mode.
//
//////////////////////////////////////////////////////////////////////////////////
//
// This source is an example of the IBM Developer for Java JDBC driver.
// IBM grants you a nonexclusive license to use this as an example
// from which you can generate similar function tailored to
// your own specific needs.
//
// This sample code is provided by IBM for illustrative purposes
// only. These examples have not been thoroughly tested under all
// conditions. IBM, therefore, cannot guarantee or imply
// reliability, serviceability, or function of these programs.
//
// All programs contained herein are provided to you "AS IS"
// without any warranties of any kind.  The implied warranties of
// merchantability and fitness for a particular purpose are
// expressly disclaimed.
//
// IBM Developer Kit for Java
// (C) Copyright IBM Corp. 2001
// All rights reserved.
// US Government Users Restricted Rights -
// Use, duplication, or disclosure restricted
// by GSA ADP Schedule Contract with IBM Corp.
//
//////////////////////////////////////////////////////////////////////////////////
import java.sql.*;
import java.util.*;

public class InvalidConnect {

    public static void main(java.lang.String[] args)
    {
        // Register the driver.
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
        } catch (ClassNotFoundException cnf) {
            System.out.println("ERROR: JDBC driver did not load.");
            System.exit(0);
        }

        // Attempt to obtain a connection without specifying any user or
        // password. The attempt works and the connection uses the
        // same user profile under which the job is running.
        try {
            Connection c1 = DriverManager.getConnection("jdbc:db2:*local");
            c1.close();
        } catch (SQLException e) {
            System.out.println("This test should not get into this exception path.");
            e.printStackTrace();
            System.exit(1);
        }

        try {
            Connection c2 = DriverManager.getConnection("jdbc:db2:*local",
                                                        "notvalid", "notvalid");
        } catch (SQLException e) {
            System.out.println("This is an expected error.");
            System.out.println("Message is " + e.getMessage());
            System.out.println("SQLSTATE is " + e.getSQLState());
```

```
        }
    }
}
```

JDBC ドライバーの接続プロパティー:

以下の表は、JDBC ドライバーの接続プロパティーとその値、およびその説明を示しています。

プロパティー	値	意味
access	all、read call、read only	この値を使用すると、特定の接続で実行できる操作のタイプを制限できます。デフォルト値は all であり、これは基本的に、その接続が JDBC API への完全アクセスを持つことを意味します。 read call 値は、照会の実行とストアード・プロシージャーの呼び出しだけをその接続に許可します。SQL ステートメントを介してデータベースを更新しようとしても、その処理は停止します。 read only 値を使用すれば、接続を照会だけに制限できます。ストアード・プロシージャー呼び出しと更新ステートメントは停止します。
auto commit	true、false	この値は接続の自動コミット設定の設定に使用されます。トランザクション分離プロパティーが none 以外の値に設定されていなければ、デフォルト値は true です。設定されている場合は、デフォルト値は false になります。
batch style	2.0、2.1	JDBC 2.1 仕様では、バッチ処理での更新で例外が発生した場合の処理として、2 番目のメソッドが定義されています。ドライバーは、どちらかの仕様に準拠できます。デフォルトでは、JDBC 2.0 仕様で定義された方法で処理されます。
block size	0、8、16、32、64、128、256、512	これは、ResultSet として一度に取り出される行数です。順方向に限られた通常の ResultSet の処理では、このサイズのブロックが取得されます。その後、ユーザーのアプリケーションが各行を処理するので、データベースはアクセスされません。ブロックの終わりに達したときに初めて、データベースは別のデータ・ブロックを要求します。 この値は、blocking enabled プロパティーが true に設定されているときにのみ使用されます。 block size プロパティーを 0 に設定すると、blocking enabled プロパティーを "false" に設定した場合と同じ効果が得られます。 デフォルトでは、32 のブロック・サイズでブロック化が行われます。これは、非常に独断的な決定であり、デフォルトは将来に変更される可能性があります。スクロール可能な ResultSet でブロック化は行われません。
blocking enabled	true、false	この値は、接続で ResultSet 行を検索するときにブロック化を行うかどうかを決定します。ブロック化により、ResultSet の処理のパフォーマンスは大きく改善される可能性があります。 このプロパティーは、デフォルトでは true に設定されています。

プロパティー	値	意味
commit hold	false、true	この値は、connection.commit() メソッドを呼び出す際に "commit hold" を使用するかどうかを指定します。"commit hold" を使用した場合は、コミットが呼び出されたときにカーソルや他のデータベース・リソースがクローズされたり解放されたりしません。 デフォルト値は false です。
concurrent access resolution	1、2、3	このプロパティーは、「現在コミット済み」アクセスを接続上で使用するかどうかを指定します。以下のいずれかの値を指定できます。 **1** 「現在コミット済み」が使用される **2** 「結果待ち」が使用される **3** 「ロックのスキップ」が使用される デフォルト値は 2 です。
cursor hold	true、false	この値は、トランザクションがコミットされた後も ResultSet をオープンにしたままにするかどうかを設定します。この値を true にすると、コミットが呼び出された後でも、アプリケーションはオープンしている ResultSet にアクセスすることができます。この値を false にすると、その接続の中でオープンしているカーソルはコミット時にすべてクローズされます。 このプロパティーは、デフォルトでは true に設定されています。 この値プロパティーは、その接続で作成されたすべての ResultSet のデフォルト値となります。 JDBC 3.0 では、カーソルの保持機能がサポートされました。このデフォルトは、アプリケーションによって別の保持機能が指定されると、置き換えられます。以前のバージョンから JDBC 3.0 にマイグレーションする場合、カーソルの保持機能のサポートが JDBC 3.0 までは追加されていなかったことに留意してください。以前のバージョンでは、デフォルト値「true」が接続時に送信されていたものの、JVM では認識されていませんでした。したがって、カーソル保持プロパティーが JDBC 3.0 までのデータベース機能に影響を与えることはありません。
cursor sensitivity	asensitive、sensitive	ResultSet.TYPE_SCROLL_SENSITIVE カーソルで使用するカーソル・センシティビティーを指定します。デフォルトでは、ネイティブ JDBC ドライバーが ResultSet.TYPE_SCROLL_SENSITIVE カーソルにアセンシティブ・カーソルを作成します。
data truncation	true、false	この値は、文字データが切り捨てられた場合に警告および例外を生成するか (true)、黙って切り捨てるか (false) を設定します。デフォルトは true で、文字フィールドのデータ切り捨てが有効です。
date format	julian、mdy、dmy、ymd、usa、iso、eur、jis	このプロパティーは、日付のフォーマット方法を変更することができます。

プロパティー	値	意味
date separator	/(スラッシュ)、-(ダッシュ)、.(ピリオド)、,(コンマ)、b	このプロパティーを使って、日付区切り文字を変更することができます。このプロパティーは、(システム規則に従って) 一部の dateFormat 値との組み合わせでのみ有効です。
decfloat rounding mode	round half even、round half up、round down、round ceiling、round floor、round half down、round up、round half even	このプロパティーは、10 進浮動小数点演算で使用する丸めモードを指定します。デフォルト値は「round half even」です。
decimal separator	.(ピリオド)、,(コンマ)	このプロパティーを使って、小数点を変更できます。
direct map	true、false	このプロパティーは、データベースから ResultSet を取得する際にデータベース・ダイレクト・マップの最適化を使用するかどうかを指定します。デフォルト値は true です。
do escape processing	true、false	このプロパティーは、接続でステートメントがエスケープ処理を実行するかどうかを示すフラグを設定します。エスケープ処理の使用は、SQL ステートメントをすべてのプラットフォームで汎用および類似のものとなるようにコード化する 1 つの方法です。データベースはエスケープ文節を読み取ると、ユーザーのシステム固有のバージョンに置き換えます。 これは、システムに余分な作業を強制する点を除けば、優れた機能です。既に有効な IBM i SQL 構文が含まれている SQL ステートメントだけを使用することが分かっているなら、パフォーマンスの向上のために、この値を false に設定することを勧めします。 JDBC 仕様に準拠するため、このプロパティーのデフォルト値は true になっています (エスケープ処理はデフォルトでアクティブになっています)。この値は、JDBC 仕様の欠点を補うために追加されています。 Statement クラスでは、エスケープ処理は off に設定することしかできません。単純なステートメントを処理する場合は、これでもうまくいきます。ステートメントを作成し、エスケープ処理を off にしてから、ステートメントの処理を開始することができます。ただし、PreparedStatement や呼び出し可能ステートメントの場合、この方式ではうまくいきません。 PreparedStatement や呼び出し可能ステートメントの構成時に指定した SQL ステートメントは、後から変更されることはありません。そのため、ステートメントを準備し、その後からエスケープ処理を変更しても、効果はありません。この接続プロパティーを設定することは、余分なオーバーヘッドを回避するための 1 つの手段となります。
errors	basic、full	このプロパティーは、全システムの 2 次レベル・エラー・テキストを SQLException オブジェクト・メッセージに戻すかどうかを指定します。デフォルトは basic になっており、標準メッセージ・テキストのみを戻します。

プロパティー	値	意味
extended metadata	true、false	このプロパティーは、ドライバーにデータベースから拡張メタデータを要求させるかどうかを指定します。このプロパティーを true に設定すると、以下の ResultSetMetaData メソッドから戻される情報の精度が上がります。 • getColumnLabel(int) • getSchemaName(int) • getTableName(int) • isReadOnly(int) • isSearchable(int) • isWriteable(int) このプロパティーを true に設定すると、データベースから取得しなければならない情報の量が増えるため、パフォーマンスが低下することがあります。
ignore warnings	無視する SQL 状態に関するコンマ区切りリスト	デフォルトでは、ネイティブ JDBC ドライバーは、データベースから戻される警告ごとに java.sql.SQLWarning オブジェクトを内部的に作成します。このプロパティーは、ネイティブ JDBC ドライバーに警告オブジェクトを作成させない SQL 状態のリストを指定します。例えば、SQLSTATE 0100C の警告はストアード・プロシージャーから ResultSet が戻されるたびに作成されます。この警告は、ストアード・プロシージャーを呼び出すアプリケーションのパフォーマンスを向上させるために無視する設定にしても問題はありません。
libraries	スペースで区切られたライブラリーのリスト。(ライブラリーのリストは、コロンまたはコンマで区切ることもできます。)	このプロパティーを使うと、ライブラリーのリストをサーバー・ジョブのライブラリー・リスト、または指定されたデフォルト・ライブラリーに設定することができます。 このプロパティーがどのように動作するかは、naming プロパティーによって影響されます。デフォルトでは、naming は "sql" に設定されており、JDBC は ODBC と同様の動作をします。ライブラリー・リストは、どのように接続処理が行われるかには影響しません。これは、すべての非修飾テーブルのデフォルト・ライブラリーになります。デフォルトでは、接続しているユーザー・プロファイルと同じ名前を持つライブラリーです。 libraries プロパティーが指定されると、リストで最初に指定されているライブラリーがデフォルト・ライブラリーとなります。デフォルト・ライブラリーが接続 URL (jdbc:db2:*local/mylibrary 内) で指定されている場合は、このプロパティーの設定よりも優先されます。naming が system に設定されている場合は、このプロパティーで指定したそれぞれのライブラリーがユーザーのライブラリー・リストに追加され、非修飾テーブル参照を解決するために検索されます。

プロパティー	値	意味
lob threshold	500000 未満の任意の値	このプロパティーは、ドライバーに対して、lob 列がしきい値よりも小さかった場合、lob 列のロケーターの代わりに ResultSet 記憶域に実際の値を格納することを指定します。このプロパティーは、列のサイズに対してのもので、それ自身の lob データ・サイズに対するものではありません。たとえば、各 lob 列のサイズが最大 1 MB として定義されていても、各列の値が 500 KB 以下であれば、引き続きロケーターが使用されます。 このサイズ制限は、常にデータ・ブロックが 16 MB の最大割り振りサイズよりも大きくなる危険がなく、ブロックをフェッチできるように指定されることに注意してください。大きな ResultSet では簡単にこの制限を超えてしまい、その結果、フェッチが失敗します。データ・ブロックと相互に作用するため、ブロック・サイズ・プロパティーとこのプロパティーの設定は慎重に行ってください。デフォルトでは 0 に設定され、lob データには常にロケーターが使用されます。
maximum precision	31、63	この値は、10 進数および数値タイプに使用する最大精度を指定します。デフォルト値は 31 です。
maximum scale	0 から 63	この値は、10 進数および数値タイプで使用する (戻す) 最大位取り (小数点の右側にくる小数点以下の桁数) を指定します。値の範囲は 0 から最大精度までです。デフォルト値は 31 です。
minimum divide scale	0 から 9	この値は、中間と結果の両方のデータ・タイプで戻される最小分割位取り (小数点の右側の小数点以下の桁数) を指定します。この値は、最大位取りを超えない 0 から 9 の範囲の値です。 0 が指定されている場合は、最小分割位取りは使用されません。デフォルト値は 0 です。
naming	sql、system	このプロパティーを使用すると、従来の IBM i 命名構文か、標準の SQL 命名構文のいずれかを使用できます。system 命名は、/ (スラッシュ) 文字を使用してコレクション値とテーブル値を区切ることを意味し、SQL 命名は、. (ピリオド) 文字を使用してそれらの値を区切ることを意味します。 この値には、デフォルトのライブラリーによってさまざまな設定があります。詳しくは、上記の libraries プロパティーを参照してください。デフォルトでは SQL 命名が使用されます。
password	任意の値	このプロパティーでは、接続用のパスワードを指定することができます。このプロパティーは、user プロパティーが指定されていないと、正しく動作しません。これらのプロパティーを使うと、IBM i ジョブを実行しているユーザーではないユーザーとして、データベースに接続することができるようになります。 user および password プロパティーを指定すると、シグニチャー `getConnection(String url, String userId, String password)` の接続メソッドを使用した場合と同様の効果が得られます。

プロパティー	値	意味
prefetch	true、false	このプロパティーは、ドライバーが処理の直後に ResultSet の最初のデータをフェッチするのか、それともデータが要求されてからフェッチするのかを指定します。デフォルトは true で、データはプリフェッチされます。 ネイティブ JDBC ドライバーを使用したアプリケーションの場合、まれにこのことが問題になることがあります。このプロパティーは本来、Java ストアード・プロシージャーおよびユーザー定義関数での内部使用のために存在しています。それらにおいては、要求されるまではデータベース・エンジンがユーザーのために ResultSet からデータをフェッチしないことが重要となります。
qaqqinilib	ライブラリー名	このプロパティーは、使用する qaqqini ファイルが含まれているライブラリーを指定します。qaqqini ファイルには、DB2 for i データベース・エンジンのパフォーマンスに影響を与える可能性のあるすべての属性が含まれています。
query optimize goal	1、2	このプロパティーは、サーバーが照会を最適化する際に何を目的とするかを指定します。この設定は、サーバーの OPTIMIZATION_GOAL という QAQQINI オプションと対応しています。以下のいずれかの値を指定できます。 **1** データの最初のブロックに対する照会を最適化する (*FIRSTIO) **2** ResultSet 全体の照会を最適化する (*ALLIO) デフォルト値は 2 です。
reuse objects	true、false	このプロパティーは、一度クローズされた、ある種類のオブジェクトをドライバーが再利用するかどうかを指定します。これはパフォーマンスを向上させます。デフォルトは true です。

プロパティー	値	意味

servermode subsystem	*SAME、subsystem name	このプロパティーは、関連する QSQSRVR ジョブが実行されるサブシステムを指定します。デフォルトの動作では、ジョブを QSYSWRK サブシステムで実行させます。 *SAME の値が使用された場合は、QSQSRVR ジョブは、ネイティブ JDBC ドライバーを使用するジョブと同じサブシステムで実行されます。
		
		QSQSRVR ジョブを別のサブシステムで実行するには、そのサブシステムの QSQSRVR 事前開始ジョブ・エントリーが存在する必要があります。以下のコマンドを使用すれば、QSQSRVR 事前開始ジョブ・エントリーを作成できます。
		
		`ENDSBS` *`sbs`*
		
		`ADDPJE SBSD(`*`library/sbsd`*`)`
		`  PGM(QSYS/QSQSRVR) STRJOBS(*YES) INLJOBS(`*`x`*`)`
		`  THRESHOLD(`*`y`*`) ADLJOBS(`*`z`*`) MAXUSE(*NOMAX)`
		
		`STRSBS` *`sbs`*
		
		ここで *sbs* はサブシステム、*library* はサブシステム記述 *sbsd* が配置されたライブラリー、*x*、*y*、および *z* は、事前開始ジョブ項目追加 (ADDPJE) コマンド上の対応するパラメーターの数値です。QSQSRVR の事前開始ジョブ・エントリーがサブシステム内に存在しない場合は、QSQSRVR ジョブは、事前開始ジョブ (PJ) ではなく、バッチ即時ジョブ (BCI) を使用します。このバッチ即時ジョブは、通常、ネイティブ JDBC ドライバーを使用するジョブと同じサブシステムで実行されます。

time format	hms、usa、iso、eur、jis	このプロパティーを使って、時刻のフォーマット方法を変更することができます。
time separator	:(コロン)、.(ピリオド)、,(コンマ)、b	このプロパティーを使って、時刻区切り文字を変更することができます。このプロパティーは、(システム規則に従って) 一部の timeFormat 値との組み合わせでのみ有効です。
trace	true、false	このプロパティーによって、接続のトレースをオンにすることができます。これは、単純なデバッグ支援機能として使用することができます。 デフォルト値は false で、トレースを使用しません。
transaction isolation	none、read committed、read uncommitted、repeatable read、serializable	このプロパティーを使って、接続のトランザクション分離レベルを設定することができます。このプロパティーで特定のレベルを設定することは、Connection インターフェースの setTransactionIsolation メソッドでレベルを指定するのと同じことです。 JDBC のデフォルトは自動コミット・モードなので、このプロパティーのデフォルト値は none です。

プロパティー	値	意味
translate binary	true、false	このプロパティーを使用すれば、JDBC ドライバーで binary データ値や varbinary データ値を強制的に char データ値や varchar データ値と同様に扱うことができます。バイナリー・データを文字データと同様に扱う場合は、ジョブの CCSID がデータの CCSID として使用されます。 このプロパティーのデフォルトは false で、バイナリー・データは文字データと同様には扱われません。
translate hex	binary、character	この値を使って、SQL 式の 16 進定数が使用するデータ・タイプを選択することができます。 binary が設定されている場合、それは 16 進数が BINARY データ・タイプを使用することを意味します。 character が設定されている場合、それは 16 進数が CHARACTER FOR BIT DATA データ・タイプを使用することを意味します。デフォルト設定は character です。
use block insert	true、false	このプロパティーは、データベースにデータのブロックを挿入するために、ネイティブ JDBC ドライバーがブロック挿入モードになることを可能にします。これはバッチ更新の最適化バージョンです。この最適化モードは、あるシステム制約への違反、またデータ挿入の障害およびデータの破壊が発生する恐れのないアプリケーション内でのみ使用できます。 このプロパティーをオンにしたアプリケーションは、バッチ更新を行おうとするときにはローカル・システムにのみ接続します。ブロック挿入は DRDA 上では管理できないため、DRDA を使ってリモート接続を確立します。 また、アプリケーションでは SQL 挿入ステートメントで PreparedStatements が使われており、すべての挿入値パラメーターが VALUES 文節によって指定されている必要があります。値リストで定数を使うことは許可されていません。これは、システムのブロック挿入エンジンの要件です。デフォルトは false です。
user	任意の値	このプロパティーでは、接続用のユーザー ID を指定することができます。このプロパティーは、password プロパティーも指定されていないと機能しません。これらのプロパティーを使うと、IBM i ジョブを実行しているユーザーではないユーザーとして、データベースに接続することができるようになります。 user および password プロパティーを指定すると、シグニチャー getConnection(String url, String userId, String password) の接続メソッドを使用した場合と同様の効果が得られます。

DataSource を UDBDataSource と共に使用する:

DataSource インターフェースは、Java Database Connectivity (JDBC) ドライバーを使用すると柔軟性が向上します。

DataSource の使用法は、以下の 2 つの段階に分けることができます。

- **配置**

  配置 (デプロイメント) とは、JDBC アプリケーションを実際に実行する前に行うセットアップの段階のことです。一般的には、配置は特定のプロパティーで DataSource をセットアップすること、その後 Java Naming and Directory Interface (JNDI) を使用してそれをディレクトリー・サービスにバインドすることを意味します。最も一般的な Lightweight Directory Access Protocol (LDAP) はディレクトリー・サービスですが、他にも Common Object Request Broker Architecture (CORBA) オブジェクト・サービス、Java リモート・メソッド呼び出し (RMI)、または基礎となるファイル・システムなど多数あります。

- **使用**

  DataSource の実行時使用と配置を分離することにより、多数のアプリケーションで DataSource のセットアップを再使用できます。配置の一部の局面を変更すると、DataSource を使用するすべてのアプリケーションに、自動的に変更内容が反映されます。

注: RMI を使用することは、複雑な作業になる可能性があるということを銘記しておいてください。 RMI をソリューションとして選ぶ前に、それにより多方面に波及する影響について必ず理解しておいてください。

DataSource の利点には、アプリケーション開発プロセスに直接影響を与えずに、アプリケーションに代わって JDBC ドライバーを稼働できることがあります。詳しくは、以下を参照してください。

- 126 ページの『オブジェクト・プーリングのための DataSource サポートの使用』
- 130 ページの『DataSource ベースのステートメント・プーリング』
- 75 ページの『JDBC 分散トランザクション』

UDBDataSourceBind

UDBDataSource を作成して JNDI とのバインドを取得する例として、 51 ページの『例: UDBDataSource を作成して JNDI でバインドする』 プログラムがあります。このプログラムにより、要求されている基本タスクがすべて実行されます。つまり、UDBDataSource オブジェクトがインスタンス化され、このオブジェクトに関するプロパティーが設定され、JNDI コンテキストが取り出され、このオブジェクトが JNDI コンテキスト中の名前とバインドされます。

配置時のコードは、ベンダーごとに固有です。このアプリケーションは、処理したい特定の DataSource の実装をインポートしなければなりません。インポート・リスト中で、パッケージ修飾 UDBDataSource クラスがインポートされます。このアプリケーションの最も知られていない点として、JNDI との協働があります (Context オブジェクトを取り出してバインド呼び出しするなど)。追加情報については、Sun Microsystems, Inc. の JNDI を参照してください。

このプログラムを実行して正常に完了すると、JNDI ディレクトリー・サービス中に SimpleDS という新しい項目が入れられます。この項目は、JNDI コンテキストによって指定されている場所に入れられます。これで、DataSource のインプリメンテーションが配置されたことになります。アプリケーション・プログラムは、この DataSource を使用して、データベース接続と JDBC 関連作業を取り出します。

UDBDataSourceUse

前述の配置したアプリケーションを使用する JDBC アプリケーションの例として、 53 ページの『例: UDBDataSource をバインドする前に初期コンテキストを取得する』 プログラムがあります。

前述の例で、JDBC アプリケーションは、UDBDataSource をバインドする前の状態の初期コンテキストを入手します。その後、そのコンテキスト上で lookup メソッドを使用して、アプリケーションが使用する DataSource タイプのオブジェクトを戻します。

注: ランタイム・アプリケーションは、DataSource インターフェースのメソッドだけを作業対象とするので、インプリメンテーション・クラスを認識する必要はありません。このようにして、アプリケーションは移植可能になります。

UDBDataSourceUse は、ある企業で大規模な運用を実行する複雑なアプリケーションとします。その企業には、十指に余る大規模アプリケーションがあります。ネットワークにあるシステムの 1 つの名前を変更する必要があるとします。配置ツールを実行して、1 つの UDBDataSource プロパティーに変更を加えれば、すべてのアプリケーションのコードに変更を加えなくても、それらのアプリケーションにこの新しい名前を認識させることができます。 DataSources の利点の 1 つに、システム・セットアップ情報を統合できることがあります。別の大きな利点は、接続プーリング、ステートメント・プーリング、および分散トランザクション・サポートなどの、アプリケーションに認識されない機能をドライバーが実装できることです。

UDBDataSourceBind と UDBDataSourceUse について綿密に分析したので、DataSource オブジェクトが実行内容を認識する方法を知りたいと思われることでしょう。どちらのプログラムにも、システム、ユーザー ID、またはパスワードを指定するコードはありません。 UDBDataSource クラスのすべてのプロパティーにデフォルト値があります。デフォルトでは、実行中のアプリケーションのユーザー・プロファイルとパスワードを使用して、ローカル IBM i サーバーに接続します。その代わりにユーザー・プロファイル cujo を使用して確実に接続したい場合は、次の 2 つの方法で行うことができます。

- ユーザー ID とパスワードを DataSource プロパティーとして設定する。この手法の使用例については、 52 ページの『例: UDBDataSourceBind を作成して DataSource プロパティーを設定する』を参照してください。
- DataSource getConnection メソッドを使用して、実行時にユーザー ID とパスワードを取得する。この手法の使用例については、 53 ページの『例: UDBDataSource の作成およびユーザー ID とパスワードの取得』 53 ページの『例: UDBDataSource の作成およびユーザー ID とパスワードの取得』を参照してください。

UDBDataSource のプロパティーを多数指定できるのと同様に、DriverManager を使用して作成する接続のプロパティーも指定できます。ネイティブ JDBC ドライバーの、サポートされているプロパティーのリストについては、 54 ページの『DataSource プロパティー』を参照してください。

これらのリストは類似していますが、今後のリリースでも同様かどうかは保証しません。 DataSource インターフェースのコーディングを始めることをお勧めします。

注: ネイティブ JDBC ドライバーには、他にも DB2DataSource と DB2StdDataSource という 2 つの DataSource インプリメンテーションがあります。これらのインプリメンテーションは使用されなくなりました。これらのインプリメンテーションの直接使用は推奨されていません。これらのインプリメンテーションは将来のリリースで除去される可能性があります。

例: UDBDataSource を作成して JNDI でバインドする:

次に、UDBDataSource を作成し、それを JNDI でバインドする方法の例を示します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
// Import the required packages. At deployment time,
// the JDBC driver-specific class that implements
// DataSource must be imported.
```

```
import java.sql.*;
import javax.naming.*;
import com.ibm.db2.jdbc.app.UDBDataSource;

public class UDBDataSourceBind
{
    public static void main(java.lang.String[] args)
    throws Exception
    {
        // Create a new UDBDataSource object and give it
        // a description.
        UDBDataSource ds = new UDBDataSource();
        ds.setDescription("A simple UDBDataSource");

        // Retrieve a JNDI context. The context serves
        // as the root for where objects are bound or
        // found in JNDI.
        Context ctx = new InitialContext();

        // Bind the newly created UDBDataSource object
        // to the JNDI directory service, giving it a name
        // that can be used to look up this object again
        // at a later time.
        ctx.rebind("SimpleDS", ds);
    }
}
```

例: UDBDataSourceBind を作成して DataSource プロパティーを設定する:

次に、UDBDataSource を作成し、DataSource のプロパティーとしてユーザー ID とパスワードを設定する方法の例を示します。

注: サンプル・コードをご使用の場合は、560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
// Import the required packages. At deployment time,
// the JDBC driver-specific class that implements
// DataSource must be imported.
import java.sql.*;
import javax.naming.*;
import com.ibm.db2.jdbc.app.UDBDataSource;

public class UDBDataSourceBind2
{
    public static void main(java.lang.String[] args)
    throws Exception
    {
        // Create a new UDBDataSource object and give it
        // a description.
        UDBDataSource ds = new UDBDataSource();
        ds.setDescription("A simple UDBDataSource "  +
                          "with cujo as the default " +
                          "profile to connect with.");

        // Provide a user ID and password to be used for
        // connection requests.
        ds.setUser("cujo");
        ds.setPassword("newtiger");

        // Retrieve a JNDI context. The context serves
        // as the root for where objects are bound or
        // found in JNDI.
        Context ctx = new InitialContext();

        // Bind the newly created UDBDataSource object
        // to the JNDI directory service, giving it a name
```

```
        // that can be used to look up this object again
        // at a later time.
        ctx.rebind("SimpleDS2", ds);
    }
}
```

例: UDBDataSource をバインドする前に初期コンテキストを取得する:

次の例では、UDBDataSource をバインドするにあたって、その前に初期コンテキストを取得します。その後、そのコンテキスト上で lookup メソッドを使用して、アプリケーションが使用する DataSource タイプのオブジェクトを戻します。

注: サンプル・コードをご使用の場合は、560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
// Import the required packages. There is no
// driver-specific code needed in runtime
// applications.
import java.sql.*;
import javax.sql.*;
import javax.naming.*;

public class UDBDataSourceUse
{
    public static void main(java.lang.String[] args)
    throws Exception
    {
        // Retrieve a JNDI context. The context serves
        // as the root for where objects are bound or
        // found in JNDI.
        Context ctx = new InitialContext();

        // Retrieve the bound UDBDataSource object using the
        // name with which it was previously bound. At runtime,
        // only the DataSource interface is used, so there
        // is no need to convert the object to the UDBDataSource
        // implementation class. (There is no need to know what
        // the implementation class is. The logical JNDI name is
        // only required).
        DataSource ds = (DataSource) ctx.lookup("SimpleDS");


        // Once the DataSource is obtained, it can be used to establish
        // a connection. This Connection object is the same type
        // of object that is returned if the DriverManager approach
        // to establishing connection is used. Thus, so everything from
        // this point forward is exactly like any other JDBC
        // application.
        Connection connection = ds.getConnection();

        // The connection can be used to create Statement objects and
        // update the database or process queries as follows.
        Statement statement = connection.createStatement();
        ResultSet rs = statement.executeQuery("select * from qsys2.sysprocs");
        while (rs.next()) {
            System.out.println(rs.getString(1) + "." + rs.getString(2));
        }

        // The connection is closed before the application ends.
        connection.close();
    }
}
```

例: UDBDataSource の作成およびユーザー ID とパスワードの取得:

以下は、UDBDataSource を作成し、実行時に getConnection メソッドを使用してユーザー ID とパスワードを取得する方法の例です。

注: サンプル・コードをご使用の場合は、560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
/// Import the required packages. There is
// no driver-specific code needed in runtime
// applications.
import java.sql.*;
import javax.sql.*;
import javax.naming.*;

public class UDBDataSourceUse2
{
    public static void main(java.lang.String[] args)
    throws Exception
    {
        // Retrieve a JNDI context. The context serves
        // as the root for where objects are bound or
        // found in JNDI.
        Context ctx = new InitialContext();

        // Retrieve the bound UDBDataSource object using the
        // name with which it was previously bound. At runtime,
        // only the DataSource interface is used, so there
        // is no need to convert the object to the UDBDataSource
        // implementation class. (There is no need to know
        // what the implementation class is. The logical JNDI name
        // is only required).
        DataSource ds = (DataSource) ctx.lookup("SimpleDS");


        // Once the DataSource is obtained, it can be used to establish
        // a connection. The user profile cujo and password newtiger
        // used to create the connection instead of any default user
        // ID and password for the DataSource.
        Connection connection = ds.getConnection("cujo", "newtiger");

        // The connection can be used to create Statement objects and
        // update the database or process queries as follows.
        Statement statement = connection.createStatement();
        ResultSet rs = statement.executeQuery("select * from qsys2.sysprocs");
        while (rs.next()) {
            System.out.println(rs.getString(1) + "." + rs.getString(2));
        }

        // The connection is closed before the application ends.
        connection.close();
    }
}
```

DataSource プロパティー:

各 JDBC ドライバー接続プロパティーには、対応するデータ・ソース・メソッドがあります。この表では、有効なデータ・ソース・プロパティーを示します。

一部のプロパティーについては、対応するドライバー接続プロパティーを参照して詳細を確認することができます。

Set メソッド (データ・タイプ)	値	説明
setAccess(String)	"all"、"read call"、"read only"	access 接続プロパティーを参照してください。

Set メソッド (データ・タイプ)	値	説明
setAutoCommit(boolean)	"true"、"false"	auto commit 接続プロパティーを参照してください。
setBatchStyle(String)	"2.0"、"2.1"	batch style 接続プロパティーを参照してください。
setBlockSize(int)	"0"、"8"、"16"、"32"、"64"、"128"、"256"、"512"	block size 接続プロパティーを参照してください。
setCommitHold(boolean)	"true"、"false"	commit hold 接続プロパティーを参照してください。
setConcurrentAccessResolution(int)	1、2、3	concurrent access resolution プロパティーを参照してください。
setCursorHold(boolean)	"true"、"false"	cursor hold 接続プロパティーを参照してください。
setCursorSensitivity(String)	"sensitive"、"asensitive"	cursor sensitivity 接続プロパティーを参照してください。
setDataTruncation(boolean)	"true"、"false"	data truncation 接続プロパティーを参照してください。
setDatabaseName(String)	任意の名前	このプロパティーでは、DataSource が接続しようとするデータベースを指定します。デフォルトは *LOCAL です。データベース名はアプリケーションが動作しているシステム上のリレーショナル・データベース・ディレクトリーに存在しているか、ローカル・システムを指定する特殊値 *LOCAL か localhost でなければなりません。
setDataSourceName(String)	任意の名前	このプロパティーは、接続プーリングをサポートするために、ConnectionPoolDataSource Java Naming and Directory Interface (JNDI) 名を渡すことを許可します。
setDateFormat(String)	"julian"、"mdy"、"dmy"、"ymd"、"usa"、"iso"、"eur"、"jis"	date format 接続プロパティーを参照してください。
setDateSeparator(String)	"/", "-", "."、","、"b"	date separator 接続プロパティーを参照してください。
setDecimalSeparator(String)	".", ","	decimal separator 接続プロパティーを参照してください。
setDecfloatRoundingMode(String)	"round half even"、"round half up"、"round down"、"round ceiling"、"round floor"、"round half down"、"round up"、"round half even"	decfloat rounding mode 接続プロパティーを参照してください。
setDescription(String)	任意の名前	このプロパティーでは、DataSource オブジェクトの記述を設定します。
setDirectMap(boolean)	"true"、"false"	direct map 接続プロパティーを参照してください。
setDoEscapeProcessing(boolean)	"true"、"false"	do escape processing 接続プロパティーを参照してください。

Set メソッド (データ・タイプ)	値	説明
setFullErrors(boolean)	"true"、"false"	errors 接続プロパティーを参照してください。
setIgnoreWarnings(String)	SQLSTATE のコンマ区切りリスト。	ignore warnings 接続プロパティーを参照してください。
setLibraries(String)	スペースで区切られたライブラリーのリスト	libraries 接続プロパティーを参照してください。
setLobThreshold(int)	500000 未満の任意の値	lob threshold 接続プロパティーを参照してください。
setLoginTimeout(int)	任意の値	このプロパティーは、現在は無視されますが、今後使用される予定です。
setMaximumPrecision(int)	31、63	maximum precision 接続プロパティーを参照してください。
setMaximumScale(int)	0 から 63	maximum scale 接続プロパティーを参照してください。
setMinimumDivideScale(int)	0 から 9	minimum divide scale 接続プロパティーを参照してください。
setNetworkProtocol(int)	任意の値	このプロパティーは、現在は無視されますが、今後使用される予定です。
setPassword(String)	任意のストリング	password 接続プロパティーを参照してください。
setPortNumber(int)	任意の値	このプロパティーは、現在は無視されますが、今後使用される予定です。
setPrefetch(boolean)	"true"、"false"	prefetch 接続プロパティーを参照してください。
setQaqqinilib(String)	ライブラリー名	qaqqinilib 接続プロパティーを参照してください。
setQueryOptimizeGoal(String)	1、2	query optimize goal 接続プロパティーを参照してください。
setReuseObjects(boolean)	"true"、"false"	reuse objects 接続プロパティーを参照してください。
setServermodeSubsystem(String)	"*SAME"、subsystem name	servermode subsystem プロパティーを参照してください。
setServerName(String)	任意の名前	このプロパティーは、現在は無視されますが、今後使用される予定です。
setSystemNaming(boolean)	"true"、"false"	naming 接続プロパティーを参照してください。
setTimeFormat(String)	"hms"、"usa"、"iso"、"eur"、"jis"	time format 接続プロパティーを参照してください。
setTimeSeparator(String)	":"、"."、","、"b"	time separator 接続プロパティーを参照してください。
setTransactionIsolationLevel(String)	"none"、"read committed"、"read uncommitted"、"repeatable read"、"serializable"	transaction isolation 接続プロパティーを参照してください。
setTranslateBinary(Boolean)	"true"、"false"	translate binary 接続プロパティーを参照してください。

Set メソッド (データ・タイプ)	値	説明
setUseBlockInsert(boolean)	"true"、"false"	use block insert 接続プロパティーを参照してください。
setUser(String)	任意の値	user 接続プロパティーを参照してください。

JDBC 用の JVM プロパティー

ネイティブ JDBC ドライバーが使用する設定の一部は、接続プロパティーを使用しては設定できません。これらの設定は、ネイティブ JDBC ドライバーが実行する JVM に対して設定しなければならないものです。これらの設定は、ネイティブ JDBC ドライバーで作成されるすべての接続に使用されます。

ネイティブ・ドライバーは以下の JVM プロパティーを認識します。

プロパティー	値	意味
jdbc.db2.job.sort.sequence	デフォルト値 = *HEX	このプロパティーを true に設定すると、ネイティブ JDBC ドライバーは、デフォルト値の *HEX を使用する代わりに、ジョブを開始したユーザーのジョブ・ソート・シーケンスを使用するようになります。これを他の値に設定する場合や、設定しないでおく場合、JDBC は引き続きデフォルトの *HEX を使用します。これが意味することに十分注意してください。 JDBC 接続が接続要求で異なるユーザー・プロファイルを渡す場合、すべての接続に対して、サーバーを開始したユーザー・プロファイルのソート・シーケンスが使用されます。これは始動時に設定される環境属性であり、動的な接続属性ではありません。
jdbc.db2.trace	1 または error = エラー情報をトレース。 2 または info = 情報およびエラー情報をトレース。 3 または verbose = 詳細、情報、およびエラー情報をトレース。 4、all、または true = 可能なすべての情報をトレース。	このプロパティーは JDBC ドライバーのトレースをオンにします。これは、問題を報告する場合に使用してください。

プロパティー	値	意味
jdbc.db2.trace.config	stdout = トレース情報は stdout に送信されます (デフォルト値)。 usrtrc = トレース情報はユーザー・トレースに送信されます。トレース情報を入手するには、CL コマンド「ユーザー・トレース・バッファーのダンプ (DMPUSRTRC)」を使用することができます。 file://<pathtofile> = トレース情報はファイルに送信されます。ファイル名に "%j" が含まれている場合、"%j" はジョブ名で置き換えられます。 <pathtofile> の例は、/tmp/jdbc.%j.trace.txt です。	このプロパティーは、トレースの出力先を指定するために使用します。

DatabaseMetaData インターフェース

DatabaseMetaData インターフェースは、IBM Developer Kit for Java JDBC ドライバーによってインプリメントされ、基礎となるデータ・ソースに関する情報を提供します。これは、提供されているデータ・ソースとの対話方法を決定するため、主にアプリケーション・サーバーとツールによって使用されます。アプリケーションは、DatabaseMetaData メソッドを使用してもデータ・ソースの情報を入手することができますが、こちらはそれほど一般的ではありません。

DatabaseMetaData インターフェースには 150 を超えるメソッドが組み込まれており、これらのメソッドは提供する情報のタイプによって分類されています。これらについては以下で説明します。DatabaseMetaData インターフェースには、40 を超えるフィールドも含まれており、それらのフィールドは、さまざまな DatabaseMetaData メソッドの戻り値として使用される定数となります。

DatabaseMetaData インターフェースのメソッドの変更点について詳しくは、この後の『JDBC 3.0 の変更点』および『JDBC 4.0 の変更点』を参照してください。

DatabaseMetaData オブジェクトを作成する

DatabaseMetaData オブジェクトは、Connection メソッド getMetaData によって作成されます。オブジェクトが作成されると、これを用いて基礎となるデータ・ソースの情報を動的に発見することができます。以下の例では、DatabaseMetaData オブジェクトを作成し、これを使ってテーブル名の最大文字数を判別する方法を示します。

例: DatabaseMetaData オブジェクトを作成する

```
// con is a Connection object
DatabaseMetaData dbmd = con.getMetadata();
int maxLen = dbmd.getMaxTableNameLength();
```

一般情報を取得する

DatabaseMetaData の一部のメソッドは、データ・ソースに関する一般情報と、その実装に関する詳細を動的に発見するために使用されます。これらの重要なメソッドには、次のようなものがあります。

- getURL
- getUserName
- getDatabaseProductVersion、getDriverMajorVersion、および getDriverMinorVersion

- getSchemaTerm、getCatalogTerm、および getProcedureTerm
- nullsAreSortedHigh、および nullsAreSortedLow
- usesLocalFiles、および usesLocalFilePerTable
- getSQLKeywords

フィーチャー・サポートを判別する

DatabaseMetaData メソッドの多くは、特定のフィーチャーやフィーチャー・セットが、ドライバーや基礎となっているデータ・ソースでサポートされているかどうかを判別するために使用できます。これに加えて、一部のメソッドは提供されているサポートのレベルを記述します。個々のフィーチャーのサポートを記述するメソッドには、次のようなものがあります。

- supportsAlterTableWithDropColumn
- supportsBatchUpdates
- supportsTableCorrelationNames
- supportsPositionedDelete
- supportsFullOuterJoins
- supportsStoredProcedures
- supportsMixedCaseQuotedIdentifiers

フィーチャー・サポートのレベルを記述するためのメソッドには、次のようなものがあります。

- supportsANSI92EntryLevelSQL
- supportsCoreSQLGrammar

データ・ソースの制限

別のグループのメソッドは、特定のデータ・ソースによって課されている制限を提供します。このカテゴリーのメソッドには次のようなものがあります。

- getMaxRowSize
- getMaxStatementLength
- getMaxTablesInSelect
- getMaxConnections
- getMaxCharLiteralLength
- getMaxColumnsInTable

このグループのメソッドは、制限値を integer として返します。戻り値ゼロは、制限がないか、不明であることを示します。

SQL オブジェクトとその属性

DatabaseMetaData のいくつかのメソッドは、特定のデータ・ソースに移植される SQL オブジェクトに関する情報を提供します。これらのメソッドは SQL オブジェクトの属性を判別することができます。また、これらのメソッドは各行に特定のオブジェクトが記述された、ResultSet オブジェクトを戻します。たとえば、getUDTs メソッドはデータ・ソース内で定義されている各 UDT に対応した行を持つ ResultSet オブジェクトを戻します。このカテゴリーには、たとえば次のようなメソッドがあります。

- getSchemas および getCatalogs
- getTables

- getPrimaryKeys
- getProcedures および getProcedureColumns
- getUDTs

トランザクション・サポート

少数のメソッドのグループは、データ・ソースがサポートするトランザクション・セマンティクスに関する情報を提供します。このカテゴリーには、たとえば次のようなメソッドがあります。

- supportsMultipleTransactions
- getDefaultTransactionIsolation

DatabaseMetaData インターフェースの使用法の例については、 63 ページの『例: DatabaseMetaData インターフェースを使用して表のリストを戻す』を参照してください。

JDBC 3.0 の変更点

JDBC 3.0 のいくつかのメソッドでは、戻り値が変更されています。 JDBC 3.0 では、以下のメソッドが戻す ResultSet にフィールドが追加されました。

- getTables
- getColumns
- getUDTs
- getSchemas

注: Java Development Kit (JDK) 1.4 を使用してアプリケーションを開発している場合、テストの際に一定数の列が戻されることがあります。ユーザーが作成するアプリケーションで、すべての列にアクセスすることを想定しています。しかし、そのアプリケーションが以前のリリースの JDK でも動作するように設計した場合、これらのフィールドにアクセスしようとすると、アプリケーションは SQLException を受け取ります。以前の JDK のリリースではこれらのフィールドが存在しないためです。 64 ページの『例: 複数の列を持ったメタデータ ResultSet を使用する』では、いくつかの JDK のリリースで動作するアプリケーションを記述する方法の例が示されています。

JDBC 4.0 の変更点

V6R1 では、コマンド言語インターフェース (CLI) での MetaData API のインプリメンテーションが変更され、さらに SYSIBM ストアード・プロシージャーの呼び出しが行われるようになりました。このため V6R1 では、JDK レベルにかかわらず、JDBC MetaData メソッドが直接 SYSIBM プロシージャーを使用します。この変更により、次の点で違いが生じていることに気付かれるでしょう。

- 以前のネイティブ JDBC ドライバーでは、ほとんどのメソッドで `localhost` のユーザーをカタログ名として使用することができました。 JDBC 4.0 では、`localhost` が指定されていると、ネイティブ JDBC ドライバーは一切情報を戻しません。
- getBestRowIdentifier の NULL 可能パラメーターが false に設定されていると、ネイティブ JDBC ドライバーは常に空の ResultSet を戻します。これは、正しい結果を戻すように将来修正されます。
- BUFFER_LENGTH 列、SQL_DATA_TYPE 列、および SQL_DATETIME_SUB 列で getColumns によって戻される値が変わっている場合があります。JDBC の仕様ではこれらの列が「未使用」として定義されているため、JDBC アプリケーションではこれらの値を使用しないでください。

- 以前のネイティブ JDBC ドライバーでは、getCrossReference、getExportedKeys、getImportedKeys、および getPrimaryKeys の表とスキーマのパラメーターが「パターン」として認識されていました。この変更により、表とスキーマのパラメーターはデータベースに保管されている名前と一致していなければなりません。
- システム定義ビューのインプリメントに使用されるビューは、以前は、getTables() で SYSTEM TABLES として記述されていました。DB2 ファミリーとの整合を図るため、これらのビューは VIEWS として記述されるようになります。
- getProcedures によって戻される列名が変わっています。これらの列名は、JDBC 4.0 の仕様では定義されていません。また、情報が取得できないときは、getProcedures の注釈列に "" が戻されていましたが、この変更によって、NULL が戻されるようになりました。

表 4. JDBC 4.0 で getProcedures によって戻される列名

列番号	以前の列名	JDBC 4.0 での列名
4	RESERVED1	NUM_INPUT_PARAMS
5	RESERVED2	NUM_OUTPUT_PARAMS
6	RESERVED3	NUM_RESULT_SETS

- さまざまなデータ・タイプで getProcedureColumns によって戻される値のいくつかは、以下のように変更されています。

表 5. JDBC 4.0 で getProcedureColumns によって戻される値

データ・タイプ	列	以前の値	JDBC 4.0 での値
ALL	注釈	""	NULL
INTEGER	Length	NULL	4
SMALLINT	Length	NULL	2
BIGINT	dataType	19 (正しくない)	-5
BIGINT	Length	NULL	8
DECIMAL	Length	NULL	precision + scale
NUMERIC	Length	NULL	precision + scale
DOUBLE	TypeName	DOUBLE PRECISION	DOUBLE
DOUBLE	Length	NULL	8
FLOAT	TypeName	DOUBLE PRECISION	DOUBLE
FLOAT	Length	NULL	8
REAL	Length	NULL	4
DATE	Precision	NULL	10
DATE	Length	10	6
TIME	Precision	NULL	8
TIME	Length	8	6
TIME	Scale	NULL	0
TIMESTAMP	Precision	NULL	26
TIMESTAMP	Length	26	16
TIMESTAMP	Scale	NULL	6
CHAR	typeName	CHARACTER	CHAR
CHAR	Precision	NULL	Length と同じ

表 5. *JDBC 4.0 で getProcedureColumns によって戻される値 (続き)*

データ・タイプ	列	以前の値	JDBC 4.0 での値
VARCHAR	typeName	CHARACTER VARYING	VARCHAR
VARCHAR	Precision	NULL	Length と同じ
CLOB	dataType	NULL (正しくない)	2005
CLOB	typeName	CHARACTER LARGE OBJECT	CLOB
CLOB	Precision	NULL	Length と同じ
CHAR FOR BIT DATA	dataType	1 (CHAR)	-2 (BINARY)
CHAR FOR BIT DATA	typeName	CHARACTER	CHAR () FOR BIT DATA
CHAR FOR BIT DATA	Precision	NULL	Length と同じ
BLOB	dataType	NULL (正しくない)	2004
BLOB	typeName	BINARY LARGE OBJECT	BLOB
BLOB	Precision	NULL	Length と同じ
DATALINK	dataType	NULL (正しくない)	70
DATALINK	Precision	NULL	Length と同じ
VARCHAR FOR BIT DATA	dataType	12 (VARCHAR)	-3 (VARBINARY)
VARCHAR FOR BIT DATA	typeName	CHARACTER VARYING	VARCHAR () FOR BIT DATA
VARCHAR FOR BIT DATA	Precision	NULL	Length と同じ

READ ONLY ストアード・プロシージャーでの制約事項

ネイティブ JDBC は `access = read only` プロパティーをサポートしています。このプロパティーは、JDBC レベルで強制されます。このため、このプロパティーが設定されている場合でも MetaData プロシージャーは引き続き機能します。ただし、READ ONLY として定義されているデータベース・ストアード・プロシージャーからネイティブ JDBC ドライバーが使用されることがあります。この場合は、MetaData プロシージャーは機能しません。

新しいメソッド: getClientInfoProperties()

getClientInfoProperties メソッドは、ドライバーがサポートしているクライアント情報プロパティーのリストを取得します。各クライアント情報プロパティーは、SQL 特殊レジスターに保管されます。ネイティブ JDBC ドライバーで戻される ResultSet には、以下の情報が含まれます。

表 6. *getClientInfoProperties メソッドで戻される情報*

名前	最大長	デフォルト値	説明
ApplicationName	255	ブランク	現在接続を使用しているアプリケーションの名前。

表 6. *getClientInfoProperties メソッドで戻される情報 (続き)*

名前	最大長	デフォルト値	説明
ClientUser	255	ブランク	接続を使用しているアプリケーションで作業を実行しているユーザーの名前。これは、接続の確立に使用されたユーザー名とは同じではない場合があります。
ClientHostname	255	ブランク	接続を使用しているアプリケーションが実行されているコンピューターのホスト名。
ClientAccounting	255	ブランク	アカウント情報。

クライアント情報プロパティーに対応する SQL 特殊レジスターは以下のとおりです。

表 7. *SQL 特殊レジスター*

名前	SQL 特殊レジスター
ApplicationName	CURRENT CLIENT_APPLNAME
ClientUser	CURRENT CLIENT_USERID
ClientHostname	CURRENT CLIENT_WRKSTNNAME
ClientAccounting	CURRENT CLIENT_ACCTNG

clientInfoProperties は、Connection オブジェクトの setClientInfo メソッドを使用して設定できます。

関連概念

108 ページの『ResultSet』
ResultSet インターフェースは、照会の実行によって生成された結果へのアクセスを提供します。 概念上、ResultSet のデータは、特定数の列と特定数の行を含むテーブルとして考えることができます。 デフォルトでは、テーブル行は順番に検索されます。 検索の対象が 1 行であれば、列の値は、任意の順序でアクセスできます。

例: DatabaseMetaData インターフェースを使用して表のリストを戻す:

次の例は、テーブルのリストを戻す方法を示しています。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
// Connect to the server.
Connection c = DriverManager.getConnection("jdbc:db2:mySystem");

// Get the database meta data from the connection.
DatabaseMetaData dbMeta = c.getMetaData();

// Get a list of tables matching this criteria.
String catalog = "myCatalog";
String schema = "mySchema";
String table  = "myTable%"; // % indicates search pattern
String types[] = {"TABLE", "VIEW", "SYSTEM TABLE"}:
ResultSet rs = dbMeta.getTables(catalog, schema, table, types);
```

```
// ... iterate through the ResultSet to get the values.

// Close the connection.
c.close():
```

例: 複数の列を持ったメタデータ ResultSet を使用する:

以下は、複数の列があるメタデータ ResultSet を使用する方法の例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
//////////////////////////////////////////////////////////////////////////////////
//
// SafeGetUDTs example. This program demonstrates one way to deal with
// metadata ResultSets that have more columns in JDK 1.4 than they
// had in previous releases.
//
// Command syntax:
//    java SafeGetUDTs
//
//////////////////////////////////////////////////////////////////////////////////
//
// This source is an example of the IBM Developer for Java JDBC driver.
// IBM grants you a nonexclusive license to use this as an example
// from which you can generate similar function tailored to
// your own specific needs.
//
// This sample code is provided by IBM for illustrative purposes
// only. These examples have not been thoroughly tested under all
// conditions. IBM, therefore, cannot guarantee or imply
// reliability, serviceability, or function of these programs.
//
// All programs contained herein are provided to you "AS IS"
// without any warranties of any kind.  The implied warranties of
// merchantability and fitness for a particular purpose are
// expressly disclaimed.
//
// IBM Developer Kit for Java
// (C) Copyright IBM Corp. 2001
// All rights reserved.
// US Government Users Restricted Rights -
// Use, duplication, or disclosure restricted
// by GSA ADP Schedule Contract with IBM Corp.
//
//////////////////////////////////////////////////////////////////////////////////

import java.sql.*;

public class SafeGetUDTs {

    public static int jdbcLevel;

    // Note: Static block runs before main begins.
    // Therefore, there is access to jdbcLevel in
    // main.
    {
        try {
            Class.forName("java.sql.Blob");

            try {
                Class.forName("java.sql.ParameterMetaData");
                // Found a JDBC 3.0 interface.  Must support JDBC 3.0.
                jdbcLevel = 3;
            } catch (ClassNotFoundException ez) {
                // Could not find the JDBC 3.0 ParameterMetaData class.
```

```
                // Must be running under a JVM with only JDBC 2.0
                // support.
                jdbcLevel = 2;
            }

        } catch (ClassNotFoundException ex) {
            // Could not find the JDBC 2.0 Blob class.  Must be
            // running under a JVM with only JDBC 1.0 support.
            jdbcLevel = 1;
        }
    }

    // Program entry point.
    public static void main(java.lang.String[] args)
    {
        Connection c = null;

        try {
            // Get the driver registered.
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");

            c = DriverManager.getConnection("jdbc:db2:*local");
            DatabaseMetaData dmd = c.getMetaData();

            if (jdbcLevel == 1) {
                System.out.println("No support is provided for getUDTs.  Just return.");
                System.exit(1);
            }

            ResultSet rs = dmd.getUDTs(null, "CUJOSQL", "SSN%", null);
            while (rs.next()) {

                // Fetch all the columns that have been available since the
                // JDBC 2.0 release.
                System.out.println("TYPE_CAT is " + rs.getString("TYPE_CAT"));
                System.out.println("TYPE_SCHEM is " + rs.getString("TYPE_SCHEM"));
                System.out.println("TYPE_NAME is " + rs.getString("TYPE_NAME"));
                System.out.println("CLASS_NAME is " + rs.getString("CLASS_NAME"));
                System.out.println("DATA_TYPE is " + rs.getString("DATA_TYPE"));
                System.out.println("REMARKS is " + rs.getString("REMARKS"));

                // Fetch all the columns that were added in JDBC 3.0.
                if (jdbcLevel > 2) {
                    System.out.println("BASE_TYPE is " + rs.getString("BASE_TYPE"));
                }
            }
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        } finally {
            if (c != null) {
                try {
                    c.close();
                } catch (SQLException e) {
                    // Ignoring shutdown exception.
                }
            }
        }
    }
}
```

Java 例外

Java 言語では、プログラムのエラー処理機能を提供するために、例外を使用します。例外は、プログラムを実行しているときに、命令の通常フローが中断されたときに発生するイベントです。

Java ランタイム・システムおよび Java パッケージの多くのクラスは、いくつかの状況で throw ステートメントを使って例外をスローしています。この同じメカニズムを使って、ユーザーの Java プログラムでも例外をスローすることができます。

Java SQLException クラス:

SQLException クラスとそのサブタイプは、データ・ソースへのアクセス時に発生したエラーや警告に関する情報を提供します。

インターフェースで定義されている大半の JDBC とは異なり、例外サポートはクラスによって提供されています。 JDBC アプリケーションが実行されているときに発生する例外のための基本クラスは、SQLException です。 JDBC API のすべてのメソッドは、SQLException がスローできるように宣言されています。 SQLException は java.lang.Exception の拡張で、データベース・コンテキスト内で発生した障害に関連した追加情報を提供します。具体的には、SQLException からの次のような情報が使用可能です。

- テキスト記述
- SQLState
- エラー・コード
- 共に発生した他の例外への参照

ExceptionExample。これは、(この場合は、予測された) SQLException のキャッチと、提供されたすべての情報をダンプする適切な処理を行うプログラムです。

注: JDBC では、複数の例外を合わせてチェーニングするメカニズムが提供されています。これにより、ドライバーまたはデータベースが、一度の要求で複数のエラーをレポートすることができます。現在のところ、ネイティブ JDBC ドライバーが実際にこれを行う実例はありません。この情報は参照用として提供されており、ドライバーが今後もこれを行わないことを明示するものではありません。

前述のように、エラーが発生すると SQLException オブジェクトがスローされます。これは正しい動作ですが、完全な全体像ではありません。現実には、ネイティブ JDBC ドライバーが実際にスローすることはまれです。そのSQLException サブクラスのインスタンスがスローされます。これにより、以下にあるように、実際にどんな障害であったのかに関する詳しい情報を判別することができます。

DB2Exception.java

DB2Exception オブジェクトが直接スローされることはありません。この基本クラスは、すべての JDBC 例外に共通の機能性を保持するために使用されます。このクラスには、JDBC がスローする標準例外となる 2 つのサブクラスがあります。これらのサブクラスとは、DB2DBException.java および DB2JDBCException.java です。 DB2DBException は、データベースから直接レポートされた例外です。 DB2JDBCExceptions は、JDBC ドライバーが、ドライバー自身に問題を発見したときにスローされます。この方法で例外クラスが階層に分割されていることで、2 つのタイプの例外を個別に処理することができます。

DB2DBException.java

前述のように、DB2DBExceptions はデータベースから直接送信される例外です。これらは、JDBC ドライバーが CLI への呼び出しを行い、SQLERROR 戻りコードが返されたときに発生します。このケースでは、CLI 機能の SQLError は、メッセージ・テキスト、SQLState、およびベンダー・コードを取得するために呼び出されます。 SQLMessage の置換テキストも取得され、戻されます。 DatabaseException クラスは、データベースが認識し、例外オブジェクトを構築するために JDBC ドライバーにレポートするエラーを発生させます。

DB2JDBCException.java

DB2JDBCException は JDBC ドライバー自身からのエラー状態によって生成されます。この例外クラスの機能には基本的な違いがあります。 JDBC ドライバーそのものは例外のメッセージの言語翻訳を処理し、オペレーティング・システムおよびデータベースが処理するその他の問題の例外については、データベース内で生成されます。可能な限り、JDBC ドライバーはデータベースの SQLStates を順守します。 JDBC ドライバーがスローする例外のベンダー・コードは、常に -99999 です。しばしば、CLI 層によって認識され、戻される DB2DBException のエラー・コードも -99999 です。 JDBCException クラスは、JDBC ドライバーが自身の例外を認識し、構築するエラーを発生させます。リリースの開発中の実行時には、後続の出力が生成されます。スタックの最上部に DB2JDBCException が含まれていることに注意してください。これは、常にデータベースへの要求が行われる前にエラーが JDBC ドライバーからレポートされることを示しています。

例: SQLException:

以下に、SQLException をキャッチし、提供されたすべての情報をダンプする例を示します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;

public class ExceptionExample {

    public static Connection connection = null;

    public static void main(java.lang.String[] args) {

        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            connection = DriverManager.getConnection("jdbc:db2:*local");

            Statement s = connection.createStatement();
            int count = s.executeUpdate("insert into cujofake.cujofake values(1, 2,3)");

            System.out.println("Did not expect that table to exist.");


        } catch (SQLException e) {
            System.out.println("SQLException exception: ");
            System.out.println("Message:....." + e.getMessage());
            System.out.println("SQLState:...." + e.getSQLState());
            System.out.println("Vendor Code:." + e.getErrorCode());
            System.out.println("-----------------------------------------------------");
            e.printStackTrace();
        } catch (Exception ex) {
            System.out.println("An exception other than an SQLException was thrown: ");
            ex.printStackTrace();
        } finally {
            try {
                if (connection != null) {
                    connection.close();
                }
            } catch (SQLException e) {
                System.out.println("Exception caught attempting to shutdown...");
            }
        }
    }
}
```

SQLWarning:

一部のインターフェースのメソッドでは、データベース・アクセス警告を出すときに SQLWarning オブジェクトが生成されます。

以下のインターフェースのメソッドが SQLWarning を生成できます。

- 接続
- Statement とそのサブタイプ PreparedStatement および CallableStatement
- ResultSet

メソッドが SQLWarning オブジェクトを生成しても、データ・アクセス警告が発生したことは呼び出し元には通知されません。 SQLWarning オブジェクトを取り出すには、適当なオブジェクトで getWarnings メソッドを呼び出す必要があります。ただし、ある状況では、SQLWarning の DataTruncation サブクラスがスローされることがあります。ネイティブ JDBC ドライバーは、効率の向上のため、データベースが生成した一部の警告を無視する場合が多いという点に注意してください。たとえば、ResultSet の終わりに達した後でユーザーが ResultSet.next メソッドを使用してデータを取り出そうとすると、システムは警告を生成します。このような場合、next メソッドは、true ではなく false を返してユーザーにエラーを通知するように定義されています。これを改めて記述するオブジェクトを作成する必要はないので、この警告はただ無視されます。

複数のデータ・アクセス警告が発生すると、それらは最初の警告に連鎖されます。これらは、SQLWarning.getNextWarning メソッドを呼び出すことによって検索できます。連鎖した警告の終わりに達すると、getNextWarning は `null` を返します。

以降の SQLWarning オブジェクトは、次のステートメントが処理されるまで、その連鎖に追加され続けていきます。 ResultSet オブジェクトの場合には、カーソルが再配置されて連鎖した SQLWarning オブジェクトがすべて除去されるまで、その連鎖に追加され続けていきます。これにより、結果としてチェーン内のすべての SQLWarning オブジェクトが除去されます。

Connection、Statement、および ResultSet オブジェクトの使用は、SQLWarnings の生成される原因となることがあります。 SQLWarning は、特定の操作が正常に完了される間に注意すべき他の情報が存在した可能性を示す、通知メッセージです。 SQLWarning は、SQLException クラスの拡張機能ですが、スローされるものではなく、代わりに、その生成の原因となったオブジェクトに付加されます。なお、SQLWarning が生成されても、警告が生成されたことをアプリケーションに通知するイベントは何も起こりません。アプリケーションは、自発的に警告の情報を要求する必要があります。

SQLException と同様、SQLWarning は、相互に連鎖させることができます。オブジェクトの警告を消去するには、その Connection、Statement、または ResultSet といった各オブジェクトで、clearWarnings メソッドを呼び出すことができます。

注: clearWarnings メソッドを呼び出しても、すべての警告が消去されるわけではありません。消去されるのは、その特定のオブジェクトに関連付けられている警告だけです。

SQLWarning オブジェクトが手動で消去されない場合は、JDBC ドライバーが、ある特定のタイミングでそれらを消去します。 SQLWarning オブジェクトは、以下のアクションが行われたときに消去されます。

- Connection インターフェースの場合は、新規の Statement、PreparedStatement、または CallableStatement オブジェクトが作成されると、警告が消去されます。
- Statement インターフェースの場合は、次のステートメントが処理される (あるいは、PreparedStatement や CallableStatement のためにもう一度同じステートメントが処理される) と、警告が消去されます。
- ResultSet インターフェースの場合は、カーソルが再配置されると、警告が消去されます。

DataTruncation およびサイレント・トランケーション:

DataTruncation は SQLWarning のサブクラスです。 SQLWarning がスローされていないときに DataTruncation オブジェクトがスローされることがあり、他の SQLWarning オブジェクトのように付加されます。サイレント・トランケーションは、列のサイズが setMaxFieldSize ステートメント・メソッドで指定されたサイズを超える場合に起こります。しかし警告または例外は報告されません。

DataTruncation オブジェクトは SQLWarning が戻す情報よりも詳細な追加情報を提供します。そのような追加情報には、次のようなものがあります。

- 転送されたデータのバイト数。
- 切り捨てられた列、またはパラメーター索引。
- 索引がパラメーター用か、あるいは ResultSet 列用か。
- 切り捨てが、データベースから読み取り中に発生したか、書き込み中に発生したか。
- 実際に転送されたデータ量。

場合によっては、この情報を判読することはできますが、直感的にはわからないこともあります。たとえば、PreparedStatement の setFloat メソッドを使って、整数値を持つ列に値を挿入しようとした場合、その列が保持できる最大値よりも大きな値を挿入しようとして DataTruncation が返されることがあります。この場合は、切り捨てられたバイト数は意味がなく、ドライバーにとっては切り捨て情報が提供されることが重要です。

set() および update() メソッドの報告

JDBC ドライバーの間にも若干の違いがあります。ネイティブのドライバーや IBM Toolbox for Java JDBC ドライバーなどの一部のドライバーでは、パラメーターで設定した時点でデータ切り捨てをキャッチし、報告します。これは、PreparedStatement の set メソッドか、ResultSet の update メソッドのどちらかで行われます。その他のドライバーでは、ステートメントを処理したときに問題が報告され、execute、executeQuery、または updateRow メソッドが完了したときに報告されます。

処理が継続できなくなったときに問題を報告するのではなく、不正なデータが入力されたときに問題を報告することには、次のような利点があります。

- 処理時に問題に対処する代わりに、問題が発生したときにアプリケーション内の障害に対処することができる。
- パラメーターを設定するときに検査することにより、JDBC ドライバーは、ステートメントの処理時にデータベースに渡す値が正しいことを確認できる。これにより、データベースは作業を最適化して、処理を早く完了することができます。

ResultSet.update() メソッドが DataTruncation 例外をスローする

以前のいくつかのリリースでは、ResultSet.update() メソッドは切り捨て条件が存在すると警告を通知しました。これは、データ値をデータベースに挿入しようとするときに発生します。仕様では、このような状況が発生した場合、JDBC ドライバーは例外をスローするように決められています。その結果、JDBC ドライバーはこの方法で動作します。

データ切り捨てエラーを受け取る ResultSet 更新機能を処理することと、エラーを受け取る更新または挿入ステートメントの PreparedStatement パラメーター・セットを処理することには、大きな違いはありません。どちらのケースでも、問題は同一です。ユーザーが望んでいたものに適合しないデータが提供されたことです。

NUMERIC および DECIMAL は、小数点の右側を黙って切り捨てます。 JDBC for UDB NT の動作でも、IBM i プラットフォーム上で対話式 SQL で動作させた場合でも、この切り捨てが行われます。

注: データ切り捨てが発生すると、値は丸められません。 NUMERIC または DECIMAL 列に適合しない、パラメーターの端数部分は、警告なしに単に失われます。

以下の例では、PreparedStatement 上のパラメーターによって、VALUES 文節の値が実際にセットされていることを想定しています。

```
create table cujosql.test (col1 numeric(4,2))
a) insert into cujosql.test values(22.22)  // works - inserts 22.22
b) insert into cujosql.test values(22.223) // works - inserts 22.22
c) insert into cujosql.test values(22.227) // works - inserts 22.22
d) insert into cujosql.test values(322.22) // fails - Conversion error on assignment to column COL1.
```

データ切り捨て警告と、データ切り捨て例外との違い

仕様では、データベースに書き込まれる値のデータ切り捨ては、例外をスローするように決められています。データベースに書き込まれる値のデータ切り捨てが行われなかった場合は、警告が生成されます。これは、データ切り捨ての状況がどのポイントかを識別されることを意味しており、データの切り捨てが処理されたステートメントのタイプにも注意する必要があります。この要件に関連して、以下にいくつかの SQL ステートメントのタイプの動作を示します。

- SELECT ステートメントでは、照会パラメーターはデータベースの内容に損傷を与えることはありません。したがって、データ切り捨ては常に警告の通知として扱われます。
- VALUES INTO および SET ステートメントでは、入力値は出力値を生成するためだけに使用されます。その結果、警告が発行されます。
- CALL ステートメントでは、JDBC ドライバーはパラメーターが与えられたストアード・プロシージャーを判別することができません。ストアード・プロシージャーのパラメーターの切り捨てが行われると、常に例外がスローされます。
- その他のすべてのステートメント・タイプでは、警告が通知されるのではなく、例外がスローされます。

Connection および DataSource のデータ切り捨てプロパティー

多くのリリースでデータ切り捨てプロパティーが使用可能になっています。このプロパティーのデフォルトは true で、データ切り捨ての事象はチェックされ、警告の通知または例外のスローが行われます。このプロパティーは、値がデータベースの列に適合するかどうかが問題にならない場合に、便宜とパフォーマンスのために提供されています。列に挿入できる形で値を挿入するため、ドライバーが使用されます。

文字およびバイナリー・ベースのデータ・タイプにのみ効果のあるデータ切り捨てプロパティー

2 つ前のリリースでは、データ切り捨て例外をスローするかどうかはデータ切り捨てプロパティーによって判断されました。このデータ切り捨てプロパティーは、JDBC アプリケーションにとって切り捨てが重要ではないときに、切り捨てられた値を無視できるように用意されています。アプリケーションが DECIMAL(2,0) に 100 を挿入しようとしたとき、データベースに 00 または 10 のどちらかを格納したいというケースがあります。そのため、JDBC データ切り捨てプロパティーは、パラメーターが CHAR、VARCHAR、CHAR FOR BIT DATA、および VARCHAR FOR BIT DATA のような文字ベースのタイプの状況でのみ有効になるよう、変更されました。

パラメーターにのみ適用されるデータ切り捨てプロパティー

データ切り捨てプロパティーは、JDBC ドライバーの設定で、データベースの設定ではありません。そのため、ステートメント・リテラルには影響はありません。たとえば、データベース内の CHAR(8) 列に値を挿入する処理を行う以下のステートメントは、データ切り捨てフラグが false に設定されて失敗します (接続は java.sql.Connection オブジェクトとして別の場所で割り当てられていることを前提としています)。

```
Statement stmt = connection.createStatement();
Stmt.executeUpdate("create table cujosql.test (col1 char(8))");
Stmt.executeUpdate("insert into cujosql.test values('dettinger')");
// Fails as the value does not fit into database column.
```

問題とならないデータ切り捨てに対するネイティブ JDBC ドライバーの例外のスロー

ネイティブ JDBC ドライバーは、パラメーターとして提供されたデータを確認することは行いません。これは処理をスローダウンさせるだけです。しかし、値の切り捨てが問題にならない状況で、データ切り捨て接続プロパティーを false に設定していない状態では、この状況が発生することがあります。

たとえば、CHAR(10) である 'dettinger ' が渡されると、適合する値のすべてが重要なものであるとしても、例外がスローされます。これは、JDBC for UDB NT で発生する動作です。しかし、SQL ステートメント内でリテラルとして値を渡した場合は、このような振る舞いは得られません。このような場合、データベース・エンジンは追加のスペースを暗黙のうちにスローアウトします。

JDBC が例外をスローしない問題は、次のとおりです。

- 必要かどうかにかかわらず、すべての set メソッドでパフォーマンスのオーバーヘッドが拡大します。たいていの場合、これは有益なことではなく、setString() のような関数で相当なパフォーマンスのオーバーヘッドがあります。
- 渡されたストリング値でトリム関数を呼び出すなど、次善策は小さなものです。
- データベースの列に関する考慮すべき問題があります。 CCSID 37 でのスペースは、CCSID 65535 または 13488 でのスペースとは全く異なります。

サイレント・トランケーション

setMaxFieldSize ステートメント・メソッドは、任意の列の最大フィールド・サイズを指定できます。最大フィールド・サイズ値を超えたためにデータの切り捨てが行われた場合、警告や例外は報告されません。このメソッドは、前述のデータ切り捨てプロパティーと同じように、CHAR、VARCHAR、CHAR FOR BIT DATA、および VARCHAR FOR BIT DATA のような文字ベースのタイプでのみ有効です。

JDBC トランザクション

トランザクションは作業の論理単位です。作業の論理単位を完了するには、データベースに対していくつかのアクションを実行しなければならない場合があります。

トランザクション・サポートは、アプリケーションが以下を行うことを可能にします。

- 作業論理単位を完了するためのすべてのステップに従う。
- 作業単位を完了するためのステップの 1 つが失敗した場合に、作業論理単位一部として実行されたすべての作業を元に戻し、データベースをトランザクションが開始される前の状態に戻す。

トランザクションは、同時アクセスの間、データの整合性、正しいアプリケーション・セマンティクス、および矛盾のないデータのビューを提供するために使用されます。トランザクションのサポートには、常に Java Database Connectivity (JDBC) に準拠したドライバーが使用されなければなりません。

注: このセクションは、ローカル・トランザクションと、 トランザクションの標準的な JDBC 概念を説明しているに過ぎません。 Java やネイティブ JDBC ドライバーは、Java Transaction API (JTA)、分散トランザクション、および 2 フェーズ・コミット・プロトコル (2PC) をサポートします。

すべてのトランザクション作業は、Connection オブジェクト・レベルで処理されます。トランザクションの作業が完了すると、作業は、commit メソッドを呼び出すことによって終了処理できます。アプリケーションがトランザクションを打ち切る場合は、rollback メソッドが呼び出されます。

接続の下にあるすべての Statement オブジェクトは、トランザクションの一部になります。つまり、アプリケーションが 3 つの Statement オブジェクトを作成して、データベースに変更を加えるためにそれらの各オブジェクトを使用する場合は、commit または rollback 呼び出しが行われると、その 3 つのステートメントすべての作業が永続的にコミットされたり、ロールバックして廃棄されたりします。

純粋に SQL を使用して作業する場合は、トランザクションの終了処理に commit および rollback SQL ステートメントを使用します。これらの SQL ステートメントは、動的には作成できません。また、JDBC アプリケーションでは、これらのステートメントを使用したトランザクションの終了は試さないでください。

JDBC 自動コミット・モード:

デフォルトでは、JDBC は自動コミットと呼ばれる操作モードを使用します。このモードでは、データベースに対するすべての更新が即時に永続的にコミットされます。

ただし、自動コミット・モードは、作業論理単位がデータベースに複数の更新を必要とする状況では、安全性に欠けます。自動コミット・モードを使用した場合、1 つの更新が行われてから他の更新が行われるまでの間にアプリケーションやシステムで何らかの問題が発生すると、最初の更新は元に戻せなくなります。

自動コミット・モードでは、変更がすぐに永続的にコミットされるため、アプリケーションでは commit メソッドや rollback メソッドを呼び出す必要がありません。このため、アプリケーションの作成は容易になります。

自動コミット・モードの使用可能化/使用不可能化は、接続が存在している間に動的に行うことができます。自動コミットは、次のようにして使用可能にされます (データ・ソースがすでに存在していると想定した場合)。

```
Connection connection = dataSource.getConnection();

Connection.setAutoCommit(false);        // Disables auto-commit.
```

トランザクションの途中で自動コミットの設定が変更されると、保留中の作業はすべて自動的にコミットされます。分散トランザクションの一部となっている接続に対して自動コミットが使用可能にされると、SQLException が生成されます。

トランザクション分離レベル:

トランザクション分離レベルは、トランザクション内でステートメントが認識できるデータを指定します。これらのレベルは、同じターゲット・データ・ソースに対するトランザクション間で可能な相互作用を定義することにより、同時に行われるアクセスのレベルに直接影響します。

データベース異常

データベース異常とは、単一のトランザクションの視点から見ると誤っているように見え、すべてのトランザクションの視点から見ると正しく見える、生成された結果のことを言います。データベース異常の各タイプは、次のように説明できます。

- **ダーティー**読み取りは、次の場合に行われます。
  1. トランザクション A がテーブルに行を挿入する。
  2. トランザクション B が新しい行を読み取る。
  3. トランザクション A がロールバックする。

  トランザクション B は、トランザクション A によって挿入された行に基づいてシステムに対する作業を完了できますが、この行は、永続的なデータベースの一部にはなりません。
- **繰り返し不可**の読み取りは、次の場合に行われます。
  1. トランザクション A が行を読み取る。
  2. トランザクション B が行を変更する。
  3. トランザクション A は、2 度目に同じ行を読み取るが、最初とは違う新しい結果を得る。
- **ファントム**読み取りは、次の場合に行われます。
  1. トランザクション A が SQL 照会で WHERE 文節を満たすすべての行を読み取る。
  2. トランザクション B が WHERE 文節を満たす別の行を挿入する。
  3. トランザクション A が WHERE 条件を再評価すると、追加された行が検出される。

注: DB2 for i は、ロック・ストラテジーにより規定されているレベルでの許容データベース異常に対して、常にアプリケーションをさらしているわけではありません。

JDBC トランザクション分離レベル

IBM Developer Kit for Java の JDBC API には、5 つのトランザクション分離レベルがあります。最小のものから最大のものをリストすると、次のようになります。

JDBC_TRANSACTION_NONE
これは、JDBC ドライバーがトランザクションをサポートしないことを示す特殊な定数です。

JDBC_TRANSACTION_READ_UNCOMMITTED
このレベルでは、トランザクションは、データに対するコミットされていない変更を認識できます。データベース異常が起こるのは、すべてこのレベルです。

JDBC_TRANSACTION_READ_COMMITTED
このレベルは、トランザクション内で行われる一切の変更が、トランザクションがコミットされるまで外から認識されないことを意味します。これにより、ダーティー読み取りが行われる可能性はなくなります。

JDBC_TRANSACTION_REPEATABLE_READ
このレベルは、読み取られる行がロックしたまま保持されることにより、トランザクションが完了するまで他のトランザクションが行を変更できなくなることを意味します。これにより、ダーティー読み取りや繰り返し不可の読み取りは行えなくなります。ファントム読み取りは可能です。

JDBC_TRANSACTION_SERIALIZABLE
テーブルはトランザクション用にロックされ、テーブルに値を追加したりテーブルから値を除去したりする他のトランザクションによって WHERE 条件が変更されなくなります。これにより、すべてのタイプのデータベース異常が起こらなくなります。

接続のトランザクション分離レベルを変更するには、setTransactionIsolation メソッドを使用できます。

考慮事項

前述の 5 つのトランザクション・レベルについては、これが JDBC 仕様で定義されていると誤解される場合がよくあります。一般に、TRANSACTION_NONE の値は、コミットメント制御なしで実行する概念を表していると考えられています。 JDBC 仕様は、これと同じ方法で TRANSACTION_NONE を定義するものではありません。 TRANSACTION_NONE は、JDBC 仕様において、ドライバーがトランザクションをサポートしないレベルとして定義されており、JDBC 互換のドライバーであるわけではありません。getTransactionIsolation メソッドが呼び出されたときに NONE というレベルが報告されることは決してありません。

問題が少し複雑になっているのは、JDBC ドライバーのデフォルト・トランザクション分離レベルがインプリメンテーションで定義されるためです。ネイティブ JDBC ドライバーのデフォルトのトランザクション分離レベルは、NONE です。このレベルでは、ジャーナルがないファイルでもドライバーで処理することができ、QGPL ライブラリーのファイルのような指定を一切作成する必要がありません。

ネイティブ JDBC ドライバーでは、setTransactionIsolation メソッドに JDBC_TRANSACTION_NONE を渡したり、接続のプロパティーに NONE を指定することが可能です。とはいえ、getTransactionIsolation メソッドは、値が NONE であると常に JDBC_TRANSACTION_READ_UNCOMMITTED を報告します。アプリケーションで、どのレベルで実行しているかをトラッキングし続ける必要がある場合、そのトラッキングは、アプリケーションの責任で行われます。

過去のリリースでは、システムに本当の意味での自動コミット・モードの概念がなかったため、JDBC ドライバーは、自動コミットに真が指定されるとトランザクション分離レベルを NONE に変更することにでこれを処理していました。これは、機能としては近いものでしたが、すべてのシナリオにおいて正確な結果をもたらす処理ではありませんでした。この処理は行われなくなり、データベースは、トランザクション分離レベルの概念と自動コミットの概念を切り離します。これにより、自動コミットを使用可能にしたまま JDBC_TRANSACTION_SERIALIZABLE レベルでシステムを稼働させることが、完全に有効になります。唯一有効でないシナリオは、自動コミット・モードを使用せずに JDBC_TRANSACTION_NONE レベルでシステムを実行するシナリオです。トランザクション分離レベルを指定せずにシステムが実行された場合、アプリケーションはコミットの境界を制御することができません。

JDBC 仕様と IBM i プラットフォームの間のトランザクション分離レベル

IBM i プラットフォームで使用されるトランザクション分離レベルの共通名は、JDBC 仕様で指定されている名前と一致しません。次の表は、IBM i プラットフォームで使用される名前と付き合わせたものですが、これは JDBC 仕様で使用されるものに対応するものではありません。

JDBC レベル*	IBM i レベル
JDBC_TRANSACTION_NONE	*NONE または *NC
JDBC_TRANSACTION_READ_UNCOMMITTED	*CHG または *UR
JDBC_TRANSACTION_READ_COMMITTED	*CS
JDBC_TRANSACTION_REPEATABLE_READ	*ALL または *RS
JDBC_TRANSACTION_SERIALIZABLE	*RR

* この表では、分かりやすくするために JDBC_TRANSACTION_NONE の値と IBM i レベル *NONE および *NC を並べてあります。これは、仕様と IBM i レベルが直接一致することを示すものではありません。

保管ポイント:

保管ポイントを使用して、トランザクション内に「ステージング・ポイント」を設定できます。保管ポイントは、アプリケーションがトランザクション全体を取り消さなくてもロールバックできるチェックポイントです。

JDBC 3.0 では保管ポイントが新しくなりました。したがって、アプリケーションが保管ポイントを使用するには、そのアプリケーションを Java Development Kit (JDK) 1.4 または後続のリリース上で実行しなければなりません。さらに、Developer Kit for Java にとっては保管ポイントは新しい機能なので、旧リリースの Developer Kit for Java で JDK 1.4 または後続のリリースを使用していない場合は、これらの保管ポイントはサポートされません。

注: システムには保管ポイントを処理する SQL ステートメントが備えられています。 JDBC アプリケーションの中で、これらのステートメントを直接使用しないようにすることをお勧めします。直接使用しても作動しますが、JDBC ドライバーの実行時に保管ポイントを追跡する機能は失われます。最低限、2 つのモデルを混合する (つまり、独自の保管ポイント SQL ステートメントと JDBC API を使用する) ことは避ける必要があります。

保管ポイントの設定とロールバック

トランザクションの作業全体のどこにでも、保管ポイントを設定できます。その後、誤りが発生した場合に、アプリケーションはこれらのいずれかの保管ポイントにロールバックし、そのポイントから処理を続行することができます。以下の例では、アプリケーションはデータベース・テーブル中に値 FIRST を挿入します。その後、保管ポイントが設定され、別の値 SECOND がデータベース中に挿入されます。保管ポイントへのロールバックが発行されると、SECOND の挿入作業は取り消されますが、FIRST は保留トランザクションの一部として残ります。最後に、値 THIRD が挿入され、トランザクションがコミットされます。データベース・テーブルには、値 FIRST と THIRD が含まれます。

例: 保管ポイントの設定とロールバック

```
Statement s = Connection.createStatement();
s.executeUpdate("insert into table1 values ('FIRST')");
Savepoint pt1 = connection.setSavepoint("FIRST SAVEPOINT");
s.executeUpdate("insert into table1 values ('SECOND')";);
connection.rollback(pt1);        // Undoes most recent insert.
s.executeUpdate("insert into table1 values ('THIRD')");
connection.commit();
```

自動コミット・モードの場合に、保管ポイントの設定時に問題が起きることは考えられませんが、トランザクションの終了時に保管ポイントの存続期間が終了するとロールバックできません。

保管ポイントの解放

Connection オブジェクト上に releaseSavepoint メソッドを指定したアプリケーションにより、保管ポイントを解放できます。保管ポイントの解放後にロールバックを試行すると例外が生じます。トランザクションのコミット時やロールバック時に、保管ポイントはすべて解放されます。特定の保管ポイントがロールバックされた時点にも、後続の他の保管ポイントは解放されます。

JDBC 分散トランザクション

通常、Java Database Connectivity (JDBC) のトランザクションはローカルです。これは、単一の接続がトランザクションのすべての作業を行い、その接続では一度に 1 つのトランザクションだけが動作できることを意味します。

このトランザクションのすべての動作が完了するか、失敗すると、永続化するためにコミットまたはロールバックが呼び出され、新しいトランザクションが開始されます。ただしこれは、ローカル・トランザクショ

ン以上の機能を提供する Java で使用可能な、トランザクションの拡張サポートです。このサポートの完全な仕様は、「Java Transaction API」を参照してください。

Java Transaction API (JTA) は、複雑なトランザクションをサポートします。また、Connection オブジェクトからのトランザクションの分離もサポートします。 JDBC は ODBC、および X/Open Call Level Interface (CLI) 仕様を、また JTA は X/Open Extended Architecture (XA) 仕様をモデルにしています。 JTA と JDBC は共に動作して、Connection オブジェクトからトランザクションを分離します。 Connection オブジェクトからトランザクションを分離することにより、同時に複数のトランザクション上で単一の接続を動作させることができるようになります。逆に、単一のトランザクションで複数の接続を機能させることもできます。

注: JTA を使用する計画の場合は、「JDBC の入門」のトピックで、拡張クラスパス内に必要な Java Archive (JAR) ファイルに関する詳細情報を参照してください。JDBC 2.0 のオプショナル・パッケージと JTA JAR ファイルの両方が必要です (JDK 1.4 またはそれ以降のバージョンを実行している場合は、これらのファイルが JDK によって自動的に検索されます)。デフォルトでは検索されません。

JTA を使ったトランザクション

JTA と JDBC を同時に使用するときは、これらの間に、トランザクションの作業を完遂するための一連のステップがあります。 XA のサポートは、XADataSource クラスを通して提供されます。このクラスは、ConnectionPoolDataSource スーパークラスとまったく同じ方法による接続プーリングの設定のためのサポートが含まれています。

XADataSource インスタンスを使うと、XAConnection オブジェクトを取得できます。 XAConnection オブジェクトは、JDBC 接続と XAResource オブジェクトの両方のためのコンテナーを提供します。 XAResource オブジェクトは、XA トランザクション・サポートを処理するために設計されました。 XAResource は、オブジェクトを経由したトランザクションを、トランザクション ID (XID) によって処理します。

XID は、必ずインプリメントする必要のあるインターフェースです。これは、X/Open トランザクション ID の XID 構造の Java マッピングに相当します。このオブジェクトには、次の 3 つの部分が含まれます。

- グローバル・トランザクションのフォーマット ID
- グローバル・トランザクション ID
- ブランチ修飾子

このインターフェースの完全な詳細については、JTA 仕様を参照してください。

プーリングおよび分散トランザクション用の UDBXADataSource サポートを使用する

Java Transaction API サポートは、接続プーリングの直接サポートを提供しています。 UDBXADataSource は ConnectionPoolDataSource の拡張で、プールされた XAConnection オブジェクトにアプリケーションがアクセスすることを可能にします。 UDBXADataSource は ConnectionPoolDataSource なので、UDBXADataSource の構成および使用方法は、「オブジェクト・プーリング用の DataSource サポートの使用」トピックで説明されている方法と同一です。

XADataSource プロパティー

ConnectionPoolDataSource によって提供されているプロパティーに加えて、XADataSource インターフェースは次のようなプロパティーを提供しています。

Set メソッド (データ・タイプ)	値	説明
setLockTimeout (int)	0 または任意の正の値	トランザクション・レベルの有効なロック・タイムアウト (秒) で、任意の正の値です。 ロック・タイムアウトを 0 にするのは、たとえ他のレベル (ジョブまたはテーブル) で強制される値がある場合でも、トランザクション・レベルで強制されるロック・タイムアウトが無いことを意味します。 デフォルト値は 0 です。
setTransactionTimeout (int)	0 または任意の正の値	有効なトランザクション・タイムアウト (秒) で、任意の正の値です。 トランザクション・タイムアウトが 0 に指定されるのは、強制されるトランザクション・タイムアウト値がないことを意味します。トランザクションがタイムアウト値よりも長時間、活動状態になった場合は、ロールバックのみとしてマークされ、このトランザクション下の以降の操作の試行では例外が発生します。 デフォルト値は 0 です。

ResultSets およびトランザクション

前述の例でも示されていたように、トランザクションの開始と終了は区別されているため、トランザクションをしばらく中断し、後で再開することができます。これにより、トランザクションの間に作成された ResultSet リソースのたくさんのシナリオが提供されます。

単純なトランザクションの終了

トランザクションを終了すると、トランザクションの中で作成され、開かれているすべての ResultSet は自動的にクローズされます。最大の並列処理を行うため、ResultSet は使用が完了したときに明示的にクローズすることをお勧めします。しかし、トランザクション中で開かれている ResultSet に XAResource.end 呼び出しを行った後にアクセスしようとすると、結果として例外が発生します。

中断および再開

トランザクションが中断されている間、トランザクションが活動状態にあるときに作成された ResultSet にアクセスすることはできず、例外になります。しかし、トランザクションを再開すると、再び ResultSet は使用可能になり、トランザクションが中断される前と同じ状態に戻ります。

ResultSet の中断の影響

トランザクションを中断している間は、ResultSet にアクセスできません。しかし、動作を実行するため、他のトランザクションの下で Statement オブジェクトを再処理することができます。 JDBC Statement オブジェクトは同時に 1 つの ResultSet しか持つことができず (JDBC 3.0 の、ストアード・プロシージャー呼

び出しからの複数の同時 ResultSet のサポートは除きます)、新しいトランザクションからの要求を満たすために、中断されたトランザクションの ResultSet はクローズしなければなりません。このことが発生します。

注: JDBC 3.0 では、Statement がストアード・プロシージャー呼び出しで同時に複数の ResultSet を開くことができますが、これは 1 つの単位として見なされ、Statement が新しいトランザクションで再処理されると、すべてクローズされます。単一のステートメントで、同時に活動状態にある 2 つのトランザクションで ResultSet を持つことはできません。

多重化

JTA API は、JDBC 接続からトランザクションの分離のために設計されています。この API によって、単一のトランザクション上で複数の接続が動作させることも、同時に複数のトランザクション上で単一の接続を動作させることも可能です。これは**多重化**と呼ばれ、JDBC 単体では実行できない数多くの複雑なタスクを実行できます。

JTA を使用するためのさらに詳細な情報は、JTA 仕様を参照してください。 JDBC 3.0 の仕様でも、これらの 2 つのテクノロジーが共に分散トランザクションをサポートする方法についての情報が含まれています。

2 フェーズ・コミットおよびトランザクション・ログ

JTA API は、分散 2 フェーズ・コミット・プロトコルの役割をアプリケーションに対して完全に外部化します。これまでの例で示されているとおり、JTA トランザクション下で JTA および JDBC を使ってデータベースにアクセスするとき、アプリケーションは変更をコミットするために XAResource.prepare() および XAResource.commit() メソッド、または単に XAResource.commit() メソッドを使用します。

さらに、単一のトランザクションを使用して複数の個別のデータベースにアクセスするとき、これらのデータベースにあるトランザクションの非分離性に必要とされる、2 フェーズ・コミット・プロトコル、およびすべての関連するロギングを保証するのは、アプリケーションの責任です。一般的には、複数のデータベースにまたがる 2 フェーズ・コミットの処理 (つまり、XAResource)、およびロギングは、アプリケーション・サーバーまたはトランザクション・モニターの制御下で実行されます。これは、アプリケーションそのものが、実際にこれらの問題に関係しないためです。

たとえば、アプリケーションがいくつかの commit() メソッドを呼び出し、エラーなしで処理が戻されたとします。その後、基礎となるアプリケーション・サーバーまたはトランザクション・モニターは、単一の分散トランザクションに加わっている各データベース (XAResource) の処理を開始します。

アプリケーション・サーバーは 2 フェーズ・コミット処理の間は、広範囲のロギングを使用します。 XAResource.prepare() メソッドがそれぞれの参加データベース (XAResource) で順番に呼び出され、続いて XAResource.commit() メソッドがそれぞれの参加データベース (XAResource) で呼び出されます。

この処理中に障害が発生した場合、アプリケーション・サーバーのトランザクション・モニターのログにより、アプリケーション・サーバーそのものが後で分散トランザクションを回復するために JTA API を使用することができるようにします。アプリケーション・サーバーまたはトランザクション・モニターの制御下で行われる回復は、アプリケーション・サーバーがそれぞれの参加データベース (XAResource) の既知の状態へのトランザクションを取得できるようにします。これで、すべての参加データベースをまたいだ分散トランザクション全体の既知の状態を確実に得られます。

関連概念

29 ページの『JDBC 入門』
IBM i 上の Java に付属している Java Database Connectivity (JDBC) ドライバーのことを、IBM Developer Kit for Java の JDBC ドライバーと呼びます。 このドライバーは、一般にネイティブ JDBC ドライバーとも呼ばれます。

126 ページの『オブジェクト・プーリングのための DataSource サポートの使用』
データベースにアクセスするための共通の構成を複数のアプリケーションで共用するために、 DataSources を使用できます。 このことは、各アプリケーションで同じ DataSource 名を参照させることによって実現します。

関連資料

129 ページの『ConnectionPoolDataSource のプロパティー』
ConnectionPoolDataSource インターフェースは、用意されている一連のプロパティーを使用することによって構成できます。

関連情報

Java Transaction API 1.0.1 仕様

例: トランザクションを処理するために JTA を使用する:

以下は、アプリケーション内でトランザクションを処理するための Java Transaction API (JTA) の使用法の例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.transaction.*;
import javax.transaction.xa.*;
import com.ibm.db2.jdbc.app.*;

public class JTACommit {

    public static void main(java.lang.String[] args) {
        JTACommit test = new JTACommit();

        test.setup();
        test.run();
    }


    /**
     * Handle the previous cleanup run so that this test can recommence.
     */
    public void setup() {

        Connection c = null;
        Statement s = null;
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            c = DriverManager.getConnection("jdbc:db2:*local");
            s = c.createStatement();

            try {
                s.executeUpdate("DROP TABLE CUJOSQL.JTATABLE");
            } catch (SQLException e) {
                // Ignore... does not exist
```

```
        }

        s.executeUpdate("CREATE TABLE CUJOSQL.JTATABLE (COL1 CHAR (50))");
        s.close();
    } finally {
        if (c != null) {
            c.close();
        }
    }
}


/**
 * This test uses JTA support to handle transactions.
 */
public void run() {
    Connection c = null;

    try {
        Context ctx = new InitialContext();

        // Assume the data source is backed by a UDBXADataSource.
        UDBXADataSource ds = (UDBXADataSource) ctx.lookup("XADataSource");

        // From the DataSource, obtain an XAConnection object that
        // contains an XAResource and a Connection object.
        XAConnection  xaConn = ds.getXAConnection();
        XAResource    xaRes  = xaConn.getXAResource();
        Connection    c      = xaConn.getConnection();

        // For XA transactions, a transaction identifier is required.
        // An implementation of the XID interface is not included with the
        // JDBC driver. See Transactions with JTA for a description of
        // this interface to build a class for it.
        Xid xid = new XidImpl();

        // The connection from the XAResource can be used as any other
        // JDBC connection.
        Statement stmt = c.createStatement();

        // The XA resource must be notified before starting any
        // transactional work.
        xaRes.start(xid, XAResource.TMNOFLAGS);

        // Standard JDBC work is performed.
        int count =
          stmt.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('JTA is pretty fun.')");

        // When the transaction work has completed, the XA resource must
        // again be notified.
        xaRes.end(xid, XAResource.TMSUCCESS);

        // The transaction represented by the transaction ID is prepared
        // to be committed.
        int rc = xaRes.prepare(xid);

        // The transaction is committed through the XAResource.
        // The JDBC Connection object is not used to commit
        // the transaction when using JTA.
        xaRes.commit(xid, false);


    } catch (Exception e) {
        System.out.println("Something has gone wrong.");
        e.printStackTrace();
    } finally {
        try {
```

```
                if (c != null)
                    c.close();
            } catch (SQLException e) {
                System.out.println("Note:  Cleaup exception.");
                e.printStackTrace();
            }
        }
    }
}
```

例: 単一トランザクション上で動作する複数の接続:

以下は、単一トランザクション上で動作する複数の接続の使用法の例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.transaction.*;
import javax.transaction.xa.*;
import com.ibm.db2.jdbc.app.*;
public class JTAMultiConn {
   public static void main(java.lang.String[] args) {
      JTAMultiConn test = new JTAMultiConn();
      test.setup();
      test.run();
   }
/**
* Handle the previous cleanup run so that this test can recommence.
*/
   public void setup() {
      Connection c = null;
      Statement s = null;
      try {
         Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
         c = DriverManager.getConnection("jdbc:db2:*local");
         s = c.createStatement();
         try {
            s.executeUpdate("DROP TABLE CUJOSQL.JTATABLE");
         }
         catch (SQLException e) {
         // Ignore... does not exist
         }
         s.executeUpdate("CREATE TABLE CUJOSQL.JTATABLE (COL1 CHAR
                         (50))");
               s.close();
      }
      finally {
         if (c != null) {
            c.close();
         }
      }
   }
/**
* This test uses JTA support to handle transactions.
*/
   public void run() {
      Connection c1 = null;
      Connection c2 = null;
      Connection c3 = null;
      try {
         Context ctx = new InitialContext();
         // Assume the data source is backed by a UDBXADataSource.
         UDBXADataSource ds = (UDBXADataSource)
```

```
                        ctx.lookup("XADataSource");
   // From the DataSource, obtain some XAConnection objects that
   // contain an XAResource and a Connection object.
   XAConnection xaConn1 = ds.getXAConnection();
   XAConnection xaConn2 = ds.getXAConnection();
   XAConnection xaConn3 = ds.getXAConnection();
   XAResource xaRes1 = xaConn1.getXAResource();
   XAResource xaRes2 = xaConn2.getXAResource();
   XAResource xaRes3 = xaConn3.getXAResource();
   c1 = xaConn1.getConnection();
   c2 = xaConn2.getConnection();
   c3 = xaConn3.getConnection();
   Statement stmt1 = c1.createStatement();
   Statement stmt2 = c2.createStatement();
   Statement stmt3 = c3.createStatement();
   // For XA transactions, a transaction identifier is required.
   // Support for creating XIDs is again left to the application
   // program.
   Xid xid = JDXATest.xidFactory();
   // Perform some transactional work under each of the three
   // connections that have been created.
   xaRes1.start(xid, XAResource.TMNOFLAGS);
   int count1 = stmt1.executeUpdate("INSERT INTO " + tableName + "VALUES('Value 1-A')");
   xaRes1.end(xid, XAResource.TMNOFLAGS);

   xaRes2.start(xid, XAResource.TMJOIN);
   int count2 = stmt2.executeUpdate("INSERT INTO " + tableName + "VALUES('Value 1-B')");
   xaRes2.end(xid, XAResource.TMNOFLAGS);

   xaRes3.start(xid, XAResource.TMJOIN);
   int count3 = stmt3.executeUpdate("INSERT INTO " + tableName + "VALUES('Value 1-C')");
   xaRes3.end(xid, XAResource.TMSUCCESS);
   // When completed, commit the transaction as a single unit.
   // A prepare() and commit() or 1 phase commit() is required for
   // each separate database (XAResource) that participated in the
   // transaction. Since the resources accessed (xaRes1, xaRes2, and xaRes3)
   // all refer to the same database, only one prepare or commit is required.
   int rc = xaRes.prepare(xid);
   xaRes.commit(xid, false);
}
catch (Exception e) {
   System.out.println("Something has gone wrong.");
   e.printStackTrace();
}
finally {
   try {
      if (c1 != null) {
         c1.close();
      }
   }
   catch (SQLException e) {
      System.out.println("Note: Cleaup exception " +
                         e.getMessage());
   }
   try {
      if (c2 != null) {
         c2.close();
      }
   }
   catch (SQLException e) {
      System.out.println("Note: Cleaup exception " +
                         e.getMessage());
   }
   try {
      if (c3 != null) {
         c3.close();
      }
```

```
        }
        catch (SQLException e) {
           System.out.println("Note: Cleaup exception " +
                              e.getMessage());
        }
     }
   }
}
```

例: 複数のトランザクションで単一の接続を使用する:

以下は、複数のトランザクションでの単一接続の使用法の例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.transaction.*;
import javax.transaction.xa.*;
import com.ibm.db2.jdbc.app.*;

public class JTAMultiTx {

    public static void main(java.lang.String[] args) {
        JTAMultiTx test = new JTAMultiTx();

        test.setup();
        test.run();
    }


    /**
     * Handle the previous cleanup run so that this test can recommence.
     */
    public void setup() {

        Connection c = null;
        Statement s = null;
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            c = DriverManager.getConnection("jdbc:db2:*local");
            s = c.createStatement();

            try {
                s.executeUpdate("DROP TABLE CUJOSQL.JTATABLE");
            } catch (SQLException e) {
                // Ignore... does not exist
            }

            s.executeUpdate("CREATE TABLE CUJOSQL.JTATABLE (COL1 CHAR (50))");

            s.close();
        } finally {
            if (c != null) {
                c.close();
            }
        }
    }


    /**
     * This test uses JTA support to handle transactions.
     */
    public void run() {
```

```
        Connection c = null;

        try {
            Context ctx = new InitialContext();

            // Assume the data source is backed by a UDBXADataSource.
            UDBXADataSource ds = (UDBXADataSource) ctx.lookup("XADataSource");

            // From the DataSource, obtain an XAConnection object that
            // contains an XAResource and a Connection object.
            XAConnection  xaConn = ds.getXAConnection();
            XAResource    xaRes  = xaConn.getXAResource();
            Connection    c      = xaConn.getConnection();
            Statement     stmt   = c.createStatement();

            // For XA transactions, a transaction identifier is required.
            // This is not meant to imply that all the XIDs are the same.
            // Each XID must be unique to distinguish the various transactions
            // that occur.
            // Support for creating XIDs is again left to the application
            // program.
            Xid xid1 = JDXATest.xidFactory();
            Xid xid2 = JDXATest.xidFactory();
            Xid xid3 = JDXATest.xidFactory();

            // Do work under three transactions for this connection.
            xaRes.start(xid1, XAResource.TMNOFLAGS);
            int count1 = stmt.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('Value 1-A')");
            xaRes.end(xid1, XAResource.TMNOFLAGS);

            xaRes.start(xid2, XAResource.TMNOFLAGS);
            int count2 = stmt.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('Value 1-B')");
            xaRes.end(xid2, XAResource.TMNOFLAGS);

            xaRes.start(xid3, XAResource.TMNOFLAGS);
            int count3 = stmt.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('Value 1-C')");
            xaRes.end(xid3, XAResource.TMNOFLAGS);

            // Prepare all the transactions
            int rc1 = xaRes.prepare(xid1);
            int rc2 = xaRes.prepare(xid2);
            int rc3 = xaRes.prepare(xid3);

            // Two of the transactions commit and one rolls back.
            // The attempt to insert the second value into the table is
            // not committed.
            xaRes.commit(xid1, false);
            xaRes.rollback(xid2);
            xaRes.commit(xid3, false);


        } catch (Exception e) {
            System.out.println("Something has gone wrong.");
            e.printStackTrace();
        } finally {
            try {
                if (c != null)
                    c.close();
            } catch (SQLException e) {
                System.out.println("Note:  Cleaup exception.");
                e.printStackTrace();
            }
        }
    }
}
```

例: 中断状態の ResultSets:

以下は、作業を実行するために Statement オブジェクトを別のトランザクションで再処理する方法の例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.transaction.*;
import javax.transaction.xa.*;
import com.ibm.db2.jdbc.app.*;

public class JTATxEffect {

    public static void main(java.lang.String[] args) {
        JTATxEffect test = new JTATxEffect();

        test.setup();
        test.run();
    }


    /**
     * Handle the previous cleanup run so that this test can recommence.
     */
    public void setup() {

        Connection c = null;
        Statement s = null;
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            c = DriverManager.getConnection("jdbc:db2:*local");
            s = c.createStatement();

            try {
                s.executeUpdate("DROP TABLE CUJOSQL.JTATABLE");
            } catch (SQLException e) {
                // Ignore... does not exist
            }

            s.executeUpdate("CREATE TABLE CUJOSQL.JTATABLE (COL1 CHAR (50))");
            s.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('Fun with JTA')");
            s.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('JTA is fun.)");

            s.close();
        } finally {
            if (c != null) {
                c.close();
            }
        }
    }


    /**
     * This test uses JTA support to handle transactions.
     */
    public void run() {
        Connection c = null;

        try {
            Context ctx = new InitialContext();

            // Assume the data source is backed by a UDBXADataSource.
            UDBXADataSource ds = (UDBXADataSource) ctx.lookup("XADataSource");
```

```
        // From the DataSource, obtain an XAConnection object that
        // contains an XAResource and a Connection object.
        XAConnection  xaConn = ds.getXAConnection();
        XAResource    xaRes  = xaConn.getXAResource();
        Connection    c      = xaConn.getConnection();

        // For XA transactions, a transaction identifier is required.
        // An implementation of the XID interface is not included with
        // the JDBC driver. See Transactions with JTA
        // for a description of this interface to build a
        // class for it.
        Xid xid = new XidImpl();

        // The connection from the XAResource can be used as any other
        // JDBC connection.
        Statement stmt = c.createStatement();

        // The XA resource must be notified before starting any
        // transactional work.
        xaRes.start(xid, XAResource.TMNOFLAGS);

        // Create a ResultSet during JDBC processing and fetch a row.
        ResultSet rs = stmt.executeUpdate("SELECT * FROM CUJOSQL.JTATABLE");
        rs.next();

        // The end method is called with the suspend option.
        // ResultSets associated with the current transaction are 'on hold'.
        // They are neither gone nor accessible in this state.
        xaRes.end(xid, XAResource.TMSUSPEND);


        // In the meantime, other work can be done outside the transaction.
        // The ResultSets under the transaction can be closed if the
        // Statement object used to create them is reused.
        ResultSet nonXARS = stmt.executeQuery("SELECT * FROM CUJOSQL.JTATABLE");
        while (nonXARS.next()) {
            // Process here...
        }


        // Attempt to go back to the suspended transaction. The suspended
        // transaction's ResultSet has disappeared because the statement
        // has been processed again.
        xaRes.start(newXid, XAResource.TMRESUME);
        try {
            rs.next();
        } catch (SQLException ex) {
            System.out.println("This exception is expected. " +
                               "The ResultSet closed due to another process.");
        }


        // When the transaction had completed, end it
        // and commit any work under it.
        xaRes.end(xid, XAResource.TMNOFLAGS);
        int rc = xaRes.prepare(xid);
        xaRes.commit(xid, false);


    } catch (Exception e) {
        System.out.println("Something has gone wrong.");
        e.printStackTrace();
    } finally {
        try {
            if (c != null)
                c.close();
        } catch (SQLException e) {
```

```
                System.out.println("Note:  Cleaup exception.");
                e.printStackTrace();
            }
        }
    }
}
```

例: トランザクションを終了する:

以下は、アプリケーション内でトランザクションを終了する方法の例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.transaction.*;
import javax.transaction.xa.*;
import com.ibm.db2.jdbc.app.*;

public class JTATxEnd {

    public static void main(java.lang.String[] args) {
        JTATxEnd test = new JTATxEnd();

        test.setup();
        test.run();
    }


    /**
     * Handle the previous cleanup run so that this test can recommence.
     */
    public void setup() {

        Connection c = null;
        Statement s = null;
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            c = DriverManager.getConnection("jdbc:db2:*local");
            s = c.createStatement();

            try {
                s.executeUpdate("DROP TABLE CUJOSQL.JTATABLE");
            } catch (SQLException e) {
                // Ignore... does not exist
            }

            s.executeUpdate("CREATE TABLE CUJOSQL.JTATABLE (COL1 CHAR (50))");
            s.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('Fun with JTA')");
            s.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('JTA is fun.)");

            s.close();
        } finally {
            if (c != null) {
                c.close();
            }
        }
    }


    /**
     * This test use JTA support to handle transactions.
     */
    public void run() {
```

```
        Connection c = null;

        try {
            Context ctx = new InitialContext();

            // Assume the data source is backed by a UDBXADataSource.
            UDBXADataSource ds = (UDBXADataSource) ctx.lookup("XADataSource");

            // From the DataSource, obtain an XAConnection object that
            // contains an XAResource and a Connection object.
            XAConnection  xaConn = ds.getXAConnection();
            XAResource    xaRes  = xaConn.getXAResource();
            Connection    c      = xaConn.getConnection();

            // For XA transactions, transaction identifier is required.
            // An implementation of the XID interface is not included
            // with the JDBC driver. See Transactions with JTA for a
            // description of this interface to build a class for it.
            Xid xid = new XidImpl();

            // The connection from the XAResource can be used as any other
            // JDBC connection.
            Statement stmt = c.createStatement();

            // The XA resource must be notified before starting any
            // transactional work.
            xaRes.start(xid, XAResource.TMNOFLAGS);

            // Create a ResultSet during JDBC processing and fetch a row.
            ResultSet rs = stmt.executeUpdate("SELECT * FROM CUJOSQL.JTATABLE");
            rs.next();

            // When the end method is called, all ResultSet cursors close.
            // Accessing the ResultSet after this point results in an
            // exception being thrown.
            xaRes.end(xid, XAResource.TMNOFLAGS);

            try {
                String value = rs.getString(1);
                System.out.println("Something failed if you receive this message.");
            } catch (SQLException e) {
                System.out.println("The expected exception was thrown.");
            }

            // Commit the transaction to ensure that all locks are
            // released.
            int rc = xaRes.prepare(xid);
            xaRes.commit(xid, false);


        } catch (Exception e) {
            System.out.println("Something has gone wrong.");
            e.printStackTrace();
        } finally {
            try {
                if (c != null)
                    c.close();
            } catch (SQLException e) {
                System.out.println("Note:  Cleaup exception.");
                e.printStackTrace();
            }
        }
    }
}
```

75 ページの『JDBC 分散トランザクション』
通常、Java Database Connectivity (JDBC) のトランザクションはローカルです。 これは、単一の接続がトランザクションのすべての作業を行い、その接続では一度に 1 つのトランザクションだけが動作できることを意味します。

例: トランザクションを中断して再開する:

以下は、中断され、その後再開されるトランザクションの例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.transaction.*;
import javax.transaction.xa.*;
import com.ibm.db2.jdbc.app.*;
import javax.naming.InitialContext;
import javax.naming.Context;

public class JTATxSuspend {

    public static void main(java.lang.String[] args) {
        JTATxSuspend test = new JTATxSuspend();

        test.setup();
        test.run();
    }


    /**
     * Handle the previous cleanup run so that this test can recommence.
     */
    public void setup() {

        Connection c = null;
        Statement s = null;
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            c = DriverManager.getConnection("jdbc:db2:*local");
            s = c.createStatement();

            try {
                s.executeUpdate("DROP TABLE CUJOSQL.JTATABLE");
            } catch (SQLException e) {
                // Ignore... doesn't exist
            }

            s.executeUpdate("CREATE TABLE CUJOSQL.JTATABLE (COL1 CHAR (50))");
            s.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('Fun with JTA')");
            s.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('JTA is fun.)");

            s.close();
        } finally {
            if (c != null) {
                c.close();
            }
        }
    }


    /**
     * This test uses JTA support to handle transactions.
     */
```

```
public void run() {
    Connection c = null;

    try {
        Context ctx = new InitialContext();

        // Assume the data source is backed by a UDBXADataSource.
        UDBXADataSource ds = (UDBXADataSource) ctx.lookup("XADataSource");

        // From the DataSource, obtain an XAConnection object that
        // contains an XAResource and a Connection object.
        XAConnection  xaConn = ds.getXAConnection();
        XAResource    xaRes  = xaConn.getXAResource();
        c      = xaConn.getConnection();

        // For XA transactions, a transaction identifier is required.
        // An implementation of the XID interface is not included with
        // the JDBC driver. See topic "Transactions with JTA" for a
        // description of this interface to build a class for it.
        Xid xid = new XidImpl();

        // The connection from the XAResource can be used as any other
        // JDBC connection.
        Statement stmt = c.createStatement();

        // The XA resource must be notified before starting any
        // transactional work.
        xaRes.start(xid, XAResource.TMNOFLAGS);

        // Create a ResultSet during JDBC processing and fetch a row.
        ResultSet rs = stmt.executeQuery("SELECT * FROM CUJOSQL.JTATABLE");
        rs.next();

        // The end method is called with the suspend option.
        // ResultSets associated with the current transaction are 'on hold'.
        // They are neither gone nor accessible in this state.
        xaRes.end(xid, XAResource.TMSUSPEND);


        // Other work can be performed with the transaction.
        // As an example, you can create a statement and process a query.
        // This work and any other transactional work that the transaction may
        // perform is separate from the work done previously under the XID.
        Statement nonXAStmt = c.createStatement();
        ResultSet nonXARS = nonXAStmt.executeQuery("SELECT * FROM CUJOSQL.JTATABLE");
        while (nonXARS.next()) {
            // Process here...
        }
        nonXARS.close();
        nonXAStmt.close();


        // If an attempt is made to use any suspended transactions
        // resources, an exception results.
        try {
            rs.getString(1);
            System.out.println("Value of the first row is " + rs.getString(1));
        } catch (SQLException e) {
            System.out.println("This was an expected exception - " +
                               "suspended ResultSet was used.");
        }


        // Resume the suspended transaction and complete the work on it.
        // The ResultSet is exactly as it was before the suspension.
        xaRes.start(newXid, XAResource.TMRESUME);
        rs.next();
```

```
            System.out.println("Value of the second row is " + rs.getString(1));


            // When the transaction has completed, end it
            // and commit any work under it.
            xaRes.end(xid, XAResource.TMNOFLAGS);
            int rc = xaRes.prepare(xid);
            xaRes.commit(xid, false);


        } catch (Exception e) {
            System.out.println("Something has gone wrong.");
            e.printStackTrace();
        } finally {
            try {
                if (c != null)
                    c.close();
            } catch (SQLException e) {
                System.out.println("Note:  Cleaup exception.");
                e.printStackTrace();
            }
        }
    }
}
```

ステートメントのタイプ

Statement インターフェースとその PreparedStatement および CallableStatement サブクラスは、データベースに対する構造化照会言語 (SQL) コマンドの処理に使用されます。 SQL ステートメントが処理されると、ResultSet オブジェクトが生成されます。

Statement インターフェースのサブクラスは、Connection インターフェース上のいくつかのメソッドを使用して作成されます。 1 つの Connection オブジェクトには、そのオブジェクトの下で同時に作成された多数の Statement オブジェクトを含めることができます。過去のリリースでは、作成できる Statement オブジェクトの正確な数を示すことができましたが、このリリースではそれはできません。というのは、様々なタイプの Statement オブジェクトがあり、そのタイプによって、データベース・エンジンに持ち込む「ハンドル」の数が異なるからです。したがって、1 回の接続でアクティブにできるステートメントの数は、使用する Statement オブジェクトのタイプによって異なります。

アプリケーションは、Statement.close メソッドを呼び出して、ステートメントの処理が終了したことを示します。すべての Statement オブジェクトは、作成元の接続がクローズされたときにクローズされます。ただし、Statement オブジェクトをクローズするときに、この動作に 100% の信頼を置かないようにしてください。たとえば、アプリケーションが変更され、接続を明示的にクローズする代わりに接続プールが使用される場合は、接続がクローズされないため、アプリケーションからステートメント・ハンドルが「リーク」します。必要なくなった時点で Statement オブジェクトをすぐにクローズすれば、そのステートメントが使用している外部データベース・リソースをすぐに解放できます。

ネイティブの JDBC ドライバーは、リークしているステートメントを検出し、それを代わりに処理しようとします。ただし、このサポートに頼ると、パフォーマンスは低くなります。

各インターフェースは、CallableStatement が PreparedStatement を拡張し、PreparedStatement が Statement を拡張する、というように階層状の継承関係にあるため、各インターフェースの機能は、そのインターフェースを拡張するクラスで使用できます。たとえば、Statement クラスの機能は、PreparedStatement クラスや CallableStatement クラスでもサポートされます。ただし、これに対する主な例外として、Statement クラスの executeQuery、executeUpdate、および execute メソッドがあります。これらのメソッドは、動的な処理を行うための SQL ステートメントを扱うので、PreparedStatement オブジェクトや CallableStatement オブジェクトでこれらのメソッドを使用すると、例外が発生します。

Statement オブジェクト:

Statement オブジェクトは、静的 SQL ステートメントの処理と、それによって生成される結果の取得に使用されます。一度にオープンできるのは、各 Statement オブジェクトにつき 1 つの ResultSet だけです。SQL ステートメントを処理するすべてのステートメント・メソッドは、すでにオープンされている ResultSet があると、暗黙的にステートメントの現行の ResultSet をクローズします。

ステートメントの作成

Statement オブジェクトは、createStatement メソッドを使用して Connection オブジェクトから作成されます。たとえば、conn という Connection オブジェクトがすでに存在しているとした場合、データベースに SQL ステートメントを渡すための Statement オブジェクトは、次のようなコードの行で作成されます。

```
Statement stmt = conn.createStatement();
```

ResultSet 特性の指定

ResultSet の特性は、最終的にその ResultSet を作成するステートメントに関連付けられています。これらの ResultSet の特性は、Connection.createStatement メソッドで指定できます。以下は、createStatement メソッドに対する有効な呼び出しの例を示しています。

例: createStatement メソッド

```
// The following is new in JDBC 2.0

Statement stmt2 = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATEABLE);

// The following is new in JDBC 3.0

Statement stmt3 = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY, ResultSet.HOLD_CURSOR_OVER_COMMIT);
```

これらの特性に関する詳細は、ResultSetを参照してください。

ステートメントの処理

Statement オブジェクトを使用した SQL ステートメントの処理は、executeQuery()、executeUpdate()、および execute() メソッドで行われます。

SQL 照会からの戻り結果

ResultSet を戻す SQL 照会ステートメントを処理する場合は、executeQuery() メソッドを使用します。Statement オブジェクトの executeQuery メソッドを使用して ResultSet を取得する サンプル・プログラムを参照してください。

注: executeQuery で処理される SQL ステートメントが ResultSet を戻さない場合は、SQLException がスローされます。

SQL ステートメントの更新カウントの戻り

SQL が、更新カウントを戻すデータ定義言語 (DDL) ステートメントまたはデータ操作言語 (DML) ステートメントであると分かっている場合は、executeUpdate() メソッドを使用します。 StatementExample プログラムは、Statement オブジェクトの executeUpdate メソッドを使用します。

何が戻されるか分からない SQL ステートメントの処理

SQL ステートメントのタイプが不明な場合、execute メソッドを使用します。このメソッドが一度処理されると、JDBC ドライバーはアプリケーションに、API 呼び出しを通して SQL ステートメントが生成した結果のタイプを通知することができます。 execute メソッドは、結果が少なくとも 1 つの ResultSet である場合は true を、戻り値が更新カウントである場合は false を戻します。この情報を得た後、アプリケーションは statement メソッドの getUpdateCount または getResultSet を使用して、SQL ステートメントの処理から戻り値を取り出すことができます。 StatementExecute プログラムは、Statement オブジェクトで execute メソッドを使用します。このプログラムは、パラメーターとして SQL ステートメントが渡されることを期待します。プログラムは、渡された SQL のテキストを確認しなくても、ステートメントを処理することによって、何についての情報が処理されたのかを判別します。

注: 結果が ResultSet の場合に getUpdateCount メソッドを呼び出すと、-1 が戻されます。結果が更新カウントの場合に getResultSet メソッドを呼び出すと、ヌルが戻されます。

cancel メソッド

ネイティブの JDBC ドライバーのメソッドは、同じオブジェクトに対して 2 つのスレッドが実行されてオブジェクトが壊れないよう、同期されます。ただし、cancel メソッドは例外です。 cancel メソッドは、同じオブジェクトの別のスレッドで長時間実行されている SQL ステートメントを停止させるのに使用できます。ネイティブの JDBC ドライバーでは、実行していたタスクすべてを停止するように要求することしかできず、強制的にスレッドに作業を停止させることはできません。この理由で、JDBC ドライバーでは、キャンセルされたステートメントを停止させるのにも時間がかかります。 cancel メソッドは、システム上のランナウェイ SQL 照会を停止させるのに使用できます。

例: Statement オブジェクトの executeUpdate メソッドを使用する:

次に、Statement オブジェクトの executeUpdate メソッドを使用する方法の例を示します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;
import java.util.Properties;

public class StatementExample {

    public static void main(java.lang.String[] args)
    {

        // Suggestion:  Load these from a properties object.
        String DRIVER = "com.ibm.db2.jdbc.app.DB2Driver";
        String URL    = "jdbc:db2://*local";

        // Register the native JDBC driver. If the driver cannot be
        // registered, the test cannot continue.
        try {
            Class.forName(DRIVER);
        } catch (Exception e) {
            System.out.println("Driver failed to register.");
            System.out.println(e.getMessage());
            System.exit(1);
        }

        Connection c = null;
        Statement s = null;

        try {
```

```
            // Create the connection properties.
            Properties properties = new Properties ();
            properties.put ("user", "userid");
            properties.put ("password", "password");

            // Connect to the local database.
            c = DriverManager.getConnection(URL, properties);

            // Create a Statement object.
            s = c.createStatement();
            // Delete the test table if it exists. Note: This
            // example assumes that the collection MYLIBRARY
            // exists on the system.
            try {
                s.executeUpdate("DROP TABLE MYLIBRARY.MYTABLE");
            } catch (SQLException e) {
                // Just continue... the table probably does not exist.
            }

            // Run an SQL statement that creates a table in the database.
            s.executeUpdate("CREATE TABLE MYLIBRARY.MYTABLE (NAME VARCHAR(20), ID INTEGER)");

            // Run some SQL statements that insert records into the table.
            s.executeUpdate("INSERT INTO MYLIBRARY.MYTABLE (NAME, ID) VALUES ('RICH', 123)");
            s.executeUpdate("INSERT INTO MYLIBRARY.MYTABLE (NAME, ID) VALUES ('FRED', 456)");
            s.executeUpdate("INSERT INTO MYLIBRARY.MYTABLE (NAME, ID) VALUES ('MARK', 789)");

            // Run an SQL query on the table.
            ResultSet rs = s.executeQuery("SELECT * FROM MYLIBRARY.MYTABLE");

            // Display all the data in the table.
            while (rs.next()) {
                System.out.println("Employee " + rs.getString(1) + " has ID " + rs.getInt(2));
            }

        } catch (SQLException sqle) {
            System.out.println("Database processing has failed.");
            System.out.println("Reason: " + sqle.getMessage());
        } finally {
            // Close database resources
            try {
                if (s != null) {
                    s.close();
                }
            } catch (SQLException e) {
                System.out.println("Cleanup failed to close Statement.");
            }
             }

            try {
                if (c != null) {
                    c.close();
                }
            } catch (SQLException e) {
                System.out.println("Cleanup failed to close Connection.");
            }
        }
    }
}
```

PreparedStatement:

PreparedStatement は Statement インターフェースを拡張し、SQL ステートメントへのパラメーターの追加をサポートします。

データベースに渡される SQL ステートメントは、結果を戻すまでに 2 段階のプロセスを通ります。これらはまず準備され、次いで処理されます。 Statement オブジェクトの場合、これらの 2 つのフェーズは、アプリケーションには 1 つのフェーズとして映ります。 PreparedStatement では、この 2 つのステップは分離が可能です。準備ステップは、オブジェクトが作成されるときに発生し、処理ステップは PreparedStatement オブジェクトに対して executeQuery、executeUpdate、または execute メソッドが呼び出されるときに発生します。

SQL 処理を個々のフェーズに分割できても、パラメーター・マーカーが追加されなければ、それは無意味です。アプリケーションにパラメーター・マーカーが置かれることによって、アプリケーションは、準備時には特定の値を持たないこと、しかし、処理の前には値を指定していることをデータベースに知らせることができます。パラメーター・マーカーは SQL ステートメントでは疑問符で表示されます。

パラメーター・マーカーを使用すれば、汎用 SQL ステートメントを作成し、それを特定の要求に合わせて使用することができます。例として、以下の SQL 照会ステートメントについて取り上げてみます。

```
SELECT * FROM EMPLOYEE_TABLE WHERE LASTNAME = 'DETTINGER'
```

これは、ただ 1 つの値、つまり Dettinger という名前の従業員についての情報を戻す特定の SQL ステートメントです。このステートメントは、以下のようなパラメーター・マーカーを追加することによって、より柔軟なものにすることができます。

```
SELECT * FROM EMPLOYEE_TABLE WHERE LASTNAME = ?
```

単純に値にパラメーター・マーカーを設定することによって、テーブル内のどの従業員についての情報でも取得することができます。

上の Statement の例は、準備フェーズを一度だけ通過すれば、パラメーターに異なる値を指定して繰り返し処理できるので、PreparedStatement によって Statement のパフォーマンスは大幅に向上しています。

注: PreparedStatement の使用は、ネイティブ JDBC ドライバーの Statement プーリングをサポートするための要件です。

PreparedStatement の作成、ResultSet 特性の指定、自動生成キーの処理、およびパラメーター・マーカーの設定を含め、PreparedStatement について詳しくは、以下のページを参照してください。

***PreparedStatements* を作成し、使用する:**

新しい PreparedStatement オブジェクトを作成するには、prepareStatement メソッドを使用します。 createStatement メソッドとは違って、 SQL ステートメントは PreparedStatement オブジェクトの作成時に指定する必要があります。その時点で SQL ステートメントは使用のためにプリコンパイルされます。

たとえば、conn という名前の Connection オブジェクトがすでに存在しているとするなら、以下の例では PreparedStatement オブジェクトが作成され、データベース内の処理のための SQL ステートメントが準備されます。

```
PreparedStatement ps = conn.prepareStatement("SELECT * FROM EMPLOYEE_TABLE
                                              WHERE LASTNAME = ?");
```

ResultSet 特性の指定および自動生成キー・サポート

createStatement メソッドと同様に、prepareStatement メソッドは、ResultSet の特性の指定をサポートするよう多重定義されています。さらに prepareStatement メソッドには、自動生成キーを処理するためのバリエーションがあります。以下に、prepareStatement メソッドの有効な呼び出し例をいくつか示します。

例: prepareStatement メソッド

注: 法律上の重要な情報に関しては、コードの特記事項情報をお読みください。

```
// New in JDBC 2.0

PreparedStatement ps2 = conn.prepareStatement("SELECT * FROM
    EMPLOYEE_TABLE WHERE LASTNAME = ?",

ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATEABLE);

// New in JDBC 3.0

PreparedStatement ps3 = conn.prepareStatement("SELECT * FROM
    EMPLOYEE_TABLE WHERE LASTNAME = ?",
    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATEABLE,
    ResultSet.HOLD_CURSOR_OVER_COMMIT);


PreparedStatement ps4 = conn.prepareStatement("SELECT * FROM
    EMPLOYEE_TABLE WHERE LASTNAME = ?", Statement.RETURN_GENERATED_KEYS);
```

パラメーターのハンドリング

PreparedStatement オブジェクトを処理する前に、各パラメーター・マーカーに何らかの値を設定する必要があります。 PreparedStatement オブジェクトには、パラメーターを設定するためのいくつかのメソッドが備わっています。どのメソッドも、set<Type> (<Type> は Java データ・タイプ) という形式です。これらのメソッドの一部の例には、setInt、setLong、setString、setTimestamp、setNull、および setBlob が含まれています。ほとんどすべてのメソッドは 2 つのパラメーターをとります。

- 最初のパラメーターは、ステートメント内のパラメーターの指標です。パラメーター・マーカーには番号が付けられます。番号は 1 から始まります。
- 2 番目のパラメーターは、パラメーターに設定する値です。 setBinaryStream の length パラメーターなど、追加パラメーターを持つ set<Type> メソッドもいくつかあります。

詳しくは、java.sql パッケージの Javadoc を参照してください。 ps オブジェクト用の上記の例の準備済み SQL ステートメントの場合、処理前のパラメーター値の指定方法は、以下のコードのようになります。

```
ps.setString(1,'Dettinger');
```

設定されていないパラメーター・マーカーを用いて PreparedStatement を処理しようとすると、SQLException が出されます。

注: パラメーター・マーカーは一度設定されると、以下の状況が発生しない限り、処理の間で同じ値を保持します。

- 別の set メソッドの呼び出しによって値が変更された。
- clearParameters メソッドが呼び出されたときに値が除去された。

clearParameters メソッドは、すべてのパラメーターに設定解除済みのフラグを立てます。clearParameters の呼び出しが行われた後、次の処理の前にすべてのパラメーターに対してもう一度 set メソッドを呼び出す必要があります。

ParameterMetaData サポート

新しい ParameterMetaData インターフェースでは、パラメーターについての情報を検索することができます。このサポートは、ResultSetMetaData に対する補足で、これと類似しています。精度、位取り、データ・タイプ、および、パラメーターがヌル値を許可するかどうか、などの情報がすべて供給されます。

例: ParameterMetaData:

これは、ParameterMetaData インターフェースを使用して、パラメーターについての情報を検索するときの一例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
//////////////////////////////////////////////////////////////////////////////////
//
// ParameterMetaData example. This program demonstrates
// the new support of JDBC 3.0 for learning information
// about parameters to a PreparedStatement.
//
// Command syntax:
//    java PMD
//
//////////////////////////////////////////////////////////////////////////////////
//
// This source is an example of the IBM Developer for Java JDBC driver.
// IBM grants you a nonexclusive license to use this as an example
// from which you can generate similar function tailored to
// your own specific needs.
//
// This sample code is provided by IBM for illustrative purposes
// only. These examples have not been thoroughly tested under all
// conditions. IBM, therefore, cannot guarantee or imply
// reliability, serviceability, or function of these programs.
//
// All programs contained herein are provided to you "AS IS"
// without any warranties of any kind.  The implied warranties of
// merchantability and fitness for a particular purpose are
// expressly disclaimed.
//
// IBM Developer Kit for Java
// (C) Copyright IBM Corp. 2001
// All rights reserved.
// US Government Users Restricted Rights -
// Use, duplication, or disclosure restricted
// by GSA ADP Schedule Contract with IBM Corp.
//
//////////////////////////////////////////////////////////////////////////////////

import java.sql.*;

public class PMD {

    // Program entry point.
    public static void main(java.lang.String[] args)
    throws Exception
    {
        // Obtain setup.
        Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
        Connection c = DriverManager.getConnection("jdbc:db2:*local");
        PreparedStatement ps = c.prepareStatement("INSERT INTO CUJOSQL.MYTABLE VALUES(?, ?, ?)");
        ParameterMetaData pmd = ps.getParameterMetaData();

        for (int i = 1; i < pmd.getParameterCount(); i++) {
            System.out.println("Parameter number " + i);
            System.out.println("  Class name is " + pmd.getParameterClassName(i));
            // Note: Mode relates to input, output or inout
            System.out.println("  Mode is " + pmd.getParameterClassName(i));
            System.out.println("  Type is " + pmd.getParameterType(i));
            System.out.println("  Type name is " + pmd.getParameterTypeName(i));
            System.out.println("  Precision is " + pmd.getPrecision(i));
            System.out.println("  Scale is " + pmd.getScale(i));
```

```
            System.out.println("  Nullable? is " + pmd.isNullable(i));
            System.out.println("  Signed? is " + pmd.isSigned(i));
        }
    }
}
```

***PreparedStatements* の処理:**

PreparedStatement オブジェクトを含む SQL ステートメントの処理は、Statement オブジェクトの処理と同様に executeQuery、executeUpdate、および execute メソッドによって行われます。しかし、この SQL ステートメントはオブジェクトの作成時に既に指定されているものなので、Statement 用のメソッドとは違って、これらのメソッドにパラメーターは渡されません。 PreparedStatement は Statement を拡張するものなので、アプリケーションは SQL ステートメントを取る種類の executeQuery、executeUpdate、および execute メソッドを呼び出そうとする可能性もあります。そうすると、SQLException が出されます。

SQL 照会からの戻り結果

ResultSet オブジェクトを戻す SQL 照会ステートメントを処理する場合は、executeQuery メソッドを使用します。 PreparedStatementExample プログラムは、PreparedStatement オブジェクトの executeQuery メソッドを使用して ResultSet を取得します。

注: executeQuery メソッドによって処理された SQL ステートメントが ResultSet を戻さない場合、SQLException が出されます。

SQL ステートメントの更新カウントの戻り

SQL が、更新カウントを戻すデータ定義言語 (DDL) ステートメントまたはデータ操作言語 (DML) ステートメントであると分かっている場合は、executeUpdate() メソッドを使用します。 PreparedStatementExample サンプル・プログラムは、PreparedStatement オブジェクトの executeUpdate メソッドを使用します。

何が戻されるか分からない SQL ステートメントの処理

SQL ステートメントのタイプが不明な場合、execute メソッドを使用します。一度このメソッドが処理されると、JDBC ドライバーは、SQL ステートメントが API 呼び出しを通して生成した結果のタイプをアプリケーションに通知することができるようになります。 execute メソッドは、結果が少なくとも 1 つの ResultSet である場合は true を、戻り値が更新カウントである場合は false を戻します。この情報を得た後、アプリケーションは getUpdateCount または getResultSet ステートメント・メソッドを使用して、SQL ステートメントの処理から戻り値を取り出すことができます。

注: 結果が ResultSet の場合に getUpdateCount メソッドを呼び出すと、-1 が戻されます。結果が更新カウントの場合に getResultSet メソッドを呼び出すと、ヌルが戻されます。

***例: ResultSet* を取得するために *PreparedStatement* を使用する:**

これは、PreparedStatement オブジェクトの executeQuery メソッドを使用して、ResultSet を入手するときの一例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;
import java.util.Properties;

public class PreparedStatementExample {
```

```
public static void main(java.lang.String[] args)
{
    // Load the following from a properties object.
    String DRIVER = "com.ibm.db2.jdbc.app.DB2Driver";
    String URL    = "jdbc:db2://*local";

    // Register the native JDBC driver. If the driver cannot
    // be registered, the test cannot continue.
    try {
        Class.forName(DRIVER);
    } catch (Exception e) {
        System.out.println("Driver failed to register.");
        System.out.println(e.getMessage());
        System.exit(1);
    }

    Connection c = null;
    Statement s = null;

    //    This program creates a table that is
    //    used by prepared statements later.
    try {
        // Create the connection properties.
        Properties properties = new Properties ();
        properties.put ("user", "userid");
        properties.put ("password", "password");

        // Connect to the local database.
        c = DriverManager.getConnection(URL, properties);

        // Create a Statement object.
        s = c.createStatement();
        // Delete the test table if it exists.  Note that
        // this example assumes throughout that the collection
        // MYLIBRARY exists on the system.
        try {
            s.executeUpdate("DROP TABLE MYLIBRARY.MYTABLE");
        } catch (SQLException e) {
            // Just continue... the table probably did not exist.
        }

        // Run an SQL statement that creates a table in the database.
        s.executeUpdate("CREATE TABLE MYLIBRARY.MYTABLE (NAME VARCHAR(20), ID INTEGER)");

    } catch (SQLException sqle) {
        System.out.println("Database processing has failed.");
        System.out.println("Reason: " + sqle.getMessage());
    } finally {
        // Close database resources
        try {
            if (s != null) {
                s.close();
            }
        } catch (SQLException e) {
            System.out.println("Cleanup failed to close Statement.");
        }
    }


    //    This program then uses a prepared statement to insert many
    //    rows into the database.
    PreparedStatement ps = null;
    String[] nameArray = {"Rich", "Fred", "Mark", "Scott", "Jason",
        "John", "Jessica", "Blair", "Erica", "Barb"};
    try {
        // Create a PreparedStatement object that is used to insert data into the
        // table.
```

```
            ps = c.prepareStatement("INSERT INTO MYLIBRARY.MYTABLE (NAME, ID) VALUES (?, ?)");

            for (int i = 0; i < nameArray.length; i++) {
                ps.setString(1, nameArray[i]);      // Set the Name from our array.
                ps.setInt(2, i+1);                  // Set the ID.
                ps.executeUpdate();
            }

        } catch (SQLException sqle) {
            System.out.println("Database processing has failed.");
            System.out.println("Reason: " + sqle.getMessage());
        } finally {
            // Close database resources
            try {
                if (ps != null) {
                    ps.close();
                }
            } catch (SQLException e) {
                System.out.println("Cleanup failed to close Statement.");
            }
        }


        //    Use a prepared statement to query the database
        //    table that has been created and return data from it. In
        //    this example, the parameter used is arbitrarily set to
        //    5, meaning return all rows where the ID field is less than
        //    or equal to 5.
        try {
            ps = c.prepareStatement("SELECT * FROM MYLIBRARY.MYTABLE " +
                                    "WHERE ID <= ?");

            ps.setInt(1, 5);

            // Run an SQL query on the table.
            ResultSet rs = ps.executeQuery();
            // Display all the data in the table.
            while (rs.next()) {
                System.out.println("Employee " + rs.getString(1) + " has ID " + rs.getInt(2));
            }

        } catch (SQLException sqle) {
            System.out.println("Database processing has failed.");
            System.out.println("Reason: " + sqle.getMessage());
        } finally {
            // Close database resources
            try {
                if (ps != null) {
                    ps.close();
                }
            } catch (SQLException e) {
                System.out.println("Cleanup failed to close Statement.");
            }

            try {
                if (c != null) {
                    c.close();
                }
            } catch (SQLException e) {
                System.out.println("Cleanup failed to close Connection.");
            }

        }
    }
}
```

CallableStatement:

JDBC CallableStatement インターフェースは PreparedStatement を拡張し、パラメーターの出力および入出力のサポートを提供します。 CallableStatement インターフェースは、PreparedStatement インターフェースによって提供される入力パラメーターもサポートします。

CallableStatement インターフェースは、SQL ステートメントを使ってストアード・プロシージャーを呼び出せるようにします。ストアード・プロシージャーは、データベース・インターフェースを持ったプログラムです。このプログラムは、次のような機能を持っています。

- 入力および出力パラメーター、または入出力両方のパラメーターを持つことができます。
- 戻り値を持つことができます。
- 複数の ResultSet を戻すことができます。

JDBC では概念的に、ストアード・プロシージャー呼び出しはデータベースに対する単一の呼び出しですが、ストアード・プロシージャーに関連したプログラムは多数のデータベース要求を処理することがあります。さらにストアード・プロシージャー・プログラムは、通常は SQL ステートメントでは実行されないような、プログラマチックな他の多くのタスクを実行するのに使用されることがあります。

CallableStatements は、準備および処理フェーズを分離した PreparedStatement モデルに従っているため、再利用するために最適化することが可能です (詳しくは、94 ページの『PreparedStatement』 を参照してください)。ストアード・プロシージャーの SQL ステートメントがプログラムに結合されると、それらのステートメントは静的 SQL として処理され、さらにパフォーマンスの向上も期待できます。多くのデータベース処理を単一で、再利用可能なデータベース呼び出しにカプセル化することは、ストアード・プロシージャーの良い使用例です。この呼び出しは他のシステムまでネットワーク上を通過しますが、多くの作業の要求はリモート・システム上で完了します。

CallableStatements を作成する

新規 CallableStatement オブジェクトを作成するには、prepareCall メソッドを使用します。 CallableStatement オブジェクトが作成されるときには、prepareStatement メソッドの場合と同様、SQL ステートメントが提供される必要があります。 SQL ステートメントのプリコンパイルはその時点で行われます。たとえば、conn という名前の Connection オブジェクトが既に存在していると想定した場合に、 CallableStatement オブジェクトを作成し、SQL ステートメントを取得する準備フェーズを完了して、データベース内で処理可能な状態にするには、次のようにします。

```
PreparedStatement ps = conn.prepareStatement("? = CALL ADDEMPLOYEE(?, ?, ?");
```

ADDEMPLOYEE ストアード・プロシージャーは入力パラメーターとして、新しい従業員の名前と社会保障番号、およびその従業員のマネージャーのユーザー ID を受け取ります。この情報によって、会社のデータベースの複数のテーブルがその従業員の勤務開始日、部署などの情報によって更新されます。さらに、ストアード・プロシージャーは、その従業員の標準ユーザー ID と E メール・アドレスを生成することもできるプログラムです。ストアード・プロシージャーが初期ユーザー名とパスワード付きの E メールを雇用管理者あてに送信することもできます。その後、雇用管理者はその従業員に対して、それらの情報を知らせることができます。

ADDEMPLOYEE ストアード・プロシージャーは戻り値を持つようにセットアップされます。呼び出し側プログラムが、障害が発生した際に利用できるよう、戻りコードとして成功または失敗コードを戻すことができます。戻り値は新しい従業員の会社 ID 番号として定義することもできます。最後に、ストアード・プロシージャー・プログラムは内部で処理される照会を持つことがあり、それらの照会によって開かれた ResultSet を呼び出し側プログラムが利用できるようにしておきます。すべての新しい従業員の情報を照会し、戻された ResultSet を通して呼び出し側プログラムがそれらを利用できるようにするのは妥当なことです。

これらのタスクのそれぞれのタイプを完了する方法については、以下のセクションで扱われています。

ResultSet 特性の指定および自動生成キー・サポート

CreateStatement および PrepareStatement では、ResultSet 特性を指定するためのサポートを提供する PrepareCall の複数のバージョンがあります。 PrepareStatement とは異なり、PrepareCall メソッドは CallableStatement からの自動生成キーを処理するバリエーションは提供されていません (JDBC 3.0 では、この概念はサポートされていません)。以下に、PrepareCall メソッドの正しい呼び出し方法のいくつかの例を示します。

例: PrepareCall メソッド

```
// The following is new in JDBC 2.0

CallableStatement cs2 = conn.prepareCall("? = CALL ADDEMPLOYEE(?, ?, ?)",
    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATEABLE);

// New in JDBC 3.0

CallableStatement cs3 = conn.prepareCall("? = CALL ADDEMPLOYEE(?, ?, ?)",
    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATEABLE,
    ResultSet.HOLD_CURSOR_OVER_COMMIT);
```

パラメーターのハンドリング

前述のとおり、CallableStatement オブジェクトは 3 タイプのパラメーターを取ることができます。

- **IN**

  IN パラメーターは PreparedStatement と同じ方法でハンドルされます。 PreparedStatement クラスから継承されたさまざまな set メソッドを使って、パラメーターを設定することができます。

- **OUT**

  OUT パラメーターは、registerOutParameter メソッドによってハンドルされます。最も一般的な形式の registerOutParameter では、最初のパラメーターとして索引パラメーターが、2 番目のパラメーターとして SQL タイプが取られます。これにより、JDBC ドライバーは、ステートメントが処理されるときにパラメーターがどのようなデータであるかが分かります。 registerOutParameter メソッドの他の 2 つのバリエーションは、java.sql パッケージ Javadoc で見つけることができます。

- **INOUT**

  INOUT パラメーターは IN パラメーターおよび OUT パラメーターの両方を使って処理が行われる場合に必要です。 INOUT パラメーターごとに、ステートメントが処理される前に set メソッドおよび registerOutParameter メソッドを呼び出す必要があります。なんらかのパラメーターの設定または登録に失敗すると、ステートメントが処理されるときに SQLException がスローされます。

詳しくは、 106 ページの『例: 入出力パラメーターを持つプロシージャーを作成する』を参照してください。

PreparedStatement と同様に、CallableStatement パラメーター値は set メソッドを再び呼び出さなくても、処理の間は値が保持されます。出力として登録されたパラメーターには、clearParameters メソッドの効果はありません。 clearParameters を呼び出した後、すべての IN パラメーターは再び値を設定する必要がありますが、すべての OUT パラメーターは再登録の必要がありません。

注: このパラメーターの概念を、パラメーター・マーカーの概念と混同しないでください。ストアード・プロシージャー呼び出しは、信頼できるパラメーター数が渡されることを要求します。 SQL ステートメ

ントの中には、実行時に指定される値を表す文字 "?" (パラメーター・マーカー) が含まれるものがあります。次の例を考慮して、これら 2 つの概念の違いを確認するようにしてください。

```
CallableStatement cs = con.prepareCall("CALL PROC(?, "SECOND", ?)");

cs.setString(1, "First");     //Parameter marker 1, Stored procedure parm 1

cs.setString(2, "Third");     //Parameter marker 2, Stored procedure parm 3
```

ストアード・プロシージャー・パラメーターに名前でアクセスする

ストアード・プロシージャーのパラメーターは、次のストアード・プロシージャー宣言の例のように、関連付けられた名前を持っています。

例: ストアード・プロシージャー・パラメーター

```
CREATE
PROCEDURE MYLIBRARY.APROC
    (IN PARM1 INTEGER)
LANGUAGE SQL SPECIFIC MYLIBRARY.APROC
BODY: BEGIN
    <Perform a task here...>
END BODY
```

ここでは、PARM1 という名前が付けられた 1 つの整数パラメーターがあります。 JDBC 3.0 では、索引だけでなく、名前によるストアード・プロシージャー・パラメーターの指定がサポートされています。このプロシージャーの CallableStatement を設定するコードは、次のようになります。

```
CallableStatement cs = con.prepareCall("CALL APROC(?)");

cs.setString("PARM1", 6);     //Sets input parameter at index 1 (PARM1) to 6.
```

***CallableStatements* を処理する:**

JDBC CallableStatement オブジェクトでの SQL ストアード・プロシージャー呼び出しの処理は、PreparedStatement オブジェクトで使用されるものと同じメソッドによって行われます。

ストアード・プロシージャーの結果を戻す

ストアード・プロシージャー内で 1 つの SQL 照会ステートメントが処理されると、そのストアード・プロシージャーを呼び出したプログラムが、照会された結果を利用できます。ストアード・プロシージャー内で複数の照会を呼び出し、呼び出し側プログラムがすべての使用可能な ResultSet を処理することもできます。

詳しくは、 104 ページの『例: 複数の ResultSet を持つプロシージャーを作成する』を参照してください。

注: ストアード・プロシージャーが executeQuery で処理され、ResultSet が戻されない場合は、SQLException がスローされます。

ResultSet に並行してアクセスする

『ストアード・プロシージャーの結果を戻す』では、ResultSet およびストアード・プロシージャーについてを扱っており、すべての Java Development Kit (JDK) リリースで動作する例が提供されています。この例では、ResultSet はストアード・プロシージャーが最初に開いた ResultSet から、最後に開いた ResultSet までが順番に処理されます。次の ResultSet を使う前に、ResultSet はクローズされます。

JDK 1.4 および後続のバージョンでは、ストアード・プロシージャーから ResultSet を並行して処理する機能がサポートされています。

注: この機能は、コマンド入力行インターフェース (CLI) の V5R2 による、基礎となるシステム・サポートに追加されたものです。結果として、JDK 1.4 または後続のバージョンを V5R2 より前のバージョンのシステムで動作させた場合は、このサポートを利用できません。

ストアード・プロシージャーの更新数を戻す

ストアード・プロシージャーから更新数を戻す機能については JDBC 仕様で扱われていますが、現在のところ、IBM i プラットフォームではサポートされていません。ストアード・プロシージャー呼び出しから複数の更新数を戻す方法はありません。ストアード・プロシージャー内の準備済み SQL ステートメントからの更新数が必要な場合は、この値を戻すための 2 つの方法があります。

- 出力パラメーターとして値を戻す。
- パラメーターからの戻り値として値を戻す。これは出力パラメーターの特殊なケースです。詳しくは、『戻り値を持つストアード・プロシージャーを処理する』を参照してください。

戻り値が不明なストアード・プロシージャーを処理する

ストアード・プロシージャー呼び出しの結果が不明な場合は、実行メソッドを使用します。このメソッドが一度処理されると、JDBC ドライバーはアプリケーションに、API 呼び出しを通してストアード・プロシージャーが生成する結果のタイプを通知することができます。実行メソッドは、結果が 1 つか複数の ResultSet であった場合、True を戻します。ストアード・プロシージャー呼び出しから更新数は返されません。

戻り値を持つストアード・プロシージャーを処理する

IBM i プラットフォームは、関数の戻り値に似た、戻り値を持つストアード・プロシージャーをサポートしています。ストアード・プロシージャーからの戻り値は他のパラメーター・マークのようにラベル付けされており、ストアード・プロシージャー呼び出しによって割り当てられます。以下に、その例を示します。

```
? = CALL MYPROC(?, ?, ?)
```

ストアード・プロシージャー呼び出しからの戻り値は常に整数タイプで、他の出力パラメーターのように登録されている必要があります。

詳しくは、 107 ページの『例: 戻り値を持つプロシージャーを作成する』を参照してください。

例: 複数の ResultSet を持つプロシージャーを作成する:

この例では、JDBC を使用してデータベースにアクセスし、複数の ResultSet を持つプロシージャーを作成する方法を示しています。

注: 法律上の重要な情報に関しては、コードの特記事項情報をお読みください。

```
import java.sql.*;
import java.util.Properties;

public class CallableStatementExample1 {

    public static void main(java.lang.String[] args) {

        // Register the Native JDBC driver.  If we cannot
        // register the driver, the test cannot continue.
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");

            // Create the connection properties
            Properties properties = new Properties ();
            properties.put ("user", "userid");
```

```
properties.put ("password", "password");

// Connect to the local server database
Connection c = DriverManager.getConnection("jdbc:db2://*local", properties);

Statement s = c.createStatement();

// Create a procedure with multiple ResultSets.
String sql = "CREATE PROCEDURE MYLIBRARY.SQLSPEX1 " +
             "RESULT SET 2 LANGUAGE SQL READS SQL DATA SPECIFIC MYLIBRARY.SQLSPEX1 " +
             "EX1: BEGIN " +
             "   DECLARE C1 CURSOR FOR SELECT * FROM QSYS2.SYSPROCS " +
             "               WHERE SPECIFIC_SCHEMA = 'MYLIBRARY';  " +
             "   DECLARE C2 CURSOR FOR SELECT * FROM QSYS2.SYSPARMS  " +
             "               WHERE SPECIFIC_SCHEMA = 'MYLIBRARY';  " +
             "   OPEN C1; " +
             "   OPEN C2; " +
             "   SET RESULT SETS CURSOR C1, CURSOR C2; " +
             "END EX1 ";

try {
    s.executeUpdate(sql);
    } catch (SQLException e) {
    // NOTE:  We are ignoring the error here.  We are making
    //        the assumption that the only reason this fails
    //        is because the procedure already exists. Other
    //        reasons that it could fail are because the C compiler
    //        is not found to compile the procedure or because
    //        collection MYLIBRARY does not exist on the system.
}
s.close();

// Now use JDBC to run the procedure and get the results back. In
// this case we are going to get information about 'MYLIBRARY's stored
// procedures (which is also where we created this procedure, thereby
// ensuring that there is something to get.
CallableStatement cs = c.prepareCall("CALL MYLIBRARY.SQLSPEX1");

ResultSet rs = cs.executeQuery();

// We now have the first ResultSet object that the stored procedure
// left open.  Use it.
int i = 1;
while (rs.next()) {
    System.out.println("MYLIBRARY stored procedure
                       " + i + " is " + rs.getString(1) + "." +
                       rs.getString(2));
    i++;
}
System.out.println("");

// Now get the next ResultSet object from the system - the previous
// one is automatically closed.
if (!cs.getMoreResults()) {
    System.out.println("Something went wrong.  There should have
                        been another ResultSet, exiting.");
    System.exit(0);
}
rs = cs.getResultSet();

 // We now have the second ResultSet object that the stored procedure
// left open.  Use that one.
i = 1;
while (rs.next()) {
    System.out.println("MYLIBRARY procedure " + rs.getString(1)
                        + "." + rs.getString(2) +
                       " parameter:  " + rs.getInt(3) + " direction:
```

```
                                " + rs.getString(4) +
                                " data type: " + rs.getString(5));
                i++;
            }

            if (i == 1) {
                System.out.println("None of the stored procedures have any parameters.");
            }

            if (cs.getMoreResults()) {
                System.out.println("Something went wrong,
                                    there should not be another ResultSet.");
                System.exit(0);
            }

            cs.close();  // close the CallableStatement object
            c.close();   // close the Connection object.

        } catch (Exception e) {
            System.out.println("Something failed..");
            System.out.println("Reason: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

例: 入出力パラメーターを持つプロシージャーを作成する:

この例では、JDBC を使用してデータベースにアクセスし、入出力パラメーターを持つプロシージャーを作成する方法を示しています。

注: 法律上の重要な情報に関しては、コードの特記事項情報をお読みください。

```
import java.sql.*;
import java.util.Properties;

public class CallableStatementExample2 {

    public static void main(java.lang.String[] args) {

        // Register the Native JDBC driver.  If we cannot
        // register the driver, the test cannot continue.
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");

            // Create the connection properties
            Properties properties = new Properties ();
            properties.put ("user", "userid");
            properties.put ("password", "password");

            // Connect to the local server database
            Connection c = DriverManager.getConnection("jdbc:db2://*local", properties);

            Statement s = c.createStatement();

            // Create a procedure with in, out, and in/out parameters.
            String sql = "CREATE PROCEDURE MYLIBRARY.SQLSPEX2 " +
                         "(IN P1 INTEGER, OUT P2 INTEGER, INOUT P3 INTEGER) " +
                         "LANGUAGE SQL SPECIFIC MYLIBRARY.SQLSPEX2 " +
                         "EX2: BEGIN " +
                         "   SET P2 = P1 + 1; " +
                         "   SET P3 = P3 + 1; " +
                         "END EX2 ";

            try {
                s.executeUpdate(sql);
```

```
        } catch (SQLException e) {
            // NOTE:  We are ignoring the error here.  We are making
            //        the assumption that the only reason this fails
            //        is because the procedure already exists.  Other
            //        reasons that it could fail are because the C compiler
            //        is not found to compile the procedure or because
            //        collection MYLIBRARY does not exist on the system.
        }
        s.close();

        // Prepare a callable statement used to run the procedure.
        CallableStatement cs = c.prepareCall("CALL MYLIBRARY.SQLSPEX2(?, ?, ?)");

        // All input parameters must be set and all output parameters must
        // be registered.  Notice that this means we have two calls to make
        // for an input output parameter.
        cs.setInt(1, 5);
        cs.setInt(3, 10);
        cs.registerOutParameter(2, Types.INTEGER);
        cs.registerOutParameter(3, Types.INTEGER);

        // Run the procedure
        cs.executeUpdate();

        // Verify the output parameters have the desired values.
        System.out.println("The value of P2 should be P1 (5) + 1 = 6.  -->  " + cs.getInt(2));
        System.out.println("The value of P3 should be P3 (10) + 1 = 11.  -->  " + cs.getInt(3));

        cs.close();  // close the CallableStatement object
        c.close();   // close the Connection object.

    } catch (Exception e) {
        System.out.println("Something failed..");
        System.out.println("Reason: " + e.getMessage());
        e.printStackTrace();
    }
  }
}
```

例: 戻り値を持つプロシージャーを作成する:

この例では、JDBC を使用してデータベースにアクセスし、戻り値を持つプロシージャーを作成する方法を示しています。

注: 法律上の重要な情報に関しては、コードの特記事項情報をお読みください。

```
import java.sql.*;
import java.util.Properties;

public class CallableStatementExample3 {

    public static void main(java.lang.String[] args) {

        // Register the native JDBC driver. If the driver cannot
        // be registered, the test cannot continue.
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");

            // Create the connection properties
            Properties properties = new Properties ();
            properties.put ("user", "userid");
            properties.put ("password", "password");

            // Connect to the local server database
            Connection c = DriverManager.getConnection("jdbc:db2://*local", properties);
```

```
            Statement s = c.createStatement();

            // Create a procedure with a return value.
            String sql = "CREATE PROCEDURE MYLIBRARY.SQLSPEX3 " +
                         " LANGUAGE SQL SPECIFIC MYLIBRARY.SQLSPEX3 " +
                         " EX3: BEGIN " +
                         "      RETURN 1976; " +
                         " END EX3 ";

            try {
                s.executeUpdate(sql);
            } catch (SQLException e) {
                // NOTE:  The error is ignored here. The assumptions is
                //        made that the only reason this fails is
                //        because the procedure already exists. Other
                //        reasons that it could fail are because the C compiler
                //        is not found to compile the procedure or because
                //        collection MYLIBRARY does not exist on the system.
            }
            s.close();

            // Prepare a callable statement used to run the procedure.
            CallableStatement cs = c.prepareCall("? = CALL MYLIBRARY.SQLSPEX3");

            // You still need to register the output parameter.
            cs.registerOutParameter(1, Types.INTEGER);

            // Run the procedure.
            cs.executeUpdate();

            // Show that the correct value is returned.
            System.out.println("The return value
                                should always be 1976 for this example:
                                -->  " + cs.getInt(1));

            cs.close();  // close the CallableStatement object
            c.close();   // close the Connection object.

        } catch (Exception e) {
            System.out.println("Something failed..");
            System.out.println("Reason: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

ResultSet

ResultSet インターフェースは、照会の実行によって生成された結果へのアクセスを提供します。概念上、ResultSet のデータは、特定数の列と特定数の行を含むテーブルとして考えることができます。デフォルトでは、テーブル行は順番に検索されます。検索の対象が 1 行であれば、列の値は、任意の順序でアクセスできます。

ResultSet の特性:

このトピックでは、ResultSet の特性を説明しています。これには ResultSet のタイプ、並行性、接続オブジェクトをコミットすることで ResultSet をクローズする機能、および ResultSet 特性の仕様などがあります。

デフォルトで、作成されるすべての ResultSet は、「順方向のみ」のタイプと「読み取り専用」の並行性を持ち、カーソルはコミット境界を超えて保持されます。例外として、WebSphere では現在カーソルの保持

可能性のデフォルトが変更されていて、カーソルはコミット時に暗黙的にクローズするようになっています。これらの特性は、Statement、PreparedStatement、および CallableStatement オブジェクトでアクセス可能なメソッドを通して構成できます。

ResultSet のタイプ

ResultSet タイプは、ResultSet に関して以下の事柄を指定します。

- ResultSet はスクロール可能かどうか。
- ResultSet インターフェースの定数によって定義されている JDBC (Java Database Connectivity) ResultSet のタイプ。

これらの ResultSet タイプの定義は以下のとおりです。

TYPE_FORWARD_ONLY
ResultSet の先頭から ResultSet の末尾に向かう処理だけが行えるカーソル。これはデフォルトのタイプです。

TYPE_SCROLL_INSENSITIVE
ResultSet の中をスクロールすることができるカーソル。このタイプのカーソルは、オープンしているときにデータベースに加えられる変更を感知しません。これには、照会が処理されたときやデータが取り出されるときに、照会の条件を満たす行が含まれます。

TYPE_SCROLL_SENSITIVE
ResultSet の中を各方向にスクロールすることができるカーソル。このタイプのカーソルは、オープンしているときにデータベースに加えられる変更を感知します。データベースへの変更は、ResultSet データに直接影響します。

JDBC 1.0 ResultSet は常に「順方向のみ」です。スクロール可能なカーソルは JDBC 2.0 で追加されました。

注: blocking enabled および block size 接続プロパティーは、TYPE_SCROLL_SENSITIVE カーソルの感度に影響します。ブロック化を行うと、データが JDBC ドライバー層そのものにキャッシングされ、パフォーマンスが向上します。

並行性

並行性は、ResultSet を更新できるかどうかを決定します。このタイプも、ResultSet インターフェースの定数によって定義されています。使用可能な並行性の設定値は以下のとおりです。

CONCUR_READ_ONLY
データベースからのデータの読み取りにだけ使用される ResultSet。これはデフォルトの設定です。

CONCUR_UPDATEABLE
変更を加えることのできる ResultSet。これらの変更は、基になるデータベースに加えることができます。

JDBC 1.0 ResultSet は常に「順方向のみ」です。更新可能な ResultSet は JDBC 2.0 で追加されました。

注: JDBC 仕様によれば、JDBC ドライバーは、値を一緒に使用することができない場合、ResultSet の並行性設定の ResultSet タイプを変更することができます。その場合、JDBC ドライバーは Connection オブジェクトに対する警告を出します。

アプリケーションが TYPE_SCROLL_INSENSITIVE, CONCUR_UPDATEABLE ResultSet を指定するある状況があります。データのコピーを作成することによって、Insensitivity (不感知) がデータベース・エンジン

にインプリメントされています。このとき、このコピーを通して基になるデータベースを更新することはできません。この組み合わせが指定されている場合は、ドライバーが感度を TYPE_SCROLL_SENSITIVE に変更し、要求が変更されたことを示す警告を作成します。

保持可能性

保持可能性の特性は、Connection オブジェクトでコミットを呼び出すと ResultSet がクローズされるかどうかを決定します。保持可能性特性の処理のための JDBC API は、バージョン 3.0 での新機能です。ただし、ネイティブ JDBC ドライバーは、いくつかのリリースで接続プロパティーを提供しており、この接続プロパティーでは、接続の下で作成されたすべての ResultSet にそのデフォルトを指定することができます。 API サポートは、接続プロパティーの設定をオーバーライドします。保持可能性特性の値は、ResultSet 定数によって定義されており、それは以下のとおりです。

HOLD_CURSOR_OVER_COMMIT
: オープンしているカーソルはすべて、commit 文節が呼び出されてもオープンしたままです。これはネイティブ JDBC のデフォルト値です。

CLOSE_CURSORS_ON_COMMIT
: オープンしているカーソルは、commit 文節が呼び出されるとクローズされます。

注: いつでも接続でロールバックが呼び出されると、オープンしているカーソルはすべてクローズされます。この事実はあまりよく知られていませんが、データベースがカーソルを扱うときの共通の方法です。

JDBC 仕様によれば、カーソルの保持可能性のデフォルトは、インプリメンテーション定義です。一部のプラットフォームは CLOSE_CURSORS_ON_COMMIT をデフォルトとして使用します。大半のアプリケーションでは普通これは問題になりませんが、コミット境界を超えてカーソルを操作する場合は、使用しているドライバーの動作に注意する必要があります。 IBM Toolbox for Java JDBC ドライバーは、HOLD_CURSORS_ON_COMMIT のデフォルトも使用しますが、UDB for Windows NT® 用の JDBC ドライバーのデフォルトは、CLOSE_CURSORS_ON_COMMIT です。

ResultSet 特性の指定

ResultSet の特性は、その ResultSet オブジェクトがいったん作成されてしまうと、変化しません。したがって、特性はオブジェクトを作成する前に指定する必要があります。これらの特性は、多重定義されている createStatement、prepareStatement、および prepareCall メソッドのバリエーションを通して指定できます。

注: ResultSet タイプと ResultSet の並行性を取得するための ResultSet メソッドはありますが、ResultSet の保持可能性を取得するためのメソッドはありません。

関連概念

92 ページの『Statement オブジェクト』
Statement オブジェクトは、静的 SQL ステートメントの処理と、それによって生成される結果の取得に使用されます。 一度にオープンできるのは、各 Statement オブジェクトにつき 1 つの ResultSet だけです。 SQL ステートメントを処理するすべてのステートメント・メソッドは、すでにオープンされている ResultSet があると、暗黙的にステートメントの現行の ResultSet をクローズします。

100 ページの『CallableStatement』
JDBC CallableStatement インターフェースは PreparedStatement を拡張し、パラメーターの出力および入出力のサポートを提供します。 CallableStatement インターフェースは、PreparedStatement インターフェースによって提供される入力パラメーターもサポートします。

94 ページの『PreparedStatement』
PreparedStatement は Statement インターフェースを拡張し、SQL ステートメントへのパラメーターの追加をサポートします。

116 ページの『カーソル移動』
IBM i Java Database Connectivity (JDBC) ドライバーは、スクロール可能な ResultSet をサポートします。 スクロール可能な ResultSet では、いくつかのカーソル配置メソッドを使用して、データの行をどんな順序でも処理できます。

関連タスク

119 ページの『ResultSets の変更』
IBM i JDBC ドライバーでは、以下のタスクを実行することによって、ResultSet を変更できます。

関連資料

42 ページの『JDBC ドライバーの接続プロパティー』
以下の表は、JDBC ドライバーの接続プロパティーとその値、およびその説明を示しています。

54 ページの『DataSource プロパティー』
各 JDBC ドライバー接続プロパティーには、対応するデータ・ソース・メソッドがあります。この表では、有効なデータ・ソース・プロパティーを示します。

例: 感知および非感知の ResultSet:

以下の例は、テーブルに行が挿入される際の、感知 ResultSet と非感知 ResultSet との違いを示しています。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;

public class Sensitive {

    public Connection connection = null;

    public static void main(java.lang.String[] args) {
        Sensitive test = new Sensitive();

        test.setup();
        test.run("sensitive");
        test.cleanup();

        test.setup();
        test.run("insensitive");
        test.cleanup();
    }
```

```
public void setup() {

    try {
        Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
        connection = DriverManager.getConnection("jdbc:db2:*local");

        Statement s = connection.createStatement();
        try {
            s.executeUpdate("drop table cujosql.sensitive");
        } catch (SQLException e) {
            // Ignored.
        }

        s.executeUpdate("create table cujosql.sensitive(col1 int)");
        s.executeUpdate("insert into cujosql.sensitive values(1)");
        s.executeUpdate("insert into cujosql.sensitive values(2)");
        s.executeUpdate("insert into cujosql.sensitive values(3)");
        s.executeUpdate("insert into cujosql.sensitive values(4)");
        s.executeUpdate("insert into cujosql.sensitive values(5)");
        s.close();

    } catch (Exception e) {
        System.out.println("Caught exception: " + e.getMessage());
        if (e instanceof SQLException) {
            SQLException another = ((SQLException) e).getNextException();
            System.out.println("Another:  " + another.getMessage());
        }
    }
}


public void run(String sensitivity) {
    try {
        Statement s = null;
        if (sensitivity.equalsIgnoreCase("insensitive")) {
            System.out.println("creating a TYPE_SCROLL_INSENSITIVE cursor");
            s = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                ResultSet.CONCUR_READ_ONLY);
        } else {
            System.out.println("creating a TYPE_SCROLL_SENSITIVE cursor");
            s = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                ResultSet.CONCUR_READ_ONLY);
        }

        ResultSet rs = s.executeQuery("select * From cujosql.sensitive");

        // Fetch the five values that are there.
        rs.next();
        System.out.println("value is " + rs.getInt(1));
        rs.next();
        System.out.println("value is " + rs.getInt(1));
        rs.next();
        System.out.println("value is " + rs.getInt(1));
        rs.next();
        System.out.println("value is " + rs.getInt(1));
        rs.next();
        System.out.println("value is " + rs.getInt(1));
        System.out.println("fetched the five rows...");

        // Note:  If you fetch the last row, the ResultSet looks
        //        closed and subsequent new rows that are added
```

```
            //        are not be recognized.

            // Allow another statement to insert a new value.
            Statement s2 = connection.createStatement();
            s2.executeUpdate("insert into cujosql.sensitive values(6)");
            s2.close();

            // Whether a row is recognized is based on the sensitivity setting.
            if (rs.next()) {
                System.out.println("There is a row now: " + rs.getInt(1));
            } else {
                System.out.println("No more rows.");
            }


        } catch (SQLException e) {
            System.out.println("SQLException exception: ");
            System.out.println("Message:....." + e.getMessage());
            System.out.println("SQLState:...." + e.getSQLState());
            System.out.println("Vendor Code:." + e.getErrorCode());
            System.out.println("-------------------------------------");
            e.printStackTrace();
        }
        catch (Exception ex) {
            System.out.println("An exception other than an SQLException was thrown: ");
            ex.printStackTrace();
        }
    }



    public void cleanup() {
        try {
            connection.close();
        } catch (Exception e) {
            System.out.println("Caught exception: ");
            e.printStackTrace();
        }
    }
}
```

例: ResultSet の感度:

以下の例は、ResultSet の感度に基づいた、変更が SQL ステートメントの where 文節に与える影響を示しています。

この例にはフォーマット設定の正しくない箇所があるかもしれません。これは、この例を印刷ページに収めるためです。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;

public class Sensitive2 {

    public Connection connection = null;

    public static void main(java.lang.String[] args) {
        Sensitive2 test = new Sensitive2();
```

```
        test.setup();
        test.run("sensitive");
        test.cleanup();

        test.setup();
        test.run("insensitive");
        test.cleanup();
    }


    public void setup() {

        try {
            System.out.println("Native JDBC used");
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            connection = DriverManager.getConnection("jdbc:db2:*local");

            Statement s = connection.createStatement();
            try {
                s.executeUpdate("drop table cujosql.sensitive");
            } catch (SQLException e) {
                // Ignored.
            }

            s.executeUpdate("create table cujosql.sensitive(col1 int)");
            s.executeUpdate("insert into cujosql.sensitive values(1)");
            s.executeUpdate("insert into cujosql.sensitive values(2)");
            s.executeUpdate("insert into cujosql.sensitive values(3)");
            s.executeUpdate("insert into cujosql.sensitive values(4)");
            s.executeUpdate("insert into cujosql.sensitive values(5)");


            try {
                s.executeUpdate("drop table cujosql.sensitive2");
            } catch (SQLException e) {
                // Ignored.
            }

            s.executeUpdate("create table cujosql.sensitive2(col2 int)");
            s.executeUpdate("insert into cujosql.sensitive2 values(1)");
            s.executeUpdate("insert into cujosql.sensitive2 values(2)");
            s.executeUpdate("insert into cujosql.sensitive2 values(3)");
            s.executeUpdate("insert into cujosql.sensitive2 values(4)");
            s.executeUpdate("insert into cujosql.sensitive2 values(5)");

            s.close();

        } catch (Exception e) {
            System.out.println("Caught exception: " + e.getMessage());
            if (e instanceof SQLException) {
                SQLException another = ((SQLException) e).getNextException();
                System.out.println("Another:  " + another.getMessage());
            }
        }
    }


    public void run(String sensitivity) {
        try {

            Statement s = null;
            if (sensitivity.equalsIgnoreCase("insensitive")) {
                System.out.println("creating a TYPE_SCROLL_INSENSITIVE cursor");
```

```
            s = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                ResultSet.CONCUR_READ_ONLY);
        } else {
            System.out.println("creating a TYPE_SCROLL_SENSITIVE cursor");
            s = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                ResultSet.CONCUR_READ_ONLY);
        }

        ResultSet rs = s.executeQuery("select col1, col2 From cujosql.sensitive,
                                   cujosql.sensitive2 where col1 = col2");

        rs.next();
        System.out.println("value is " + rs.getInt(1));
        rs.next();
        System.out.println("value is " + rs.getInt(1));
        rs.next();
        System.out.println("value is " + rs.getInt(1));
        rs.next();
        System.out.println("value is " + rs.getInt(1));

        System.out.println("fetched the four rows...");

        // Another statement creates a value that does not fit the where clause.
        Statement s2 =
          connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
          ResultSet.CONCUR_UPDATEABLE);
        ResultSet rs2 = s2.executeQuery("select *
        from cujosql.sensitive where col1 = 5 FOR UPDATE");
        rs2.next();
        rs2.updateInt(1, -1);
        rs2.updateRow();
        s2.close();

        if (rs.next()) {
            System.out.println("There is still a row: " + rs.getInt(1));
        } else {
            System.out.println("No more rows.");
        }

    } catch (SQLException e) {
        System.out.println("SQLException exception: ");
        System.out.println("Message:....." + e.getMessage());
        System.out.println("SQLState:...." + e.getSQLState());
        System.out.println("Vendor Code:." + e.getErrorCode());
        System.out.println("----------------------------");
        e.printStackTrace();
    }
    catch (Exception ex) {
        System.out.println("An exception other
        than an SQLException was thrown: ");
        ex.printStackTrace();
    }
}


public void cleanup() {
    try {
        connection.close();
    } catch (Exception e) {
        System.out.println("Caught exception: ");
```

```
            e.printStackTrace();
        }
    }
}
```

カーソル移動:

IBM i Java Database Connectivity (JDBC) ドライバーは、スクロール可能な ResultSet をサポートします。スクロール可能な ResultSet では、いくつかのカーソル配置メソッドを使用して、データの行をどんな順序でも処理できます。

ResultSet.next メソッドを使用すると、ResultSet 内を 1 行ずつ移動します。 Java Database Connectivity (JDBC) 2.0 では、IBM i JDBC ドライバーはスクロール可能な ResultSet をサポートします。スクロール可能な ResultSet では、previous、absolute、relative、first、および last メソッドを使用することにより、データの行をどのような順序でも処理できます。

デフォルトでは、JDBC ResultSets は常に「順方向のみ」です。これは、唯一の有効なカーソル配置メソッドが next() であることを意味します。スクロール可能な ResultSet は明示的に要求する必要があります。詳しくは、ResultSet のタイプを参照してください。

スクロール可能な ResultSet では、以下のカーソル配置メソッドが使用できます。

メソッド	説明
Next	このメソッドは、ResultSet 内でカーソルを順方向に 1 行移動します。 このメソッドは、カーソルが有効な行に配置されれば true を、そうでなければ false を戻します。
Previous	このメソッドは、ResultSet 内でカーソルを逆方向に 1 行移動します。 このメソッドは、カーソルが有効な行に配置されれば true を、そうでなければ false を戻します。
First	このメソッドは、カーソルを ResultSet 内の最初の行に移動します。 このメソッドは、カーソルが最初の行に配置されれば true を、ResultSet が空なら false を戻します。
Last	このメソッドは、カーソルを ResultSet 内の最後の行に移動します。 このメソッドは、カーソルが最後の行に配置されれば true を、ResultSet が空なら false を戻します。
BeforeFirst	このメソッドは、カーソルを ResultSet 内の最初の行のすぐ前に移動します。 空の ResultSet には、このメソッドは無効です。このメソッドからの戻り値はありません。
AfterLast	このメソッドは、カーソルを ResultSet 内の最後の行のすぐ後ろに移動します。 空の ResultSet には、このメソッドは無効です。このメソッドからの戻り値はありません。

メソッド	説明
Relative (int rows)	このメソッドは、カーソルをその現行位置に相対する位置へ移動します。 • 行が 0 の場合、このメソッドは無効です。 • 行が正の場合、カーソルは順方向にその行数だけ移動されます。現行行から ResultSet の末尾までの行数が、入力パラメーターで指定された行数より少ない場合、このメソッドは afterLast と同様に操作します。 • 行が負の場合、カーソルは逆方向にその行数だけ移動されます。現行行から ResultSet の先頭までの行数が、入力パラメーターで指定された行数より少ない場合、このメソッドは beforeFirst と同様に操作します。 このメソッドは、カーソルが有効な行に配置されれば true を、そうでなければ false を戻します。
Absolute (int row)	このメソッドは、カーソルを行値で指定された行に移動します。 行値が正の場合、カーソルは ResultSet の先頭から数えてその行数番目に配置されます。最初の行の番号は 1、2 番目は 2 です。 ResultSet 内の行数が行値で指定された行数より少ない場合、このメソッドは afterLast と同じ方法で操作します。 行値が負の場合、カーソルは ResultSet の終了から数えてその行数番目に配置されます。最終行の番号は -1、最後から 2 番目は -2 です。 ResultSet 内の行数が行値で指定された行数より少ない場合、このメソッドは beforeFirst と同じ方法で操作します。 行値が 0 の場合、このメソッドは beforeFirst と同じ方法で操作します。 このメソッドは、カーソルが有効な行に配置されれば true を、そうでなければ false を戻します。

ResultSet データの取得:

ResultSet オブジェクトは、行の列データを取得するためのいくつかのメソッドが備わっています。どのメソッドも、get<*Type*> (<*Type*> は Java データ・タイプ) という形式です。これらのメソッドには、たとえば、getInt、getLong、getString、getTimestamp、getBlob などがあります。これらメソッドのほとんどすべては単一のパラメーターをとり、そのパラメーターは ResultSet 内の列索引か、列名のいずれかです。

ResultSet の列には番号が付けられます。番号は 1 から始まります。列名が使用され、ResultSet 内に同じ名前を持つ列が複数ある場合は、最初のものが戻されます。複数のパラメーターをとる get<Type> メソッドもあります。オプションの Calendar オブジェクトはその一例です。このオブジェクトは getTime、getDate、および getTimestamp に渡すことができます。詳しくは、java.sql パッケージの Javadoc を参照してください。

オブジェクトを戻す get メソッドの場合、ResultSet の列がヌルのときは、戻り値はヌルになります。プリミティブ・タイプの場合は、ヌルは戻せません。その場合、値は 0 か false です。アプリケーションがヌルと 0 または false を区別する必要がある場合は、呼び出しの直後に wasNull メソッドを使用します。このメソッドは、値が実際の 0 または false 値かどうか、あるいは、その値は ResultSet 値が本当にヌルであったために戻されたのかどうかを判別します。

ResultSetMetaData サポート

getMetaData メソッドは、ResultSet オブジェクトに対して呼び出されると、ResultSet オブジェクトの列を記述する ResultSetMetaData オブジェクトを戻します。実行されている SQL ステートメントが実行時まで

認識されない場合、ResultSetMetaData を使用すれば、データの検索に使用すべき get メソッドを判別できます。以下のコード例では、ResultSetMetaData を使用して ResultSet 内の各列タイプを判別しています。

```
ResultSet rs = stmt.executeQuery(sqlString);
ResultSetMetaData rsmd = rs.getMetaData();
int colType [] = new int[rsmd.getColumnCount()];
for (int idx = 0, int col = 1; idx < colType.length; idx++, col++)
colType[idx] = rsmd.getColumnType(col);
```

例: ResultSetMetaData インターフェース:

このプログラムは、実際の ResultSetMetaData と ResultSet の例を使用して、表の照会によって作成される ResultSet についてのメタデータをすべて表示します。ユーザーは、表とライブラリーの値を渡します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;

/**
ResultSetMetaDataExample.java

This program demonstrates using a ResultSetMetaData and
a ResultSet to display all the metadata about a ResultSet
created querying a table.  The user passes the value for the
table and library.
**/
public class ResultSetMetaDataExample {

    public static void main(java.lang.String[] args)
    {
        if (args.length != 2) {
            System.out.println("Usage:  java ResultSetMetaDataExample <library> <table>");
            System.out.println("where <library> is the library that contains <table>");
            System.exit(0);
        }

        Connection con = null;
        Statement s = null;
        ResultSet rs = null;
        ResultSetMetaData rsmd = null;

        try {
            // Get a database connection and prepare a statement.
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            con = DriverManager.getConnection("jdbc:db2:*local");

            s = con.createStatement();

            rs = s.executeQuery("SELECT * FROM " + args[0] + "." + args[1]);
            rsmd = rs.getMetaData();

            int colCount = rsmd.getColumnCount();
            int rowCount = 0;
        for (int i = 1; i <= colCount; i++) {
                System.out.println("Information about column " + i);
                System.out.println("   Name..........: " + rsmd.getColumnName(i));
                System.out.println("   Data Type.....: " + rsmd.getColumnType(i) +
                                   " ( " + rsmd.getColumnTypeName(i) + " )");
                System.out.println("   Precision.....: " + rsmd.getPrecision(i));
                System.out.println("   Scale.........: " + rsmd.getScale(i));
                System.out.print  ("   Allows Nulls..: ");
                if (rsmd.isNullable(i)==0)
                    System.out.println("false");
                else
                    System.out.println("true");
```

```
            }

        } catch (Exception e) {
            // Handle any errors.
            System.out.println("Oops... we have an error... ");
            e.printStackTrace();
        } finally {
            // Ensure we always clean up.  If the connection gets closed, the
            // statement under it closes as well.
            if (con != null) {
                try {
                    con.close();
                } catch (SQLException e) {
                    System.out.println("Critical error - cannot close connection object");
                }
            }
        }
    }
}
```

ResultSets の変更:

IBM i JDBC ドライバーでは、以下のタスクを実行することによって、ResultSet を変更できます。

ResultSet のデフォルト設定は、「読み取り専用」です。しかし、Java Database Connectivity (JDBC) 2.0 では、IBM i JDBC ドライバーは更新可能 ResultSet を完全にサポートしています。

ResultSet の更新方法については、 108 ページの『ResultSet の特性』を参照することができます。

行の更新

行は ResultSet インターフェースを通してデータベース・テーブル内で更新できます。このプロセスに関連したステップは、以下のとおりです。

1. 種々の update<*Type*> メソッド (<*Type*> は Java データ・タイプ) を使用して、特定の行の値を変更する。 update<*Type*> メソッドは、値の検索に使用することができる get<*Type*> メソッドと対応しています。
2. 行を基になるデータベースに適用する。

データベースそのものは、2 番目のステップを実行するまでは更新されません。 updateRow メソッドを呼び出さずに ResultSet の列を更新しても、データベースに変更は加えられません。

計画した行に対する更新は、cancelUpdates メソッドで破棄することができます。いったん updateRow メソッドを呼び出せば、データベースに対する変更は確定され、元に戻すことはできません。

注: データベースが更新済みの行を指し示す手段を持っていない場合、rowUpdated メソッドは常に false を戻します。これに対応して updatesAreDetected メソッドも false を戻します。

行の削除

行は ResultSet インターフェースを通してデータベース・テーブル内で削除できます。 deleteRow メソッドを指定すると、現行の行が削除されます。

行の挿入

行は ResultSet インターフェースを通してデータベース・テーブルに挿入できます。このプロセスでは「挿入行」を使用します。アプリケーションはこの「挿入行」に実際にカーソルを移動し、データベースに挿入する値を設定します。このプロセスに関連したステップは、以下のとおりです。

1. 挿入行にカーソルを置く。
2. 新しい行の各列の値を設定する。
3. データベースに行を挿入し、任意でカーソルを ResultSet 内の現行の行に戻す。

注: 新規行は、カーソルが置かれているテーブルには挿入されません。これらは通常、テーブル・データ・スペースの末尾に追加されます。リレーショナル・データベースは、デフォルトでは位置依存ではありません。たとえば、3 番目の行にカーソルを移動して何かを挿入しても、後のユーザーがデータを取り出すときに 4 番目の行の前にそれが表示されることはありません。

定位置更新のサポート

ResultSet を通してデータベースを更新するためのメソッド以外に、定位置更新を発行するための SQL ステートメントを使用することができます。このサポートは、名前付きカーソルを使用することを前提としています。 JDBC は、これらの値へのアクセスを可能にする Statement からの setCursorName メソッドと、ResultSet からの getCursorName メソッドを提供しています。

supportsPositionedUpdated と supportsPositionedDelete の 2 つの DatabaseMetaData メソッドはどちらも、ネイティブ JDBC ドライバーでこのフィーチャーがサポートされていれば true を戻します。

例: 他のステートメントのカーソルを介してテーブルから値を除去する:

この Java の例では、他のステートメントのカーソルを介してテーブルから値を除去する方法を示します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;

public class UsingPositionedDelete {
    public Connection connection = null;
    public static void main(java.lang.String[] args) {
        UsingPositionedDelete test = new UsingPositionedDelete();

        test.setup();
        test.displayTable();

        test.run();
        test.displayTable();

        test.cleanup();
    }



/**
Handle all the required setup work.
**/
    public void setup() {
        try {
            // Register the JDBC driver.
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");

            connection = DriverManager.getConnection("jdbc:db2:*local");

            Statement s = connection.createStatement();
            try {
                s.executeUpdate("DROP TABLE CUJOSQL.WHERECUREX");
            } catch (SQLException e) {
                // Ignore problems here.
            }
```

```
            s.executeUpdate("CREATE TABLE CUJOSQL.WHERECUREX ( " +
                            "COL_IND INT, COL_VALUE CHAR(20)) ");

            for (int i = 1; i <= 10; i++) {
                s.executeUpdate("INSERT INTO CUJOSQL.WHERECUREX VALUES(" + i + ", 'FIRST')");
            }

            s.close();

        } catch (Exception e) {
            System.out.println("Caught exception: " + e.getMessage());
            e.printStackTrace();
        }
    }


/**
In this section, all the code to perform the testing should
be added. If only one connection to the database is needed,
the global variable 'connection' can be used.
**/
    public void run() {
        try {
            Statement stmt1 = connection.createStatement();

            // Update each value using next().
            stmt1.setCursorName("CUJO");
            ResultSet rs = stmt1.executeQuery ("SELECT * FROM CUJOSQL.WHERECUREX " +
                                               "FOR UPDATE OF COL_VALUE");

            System.out.println("Cursor name is " + rs.getCursorName());

            PreparedStatement stmt2 = connection.prepareStatement
                                      ("DELETE FROM " + " CUJOSQL.WHERECUREX WHERE CURRENT OF " +
                                      rs.getCursorName ());


            // Loop through the ResultSet and update every other entry.
            while (rs.next ()) {
                if (rs.next())
                    stmt2.execute ();
            }


            // Clean up the resources after they have been used.
            rs.close ();
            stmt2.close ();


        } catch (Exception e) {
            System.out.println("Caught exception: ");
            e.printStackTrace();
        }
    }



/**
In this section, put all clean-up work for testing.
**/
    public void cleanup() {
        try {
            // Close the global connection opened in setup().
            connection.close();

        } catch (Exception e) {
```

```
            System.out.println("Caught exception: ");
            e.printStackTrace();
        }
    }



/**
Display the contents of the table.
**/
    public void displayTable()
    {
        try {
            Statement s = connection.createStatement();
            ResultSet rs = s.executeQuery ("SELECT * FROM CUJOSQL.WHERECUREX");

            while (rs.next ()) {
                System.out.println("Index " + rs.getInt(1) + " value " + rs.getString(2));
            }

            rs.close ();
            s.close();
            System.out.println("-----------------------------------------");
        } catch (Exception e) {
            System.out.println("Caught exception: ");
            e.printStackTrace();
        }
    }
}
```

例: 他のステートメントのカーソルを介してステートメントで値を変更する:

この Java の例では、他のステートメントのカーソルを介したステートメントによる値の変更方法を示します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;

public class UsingPositionedUpdate {
    public Connection connection = null;
    public static void main(java.lang.String[] args) {

        UsingPositionedUpdate test = new UsingPositionedUpdate();

        test.setup();
        test.displayTable();

        test.run();
        test.displayTable();

        test.cleanup();
    }



/**
Handle all the required setup work.
**/
    public void setup() {
        try {
            // Register the JDBC driver.
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");

            connection = DriverManager.getConnection("jdbc:db2:*local");
```

```
            Statement s = connection.createStatement();
            try {
                s.executeUpdate("DROP TABLE CUJOSQL.WHERECUREX");
            } catch (SQLException e) {
                // Ignore problems here.
            }

            s.executeUpdate("CREATE TABLE CUJOSQL.WHERECUREX ( " +
                            "COL_IND INT, COL_VALUE CHAR(20)) ");

            for (int i = 1; i <= 10; i++) {
                s.executeUpdate("INSERT INTO CUJOSQL.WHERECUREX VALUES(" + i + ", 'FIRST')");
            }

            s.close();

        } catch (Exception e) {
            System.out.println("Caught exception: " + e.getMessage());
            e.printStackTrace();
        }
    }


/**
In this section, all the code to perform the testing should
be added. If only one connection to the database is required,
the global variable 'connection' can be used.
**/
    public void run() {
        try {
            Statement stmt1 = connection.createStatement();

            // Update each value using next().
            stmt1.setCursorName("CUJO");
            ResultSet rs = stmt1.executeQuery ("SELECT * FROM CUJOSQL.WHERECUREX " +
                                               "FOR UPDATE OF COL_VALUE");

            System.out.println("Cursor name is " + rs.getCursorName());

            PreparedStatement stmt2 = connection.prepareStatement ("UPDATE "
                                                                  + " CUJOSQL.WHERECUREX
                                                                      SET COL_VALUE = 'CHANGED'
                                                                      WHERE CURRENT OF "
                                                                  + rs.getCursorName ());


            // Loop through the ResultSet and update every other entry.
            while (rs.next ()) {
                if (rs.next())
                    stmt2.execute ();
            }


            // Clean up the resources after they have been used.
            rs.close ();
            stmt2.close ();


        } catch (Exception e) {
            System.out.println("Caught exception: ");
            e.printStackTrace();
        }
    }
```

```
/**
In this section, put all clean-up work for testing.
**/
    public void cleanup() {
        try {
            // Close the global connection opened in setup().
            connection.close();

        } catch (Exception e) {
            System.out.println("Caught exception: ");
            e.printStackTrace();
        }
    }



/**
Display the contents of the table.
**/
    public void displayTable()
    {
        try {
            Statement s = connection.createStatement();
            ResultSet rs = s.executeQuery ("SELECT * FROM CUJOSQL.WHERECUREX");

            while (rs.next ()) {
                System.out.println("Index " + rs.getInt(1) + " value " + rs.getString(2));
            }

            rs.close ();
            s.close();
            System.out.println("-----------------------------------------");
        } catch (Exception e) {
            System.out.println("Caught exception: ");
            e.printStackTrace();
        }
    }
}
```

ResultSet の作成:

ResultSet オブジェクトを作成するには、executeQuery メソッド、または他のメソッドを使用できます。このトピックでは、ResultSet を作成する際のオプションについて説明します。

これらのメソッドは、Statement、PreparedStatement、または CallableStatement インターフェースから実行します。ただし、方法は他にもあります。たとえば、getColumns、getTables、getUDTs、getPrimaryKeys などの DatabaseMetaData メソッドは、ResultSet を戻します。単一の SQL ステートメントの処理で、複数の ResultSet を戻すこともできます。さらに、Statement、PreparedStatement、または CallableStatement インターフェースで提供されている execute メソッドを呼び出した後に、getResultSet メソッドを使用して ResultSet オブジェクトを検索することができます。

詳しくは、 104 ページの『例: 複数の ResultSet を持つプロシージャーを作成する』を参照してください。

ResultSets のクローズ

ResultSet オブジェクトは、関連する Statement オブジェクトがクローズすると自動的にクローズされますが、ResultSet オブジェクトは使用しなくなったらクローズすることをお勧めします。そうすれば、即時に内部データベース・リソースが解放され、それによってアプリケーションのスループットが増大する可能性があります。

DatabaseMetaData 呼び出しによって生成された ResultSet をクローズすることも大切です。これらの ResultSet の作成に使用された Statement オブジェクトに直接アクセスすることはできないので、Statement オブジェクトで直接 close を呼び出すことはしません。これらのオブジェクトは互いにリンクされ、外部の ResultSet オブジェクトをクローズすると、JDBC ドライバーが内部の Statement オブジェクトをクローズするようになっています。これらのオブジェクトを手動でクローズしない場合、システムは引き続き作動しますが、必要以上のリソースを使用することになります。

注: ResultSet の保持可能性特性でも ResultSet を自動的にクローズすることができます。 close は ResultSet オブジェクトで何回でも呼び出すことができます。

92 ページの『Statement オブジェクト』
Statement オブジェクトは、静的 SQL ステートメントの処理と、それによって生成される結果の取得に使用されます。 一度にオープンできるのは、各 Statement オブジェクトにつき 1 つの ResultSet だけです。 SQL ステートメントを処理するすべてのステートメント・メソッドは、すでにオープンされている ResultSet があると、暗黙的にステートメントの現行の ResultSet をクローズします。

94 ページの『PreparedStatement』
PreparedStatement は Statement インターフェースを拡張し、SQL ステートメントへのパラメーターの追加をサポートします。

100 ページの『CallableStatement』
JDBC CallableStatement インターフェースは PreparedStatement を拡張し、パラメーターの出力および入出力のサポートを提供します。 CallableStatement インターフェースは、PreparedStatement インターフェースによって提供される入力パラメーターもサポートします。

58 ページの『DatabaseMetaData インターフェース』
DatabaseMetaData インターフェースは、IBM Developer Kit for Java JDBC ドライバーによってインプリメントされ、基礎となるデータ・ソースに関する情報を提供します。これは、提供されているデータ・ソースとの対話方法を決定するため、主にアプリケーション・サーバーとツールによって使用されます。 アプリケーションは、DatabaseMetaData メソッドを使用してもデータ・ソースの情報を入手することができますが、こちらはそれほど一般的ではありません。

例: ResultSet インターフェース:

以下は、ResultSet インターフェースの使用法の例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;

/**
ResultSetExample.java

This program demonstrates using a ResultSetMetaData and
a ResultSet to display all the data in a table even though
the program that gets the data does not know what the table
is going to look like (the user passes in the values for the
table and library).
**/
public class ResultSetExample {

    public static void main(java.lang.String[] args)
    {
        if (args.length != 2) {
            System.out.println("Usage:  java ResultSetExample <library> <table>");
            System.out.println(" where <library> is the library that contains <table>");
            System.exit(0);
        }
```

```
        Connection con = null;
        Statement s = null;
        ResultSet rs = null;
        ResultSetMetaData rsmd = null;

        try {
            // Get a database connection and prepare a statement.
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            con = DriverManager.getConnection("jdbc:db2:*local");

            s = con.createStatement();

            rs = s.executeQuery("SELECT * FROM " + args[0] + "." + args[1]);
            rsmd = rs.getMetaData();

            int colCount = rsmd.getColumnCount();
            int rowCount = 0;
            while (rs.next()) {
                rowCount++;
                System.out.println("Data for row " + rowCount);
                for (int i = 1; i <= colCount; i++)
                    System.out.println("   Row " + i + ": " + rs.getString(i));
            }

        } catch (Exception e) {
            // Handle any errors.
            System.out.println("Oops... we have an error... ");
            e.printStackTrace();
        } finally {
            // Ensure we always clean up.  If the connection gets closed, the
            // statement under it closes as well.
            if (con != null) {
                try {
                    con.close();
                } catch (SQLException e) {
                    System.out.println("Critical error - cannot close connection object");
                }
            }
        }
    }
}
```

JDBC オブジェクト・プーリング

オブジェクト・プーリングは、Java Database Connectivity (JDBC) とパフォーマンスにとって重要な考慮事項です。JDBC で使用されるオブジェクト (Connection、Statement、および ResultSet オブジェクトなど) の多くは作成に費用がかかるので、これらのオブジェクトを必要になるたびに作成するのではなく再利用することで、パフォーマンス上の大きな利点を得ることができます。

ユーザーに代わってすでに多くのアプリケーションで、オブジェクト・プーリングはハンドルされています。たとえば、WebSphere は、JDBC オブジェクト・プーリングを広範囲にサポートしており、プールの管理方法を制御できるようになっています。このため、独自のプーリング・メカニズムを気にすることなく、必要な機能を利用することができます。しかし、このサポートがなければ、ほとんどのアプリケーションについてソリューションを自分で見つける必要があります。

オブジェクト・プーリングのための DataSource サポートの使用:

データベースにアクセスするための共通の構成を複数のアプリケーションで共用するために、 DataSources を使用できます。このことは、各アプリケーションで同じ DataSource 名を参照させることによって実現します。

DataSource を使用することにより、多くのアプリケーションを中央設置場所から変更できます。たとえば、すべてのアプリケーションで使用するデフォルト・ライブラリーの名前を変更し、 1 つの DataSource を使用して、それらすべての接続を入手した場合、その DataSource でコレクションの名前を更新できます。その後、使用しているすべてのアプリケーションは、新しいデフォルト・ライブラリーの使用を開始します。

DataSource を使用して、アプリケーションの接続を入手する場合、接続プーリングのためにネイティブ JDBC ドライバーの組み込みサポートを使用できます。このサポートは、ConnectionPoolDataSource インターフェースの実装として提供されます。

プーリングは、物理 Connection オブジェクトの代わりに、「論理」Connection オブジェクトを出すことによって実現します。**論理 Connection オブジェクト**とは、プールされた Connection オブジェクトによって戻される接続オブジェクトのことです。それぞれの論理接続オブジェクトは、プールされた接続オブジェクトで表される物理接続への一時ハンドルとして機能します。アプリケーションにとっては、Connection オブジェクトが戻されれば、それら 2 つの間には大きな違いはありません。 Connection オブジェクトでクローズ・メソッドを呼び出すときに、わずかな違いが出てくるだけです。この呼び出しは、論理接続を無効にして、別のアプリケーションが物理接続を使用できるプールに物理接続を戻します。この技法を使用すると、多くの論理接続オブジェクトで、1 つの物理接続を再利用できるようになります。

接続プーリングの設定

接続プーリングは、ConnectionPoolDataSource オブジェクトを参照する DataSource オブジェクトを作成することで実現します。 ConnectionPoolDataSource オブジェクトには、プール保守のさまざまな要素を処理するために設定できるプロパティーがあります。

UDBDataSource および UDBConnectionPoolDataSource を使用して接続プーリングをセットアップする方法の詳細の例を参照してください。この例で JNDI が担当する役割の詳細については、Java Naming and Directory Interface (JNDI) も参照できます。

例では、2 つの DataSource オブジェクトを 1 つにバインドするリンクは、dataSourceName です。このリンクは、プーリングを自動的に管理する ConnectionPoolDataSource オブジェクトへの接続の確立を延期するよう、DataSource オブジェクトに通知します。

プーリングおよび非プーリング・アプリケーション

Connection プーリングを使用するアプリケーションと、それを使用しないアプリケーションとの間には、違いはありません。したがって、プーリング・サポートは、アプリケーション・コードの完了後に追加できます。その際に、アプリケーション・コードに変更を加える必要はありません。

次に示すのは、開発時に前述のプログラムをローカルに実行するときの出力です。

非プーリング DataSource バージョンのタイミングを開始 (Start timing the non-pooling DataSource version...) 経過時間: 6410 (Time spent: 6410)

プーリング・バージョンのタイミングを開始... (Start timing the pooling version...) 経過時間: 282 (Time spent: 282)

Java プログラムが完了しました (Java program completed)

デフォルトでは、UDBConnectionPoolDataSource は 1 つの接続をプーリングします。アプリケーションが接続を複数回必要としていて、一度に 1 つの接続だけを必要とする場合、UDBConnectionPoolDataSource を使用することは、完全な解決策になります。多数の接続を同時に必要とする場合は、

ConnectionPoolDataSource を構成 129 ページの『ConnectionPoolDataSource のプロパティー』し、必要とリソースを満たす必要があります。

関連概念

559 ページの『Java Naming and Directory Interface』
Java Naming and Directory Interface (JNDI) は、JavaSoft のプラットフォーム・アプリケーション・プログラミング・インターフェース (API) の一部です。 JNDI により、複数の命名およびディレクトリー・サービスにシームレスに接続することができます。 このインターフェースを使用すると、強力で可搬性のある、ディレクトリーが使用可能な Java アプリケーションを作成することができます。

関連資料

『例: UDBDataSource および UDBConnectionPoolDataSource で接続プーリングをセットアップする』
以下に、UDBDataSource および UDBConnectionPoolDataSource で接続プーリングを使用する方法の例を示します。

129 ページの『ConnectionPoolDataSource のプロパティー』
ConnectionPoolDataSource インターフェースは、用意されている一連のプロパティーを使用することによって構成できます。

例: UDBDataSource および UDBConnectionPoolDataSource で接続プーリングをセットアップする:

以下に、UDBDataSource および UDBConnectionPoolDataSource で接続プーリングを使用する方法の例を示します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;
import javax.naming.*;
import com.ibm.db2.jdbc.app.UDBDataSource;
import com.ibm.db2.jdbc.app.UDBConnectionPoolDataSource;

public class ConnectionPoolingSetup
{
    public static void main(java.lang.String[] args)
    throws Exception
    {
        // Create a ConnectionPoolDataSource implementation
        UDBConnectionPoolDataSource cpds = new UDBConnectionPoolDataSource();
        cpds.setDescription("Connection Pooling DataSource object");

        // Establish a JNDI context and bind the connection pool data source
        Context ctx = new InitialContext();
        ctx.rebind("ConnectionSupport", cpds);

        // Create a standard data source that references it.
        UDBDataSource ds = new UDBDataSource();
        ds.setDescription("DataSource supporting pooling");
        ds.setDataSourceName("ConnectionSupport");
        ctx.rebind("PoolingDataSource", ds);
  }
}
```

例: 接続プーリングのパフォーマンスをテストする:

以下に、プーリングされたときのパフォーマンスとプーリングされていないときのパフォーマンスを対比してテストする方法を示します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;
import javax.naming.*;
import java.util.*;
import javax.sql.*;

public class ConnectionPoolingTest
{
    public static void main(java.lang.String[] args)
    throws Exception
    {
        Context ctx = new InitialContext();
        // Do the work without a pool:
        DataSource ds = (DataSource) ctx.lookup("BaseDataSource");
        System.out.println("¥nStart timing the non-pooling DataSource version...");

        long startTime = System.currentTimeMillis();
        for (int i = 0; i < 100; i++) {
            Connection c1 = ds.getConnection();
            c1.close();
        }
        long endTime = System.currentTimeMillis();
        System.out.println("Time spent: " + (endTime - startTime));

        // Do the work with pooling:
        ds = (DataSource) ctx.lookup("PoolingDataSource");
        System.out.println("¥nStart timing the pooling version...");

        startTime = System.currentTimeMillis();
        for (int i = 0; i < 100; i++) {
            Connection c1 = ds.getConnection();
            c1.close();
        }
        endTime = System.currentTimeMillis();
        System.out.println("Time spent: " + (endTime - startTime));
    }
}
```

ConnectionPoolDataSource のプロパティー:

ConnectionPoolDataSource インターフェースは、用意されている一連のプロパティーを使用することによって構成できます。

次の表には、このようなプロパティーの説明が載せられています。

プロパティー	説明
initialPoolSize	プールを初めてインスタンス化する場合、このプロパティーにより、プールに配置する接続数が決定されます。この値が minPoolSize と maxPoolSize の範囲外で指定される場合、作成する初期接続の数として minPoolSize または maxPoolSize が使用されます。

プロパティー	説明
maxPoolSize	プールが使用される場合、プールに含まれる数よりも多くの接続を要求できます。このプロパティーでは、プールに作成できる最大接続数を指定します。 プールが最大サイズになっていて、すべての接続が使用中である場合には、アプリケーションは、プールに戻される接続を「ブロック」せずに待機します。代わりに、JDBC ドライバーは、DataSource プロパティーに基づいて新しい接続を構成し、接続を戻します。 maxPoolSize として 0 が指定される場合、渡すことのできるリソースがシステムにある限り、プールを無限に拡大することができます。
minPoolSize	プールを使用中のスパイクは、それに含まれる接続の数を増やすことができます。アクティビティー・レベルが、プールからいくつかの Connection が引き出されない点まで下がると、特に理由はなくてもリソースが取られます。 そのようなケースでは、JDBC ドライバーに、累積したいくつかの接続を解放する機能があります。このプロパティーを使用すると、JDBC に接続を解放するよう通知し、使用できる特定の接続数を常に保つようにすることができます。 minPoolSize として 0 が指定される場合、プールがすべての接続を解放し、アプリケーションが接続要求ごとに接続時間を実際に処理できるようになります。
maxIdleTime	接続は、使用されずに放置されている期間を追跡します。このプロパティーは、接続を解放する前に、アプリケーションが接続を未使用にしておける期間を指定します (つまり、必要な数よりもさらに多くの接続が存在するということです)。 このプロパティーは、秒単位の時間であり、実際のクローズが行われる時刻を指定するものではありません。ここでは、接続を解放するための十分な時間がいつ経過するかを指定します。
propertyCycle	このプロパティーは、これらの規則の実行と実行の間で、経過することを認められている秒数を表します。

注: maxIdleTime または propertyCycle のいずれかの時間を 0 に設定する場合、JDBC ドライバーは、それ自体ではプールから除去される接続を検査しません。 initial、min、および max サイズに指定される規則はまだ有効です。

maxIdleTime および propertyCycle が 0 でない場合、プールを監視するために管理スレッドが使用されます。このスレッドは、propertyCycle 秒ごとにウェイクし、プール内のすべての接続を検査して、maxIdleTime 秒以上使用されていない接続を確認します。この基準に当てはまる接続は、minPoolSize に達するまでプールから除去されます。

DataSource ベースのステートメント・プーリング:

UDBConnectionPoolDataSource インターフェース上で使用できる maxStatements プロパティーを使用すると、接続プール内でのステートメント・プーリングが可能になります。ステートメント・プーリングだけが、PreparedStatements および CallableStatements に影響します。ステートメント・オブジェクトは、プールされません。

ステートメント・プーリングの実装は、接続プーリングの実装と似ています。アプリケーションが Connection.prepareStatement("select * from tablex") を呼び出すと、プーリング・モジュールが、接続下で Statement オブジェクトが準備されているかどうかを確認します。準備されている場合、物理オブジェクトではなく、論理 PreparedStatement オブジェクトが渡されます。クローズを呼び出すと、Connection オブジェクトがプールに戻され、論理 Connection オブジェクトが出され、そして Statement オブジェクトが再利用できるようになります。

maxStatements プロパティーを使用すると、DataSource は、接続下でプールできるステートメントの数を指定できます。値が 0 の場合、ステートメント・プーリングを使用しないことを示します。ステートメント・プールがいっぱいの場合、使用された一番古いアルゴリズムが適用され、出されるステートメントが判別されます。

次の例では、接続プーリングだけを使用する 1 つの DataSource と、ステートメントと接続プーリングを使用する他の DataSource をテストします。

次の例は、開発時にこのプログラムをローカルに実行するときの出力です。

ステートメント・プーリングのデータ・ソースを展開しています (Deploying statement pooling data source) 接続プーリング専用バージョンのタイミングを開始... (Start timing the connection pooling only version...) 経過時間: 26312 (Time spent: 26312)

ステートメント・プーリング・バージョンのタイミングを開始... (Starting timing the statement pooling version...) 経過時間: 2292 (Time spent: 2292) Java プログラムが完了しました (Java program completed)

例: 2 つの DataSource のパフォーマンスをテストする:

次に、接続プーリングだけを使用する 1 つの DataSource と、ステートメントと接続プーリングを使用する別の DataSource をテストする例を示します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;
import javax.naming.*;
import java.util.*;
import javax.sql.*;
import com.ibm.db2.jdbc.app.UDBDataSource;
import com.ibm.db2.jdbc.app.UDBConnectionPoolDataSource;

public class StatementPoolingTest
{

    public static void main(java.lang.String[] args)
    throws Exception
    {
        Context ctx = new InitialContext();

        System.out.println("deploying statement pooling data source");
        deployStatementPoolDataSource();

        // Do the work with connection pooling only.
        DataSource ds = (DataSource) ctx.lookup("PoolingDataSource");
```

```
        System.out.println("¥nStart timing the connection pooling only version...");

        long startTime = System.currentTimeMillis();
        for (int i = 0; i < 100; i++) {
            Connection c1 = ds.getConnection();
            PreparedStatement ps = c1.prepareStatement("select * from qsys2.sysprocs");
            ResultSet rs = ps.executeQuery();
            c1.close();
        }
        long endTime = System.currentTimeMillis();
        System.out.println("Time spent: " + (endTime - startTime));


        // Do the work with statement pooling added.
        ds = (DataSource) ctx.lookup("StatementPoolingDataSource");
        System.out.println("¥nStart timing the statement pooling version...");

        startTime = System.currentTimeMillis();
        for (int i = 0; i < 100; i++) {
            Connection c1 = ds.getConnection();
            PreparedStatement ps = c1.prepareStatement("select * from qsys2.sysprocs");
            ResultSet rs = ps.executeQuery();
            c1.close();
        }
        endTime = System.currentTimeMillis();
        System.out.println("Time spent: " + (endTime - startTime));
    }



    private static void deployStatementPoolDataSource()
    throws Exception
    {
        // Create a ConnectionPoolDataSource implementation
        UDBConnectionPoolDataSource cpds = new UDBConnectionPoolDataSource();
        cpds.setDescription("Connection Pooling DataSource object with Statement pooling");
        cpds.setMaxStatements(10);

        // Establish a JNDI context and bind the connection pool data source
        Context ctx = new InitialContext();
        ctx.rebind("StatementSupport", cpds);


        // Create a standard datasource that references it.
        UDBDataSource ds = new UDBDataSource();
        ds.setDescription("DataSource supporting statement pooling");
        ds.setDataSourceName("StatementSupport");
        ctx.rebind("StatementPoolingDataSource", ds);

    }
}
```

独自の接続プーリングの構築:

DataSources のサポートを必要としない、または別の製品に依存しない、独自の接続およびステートメント・プーリングを開発できます。

接続プーリングを使用しなかった場合、要求ごとに発生するデータベース作業の量は膨大です。つまり、接続を確立し、ステートメントを入手し、ステートメントを処理し、ステートメントをクローズし、そして接続をクローズするわけです。行ったことすべてをそれぞれの要求後に破棄してしまうのではなく、このプロセスの部分を再利用するための方法があります。**接続プーリング**は、接続の作成コードを、プールから接続を入手するコードに置き換え、接続のクローズ・コードを、使用する接続をプールに戻すコードに置き換えます。

接続プーリングのコンストラクターは、接続を作成してプールに置きます。プール・クラスには、使用する接続を探し、接続を使用した処理が終了したときに、その接続をプールに戻すための、take および put メソッドがあります。プール・オブジェクトは共用リソースであるため、これらのメソッドは同期化されますが、プーリングされたリソースを複数のスレッドで同時に操作するわけではありません。

独自のステートメント・プーリングの構築

接続プーリングを使用する場合、各ステートメントを処理するときには、ステートメントの作成とクローズに時間がかかります。これは、再利用できるオブジェクトを無駄にしている例といえます。

オブジェクトを再利用するために、準備したステートメント・クラスを使用できます。ほとんどのアプリケーションで、小さな変更が加えられた同じ SQL ステートメントが再利用されます。たとえば、アプリケーションで 1 回反復がある場合、次の照会を生成できます。

```
SELECT * from employee where salary > 100000
```

次の反復では、次の照会を生成できます。

```
SELECT * from employee where salary > 50000
```

これは同じ照会ですが、別のパラメーターを使用しています。両方の照会を、次の照会で実現できます。

```
SELECT * from employee where salary > ?
```

その後、最初の照会の処理時にパラメーター・マーカー (疑問符で示される) を 100000 に設定し、 2 番目の照会の処理時に 50000 に設定します。このようにすると、接続プールで実現できる機能以外の 3 つの理由で、パフォーマンスが拡張されます。

- 作成されるオブジェクトがより少なくて済む。要求のたびに Statement オブジェクトが作成されるのではなく、PreparedStatement オブジェクトが作成されて再利用されます。したがって、実行するコンストラクターが少なくて済みます。
- SQL ステートメントを設定するデータベース作業 (**準備**という) を再利用できる。 SQL ステートメントの準備は、SQL ステートメント・テキストの内容と、要求されたタスクをシステムで実現する方法を識別することが関係するため、それなりに高く付きます。
- 別のオブジェクト作成を除去するときに、あまり考慮されない利点がある。作成されなかったものを破棄する必要はありません。このモデルは、Java ガーベッジ・コレクター上ではより使い勝手が良く、ユーザーが多くて時間がかかっているパフォーマンスの点でも有利です。

考慮事項

パフォーマンスは、複製を行うことによって改善されます。特定の項目を再利用しない場合、その項目をプールするためにリソースを無駄にしています。

ほとんどのアプリケーションには、コードのクリティカル・セクションが含まれています。一般には、アプリケーションは、コードの 10 から 20 % だけに対して、処理時間の 80 から 90 % を費やします。アプリケーションで 10,000 個の SQL ステートメントが使用される可能性がある場合、そのすべてがプールされるわけではありません。その目的は、アプリケーションのコードのクリティカル・セクションで使用される SQL ステートメントを識別してプールすることです。

Java インプリメンテーションでオブジェクトを作成すると、コストが非常に高く付く可能性があります。この点で、プーリング・ソリューションを使用することには利点があります。プロセスで使用されるオブジェクトは、開始時に作成されますが、これは他のユーザーがシステムを使用しようとする前です。これらのオブジェクトは、必要なときに再利用されます。パフォーマンスは優秀ですし、アプリケーションをきめ細かく徐々に調整するので、大勢のユーザーが使用できるようになります。結果として、さらに多くのオブジ

ェクトがプールされます。さらに、アプリケーションのデータベース・アクセスをより効率的にマルチスレッド化することにより、より良いスループットを得ることができます。

Java (JDBC を使用した) は、動的 SQL をベースにしているため、遅くなる傾向があります。プーリングすることにより、この問題を最小限にすることができます。開始時にステートメントを準備することにより、データベースへのアクセスを静的に実現できます。ステートメントを準備した後は、動的 SQL と静的 SQL との間には、パフォーマンスの点でほとんど差はありません。

Java でのデータベース・アクセスのパフォーマンスは効率的になりますが、このことは、 オブジェクト指向設計やコードの保守容易性を犠牲にすることなく実現できます。ステートメントおよび接続プーリングを構築するためにコードを作成することは難しくありません。さらに、コードを変更して拡張することで、複数のアプリケーションやアプリケーション・タイプ (Web ベース、クライアント/サーバー) などをサポートできるようになります。

バッチ更新

バッチ更新サポートを使用することにより、データベースに対する任意の数の更新を、ユーザー・プログラムとデータベースの間の単一トランザクションとして渡すことができます。このプロシージャーは、一度に多くの更新を実行しなければならないときに、パフォーマンスをかなり向上させることができます。

たとえば、ある大規模な会社で、新しい社員たちが月曜日から業務を開始しなければならない場合、これは社員データベースに対して、一度に多くの更新 (この場合は、挿入) を行うことが必要になります。更新するためのバッチを作成し、データベースにこれを 1 つの単位としてサブミットすれば、処理時間を節約することができます。

バッチ更新には、次の 2 つのタイプがあります。

- Statement オブジェクトを使用したバッチ更新。
- PreparedStatement オブジェクトを使用したバッチ更新。

Statement バッチ更新:

Statement バッチ更新を実行するためには、自動コミットをオフにする必要があります。 Java Database Connectivity (JDBC) では、デフォルトで自動コミットがオンになっています。自動コミットは、データベースに対する更新のたびに、それぞれの SQL ステートメントの処理の後にコミットされることです。データベースへの処理のための複数のステートメントのグループを、機能的に 1 つのグループとしてまとめて扱いたい場合は、各ステートメントごとに個別にデータベースにコミットすることは望ましくありません。自動コミットをオフにせずにバッチの途中で失敗してしまった場合は、バッチ全体をロールバックすることができず、ステートメント全体を完了するためにバッチ処理を再度実行する必要があります。さらに、バッチ内の各ステートメントでコミットするという追加作業は、多大なオーバーヘッドを生み出します。

詳細は、 71 ページの『JDBC トランザクション』を参照してください。

自動コミットをオフにすると、標準の Statement オブジェクトが作成できます。 executeUpdate のようなメソッドを使ってステートメントを処理する代わりに、そのステートメントを addBatch メソッドを使ってバッチに追加します。バッチに追加したいすべてのステートメントを追加したなら、executeBatch メソッドですべてのステートメントを処理することができます。 clearBatch メソッドを使用すれば、いつでもバッチを空にすることができます。

以下に、これらのメソッドを使い方を示します。

例: Statement バッチ更新

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
connection.setAutoCommit(false);
Statement statement = connection.createStatement();
statement.addBatch("INSERT INTO TABLEX VALUES(1, 'Cujo')");
statement.addBatch("INSERT INTO TABLEX VALUES(2, 'Fred')");
statement.addBatch("INSERT INTO TABLEX VALUES(3, 'Mark')");
int [] counts = statement.executeBatch();
connection.commit();
```

この例では、executeBatch メソッドから整数の配列が戻されます。この配列は、バッチ内で処理されたステートメントごとに 1 つの整数値を持っています。データベースへ値を挿入した場合は、その各ステートメントのこの値は 1 になります (これは、処理が成功したことを想定しています)。しかし、更新ステートメントのようないくつかのステートメントでは、影響が複数の行にわたることがあります。バッチ内に INSERT、UPDATE、または DELETE 以外のステートメントを加えた場合は、例外が発生します。

PreparedStatement バッチ更新:

PreparedStatement バッチは Statement バッチと似ていますが、PreparedStatement バッチは同じ準備済みステートメントを常に取り除くので、そのステートメントに対するパラメーターを変更するだけで済みます。

以下に、PreparedStatement バッチを使用した例を示します。

注: 法律上の重要な情報に関しては、コードの特記事項情報をお読みください。

```
connection.setAutoCommit(false);
PreparedStatement statement =
  connection.prepareStatement("INSERT INTO TABLEX VALUES(?, ?)");
statement.setInt(1, 1);
statement.setString(2, "Cujo");
statement.addBatch();
statement.setInt(1, 2);
statement.setString(2, "Fred");
statement.addBatch();
statement.setInt(1, 3);
statement.setString(2, "Mark");
statement.addBatch();
int [] counts = statement.executeBatch();
connection.commit();
```

JDBC BatchUpdateException:

バッチ更新の重要な考慮事項は、executeBatch メソッドの呼び出しが失敗したときに、どのようなアクションが取られるかということです。この場合、新しいタイプの例外である BatchUpdateException がスローされます。 BatchUpdateException は SQLException のサブクラスで、普段、メッセージや SQLState、ベンダー・コードを受信するために呼び出している同様のメソッドのすべてを呼び出すことができます。

BatchUpdateException は、整数配列を戻す getUpdateCounts メソッドも提供しています。この整数配列には、バッチ内で障害が発生する時点までのすべてのステートメントによって処理された更新数が含まれています。配列の長さは、バッチのどのステートメントが失敗したのかを示します。たとえば、例外によって返された配列の長さが 3 であった場合は、バッチ内の 4 番目のステートメントが失敗したことを示しています。そのため、戻された単一の BatchUpdateException オブジェクトから、成功したすべてのステートメントの更新数、どのステートメントが失敗したのか、およびその障害に関するすべての情報を判別することができます。

バッチ更新の処理の標準的なパフォーマンスは、各ステートメントを別々に処理したときのパフォーマンスと同等です。バッチ更新の最適化サポートについて詳しくは、ブロック挿入サポートを参照してください。コードを記述する際は、将来のパフォーマンスの最適化の利点を得るため、現在でもこの新しいモデルを使用するべきです。

注: JDBC 2.1 仕様では、バッチ更新の例外条件を処理する方法として別のオプションが提供されています。 JDBC 2.1 では、バッチ項目が失敗した後もバッチの処理を継続するモデルが導入されています。特殊更新数は、失敗した各項目から戻された整数の更新数の配列の中に格納されています。これにより、大規模なバッチの一部の項目が失敗したとしても、処理を継続することができます。この操作の 2 つのモードについて詳しくは、JDBC 2.1 または JDBC 3.0 の仕様を参照してください。デフォルトでは、ネイティブ JDBC ドライバーは JDBC 2.0 定義を使用します。ドライバーは、接続を確立するために DriverManager を使用するときに使われる Connection プロパティーを提供しています。また、ドライバーは接続を確立するために DataSource を使用するときに使われる DataSource プロパティーも提供しています。これらのプロパティーを使うと、バッチ操作中に発生した障害にどのように処理するか、アプリケーションが選択することができます。

JDBC でのブロック挿入:

ブロック挿入操作を使用して、データベース・テーブルに一度に複数の行を挿入することができます。

ブロック挿入は、IBM i の操作の特殊なタイプで、データベースのテーブルに一度に複数の行を挿入する、高度に最適化された方法を提供します。ブロック挿入は、バッチ更新のサブセットと考えることができます。バッチ更新は任意の形式で更新要求ができますが、ブロック挿入は指定された形式です。しかし、バッチ更新のブロック挿入タイプは共通です。ネイティブ JDBC ドライバーはこの機能の利点を得るために変更が加えられました。

ブロック挿入サポートを使用するときに生じるシステム制約事項により、ネイティブ JDBC ドライバーのデフォルト設定では、ブロック挿入は使用不可になっています。これは、Connection プロパティーまたは DataSource プロパティーを通して使用可能にすることができます。ブロック挿入を使用するときは、ユーザーの利益のためにそれらの制約事項の多くをチェックおよびハンドルすることができますが、いくつかの制約事項ではそれができません。これが、デフォルト設定ではブロック挿入サポートがオフになっている理由です。制約事項のリストは次のとおりです。

- SQL ステートメントでは、INSERT ステートメントは SUBSELECT と共にではなく、 VALUES 文節と共に使用しなければなりません。 JDBC ドライバーはこの制約を認識し、適切な処理方針を取ります。
- PreparedStatement を必ず使用しなければならず、これによって Statement オブジェクトの最適化サポートはなくなります。 JDBC ドライバーはこの制約を認識し、適切な処理方針を取ります。
- SQL ステートメントは、テーブル内のすべての列に対するパラメーター・マーカーを指定しなければなりません。これにより、列に定数値を使用するか、データベースが挿入時に任意の列にデフォルト値を挿入できるようにするか、どちらかが使用できません。 JDBC ドライバーは、 SQL ステートメント内のパラメーター・マーカーの指定をテストするためのメカニズムを持っていません。最適化されたブロック挿入を実行するためにプロパティーを設定し、 SQL ステートメント内でデフォルト値や定数値の使用を差し控えなかった場合は、最終的なデータベース・テーブル内の値は不正なものになります。
- 接続はローカル・システムに対するものでなければなりません。ブロック挿入操作では DRDA がサポートされていないため、リモート・システムへアクセスするために DRDA を使った接続は使用できません。 JDBC ドライバーは、ローカル・システムへの接続をテストするためのメカニズムを持っていません。最適化されたブロック挿入を実行するためのプロパティーを設定し、リモート・システムへの接続を行おうとすると、バッチ更新の処理は失敗します。

以下のコード例は、ブロック挿入処理サポートを使用可能にする方法を示しています。このコードと、ブロック挿入サポートを使用していないバージョンとの違いは、接続 URL に `use block insert=true` が追加されているかどうかだけです。

例: ブロック挿入処理

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
// Create a database connection
Connection c = DriverManager.getConnection("jdbc:db2:*local;use block insert=true");
BigDecimal bd = new BigDecimal("123456");

// Create a PreparedStatement to insert into a table with 4 columns
PreparedStatement ps =
  c.prepareStatement("insert into cujosql.xxx values(?, ?, ?, ?)");

// Start timing...
for (int i = 1; i <= 10000; i++) {
  ps.setInt(1, i);                        // Set all the parameters for a row
  ps.setBigDecimal(2, bd);
  ps.setBigDecimal(3, bd);
  ps.setBigDecimal(4, bd);
  ps.addBatch();                          //Add the parameters to the batch
}

// Process the batch
int[] counts = ps.executeBatch();

// End timing...
```

同様のテスト・ケースにおいては、ブロック挿入を使用しなかった場合に同様の処理をする場合より、ブロック挿入処理を行った場合のほうが数倍処理が速くなります。たとえば、前述のコードでのテストでは、ブロック挿入を使うと 9 倍の速度になりました。オブジェクトの代わりにプリミティブ・タイプのみを使ったケースでは、最大で 16 倍まで速くなりました。相当数の処理が行われているアプリケーションでは、期待できる効果もそれ相応のものになると考えられます。

拡張データ・タイプ

拡張 SQL3 データ・タイプでは、非常に幅広い柔軟性が提供されています。これは、シリアル化された Java オブジェクト、XML (Extensible Markup Language) 文書、および音楽、製品の画像、従業員の写真やムービー・クリップといったマルチメディア・データを格納するのに理想的です。 Java Database Connectivity (JDBC) 2.0 およびそれ以降では、これらのデータ・タイプを SQL99 標準の一部として処理するためのサポートが提供されています。

特殊タイプ

特殊タイプは、標準データベース・タイプに基づくユーザー定義タイプです。たとえば、内部的に CHAR(9) の社会保障番号タイプ、SSN を定義できます。次の SQL ステートメントは、特殊タイプを作成します。

```
CREATE DISTINCT TYPE CUJOSQL.SSN AS CHAR(9)
```

特殊タイプは常に、組み込みデータ・タイプにマップされます。 SQL を使用する際、どのように、またいつ特殊タイプを使用するかについては、「SQL のリファレンス・マニュアル」を参照してください。

JDBC で特殊タイプを使用するには、その基となったタイプにアクセスする方法と同じ方法でアクセスします。新しいメソッドである getUDTs メソッドを使って、システム上でどの特殊タイプが使用可能かを調べることができます。「例: 特殊タイプ」のプログラムは、以下について示します。

- 特殊タイプの作成。
- 特殊タイプを使ったテーブルの作成。
- 特殊タイプ・パラメーターを設定するための PreparedStatement の使用。
- 特殊タイプを戻すための ResultSet の使用。
- 特殊タイプについて調べるために getUDTs を呼び出すためのメタデータ・アプリケーション・プログラミング・インターフェース (API) の使用。

詳細は、「例: 特殊タイプ」のサブトピックを参照してください。このサブトピックには、特殊タイプを使用して実行できるさまざまな共通タスクについての説明があります。

ラージ・オブジェクト

ラージ・オブジェクト (LOB) には、3 つのタイプがあります。

- バイナリー・ラージ・オブジェクト (BLOB)
- 文字ラージ・オブジェクト (CLOB)
- 2 バイト文字ラージ・オブジェクト (DBCLOB)

DBCLOB は、文字データの内部記憶域表現ということを除けば、CLOB と同じです。 Java および JDBC はすべての文字データを Unicode として外部化しますが、これは JDBC では CLOB でのみサポートされています。 DBCLOB の動作は、JDBC の観点からすると、CLOB サポートと交換可能です。

バイナリー・ラージ・オブジェクト

多くの場合、バイナリー・ラージ・オブジェクト (BLOB) 列は、大きなデータを格納できる CHAR FOR BIT DATA 列と同等です。これらの列は、変換されていないバイト・データのストリームと見なされ、どんなデータでも保管することができます。 BLOB 列はしばしば、シリアル化された Java オブジェクト、ピクチャー、音楽、および他のバイナリー・データを保管するために使用されます。

BLOB は他の標準データベース・タイプと同じ方法で使用することができます。ストアード・プロシージャーに渡したり、PreparedStatement 内で使用したり、ResultSet 内で更新することができます。 PreparedStatement クラスには BLOB をデータベースに渡すための setBlob メソッドがあり、ResultSet クラスは BLOB をデータベースから取得するための getBlob クラスが追加されています。 BLOB は Java プログラムの中では、JDBC インターフェースの BLOB オブジェクトとして扱われます。

文字ラージ・オブジェクト

文字ラージ・オブジェクト (CLOB) は、BLOB に文字データを補足するものです。変換なしでデータベースにデータを保管するのではなく、データをテキストとしてデータベースに保管し、CHAR 列と同様の方法で処理されます。 BLOB と同様に、JDBC 2.0 には CLOB と直接やり取りするための機能が提供されています。 PreparedStatement インターフェースには setClob メソッドが含まれており、ResultSet インターフェースには getClob メソッドが含まれています。

BLOB および CLOB 列は CHAR FOR BIT DATA および CHAR 列と似た動作をしますが、これは外部からのユーザーの視点でどのように動作するかを概念的に示したものです。内部的には、これらは別物です。巨大なサイズになることもあり得るラージ・オブジェクト (LOB) 列では、データは一般的に間接的に処理されます。たとえば、データベースから行ブロックをフェッチしたときは、LOB のブロックを

ResultSet に移動することはありません。その代わりに、LOB ロケーターと呼ばれるポインター (これは、4 バイトの整数) を ResultSet に移動します。しかし、JDBC 内で LOB を処理する際には、ロケーターについて知っておく必要はありません。

データ・リンク

データ・リンクは、データベースからデータベース外に保管されたファイルへの論理参照を含んだ、カプセル化された値です。データ・リンクは、JDBC 2.0 かそれ以前を使用しているか、JDBC 3.0 かそれ以降を使用しているかによって、JDBC から 2 つの異なる方法で扱われ、使用されます。

サポートされていない SQL3 データ・タイプ

この他にも、既に定義され、JDBC API によってサポートが提供されている SQL3 データ・タイプがあります。 ARRAY、REF、および STRUCT です。現在のところ、IBM i でこれらのタイプはサポートされていません。そのため、JDBC ドライバーはそれらタイプに対するいかなる形式のサポートも提供していません。

関連資料

148 ページの『例: 特殊タイプ』
以下に、特殊タイプの使用法の例を示します。

BLOB を使ったコードを記述する:

Java Database Connectivity (JDBC) アプリケーション・プログラミング・インターフェース (API) を介し、データベースのバイナリー・ラージ・オブジェクト (BLOB) 列を使って達成できる、数多くのタスクがあります。以下のトピックでは、これらのタスクについて簡単に説明し、その使い方の例を示します。

データベースからの BLOB の読み取り、およびデータベースへの BLOB の挿入

JDBC API では、データベースからの BLOB の取り出し、およびデータベースへの BLOB の書き込みにはいくつかの方法があります。しかし、BLOB オブジェクトを作成するための標準化された方法はありません。これはデータベースが BLOB を完全に利用できる場合は問題ではありませんが、JDBC 経由で最初から BLOB を処理したい場合に問題を引き起こす可能性があります。 JDBC API の BLOB および CLOB インターフェース用のコンストラクターを定義する代わりに、他のタイプとしてデータベースに BLOB を直接格納したり、BLOB をデータベースから直接取り出すサポートが提供されています。たとえば、setBinaryStream メソッドを使うと、データベース内の BLOB タイプの列を処理できます。『例: BLOB』のトピックには、データベースに BLOB を書き込んだり、データベースから BLOB を取り出したりするための一般的な方法のいくつかが示されています。

BLOB オブジェクト API の処理

BLOB は JDBC の中で、数多くのドライバーによってインプリメンテーションが提供されたインターフェースとして定義されています。このインターフェースには、BLOB オブジェクトと対話するために使用できる一連のメソッドがあります。『例: BLOB の使用』のトピックには、この API を使って実行できる一般的なタスクのいくつかが示されています。BLOB オブジェクトで使用できるすべてのメソッドのリストは、JDBC Javadoc を調べてください。

BLOB の更新のために JDBC 3.0 サポートを使用する

JDBC 3.0 では、LOB オブジェクトへ変更を加える機能がサポートされています。これらの変更は、データベース内の BLOB 列に保管することができます。『例: BLOB の更新』のトピックには、JDBC 3.0 の BLOB サポートを使って実行できる一般的なタスクのいくつかが示されています。

関連資料

『例: BLOB』
以下は、 BLOB をデータベースに書き込んだり、データベースから検索したりする方法の例です。

141 ページの『例: BLOB の更新』
以下は、Java アプリケーション中で BLOB を更新する方法の例です。

例: BLOB:

以下は、 BLOB をデータベースに書き込んだり、データベースから検索したりする方法の例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
/////////////////////////////////////////
// PutGetBlobs is an example application
// that shows how to work with the JDBC
// API to obtain and put BLOBs to and from
// database columns.
//
// The results of running this program
// are that there are two BLOB values
// in a new table. Both are identical
// and contain 500k of random byte
// data.
/////////////////////////////////////////
import java.sql.*;
import java.util.Random;

public class PutGetBlobs {
   public static void main(String[] args)
   throws SQLException
   {
       // Register the native JDBC driver.
       try {
          Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
      } catch (Exception e) {
          System.exit(1);  // Setup error.
      }

      // Establish a Connection and Statement with which to work.
      Connection c = DriverManager.getConnection("jdbc:db2:*local");
      Statement s = c.createStatement();

      // Clean up any previous run of this application.
      try {
          s.executeUpdate("DROP TABLE CUJOSQL.BLOBTABLE");
      } catch (SQLException e) {
          // Ignore it - assume the table did not exist.
      }

      // Create a table with a BLOB column. The default BLOB column
      // size is 1 MB.
      s.executeUpdate("CREATE TABLE CUJOSQL.BLOBTABLE (COL1 BLOB)");

      // Create a PreparedStatement object that allows you to put
      // a new Blob object into the database.
      PreparedStatement ps = c.prepareStatement("INSERT INTO CUJOSQL.BLOBTABLE VALUES(?)");

      // Create a big BLOB value...
      Random random = new Random ();
      byte [] inByteArray = new byte[500000];
      random.nextBytes (inByteArray);

      // Set the PreparedStatement parameter. Note: This is not
```

```
        // portable to all JDBC drivers. JDBC drivers do not have
        // support when using setBytes for BLOB columns. This is used to
        // allow you to generate new BLOBs. It also allows JDBC 1.0
        // drivers to work with columns containing BLOB data.
        ps.setBytes(1, inByteArray);

        // Process the statement, inserting the BLOB into the database.
        ps.executeUpdate();

        // Process a query and obtain the BLOB that was just inserted out
        // of the database as a Blob object.
        ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.BLOBTABLE");
        rs.next();
        Blob blob = rs.getBlob(1);


        // Put that Blob back into the database through
        // the PreparedStatement.
        ps.setBlob(1, blob);
        ps.execute();

        c.close(); // Connection close also closes stmt and rs.
    }
}
```

例: BLOB の更新:

以下は、Java アプリケーション中で BLOB を更新する方法の例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
/////////////////////////////////////////
// UpdateBlobs is an example application
// that shows some of the APIs providing
// support for changing Blob objects
// and reflecting those changes to the
// database.
//
// This program must be run after
// the PutGetBlobs program has completed.
/////////////////////////////////////////
import java.sql.*;

public class UpdateBlobs {
    public static void main(String[] args)
    throws SQLException
    {
        // Register the native JDBC driver.
        try {
           Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
        } catch (Exception e) {
           System.exit(1);  // Setup error.
        }

        Connection c = DriverManager.getConnection("jdbc:db2:*local");
        Statement s = c.createStatement();

        ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.BLOBTABLE");

        rs.next();
        Blob blob1 = rs.getBlob(1);
        rs.next();
        Blob blob2 = rs.getBlob(1);
```

```
        // Truncate a BLOB.
        blob1.truncate((long) 150000);
        System.out.println("Blob1's new length is " + blob1.length());

        // Update part of the BLOB with a new byte array.
        // The following code obtains the bytes that are at
        // positions 4000-4500 and set them to positions 500-1000.

        // Obtain part of the BLOB as a byte array.
        byte[] bytes = blob1.getBytes(4000L, 4500);

        int bytesWritten = blob2.setBytes(500L, bytes);

        System.out.println("Bytes written is " + bytesWritten);

        // The bytes are now found at position 500 in blob2
        long startInBlob2 = blob2.position(bytes, 1);

        System.out.println("pattern found starting at position " + startInBlob2);

        c.close(); // Connection close also closes stmt and rs.
    }
}
```

例: BLOB の使用:

以下は、Java アプリケーション中で BLOB を使用する方法の例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
/////////////////////////////////////////
// UseBlobs is an example application
// that shows some of the APIs associated
// with Blob objects.
//
// This program must be run after
// the PutGetBlobs program has completed.
/////////////////////////////////////////
import java.sql.*;

public class UseBlobs {
    public static void main(String[] args)
    throws SQLException
    {
        // Register the native JDBC driver.
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
        } catch (Exception e) {
            System.exit(1);  // Setup error.
        }

        Connection c = DriverManager.getConnection("jdbc:db2:*local");
        Statement s = c.createStatement();

        ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.BLOBTABLE");

        rs.next();
        Blob blob1 = rs.getBlob(1);
        rs.next();
        Blob blob2 = rs.getBlob(1);


        // Determine the length of a LOB.
        long end = blob1.length();
        System.out.println("Blob1 length is " + blob1.length());
```

```
        // When working with LOBs, all indexing that is related to them
        // is 1-based, and is not 0-based like strings and arrays.
        long startingPoint = 450;
        long endingPoint = 500;

        // Obtain part of the BLOB as a byte array.
        byte[] outByteArray = blob1.getBytes(startingPoint, (int)endingPoint);

        // Find where a sub-BLOB or byte array is first found within a
        // BLOB. The setup for this program placed two identical copies of
        // a random BLOB into the database. Thus, the start position of the
        // byte array extracted from blob1 can be found in the starting
        // position in blob2. The exception would be if there were 50
        // identical random bytes in the LOBs previously.
        long startInBlob2 = blob2.position(outByteArray, 1);

        System.out.println("pattern found starting at position " + startInBlob2);

        c.close(); // Connection close closes stmt and rs too.
    }
}
```

CLOB を使ったコードを記述する:

Java Database Connectivity (JDBC) アプリケーション・プログラミング・インターフェース (API) を介し、データベースの CLOB および DBCLOB 列を使って達成できる、数多くのタスクがあります。以下のトピックでは、これらのタスクについて簡単に説明し、その使い方の例を示します。

データベースからの CLOB の読み取り、およびデータベースへの CLOB の挿入

JDBC API では、データベースからの CLOB の取り出し、およびデータベースへの CLOB の書き込みにはいくつかの方法があります。しかし、CLOB オブジェクトを作成するための標準化された方法はありません。これはデータベースが CLOB を完全に利用できる場合は問題ではありませんが、JDBC 経由で最初から CLOB を処理したい場合に問題を引き起こす可能性があります。 JDBC API の BLOB および CLOB インターフェース用のコンストラクターを定義する代わりに、他のタイプとしてデータベースに CLOB を直接格納したり、CLOB をデータベースから直接取り出すサポートが提供されています。たとえば、setCharacterStream メソッドを使うと、データベース内の CLOB タイプの列を処理できます。『例: CLOB』のトピックには、データベースに CLOB を書き込んだり、データベースから CLOB を取り出したりするための一般的な方法のいくつかが示されています。

CLOB オブジェクト API の処理

CLOB は JDBC の中で、数多くのドライバーによってインプリメンテーションが提供されたインターフェースとして定義されています。このインターフェースには、CLOB オブジェクトと対話するために使用できる一連のメソッドがあります。『例: CLOB の使用』のトピックには、この API を使って実行できる一般的なタスクのいくつかが示されています。CLOB オブジェクトで使用できるすべてのメソッドのリストは、JDBC Javadoc を調べてください。

CLOB の更新のために JDBC 3.0 サポートを使用する

JDBC 3.0 では、LOB オブジェクトへ変更を加える機能がサポートされています。これらの変更は、データベース内の CLOB 列に保管することができます。『例: CLOB の更新』のトピックには、JDBC 3.0 の CLOB サポートを使って実行できる一般的なタスクのいくつかが示されています。

関連資料

『例: CLOB』
以下は、 CLOB をデータベースに書き込んだり、データベースから検索したりする方法の例です。

146 ページの『例: CLOB の使用』
以下は、Java アプリケーション中で CLOB を使用する方法の例です。

145 ページの『例: CLOB の更新』
以下は、Java アプリケーション中で CLOB を更新する方法の例です。

例: CLOB:

以下は、 CLOB をデータベースに書き込んだり、データベースから検索したりする方法の例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
/////////////////////////////////////////
// PutGetClobs is an example application
// that shows how to work with the JDBC
// API to obtain and put CLOBs to and from
// database columns.
//
// The results of running this program
// are that there are two CLOB values
// in a new table. Both are identical
// and contain about 500k of repeating
// text data.
/////////////////////////////////////////
import java.sql.*;

public class PutGetClobs {
   public static void main(String[] args)
   throws SQLException
   {
       // Register the native JDBC driver.
       try {
          Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
      } catch (Exception e) {
          System.exit(1);  // Setup error.
      }

      // Establish a Connection and Statement with which to work.
      Connection c = DriverManager.getConnection("jdbc:db2:*local");
      Statement s = c.createStatement();

      // Clean up any previous run of this application.
      try {
          s.executeUpdate("DROP TABLE CUJOSQL.CLOBTABLE");
      } catch (SQLException e) {
          // Ignore it - assume the table did not exist.
      }

      // Create a table with a CLOB column. The default CLOB column
      // size is 1 MB.
      s.executeUpdate("CREATE TABLE CUJOSQL.CLOBTABLE (COL1 CLOB)");

      // Create a PreparedStatement object that allow you to put
      // a new Clob object into the database.
      PreparedStatement ps = c.prepareStatement("INSERT INTO CUJOSQL.CLOBTABLE VALUES(?)");

      // Create a big CLOB value...
      StringBuffer buffer = new StringBuffer(500000);
      while (buffer.length() < 500000) {
          buffer.append("All work and no play makes Cujo a dull boy.");
```

```
        }
        String clobValue = buffer.toString();

        // Set the PreparedStatement parameter. This is not
        // portable to all JDBC drivers. JDBC drivers do not have
        // to support setBytes for CLOB columns. This is done to
        // allow you to generate new CLOBs. It also
        // allows JDBC 1.0 drivers a way to work with columns containing
        // Clob data.
        ps.setString(1, clobValue);

        // Process the statement, inserting the clob into the database.
        ps.executeUpdate();

        // Process a query and get the CLOB that was just inserted out of the
        // database as a Clob object.
        ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.CLOBTABLE");
        rs.next();
        Clob clob = rs.getClob(1);


        // Put that Clob back into the database through
        // the PreparedStatement.
        ps.setClob(1, clob);
        ps.execute();

        c.close(); // Connection close also closes stmt and rs.
    }
}
```

例: CLOB の更新:

以下は、Java アプリケーション中で CLOB を更新する方法の例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
/////////////////////////////////////////
// UpdateClobs is an example application
// that shows some of the APIs providing
// support for changing Clob objects
// and reflecting those changes to the
// database.
//
// This program must be run after
// the PutGetClobs program has completed.
/////////////////////////////////////////
import java.sql.*;

public class UpdateClobs {
    public static void main(String[] args)
    throws SQLException
    {
        // Register the native JDBC driver.
        try {
           Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
        } catch (Exception e) {
           System.exit(1);  // Setup error.
        }

        Connection c = DriverManager.getConnection("jdbc:db2:*local");
        Statement s = c.createStatement();

        ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.CLOBTABLE");

        rs.next();
```

```
      Clob clob1 = rs.getClob(1);
      rs.next();
      Clob clob2 = rs.getClob(1);


      // Truncate a CLOB.
      clob1.truncate((long) 150000);
      System.out.println("Clob1's new length is " + clob1.length());

      // Update a portion of the CLOB with a new String value.
      String value = "Some new data for once";
      int charsWritten = clob2.setString(500L, value);

      System.out.println("Characters written is " + charsWritten);

      // The bytes can be found at position 500 in clob2
      long startInClob2 = clob2.position(value, 1);

      System.out.println("pattern found starting at position " + startInClob2);

      c.close(); // Connection close also closes stmt and rs.
   }
}
```

例: CLOB の使用:

以下は、Java アプリケーション中で CLOB を使用する方法の例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
/////////////////////////////////////////
// UpdateClobs is an example application
// that shows some of the APIs providing
// support for changing Clob objects
// and reflecting those changes to the
// database.
//
// This program must be run after
// the PutGetClobs program has completed.
/////////////////////////////////////////
import java.sql.*;

public class UseClobs {
   public static void main(String[] args)
   throws SQLException
   {
       // Register the native JDBC driver.
       try {
          Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
      } catch (Exception e) {
          System.exit(1);  // Setup error.
      }

      Connection c = DriverManager.getConnection("jdbc:db2:*local");
      Statement s = c.createStatement();

      ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.CLOBTABLE");

      rs.next();
      Clob clob1 = rs.getClob(1);
      rs.next();
      Clob clob2 = rs.getClob(1);


      // Determine the length of a LOB.
```

```
        long end = clob1.length();
        System.out.println("Clob1 length is " + clob1.length());

        // When working with LOBs, all indexing that is related to them
        // is 1-based, and not 0-based like strings and arrays.
        long startingPoint = 450;
        long endingPoint = 50;

        // Obtain part of the CLOB as a byte array.
        String outString = clob1.getSubString(startingPoint, (int)endingPoint);
        System.out.println("Clob substring is " + outString);

        // Find where a sub-CLOB or string is first found within a
        // CLOB. The setup for this program placed two identical copies of
        // a repeating CLOB into the database. Thus, the start position of the
        // string extracted from clob1 can be found in the starting
        // position in clob2 if the search begins close to the position where
    // the string starts.
        long startInClob2 = clob2.position(outString, 440);

        System.out.println("pattern found starting at position " + startInClob2);

        c.close(); // Connection close also closes stmt and rs.
    }
}
```

データ・リンクを使ったコードを記述する:

データ・リンクをどのように使って処理するかどうかは、どのリリースを使用して処理するかに依存しています。 JDBC 3.0 では、getURL および putURL メソッドを使って、データ・リンク列を直接処理する機能がサポートされています。

以前のバージョンの JDBC の場合は、ストリング列のようにデータ・リンク列を処理しなければなりません。現在のところ、データベースのデータ・リンクと文字データ・タイプの自動変換はサポートされていません。その結果として、SQL ステートメント内である型キャスティングを実行する必要があります。

例: Datalink:

このサンプル・アプリケーションでは、JDBC API を使用して Datalink データベース列をハンドルする方法を示します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
/////////////////////////////////////////
// PutGetDatalinks is an example application
// that shows how to use the JDBC
// API to handle datalink database columns.
/////////////////////////////////////////
import java.sql.*;
import java.net.URL;
import java.net.MalformedURLException;

public class PutGetDatalinks {
    public static void main(String[] args)
    throws SQLException
    {
        // Register the native JDBC driver.
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
        } catch (Exception e) {
            System.exit(1);  // Setup error.
        }
```

```
        // Establish a Connection and Statement with which to work.
        Connection c = DriverManager.getConnection("jdbc:db2:*local");
        Statement s = c.createStatement();

        // Clean up any previous run of this application.
        try {
            s.executeUpdate("DROP TABLE CUJOSQL.DLTABLE");
        } catch (SQLException e) {
            // Ignore it - assume the table did not exist.
        }

        // Create a table with a datalink column.
        s.executeUpdate("CREATE TABLE CUJOSQL.DLTABLE (COL1 DATALINK)");

        // Create a PreparedStatement object that allows you to add
        // a new datalink into the database. Since conversing
        // to a datalink cannot be accomplished directly in the database, you
        // can code the SQL statement to perform the explicit conversion.
        PreparedStatement ps = c.prepareStatement("INSERT INTO CUJOSQL.DLTABLE
                                            VALUES(DLVALUE( CAST(? AS VARCHAR(100))))");

        // Set the datalink. This URL points you to a topic about
        // the new features of JDBC 3.0.
        ps.setString (1, "http://www.ibm.com/developerworks/java/library/j-jdbcnew/index.html");
        // Process the statement, inserting the CLOB into the database.
        ps.executeUpdate();

        // Process a query and obtain the CLOB that was just inserted out of the
        // database as a Clob object.
        ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.DLTABLE");
        rs.next();
        String datalink = rs.getString(1);


        // Put that datalink value into the database through
        // the PreparedStatement. Note: This function requires JDBC 3.0
        // support.
        /*
        try {
            URL url = new URL(datalink);
            ps.setURL(1, url);
            ps.execute();
        } catch (MalformedURLException mue) {
            // Handle this issue here.
        }

        rs = s.executeQuery("SELECT * FROM CUJOSQL.DLTABLE");
        rs.next();
        URL url = rs.getURL(1);
        System.out.println("URL value is " + url);
        */

        c.close(); // Connection close also closes stmt and rs.
    }
}
```

例: 特殊タイプ:

以下に、特殊タイプの使用法の例を示します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
/////////////////////////////////////////
// This example program shows examples of
// various common tasks that can be done
// with distinct types.
/////////////////////////////////////////
import java.sql.*;

public class Distinct {
    public static void main(String[] args)
    throws SQLException
    {
        // Register the native JDBC driver.
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
        } catch (Exception e) {
            System.exit(1);  // Setup error.
        }

        Connection c = DriverManager.getConnection("jdbc:db2:*local");
        Statement s = c.createStatement();

        // Clean up any old runs.
        try {
            s.executeUpdate("DROP TABLE CUJOSQL.SERIALNOS");
        } catch (SQLException e) {
            // Ignore it and assume the table did not exist.
        }

        try {
            s.executeUpdate("DROP DISTINCT TYPE CUJOSQL.SSN");
        } catch (SQLException e) {
            // Ignore it and assume the table did not exist.
        }

        // Create the type, create the table, and insert a value.
        s.executeUpdate("CREATE DISTINCT TYPE CUJOSQL.SSN AS CHAR(9)");
        s.executeUpdate("CREATE TABLE CUJOSQL.SERIALNOS (COL1 CUJOSQL.SSN)");

        PreparedStatement ps = c.prepareStatement("INSERT INTO CUJOSQL.SERIALNOS VALUES(?)");
        ps.setString(1, "399924563");
        ps.executeUpdate();
        ps.close();

        // You can obtain details about the types available with new metadata in
        // JDBC 2.0
        DatabaseMetaData dmd = c.getMetaData();

        int types[] = new int[1];
        types[0] = java.sql.Types.DISTINCT;

        ResultSet rs = dmd.getUDTs(null, "CUJOSQL", "SSN", types);
        rs.next();
        System.out.println("Type name " + rs.getString(3) +
                           " has type " + rs.getString(4));


        // Access the data you have inserted.
        rs = s.executeQuery("SELECT COL1 FROM CUJOSQL.SERIALNOS");
        rs.next();
        System.out.println("The SSN is " + rs.getString(1));

        c.close(); // Connection close also closes stmt and rs.
    }
}
```

JDBC RowSets

RowSet は、元は Java Database Connectivity (JDBC) 2.0 Optional Package に追加されていました。もっとよく知られたいくつかの JDBC 仕様のインターフェースとは異なり、RowSet 仕様は、実際のインプリメンテーションの仕様というよりは、フレームワークの仕様として設計されています。 RowSet インターフェースは、すべての RowSets に含まれているコア機能のセットを定義します。 RowSet インプリメンテーションのプロバイダーは、特定の問題スペースでのその必要を満たすために必要な機能をかなり自由に定義できます。

RowSet の特性:

RowSet によって特定のプロパティーの条件が満たされるよう要求できます。共通プロパティーには、結果の RowSet によってサポートされるインターフェースのセットが含まれます。

RowSet は ResultSet

RowSet インターフェースは ResultSet インターフェースを拡張するものです。このことは、RowSet には ResultSet が実行できるすべての機能を実行する能力があるということを意味します。たとえば、RowSet はスクロールと更新が可能です。

RowSet はデータベースから切断可能

RowSet には、以下の 2 つのカテゴリーがあります。

接続 接続 RowSet は、データが読み込まれている間、基となるデータベースとの内部接続を常に開いており、ResultSet のインプリメンテーションのラッパーとして機能します。

切断 切断 RowSet は、常にそのデータ・ソースへの接続を保持している必要はありません。切断 RowSet は、データベースから切り離してさまざまな用途に使用し、その後、加えられた変更を反映するためにデータベースに再接続することができます。

RowSets は JavaBeans™ のコンポーネント

RowSet には、JavaBeans のイベント処理モデルに基づくイベント処理のサポートがあります。これらには、設定できるプロパティーもあります。これらのプロパティーは、RowSet が以下のことを実行するときに使用されます。

- データベースへの接続を確立する。
- SQL ステートメントを処理する。
- RowSet が表すデータのフィーチャーを判別し、RowSet オブジェクトの内部フィーチャーを処理する。

RowSet は逐次化可能

RowSet は、ネットワーク接続上をフローできるように逐次化および並列化したり、フラット・ファイル (つまり、ワード・プロセッシングや他の構造文字をもたないテキスト文書) に書き込んだりすることができます。

DB2CachedRowSet:

DB2CachedRowSet オブジェクトは切断された RowSet で、データベースに接続せずに使用できることを意味します。そのインプリメンテーションは、CachedRowSet の記述に厳密に従っています。
DB2CachedRowSet は ResultSet のデータ行のためのコンテナーです。 DB2CachedRowSet はそのデータを保持しており、明示的にデータをデータベースから読み込んだり、書き込んだりするときでなければ、データベースへの接続を保つ必要がありません。

***DB2CachedRowSet* の使用:**

DB2CachedRowSet オブジェクトは切断および逐次化することができるため、全体の JDBC ドライバーを動作させることが必ずしも常に実際的でない環境 (たとえば、PDA および Java が使用できる携帯電話など) で有用です。

DB2CachedRowSet オブジェクトはメモリー内に格納され、そのデータは既に取得されているため、アプリケーションに対して、スクロール可能な ResultSet の高度に最適化された形式として提供できます。しかし、スクロール可能な DB2 ResultSet はしばしば、パフォーマンス面での弱点となります。それは、ランダムな移動が JDBC ドライバーのデータの行をキャッシュする機能と干渉してしまうためです。 RowSet にはこの問題はありません。

DB2CachedRowSet には、新しい RowSet を作成する 2 つのメソッドが提供されています。

- createCopy メソッドは、コピーされた同一の新しい RowSet を作成します。
- createShared メソッドは、オリジナルと同一の基礎データを共用する新しい RowSet を作成します。

クライアントに共通の ResultSet を配布するには、createCopy メソッドを使用できます。テーブル・データが変更されない場合、RowSet のコピーを作成して各クライアントに配布することは、毎回データベースに対して照会を実行するよりも効率的です。

createShared メソッドを使うと、同一のデータに複数のユーザーがアクセスできるようにしてデータベースのパフォーマンスを向上させることができます。たとえば、顧客が接続したとき、ホーム・ページ上で上位 20 位のベストセラーの商品を表示する Web サイトを想定してみましょう。メイン・ページ上の情報は定期的に更新する必要がありますが、顧客がメイン・ページを訪問するたびに良く売れている商品を取り出す照会を実行するのは実際的ではありません。 createShared メソッドを使うと、何度も照会を処理したり、膨大な量の情報をメモリーに保管しておくことなく、事実上、各顧客用の「カーソル」を作成することができます。必要があれば、良く売れている商品を検索する照会を再度実行することもできます。共用カーソルを作成するために使用した RowSet に新しいデータを取り込み、サーブレットがこれらを利用できます。

DB2CachedRowSets は遅延処理機能を提供しています。この機能を利用すると、複数の照会要求をグループ化し、データベースに対する 1 つの要求として処理することができます。この機能を利用しない場合に発生するであろうデータベースへの計算負荷をいくらかを軽減するためには、152 ページの『DB2CachedRowSet の作成とデータ取り込み』のトピックを参照してください。

RowSet は、データベースにデータを戻すため、また、加えた変更を元に戻したり、加えられたすべての変更を表示する機能をサポートするために、加えられた変更を注意深く追跡する必要があります。たとえば、RowSet に対して削除された行をフェッチするための、showDeleted メソッドがあります。また、cancelRowInsert および cancelRowDelete メソッドは、ユーザーが行の挿入や削除を行った後、きちんと元に戻します。

DB2CachedRowSet オブジェクトは、イベント処理サポート、および RowSet またはその一部を Java コレクションに変換するための toCollection メソッドにより、他の Java API との良好な相互運用性を提供しています。

DB2CachedRowSet のイベント処理サポートは、グラフィカル・ユーザー・インターフェース (GUI) アプリケーションの表示制御や、RowSet に加えられた変更の情報のロギング、または RowSet 以外のソースに加えられた変更に関する情報の検索などに使用できます。詳細は170 ページの『DB2JdbcRowSet イベント』を参照してください。

イベント・モデルおよびイベント処理については、 168 ページの『DB2JdbcRowSet』 を参照してください。このサポートは、どちらのタイプの RowSet でも同様に動作します。

***DB2CachedRowSet* の作成とデータ取り込み:**

DB2CachedRowSet にデータを配置する方法は、いくつか存在します。populate メソッド、DataSources での DB2CachedRowSet プロパティー、DB2CachedRowSet プロパティーと JDBC URL、setConnection (Connection) メソッド、execute(Connection) メソッド、および execute(int) メソッドなどです。

populate メソッドを使用する

DB2CachedRowSets には、DB2 ResultSet オブジェクトから RowSet にデータを書き込むための populate メソッドがあります。以下に、この方法の例を示します。

例: populate メソッドを使用する

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
// Establish a connection to the database.
Connection conn = DriverManager.getConnection("jdbc:db2:*local");

// Create a statement and use it to perform a query.
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("select col1 from cujosql.test_table");

// Create and populate a DB2CachedRowSet from it.
DB2CachedRowSet crs = new DB2CachedRowSet();
crs.populate(rs);

// Note: Disconnect the ResultSet, Statement,
// and Connection used to create the RowSet.
rs.close();
stmt.close();
conn.close();

// Loop through the data in the RowSet.
while (crs.next()) {
  System.out.println("v1 is " + crs.getString(1));
}

crs.close();
```

DB2CachedRowSet プロパティーと DataSources を使用する

DB2CachedRowSets には、DB2CachedRowSet が SQL 照会と DataSource 名を受け取るためのプロパティーがあります。 SQL 照会と DataSource 名を使って、それ自身のデータを作成することができます。以下に、この方法の例を示します。 BaseDataSource という名前の DataSource への参照が、事前に有効な DataSource としてセットアップされていることを想定しています。

例: DB2CachedRowSet プロパティーと DataSources を使用する

注: 法律上の重要な情報に関しては、コードの特記事項情報をお読みください。

```
// Create a new DB2CachedRowSet
DB2CachedRowSet crs = new DB2CachedRowSet();

// Set the properties that are needed for
// the RowSet to use a DataSource to populate itself.
crs.setDataSourceName("BaseDataSource");
crs.setCommand("select col1 from cujosql.test_table");
```

```
// Call the RowSet execute method. This causes
// the RowSet to use the DataSource and SQL query
// specified to populate itself with data. Once
// the RowSet is populated, it disconnects from the database.
crs.execute();

// Loop through the data in the RowSet.
while (crs.next()) {
  System.out.println("v1 is " + crs.getString(1));
}

// Eventually, close the RowSet.
crs.close();
```

DB2CachedRowSet プロパティーと JDBC URL を使用する

DB2CachedRowSets には、DB2CachedRowSet が SQL 照会と JDBC URL を受け取るためのプロパティーがあります。照会と JDBC URL を使って、それ自身のデータを作成することができます。以下に、この方法の例を示します。

例: DB2CachedRowSet プロパティーおよび JDBC URL を使用する

注: 法律上の重要な情報に関しては、コードの特記事項情報をお読みください。

```
// Create a new DB2CachedRowSet
DB2CachedRowSet crs = new DB2CachedRowSet();

// Set the properties that are needed for
// the RowSet to use a JDBC URL to populate itself.
crs.setUrl("jdbc:db2:*local");
crs.setCommand("select col1 from cujosql.test_table");

// Call the RowSet execute method. This causes
// the RowSet to use the DataSource and SQL query
// specified to populate itself with data. Once
// the RowSet is populated, it disconnects from the database.
crs.execute();

// Loop through the data in the RowSet.
while (crs.next()) {
  System.out.println("v1 is " + crs.getString(1));
}

// Eventually, close the RowSet.
crs.close();
```

setConnection(Connection) メソッドを使って、既存のデータベース接続を使用する

JDBC Connection オブジェクトの再利用を促進するため、DB2CachedRowSet には、確立された Connection オブジェクトを、DB2CachedRowset (RowSet にデータを取り込むために使用される) に渡すメカニズムがあります。ユーザーが提供した Connection オブジェクトが渡されると、DB2CachedRowSet は取り込みが完了しても切断しません。

例: setConnection(Connection) メソッドを使って、既存のデータベース接続を使用する

注: 法律上の重要な情報に関しては、コードの特記事項情報をお読みください。

```
// Establish a JDBC connection to the database.
Connection conn = DriverManager.getConnection("jdbc:db2:*local");

// Create a new DB2CachedRowSet
DB2CachedRowSet crs = new DB2CachedRowSet();
```

```
// Set the properties that are needed for the
// RowSet to use an already connected connection
// to populate itself.
crs.setConnection(conn);
crs.setCommand("select col1 from cujosql.test_table");

// Call the RowSet execute method. This causes
// the RowSet to use the connection that it was provided
// with previously. Once the RowSet is populated, it does not
// close the user-supplied connection.
crs.execute();

// Loop through the data in the RowSet.
while (crs.next()) {
  System.out.println("v1 is " + crs.getString(1));
}

// Eventually, close the RowSet.
crs.close();
```

execute(Connection) メソッドを使って、既存のデータベース接続を使用する

JDBC Connection オブジェクトの再利用を促進するため、DB2CachedRowSet には、確立された Connection オブジェクトを、メソッドが呼び出されたときに DB2CachedRowset に渡すメカニズムがあります。ユーザーが提供した Connection オブジェクトが渡されると、DB2CachedRowSet は取り込みが完了しても切断しません。

例: execute(Connection) メソッドを使って、既存のデータベース接続を使用する

注: 法律上の重要な情報に関しては、コードの特記事項情報をお読みください。

```
// Establish a JDBC connection to the database.
Connection conn = DriverManager.getConnection("jdbc:db2:*local");

// Create a new DB2CachedRowSet
DB2CachedRowSet crs = new DB2CachedRowSet();

// Set the SQL statement that is to be used to
// populate the RowSet.
crs.setCommand("select col1 from cujosql.test_table");

// Call the RowSet execute method, passing in the connection
// that should be used. Once the Rowset is populated, it does not
// close the user-supplied connection.
crs.execute(conn);

// Loop through the data in the RowSet.
while (crs.next()) {
  System.out.println("v1 is " + crs.getString(1));
}

// Eventually, close the RowSet.
crs.close();
```

execute(int) メソッドを使って、データベース要求をグループ化する

データベースのワークロードを軽減させるため、DB2CachedRowSet には、複数の SQL ステートメントをいくつかのスレッドでグループ化し、1 つの処理としてまとめてデータベースに要求するメカニズムがあります。

例: execute(int) メソッドを使って、データベース要求をグループ化する

注: 法律上の重要な情報に関しては、コードの特記事項情報をお読みください。

```
// Create a new DB2CachedRowSet
DB2CachedRowSet crs = new DB2CachedRowSet();

// Set the properties that are needed for
// the RowSet to use a DataSource to populate itself.
crs.setDataSourceName("BaseDataSource");
crs.setCommand("select col1 from cujosql.test_table");

// Call the RowSet execute method. This causes
// the RowSet to use the DataSource and SQL query
// specified to populate itself with data. Once
// the RowSet is populated, it disconnects from the database.
// This version of the execute method accepts the number of seconds
// that it is willing to wait for its results. By
// allowing a delay, the RowSet can group the requests
// of several users and only process the request against
// the underlying database once.
crs.execute(5);

// Loop through the data in the RowSet.
while (crs.next()) {
  System.out.println("v1 is " + crs.getString(1));
}

// Eventually, close the RowSet.
crs.close();
```

DB2CachedRowSet データへのアクセスおよびカーソル操作:

このトピックでは、DB2CachedRowSet データへのアクセス、およびさまざまなカーソル操作機能についての情報を記載しています。

RowSet は ResultSet メソッドに依存しています。 DB2CachedRowSet データ・アクセスやカーソルの移動などの多くの操作は、アプリケーション・レベルでは ResultSet の場合も RowSet の場合も違いはありません。

DB2CachedRowSet データへのアクセス

RowSet と ResultSet は同じ方式でデータにアクセスします。以下の例ではプログラムで JDBC を使用し、テーブルを作成して、さまざまなデータ・タイプのデータを取り込みます。テーブルが準備されると、DB2CachedRowSet が作成され、テーブル内の情報が取り込まれます。この例では、RowSet クラスのさまざまな get メソッドも使用されています。

例: DB2CachedRowSet データへのアクセス

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;
import javax.sql.*;
import com.ibm.db2.jdbc.app.*;
import java.io.*;
import java.math.*;

public class TestProgram
{
  public static void main(String args[])
  {
    // Register the driver.
    try {
```

```
      Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
    }
    catch (ClassNotFoundException ex) {
      System.out.println("ClassNotFoundException: " +
                 ex.getMessage());
      // No need to go any further.
      System.exit(1);
    }

    try {
      Connection conn = DriverManager.getConnection("jdbc:db2:*local");

      Statement stmt = conn.createStatement();

      // Clean up previous runs
      try {
        stmt.execute("drop table cujosql.test_table");
      }
      catch (SQLException ex) {
        System.out.println("Caught drop table: " + ex.getMessage());
      }

      // Create test table
      stmt.execute("Create table cujosql.test_table (col1 smallint, col2 int, " +
              "col3 bigint, col4 real, col5 float, col6 double, col7 numeric, " +
              "col8 decimal, col9 char(10), col10 varchar(10), col11 date, " +
              "col12 time, col13 timestamp)");
      System.out.println("Table created.");

      // Insert some test rows
      stmt.execute("insert into cujosql.test_table values (1, 1, 1, 1.5, 1.5, 1.5, 1.5, 1.5, 'one', 'one',
                   {d '2001-01-01'}, {t '01:01:01'}, {ts '1998-05-26 11:41:12.123456'})");

      stmt.execute("insert into cujosql.test_table values (null, null, null, null, null, null, null, null,
                   null, null, null, null, null)");
      System.out.println("Rows inserted");

      ResultSet rs = stmt.executeQuery("select * from cujosql.test_table");
      System.out.println("Query executed");

      // Create a new rowset and populate it...
      DB2CachedRowSet crs = new DB2CachedRowSet();
      crs.populate(rs);
      System.out.println("RowSet populated.");

      conn.close();
      System.out.println("RowSet is detached...");

      System.out.println("Test with getObject");
      int count = 0;
      while (crs.next()) {
        System.out.println("Row " + (++count));
        for (int i = 1; i <= 13; i++) {
          System.out.println("  Col " + i + " value " + crs.getObject(i));
        }
      }

      System.out.println("Test with getXXX... ");
      crs.first();
      System.out.println("Row 1");
      System.out.println("  Col 1 value " + crs.getShort(1));
      System.out.println("  Col 2 value " + crs.getInt(2));
      System.out.println("  Col 3 value " + crs.getLong(3));
      System.out.println("  Col 4 value " + crs.getFloat(4));
      System.out.println("  Col 5 value " + crs.getDouble(5));
      System.out.println("  Col 6 value " + crs.getDouble(6));
      System.out.println("  Col 7 value " + crs.getBigDecimal(7));
```

```
      System.out.println("  Col 8 value " + crs.getBigDecimal(8));
      System.out.println("  Col 9 value " + crs.getString(9));
      System.out.println("  Col 10 value " + crs.getString(10));
      System.out.println("  Col 11 value " + crs.getDate(11));
      System.out.println("  Col 12 value " + crs.getTime(12));
      System.out.println("  Col 13 value " + crs.getTimestamp(13));
      crs.next();
      System.out.println("Row 2");
      System.out.println("  Col 1 value " + crs.getShort(1));
      System.out.println("  Col 2 value " + crs.getInt(2));
      System.out.println("  Col 3 value " + crs.getLong(3));
      System.out.println("  Col 4 value " + crs.getFloat(4));
      System.out.println("  Col 5 value " + crs.getDouble(5));
      System.out.println("  Col 6 value " + crs.getDouble(6));
      System.out.println("  Col 7 value " + crs.getBigDecimal(7));
      System.out.println("  Col 8 value " + crs.getBigDecimal(8));
      System.out.println("  Col 9 value " + crs.getString(9));
      System.out.println("  Col 10 value " + crs.getString(10));
      System.out.println("  Col 11 value " + crs.getDate(11));
      System.out.println("  Col 12 value " + crs.getTime(12));
      System.out.println("  Col 13 value " + crs.getTimestamp(13));

      crs.close();
    }
    catch (Exception ex) {
      System.out.println("SQLException: " + ex.getMessage());
                 ex.printStackTrace();
    }
  }
}
```

カーソル操作

RowSet はスクロール可能で、その動作はスクロール可能な ResultSet と同じです。以下の例ではプログラムで JDBC を使用し、テーブルを作成して、データを取り込みます。テーブルが準備されると、DB2CachedRowSet オブジェクトが作成され、テーブル内の情報が取り込まれます。この例では、さまざまなカーソル操作機能も使用されています。

例: カーソル操作

```
import java.sql.*;
import javax.sql.*;
import com.ibm.db2.jdbc.app.DB2CachedRowSet;

public class RowSetSample1
{
  public static void main(String args[])
  {
    // Register the driver.
    try {
      Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
    }
    catch (ClassNotFoundException ex) {
      System.out.println("ClassNotFoundException: " +
                 ex.getMessage());
      // No need to go any further.
      System.exit(1);
  }

  try {
    Connection conn = DriverManager.getConnection("jdbc:db2:*local");

    Statement stmt = conn.createStatement();

    // Clean up previous runs
```

```
try {
  stmt.execute("drop table cujosql.test_table");
}
catch (SQLException ex) {
  System.out.println("Caught drop table: " + ex.getMessage());
}

// Create a test table
stmt.execute("Create table cujosql.test_table (col1 smallint)");
System.out.println("Table created.");

// Insert some test rows
for (int i = 0; i < 10; i++) {
  stmt.execute("insert into cujosql.test_table values (" + i + ")");
}
System.out.println("Rows inserted");

ResultSet rs = stmt.executeQuery("select col1 from cujosql.test_table");
System.out.println("Query executed");

// Create a new rowset and populate it...
DB2CachedRowSet crs = new DB2CachedRowSet();
crs.populate(rs);
System.out.println("RowSet populated.");

conn.close();
System.out.println("RowSet is detached...");

System.out.println("Use next()");
while (crs.next()) {
  System.out.println("v1 is " + crs.getShort(1));
}

System.out.println("Use previous()");
while (crs.previous()) {
  System.out.println("value is " + crs.getShort(1));
}

System.out.println("Use relative()");
crs.next();
crs.relative(9);
System.out.println("value is " + crs.getShort(1));

crs.relative(-9);
System.out.println("value is " + crs.getShort(1));

System.out.println("Use absolute()");
crs.absolute(10);
System.out.println("value is " + crs.getShort(1));
crs.absolute(1);
System.out.println("value is " + crs.getShort(1));
crs.absolute(-10);
System.out.println("value is " + crs.getShort(1));
crs.absolute(-1);
System.out.println("value is " + crs.getShort(1));

System.out.println("Test beforeFirst()");
crs.beforeFirst();
System.out.println("isBeforeFirst is " + crs.isBeforeFirst());
crs.next();
System.out.println("move one... isFirst is " + crs.isFirst());

System.out.println("Test afterLast()");
crs.afterLast();
System.out.println("isAfterLast is " + crs.isAfterLast());
crs.previous();
System.out.println("move one... isLast is " + crs.isLast());
```

```
    System.out.println("Test getRow()");
    crs.absolute(7);
    System.out.println("row should be (7) and is " + crs.getRow() +
                       " value should be (6) and is " + crs.getShort(1));

    crs.close();
  }
  catch (SQLException ex) {
    System.out.println("SQLException: " + ex.getMessage());
  }
 }
}
```

***DB2CachedRowSet* データを変更し、データ・ソースに変更を反映する:**

このトピックでは、DB2CachedRowSet 内の行に変更を行い、その後の基礎データベースの更新に関する情報を提供します。

DB2CachedRowSet では、RowSet オブジェクト内のデータを変更するための標準 ResultSet インターフェースと同じメソッドを使用します。アプリケーション・レベルでは、RowSet のデータを変更することと、ResultSet のデータを変更することには違いがありません。 DB2CachedRowSet は acceptChanges メソッドを提供しており、このメソッドは RowSet に加えられた変更をデータ元のデータベースに反映するために使用します。

DB2CachedRowSet 内の行を削除、挿入、および更新する

DB2CachedRowSet は更新可能です。以下の例ではプログラムで JDBC を使用し、テーブルを作成して、データを取り込みます。テーブルが準備されると、DB2CachedRowSet が作成され、テーブル内の情報が取り込まれます。この例では、RowSet を更新するために使用できる多くのメソッドを使っており、またアプリケーションが削除された後の行をフェッチできるようにする showDeleted プロパティーの使い方を示しています。さらにこの例では、行の挿入および削除を元に戻すことのできる cancelRowInsert および cancelRowDelete メソッドの使い方も示されています。

例: DB2CachedRowSet 内の行を削除、挿入、および更新する

注: 法律上の重要な情報に関しては、コードの特記事項情報をお読みください。

```
import java.sql.*;
import javax.sql.*;
import com.ibm.db2.jdbc.app.DB2CachedRowSet;

public class RowSetSample2
{
  public static void main(String args[])
  {
    // Register the driver.
    try {
      Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
    }
    catch (ClassNotFoundException ex) {
      System.out.println("ClassNotFoundException: " +
              ex.getMessage());

      // No need to go any further.
      System.exit(1);
    }

    try {
      Connection conn = DriverManager.getConnection("jdbc:db2:*local");
```

```
Statement stmt = conn.createStatement();

// Clean up previous runs
try {
  stmt.execute("drop table cujosql.test_table");
}

catch (SQLException ex) {
  System.out.println("Caught drop table: " + ex.getMessage());
}

// Create test table
stmt.execute("Create table cujosql.test_table (col1 smallint)");
System.out.println("Table created.");

// Insert some test rows
for (int i = 0; i < 10; i++) {
  stmt.execute("insert into cujosql.test_table values (" + i + ")");
}
System.out.println("Rows inserted");

ResultSet rs = stmt.executeQuery("select col1 from cujosql.test_table");
System.out.println("Query executed");

// Create a new rowset and populate it...
DB2CachedRowSet crs = new DB2CachedRowSet();
crs.populate(rs);
System.out.println("RowSet populated.");

conn.close();
System.out.println("RowSet is detached...");

System.out.println("Delete the first three rows");
crs.next();
crs.deleteRow();
crs.next();
crs.deleteRow();
crs.next();
crs.deleteRow();

crs.beforeFirst();
System.out.println("Insert the value -10 into the RowSet");
crs.moveToInsertRow();
crs.updateShort(1, (short)-10);
crs.insertRow();
crs.moveToCurrentRow();

System.out.println("Update the rows to be the negative of what they now are");
crs.beforeFirst();
while (crs.next())
  short value = crs.getShort(1);
  value = (short)-value;
  crs.updateShort(1, value);
  crs.updateRow();
}

crs.setShowDeleted(true);

System.out.println("RowSet is now (value - inserted - updated - deleted)");
crs.beforeFirst();
while (crs.next()) {
  System.out.println("value is " + crs.getShort(1) + " " +
             crs.rowInserted() + " " +
             crs.rowUpdated() + " " +
             crs.rowDeleted());
}
```

```
      System.out.println("getShowDeleted is " + crs.getShowDeleted());

      System.out.println("Now undo the inserts and deletes");
      crs.beforeFirst();
      crs.next();
      crs.cancelRowDelete();
      crs.next();
      crs.cancelRowDelete();
      crs.next();
      crs.cancelRowDelete();
      while (!crs.isLast()) {
        crs.next();
      }

      crs.cancelRowInsert();

      crs.setShowDeleted(false);

      System.out.println("RowSet is now (value - inserted - updated - deleted)");
      crs.beforeFirst();
      while (crs.next()) {
        System.out.println("value is " + crs.getShort(1) + " " +
                   crs.rowInserted() + " " +
                   crs.rowUpdated() + " " +
                   crs.rowDeleted());
      }

      System.out.println("finally show that calling cancelRowUpdates works");
      crs.first();
      crs.updateShort(1, (short) 1000);
      crs.cancelRowUpdates();
      crs.updateRow();
      System.out.println("value of row is " + crs.getShort(1));
      System.out.println("getShowDeleted is " + crs.getShowDeleted());

      crs.close();

    }

    catch (SQLException ex) {
      System.out.println("SQLException: " + ex.getMessage());
    }
  }
}
```

DB2CachedRowSet に加えられた変更を、元のデータベースに反映する

DB2CachedRowSet への変更が加えられると、その変更は RowSet オブジェクトが存在している間のみ存在します。つまり、切断された RowSet に加えられた変更は、データベースには影響を与えません。 RowSet に加えられた変更を元のデータベースに反映するには、acceptChanges メソッドを使用します。このメソッドは、切断された RowSet がデータベースへの接続を再確立し、RowSet に加えられた変更を元のデータベースに戻すよう試行します。 RowSet が作成された後にデータベースに加えられた他の変更との競合により、データベースに安全に変更が加えられない場合は、例外がスローされ、トランザクションがロールバックします。

例: DB2CachedRowSet に加えられた変更を、元のデータベースに反映する

注: 法律上の重要な情報に関しては、コードの特記事項情報をお読みください。

```
import java.sql.*;
import javax.sql.*;
import com.ibm.db2.jdbc.app.DB2CachedRowSet;

public class RowSetSample3
```

```
{
  public static void main(String args[])
  {
    // Register the driver.
    try {
      Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
    }
    catch (ClassNotFoundException ex) {
      System.out.println("ClassNotFoundException: " +
                ex.getMessage());
      // No need to go any further.
      System.exit(1);
    }

    try {
      Connection conn = DriverManager.getConnection("jdbc:db2:*local");

      Statement stmt = conn.createStatement();

      // Clean up previous runs
      try {
        stmt.execute("drop table cujosql.test_table");
      }
      catch (SQLException ex) {
        System.out.println("Caught drop table: " + ex.getMessage());
      }

      // Create test table
      stmt.execute("Create table cujosql.test_table (col1 smallint)");
      System.out.println("Table created.");

      // Insert some test rows
      for (int i = 0; i < 10; i++) {
        stmt.execute("insert into cujosql.test_table values (" + i + ")");
      }
      System.out.println("Rows inserted");

      ResultSet rs = stmt.executeQuery("select col1 from cujosql.test_table");
      System.out.println("Query executed");

      // Create a new rowset and populate it...
      DB2CachedRowSet crs = new DB2CachedRowSet();
      crs.populate(rs);
      System.out.println("RowSet populated.");

      conn.close();
      System.out.println("RowSet is detached...");

      System.out.println("Delete the first three rows");
      crs.next();
      crs.deleteRow();
      crs.next();
      crs.deleteRow();
      crs.next();
      crs.deleteRow();

      crs.beforeFirst();
      System.out.println("Insert the value -10 into the RowSet");
      crs.moveToInsertRow();
      crs.updateShort(1, (short)-10);
      crs.insertRow();
      crs.moveToCurrentRow();

      System.out.println("Update the rows to be the negative of what they now are");
      crs.beforeFirst();
      while (crs.next()) {
        short value = crs.getShort(1);
```

```
        value = (short)-value;
        crs.updateShort(1, value);
        crs.updateRow();
      }

      System.out.println("Now accept the changes to the database");

      crs.setUrl("jdbc:db2:*local");
      crs.setTableName("cujosql.test_table");

      crs.acceptChanges();
      crs.close();

      System.out.println("And the database table looks like this:");
      conn = DriverManager.getConnection("jdbc:db2:localhost");
      stmt = conn.createStatement();
      rs = stmt.executeQuery("select col1 from cujosql.test_table");
      while (rs.next()) {
        System.out.println("Value from table is " + rs.getShort(1));
      }

      conn.close();

    }
    catch (SQLException ex) {
      System.out.println("SQLException: " + ex.getMessage());
    }
  }
}
```

DB2CachedRowSet フィーチャー:

ResultSet のような動作に加え、DB2CachedRowSet クラスには、さらに柔軟に使用できるいくつかの追加機能があります。これらのメソッドは、完全な Java Database Connectivity (JDBC) RowSet、またはその一部を Java コレクションに変換します。さらに、それらが切断状態にあるため、DB2CachedRowSet は ResultSet と絶対的な 1 対 1 の関係を持っていません。

いくつかの例で示したように、ResultSet のような動作に加え、DB2CachedRowSet クラスには、さらに柔軟に使用できるいくつかの追加機能があります。これらのメソッドは、完全な Java Database Connectivity (JDBC) RowSet、またはその一部を Java コレクションに変換します。さらに、それらが切断状態にあるため、DB2CachedRowSet は ResultSet と絶対的な 1 対 1 の関係を持っていません。

DB2CachedRowSet で提供されているメソッドを使って、次のようなタスクを実行できます。

DB2CachedRowSets からコレクションを取得する

DB2CachedRowset オブジェクトからいくつかの形式のコレクションを戻すには、3 つのメソッドがあります。以下のメソッドです。

- **toCollection** は、ベクトル (1 項目が 1 列) の ArrayList (1 項目が 1 行) で戻します。
- **toCollection(int columnIndex)** は、各行に指定された列の値を格納したベクトルを戻します。
- **getColumn(int columnIndex)** は、各列に指定された列の値を格納した配列を戻します。

toCollection(int columnIndex) と getColumn(int columnIndex) の大きな違いは、getColumn メソッドはプリミティブ・タイプの配列を戻すことができる点です。したがって、columnIndex で整数データを持つ列を指定した場合、整数の配列が戻され、java.lang.Integer オブジェクトの配列が戻されるわけではありません。

以下に、これらのメソッドを使い方を示します。

例: DB2CachedRowSets からコレクションを取得する

注: 法律上の重要な情報に関しては、コードの特記事項情報をお読みください。

```
import java.sql.*;
import javax.sql.*;
import com.ibm.db2.jdbc.app.DB2CachedRowSet;
import java.util.*;

public class RowSetSample4
{
  public static void main(String args[])
  {
    // Register the driver.
    try {
      Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
    }
    catch (ClassNotFoundException ex) {
      System.out.println("ClassNotFoundException: " +
                 ex.getMessage());
      // No need to go any further.
      System.exit(1);
    }

    try {
      Connection conn = DriverManager.getConnection("jdbc:db2:*local");
      Statement stmt = conn.createStatement();

      // Clean up previous runs
      try {
        stmt.execute("drop table cujosql.test_table");
      }
      catch (SQLException ex) {
        System.out.println("Caught drop table: " + ex.getMessage());
      }

      // Create test table
      stmt.execute("Create table cujosql.test_table (col1 smallint, col2 smallint)");
      System.out.println("Table created.");

      // Insert some test rows
      for (int i = 0; i < 10; i++) {
        stmt.execute("insert into cujosql.test_table values (" + i + ", " + (i + 100) + ")");
      }
      System.out.println("Rows inserted");

      ResultSet rs = stmt.executeQuery("select * from cujosql.test_table");
      System.out.println("Query executed");

      // Create a new rowset and populate it...
      DB2CachedRowSet crs = new DB2CachedRowSet();
      crs.populate(rs);
      System.out.println("RowSet populated.");

      conn.close();
      System.out.println("RowSet is detached...");

      System.out.println("Test the toCollection() method");
      Collection collection = crs.toCollection();
      ArrayList map = (ArrayList) collection;

      System.out.println("size is " + map.size());
      Iterator iter = map.iterator();
      int row = 1;
      while (iter.hasNext()) {
        System.out.print("row [" + (row++) + "]: ¥t");
```

```
        Vector vector = (Vector)iter.next();
        Iterator innerIter = vector.iterator();
        int i = 1;
        while (innerIter.hasNext()) {
          System.out.print(" [" + (i++) + "]=" + innerIter.next() + "; ¥t");
        }
        System.out.println();
      }
      System.out.println("Test the toCollection(int) method");
      collection = crs.toCollection(2);
      Vector vector = (Vector) collection;

      iter = vector.iterator();

      while (iter.hasNext()) {
        System.out.println("Iter: Value is " + iter.next());
      }

      System.out.println("Test the getColumn(int) method");
      Object values = crs.getColumn(2);
      short[] shorts = (short [])values;

      for (int i =0; i < shorts.length; i++) {
        System.out.println("Array: Value is " + shorts[i]);
      }
    }
    catch (SQLException ex) {
      System.out.println("SQLException: " + ex.getMessage());
    }
  }
}
```

RowSet のコピーを作成する

createCopy メソッドは、DB2CachedRowSet のコピーを作成します。 RowSet に関連したすべてのデータが、すべての制御構造、プロパティー、および状況フラグと共に複製されます。

以下に、このメソッドを使い方を示します。

例: RowSet のコピーを作成する

注: 法律上の重要な情報に関しては、コードの特記事項情報をお読みください。

```
import java.sql.*;
import javax.sql.*;
import com.ibm.db2.jdbc.app.*;
import java.io.*;

public class RowSetSample5
{
  public static void main(String args[])
  {
    // Register the driver.
    try {
      Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
    }
    catch (ClassNotFoundException ex) {
      System.out.println("ClassNotFoundException: " +
                 ex.getMessage());
      // No need to go any further.
      System.exit(1);
    }

    try {
      Connection conn = DriverManager.getConnection("jdbc:db2:*local");
```

```
      Statement stmt = conn.createStatement();

      // Clean up previous runs
      try {
        stmt.execute("drop table cujosql.test_table");
      }
      catch (SQLException ex) {
        System.out.println("Caught drop table: " + ex.getMessage());
      }

      // Create test table
      stmt.execute("Create table cujosql.test_table (col1 smallint)");
      System.out.println("Table created.");

      // Insert some test rows
      for (int i = 0; i < 10; i++) {
        stmt.execute("insert into cujosql.test_table values (" + i + ")");
      }
      System.out.println("Rows inserted");

      ResultSet rs = stmt.executeQuery("select col1 from cujosql.test_table");
      System.out.println("Query executed");

      // Create a new rowset and populate it...
      DB2CachedRowSet crs = new DB2CachedRowSet();
      crs.populate(rs);
      System.out.println("RowSet populated.");

      conn.close();
      System.out.println("RowSet is detached...");

      System.out.println("Now some new RowSets from one.");
      DB2CachedRowSet crs2 = crs.createCopy();
      DB2CachedRowSet crs3 = crs.createCopy();

      System.out.println("Change the second one to be negated values");
      crs2.beforeFirst();
      while (crs2.next()) {
        short value = crs2.getShort(1);
        value = (short)-value;
        crs2.updateShort(1, value);
        crs2.updateRow();
      }

      crs.beforeFirst();
      crs2.beforeFirst();
      crs3.beforeFirst();
      System.out.println("Now look at all three of them again");

      while (crs.next()) {
        crs2.next();
        crs3.next();
        System.out.println("Values: crs: " + crs.getShort(1) + ", crs2: " + crs2.getShort(1) +
                   ", crs3: " + crs3.getShort(1));
      }
    }
    catch (Exception ex) {
      System.out.println("SQLException: " + ex.getMessage());
      ex.printStackTrace();
    }

  }

}
```

RowSet の共用を作成する

createShared メソッドは、高レベルの状況情報付きの新しい RowSet オブジェクトを作成し、2つの RowSet オブジェクトが同一の基礎となる物理データを共用できるようにします。

以下に、このメソッドを使い方を示します。

例: RowSet の共用を作成する

注: 法律上の重要な情報に関しては、コードの特記事項情報をお読みください。

```
import java.sql.*;
import javax.sql.*;
import com.ibm.db2.jdbc.app.*;
import java.io.*;

public class RowSetSample5
{
  public static void main(String args[])
  {
    // Register the driver.
    try {
      Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
    }
    catch (ClassNotFoundException ex) {
      System.out.println("ClassNotFoundException: " +
                ex.getMessage());
      // No need to go any further.
      System.exit(1);
    }

    try {
      Connection conn = DriverManager.getConnection("jdbc:db2:*local");

      Statement stmt = conn.createStatement();

      // Clean up previous runs
      try {
        stmt.execute("drop table cujosql.test_table");
      }
      catch (SQLException ex) {
        System.out.println("Caught drop table: " + ex.getMessage());
      }

      // Create test table
      stmt.execute("Create table cujosql.test_table (col1 smallint)");
      System.out.println("Table created.");

      // Insert some test rows
      for (int i = 0; i < 10; i++) {
        stmt.execute("insert into cujosql.test_table values (" + i + ")");
      }
      System.out.println("Rows inserted");

      ResultSet rs = stmt.executeQuery("select col1 from cujosql.test_table");
      System.out.println("Query executed");

      // Create a new rowset and populate it...
      DB2CachedRowSet crs = new DB2CachedRowSet();
      crs.populate(rs);
      System.out.println("RowSet populated.");

      conn.close();
      System.out.println("RowSet is detached...");
```

```
      System.out.println("Test the createShared functionality (create 2 shares)");
      DB2CachedRowSet crs2 = crs.createShared();
      DB2CachedRowSet crs3 = crs.createShared();

      System.out.println("Use the original to update value 5 of the table");
      crs.absolute(5);
      crs.updateShort(1, (short)-5);
      crs.updateRow();

      crs.beforeFirst();
      crs2.afterLast();

      System.out.println("Now move the cursors in opposite directions of the same data.");

      while (crs.next()) {
        crs2.previous();
        crs3.next();
        System.out.println("Values: crs: " + crs.getShort(1) + ", crs2: " + crs2.getShort(1) +
                   ", crs3: " + crs3.getShort(1));
      }
      crs.close();
      crs2.close();
      crs3.close();
    }
    catch (Exception ex) {
      System.out.println("SQLException: " + ex.getMessage());
                ex.printStackTrace();
    }
  }
}
```

DB2JdbcRowSet:

DB2JdbcRowSet は接続された RowSet で、基盤となっている Connection オブジェクト、PreparedStatement オブジェクト、または ResultSet オブジェクトのサポートでのみ使用できます。そのインプリメンテーションは、JdbcRowSet の記述に厳密に従っています。

DB2JdbcRowSet の使用

DB2JdbcRowSet オブジェクトは Java Database Connectivity (JDBC) 3.0 仕様で記述されているすべての RowSet のイベントをサポートしているため、ローカル・データベースと、データベースのデータの変更が通知される必要のある他のオブジェクトとの中間オブジェクトとして動作することができます。

たとえば、メイン・データベースと、それに接続するために無線プロトコルを使用するいくつかの PDA という環境で作業することを想定します。 DB2JdbcRowSet オブジェクトは、サーバー上で動作するマスター・アプリケーションを使用した、行への移動とその更新に使用することができます。行を更新すると、RowSet コンポーネントによってイベントが生成されます。もし PDA に対する更新の送信を担当するサービスが動作していれば、これを RowSet の「リスナー」として登録することができます。 RowSet イベントを受信するたびに、無線デバイスに対して適切な更新および送信を生成することができます。

詳しくは、例: DB2JdbcRowSet イベントを参照してください。

JDBCRowSets の作成

DB2JDBCRowSet オブジェクトを生成するために、いくつかのメソッドが提供されています。以下にそれぞれを概説します。

DB2JdbcRowSet プロパティーおよび DataSources を使用する

DB2JdbcRowSet には、SQL 照会および DataSource 名を受け取るプロパティーがあります。その後、DB2JdbcRowSet は使用可能になります。以下に、この方法の例を示します。 BaseDataSource という名前の DataSource への参照が、事前に有効な DataSource としてセットアップされていることを想定しています。

例: DB2JDBCRowSet プロパティーおよび DataSource を使用する

注: 法律上の重要な情報に関しては、コードの特記事項情報をお読みください。

```
// Create a new DB2JdbcRowSet
DB2JdbcRowSet jrs = new DB2JdbcRowSet();

// Set the properties that are needed for
// the RowSet to be processed.
jrs.setDataSourceName("BaseDataSource");
jrs.setCommand("select col1 from cujosql.test_table");

// Call the RowSet execute method. This method causes
// the RowSet to use the DataSource and SQL query
// specified to prepare itself for data processing.
jrs.execute();

// Loop through the data in the RowSet.
while (jrs.next()) {
    System.out.println("v1 is " + jrs.getString(1));
}

// Eventually, close the RowSet.
jrs.close();
```

DB2JdbcRowSet プロパティーおよび JDBC URL を使用する

DB2JdbcRowSet には、SQL 照会および JDBC URL を受け取るプロパティーがあります。その後、DB2JdbcRowSet は使用可能になります。以下に、この方法の例を示します。

例: DB2JdbcRowSet プロパティーおよび JDBC URL を使用する

注: 法律上の重要な情報に関しては、コードの特記事項情報をお読みください。

```
// Create a new DB2JdbcRowSet
DB2JdbcRowSet jrs = new DB2JdbcRowSet();

// Set the properties that are needed for
// the RowSet to be processed.
jrs.setUrl("jdbc:db2:*local");
jrs.setCommand("select col1 from cujosql.test_table");

// Call the RowSet execute method. This causes
// the RowSet to use the URL and SQL query specified
// previously to prepare itself for data processing.
jrs.execute();

// Loop through the data in the RowSet.
while (jrs.next()) {
    System.out.println("v1 is " + jrs.getString(1));
}

// Eventually, close the RowSet.
jrs.close();
```

setConnection(Connection) メソッドを使って、既存のデータベース接続を使用する

JDBC Connection オブジェクトの再利用を促進するため、DB2JdbcRowSet は確立された接続を DB2JdbcRowSet に渡すことができます。この接続は、execute メソッドが呼び出されると、その使用の準備のために DB2JdbcRowSet によって使用されます。

例: setConnection メソッド

注: 法律上の重要な情報に関しては、コードの特記事項情報をお読みください。

```
// Establish a JDBC Connection to the database.
Connection conn = DriverManager.getConnection("jdbc:db2:*local");

// Create a new DB2JdbcRowSet.
DB2JdbcRowSet jrs = new DB2JdbcRowSet();

// Set the properties that are needed for
// the RowSet to use an established connection.
jrs.setConnection(conn);
jrs.setCommand("select col1 from cujosql.test_table");

// Call the RowSet execute method. This causes
// the RowSet to use the connection that it was provided
// previously to prepare itself for data processing.
jrs.execute();

// Loop through the data in the RowSet.
while (jrs.next()) {
    System.out.println("v1 is " + jrs.getString(1));
}

// Eventually, close the RowSet.
jrs.close();
```

データのアクセスおよびカーソル移動

DB2JdbcRowSet を介したカーソル位置の操作、およびデータベース・データへのアクセスは、基盤になる ResultSet オブジェクトによって処理されます。 ResultSet オブジェクトで完了できるタスクは、DB2JdbcRowSet オブジェクトにも適用されます。

データの変更、および元のデータベースへの変更の反映

DB2JdbcRowSet を介したデータベースの更新のサポートは、基盤となる ResultSet オブジェクトによってすべて処理されます。 ResultSet オブジェクトで完了できるタスクは、DB2JdbcRowSet オブジェクトにも適用されます。

DB2JdbcRowSet イベント*:*

すべての RowSet インプリメンテーションは、他のコンポーネントにとって興味ある状態を処理するイベントをサポートしています。このサポートにより、アプリケーション・コンポーネントでイベントが発生したとき、それらのコンポーネントがお互いに「話し合う」ことが可能です。たとえば、RowSet を介してデータベースの行が更新されたときに、グラフィカル・ユーザー・インターフェース (GUI) で更新された表を表示させることができます。

以下の例で、メインメソッドは RowSet の更新を行うコア・アプリケーションです。リスナーは、そのアプリケーション領域内で切断されたクライアントが使用する、無線サーバーの一部です。これにより、2 つの処理が混在するコードを使うことなく、ビジネスの 2 つの側面を互いに結び付けることができます。RowSet のこのイベント・サポートは、主にデータベース・データによって GUI を更新することを目的に設計されており、このタイプのアプリケーションの問題に対して完全に動作します。

例: DB2JdbcRowSet イベント

注: 法律上の重要な情報に関しては、コードの特記事項情報をお読みください。

```
import java.sql.*;
import javax.sql.*;
import com.ibm.db2.jdbc.app.DB2JdbcRowSet;

public class RowSetEvents {
    public static void main(String args[])
    {
        // Register the driver.
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
        } catch (ClassNotFoundException ex) {
            System.out.println("ClassNotFoundException: " +
                               ex.getMessage());
            // No need to go any further.
            System.exit(1);
        }


        try {
            // Obtain the JDBC Connection and Statement needed to set
            // up this example.
            Connection conn = DriverManager.getConnection("jdbc:db2:*local");
            Statement stmt = conn.createStatement();

            // Clean up any previous runs.
            try {
                stmt.execute("drop table cujosql.test_table");
            } catch (SQLException ex) {
                System.out.println("Caught drop table: " + ex.getMessage());
            }

            // Create the test table
            stmt.execute("Create table cujosql.test_table (col1 smallint)");
            System.out.println("Table created.");

            // Populate the table with data.
            for (int i = 0; i < 10; i++) {
                stmt.execute("insert into cujosql.test_table values (" + i + ")");
            }
            System.out.println("Rows inserted");

            // Remove the setup objects.
            stmt.close();
            conn.close();

            // Create a new rowset and set the properties need to
            // process it.
            DB2JdbcRowSet jrs = new DB2JdbcRowSet();
            jrs.setUrl("jdbc:db2:*local");
            jrs.setCommand("select col1 from cujosql.test_table");
            jrs.setConcurrency(ResultSet.CONCUR_UPDATEABLE);

            // Give the RowSet object a listener. This object handles
            // special processing when certain actions are done on
            // the RowSet.
            jrs.addRowSetListener(new MyListener());

            // Process the RowSet to provide access to the database data.
            jrs.execute();

            // Cause a few cursor change events. These events cause the cursorMoved
            // method in the listener object to get control.
            jrs.next();
```

```
            jrs.next();
            jrs.next();

            // Cause a row change event to occur. This event causes the rowChanged method
            // in the listener object to get control.
            jrs.updateShort(1, (short)6);
            jrs.updateRow();

            // Finally, cause a RowSet change event to occur. This causes the
            // rowSetChanged method in the listener object to get control.
            jrs.execute();

            // When completed, close the RowSet.
            jrs.close();
                } catch (SQLException ex) {
            ex.printStackTrace();
        }
    }
}


/**
 * This is an example of a listener. This example prints messages that show
 * how control flow moves through the application and offers some
 * suggestions about what might be done if the application were fully implemented.
 */
class MyListener
implements RowSetListener {
    public void cursorMoved(RowSetEvent rse) {
        System.out.println("Event to do:  Cursor position changed.");
        System.out.println("  For the remote system, do nothing ");
        System.out.println("  when this event happened. The remote view of the data");
        System.out.println("  could be controlled separately from the local view.");
        try {
            DB2JdbcRowSet rs = (DB2JdbcRowSet) rse.getSource();
            System.out.println("row is " + rs.getRow() + ". ¥n¥n");
        } catch (SQLException e) {
            System.out.println("To do:  Properly handle possible problems.");
        }
    }

    public void rowChanged(RowSetEvent rse) {
        System.out.println("Event to do:  Row changed.");
        System.out.println("  Tell the remote system that a row has changed. Then,");
        System.out.println("  pass all the values only for that row to the  ");
        System.out.println("  remote system.");
        try {
            DB2JdbcRowSet rs = (DB2JdbcRowSet) rse.getSource();
            System.out.println("new values are " + rs.getShort(1) + ". ¥n¥n");
        } catch (SQLException e) {
              System.out.println("To do:  Properly handle possible problems.");
        }
    }

    public void rowSetChanged(RowSetEvent rse) {
        System.out.println("Event to do:  RowSet changed.");
        System.out.println("  If there is a remote RowSet already established, ");
        System.out.println("  tell the remote system that the values it ");
        System.out.println("  has should be thrown out. Then, pass all ");
        System.out.println("  the current values to it.¥n¥n");
    }
}
```

ネイティブ JDBC ドライバーに関するパフォーマンス上のヒント

ネイティブ JDBC ドライバーは、データベースを扱う高性能な Java インターフェースとして設計されています。ただし、最高のパフォーマンスを得るためには、ネイティブ JDBC ドライバーが提供する能力を

利用するようにアプリケーションを構築する必要があります。以下に挙げるヒントを適用すれば、JDBC プログラミングを効果的に行えるでしょう。ほとんどは、ネイティブ JDBC ドライバーに特有の情報ではありません。ですから、ここに示される指針に従って作成されたアプリケーションは、ネイティブ JDBC ドライバー以外の JDBC ドライバーと共に使用される場合でも高いパフォーマンスを示します。

SELECT * SQL 照会を避ける

SELECT * FROM... は、SQL の照会を記述する一般的な方法です。しかし、すべてのフィールドを照会する必要がない場合もよくあります。戻されるそれぞれの列ごとに、JDBC ドライバーは、行をバインドして戻す余分の作業をしなければなりません。アプリケーションが特定の列を使用しない場合でも、JDBC ドライバーはその列を認識しなければならず、それを使用するためのスペースを予約しなければなりません。使用されない列がテーブル内にほとんどない場合、このことは重大なオーバーヘッドにはなりません。しかし、使用されない列が多数ある場合、このオーバーヘッドは重大となる可能性があります。これを解決するための良い方法は、アプリケーションと関係のある列を以下のように個々にリストすることです。

```
SELECT COL1, COL2, COL3 FROM...
```

getXXX(String) の代わりに getXXX(int) を使用する

ResultSet getXXX メソッドを使用する際に、列名をとるバージョンの代わりに数値をとるバージョンを使用します。数値定数の代わりに列名を使用できることは利点のように思えますが、データベース自体は列索引を処理する方法しか認識していません。したがって、列名をとる各 getXXX メソッドを呼び出す場合、それらメソッドのはデータベースに渡される前に JDBC ドライバーによって解決されなければなりません。 getXXX メソッドは通常、何度も実行されるループ内部で呼び出されるので、この小さなオーバーヘッドは急激に蓄積します。

Java プリミティブ・タイプの getObject 呼び出しを避ける

プリミティブ・タイプ (int、long、float など) 値をデータベースから取得するときは、プリミティブ・タイプ固有の get メソッド (getInt、getLong、getFloat) を使用する方が、getObject を使用するより早く取得できます。 getObject 呼び出しは、プリミティブ・タイプに対して取得作業を行った後にオブジェクトを作成してユーザーに戻します。これは通常ループの中で行われますが、存続期間の短い無数のオブジェクトが作成される可能性があります。プリミティブ・コマンドの getObject を使用することには、ガーベッジ・コレクターが頻繁に活動化されてパフォーマンスが低下するという欠点があります。

Statement よりも PreparedStatement を使用する

何回も使用される SQL ステートメントを作成する場合、Statement オブジェクトよりも PreparedStatement を使用する方がパフォーマンスが向上します。ステートメントを実行するたびに、2 つのステップのプロセス、つまりステートメントが準備されてからステートメントが処理されます。 PreparedStatement を使用する際、そのステートメントの準備は、実行されるたびに行われるのではなく、構成されたときだけに行われます。 PreparedStatement の方が Statement よりも実行が速いことは知られていますが、プログラマーたちはこの利点をしばしば無視します。 PreparedStatement によるパフォーマンス向上を考えるなら、アプリケーションを設計する際、可能な場合にはいつも PreparedStatement を使用するのが賢明と言えます (以下の 175 ページの『PreparedStatement プーリングの使用を考慮する』を参照してください)。

DatabaseMetaData 呼び出しを避ける

DatabaseMetaData 呼び出しの中には費用がかかるものがあることに注意してください。特に、getBestRowIdentifier、getCrossReference、getExportedKeys、および getImportedKeys メソッドは費用がかかる場合があります。一部の DataBaseMetaData 呼び出しには、システム・レベル・テーブルに対する複雑な

結合条件が伴います。ただ便利だからという理由でそれらを使用せずに、その情報が必要な場合のみ使用してください。

アプリケーションに対して適切なコミット・レベルを使用する

JDBC は複数のコミット・レベルを提供しており、システム内で複数のトランザクションが互いにどのように影響しあうかはこれによって決まります (詳しくは、トランザクションを参照してください)。デフォルトでは、最低のコミット・レベルが使用されます。つまり、トランザクションはコミット境界を介して互いの作業の一部を知ることができます。これは、ある種のデータベース異常を生じさせる可能性があります。そのため、一部のプログラマーはコミット・レベルを上げて、そのような異常の発生を心配しなくてもよいようにしています。コミット・レベルを上げると、より粗い細分度のロックでのデータベースのハングを引き起こすことになることに注意してください。これは、システムで可能な並行度を制限するので、いくつかのアプリケーションのパフォーマンスが極度に低下します。そもそもアプリケーションの設計によって、ロック上の異常な状況は起きない場合もよくあります。行おうとしていることを時間を取って理解し、トランザクションの分離レベルを、安全に使用できる最低レベルまでに制限してください。

Unicode 形式でのデータの保管を考慮する

Java では、処理するすべての文字データ (String) が Unicode 形式でなければなりません。そのため、Unicode データを持たないテーブルはすべて、データベースにデータを挿入したりデータベースからデータを検索したりする際にデータを変換するため、JDBC ドライバーを必要とします。テーブルがすでに Unicode 形式の場合、JDBC ドライバーはデータを変換する必要はないので、データベースのデータをより早く取り出すことができます。 Unicode 形式のデータは非 Java アプリケーションでは使用できないことがある、という点を理解しておく必要があります。なぜならそれらのアプリケーションは、Unicode を処理できないからです。また、文字データ以外の場合は、データの変換がないため、パフォーマンスが変わらないことも覚えておいてください。別の考慮事項として、Unicode 形式で保管されたデータは、単一バイトのデータと比べて 2 倍のスペースを使用します。ただし、何度も読み取られる文字列がたくさんある場合は、Unicode 形式でデータを保管することでパフォーマンスが大きく向上する場合があります。

ストアード・プロシージャーを使用する

Java ではストアード・プロシージャーの使用がサポートされています。ストアード・プロシージャーは、JDBC ドライバーが動的 SQL の代わりに静的 SQL を実行できるようにすることによって、パフォーマンスを向上させることができます。ストアード・プロシージャーは、プログラムで実行する個々の SQL ステートメントごとには作成しないでください。ただし、可能な場合は、SQL ステートメントのグループを実行するストアード・プロシージャーを作成してください。

Numeric または Decimal の代わりに BigInt を使用する

スケールが 0 である Numeric フィールドまたは Decimal フィールドを使用する代わりに、BigInt データ・タイプを使用します。 BigInt は Java プリミティブ・タイプの Long に直接変換されますが、Numeric または Decimal データ・タイプは、 String オブジェクトまたは BigDecimal オブジェクトに変換されます。 173 ページの『DatabaseMetaData 呼び出しを避ける』で述べたように、プリミティブ・データ・タイプの使用は、オブジェクトの作成が必要なタイプの使用より望ましいと言えます。

必要がなくなった JDBC リソースを明示的にクローズする

ResultSets、Statements、および Connections は、必要がなくなったときにアプリケーションによって明示的にクローズする必要があります。これにより、リソースは最も効率的にクリーンアップされ、パフォーマンスを向上させることができます。さらに、データベース・リソースが明示的にクローズされないと、リソース・リークが生じたり、データベース・ロックの時間が必要以上に長くなったりします。これは、アプリケ

ーション障害の発生や、アプリケーションでの並行性の減少につながる可能性があります。

接続プーリングを使用する

Connection プーリングは、各ユーザー要求で独自の Connection オブジェクトを作成する代わりに、複数のユーザーで JDBC Connection オブジェクトを再利用する戦略です。 Connection オブジェクトを作成するには費用がかかります。各ユーザーが新規のオブジェクトを作成する代わりに、パフォーマンスが重要なアプリケーションでオブジェクトのプールを共用する必要があります。多くの製品 (WebSphere など) は Connection プーリング・サポートを提供しており、これは、ユーザーの側の少し余分な努力で使用できます。 Connection プーリング・サポートを持つ製品を使用しない場合や、プールの動作やパフォーマンスをよりよく制御するために独自のオブジェクトの作成を望む場合は、そのほうが合理的で容易でしょう。

PreparedStatement プーリングの使用を考慮する

Statement プーリングの動作は、Connection プーリングの動作と類似しています。ただし、Connection を単にプールに入れる代わりに、Connection および PreparedStatement を含むオブジェクトをプールに入れます。その後、そのオブジェクトを検索し、使用する特定のステートメントにアクセスします。これによりパフォーマンスは劇的に向上します。

効率的な SQL を使用する

JDBC は SQL に基づいて作成されているので、SQL の効率を上げることは、JDBC の効率を上げることになります。したがって、照会の最適化、賢明な索引の選択、および優れた SQL 設計は、JDBC にとって有益です。

DB2 SQLJ サポートを使用するデータベースへのアクセス

DB2 Structured Query Language for Java (SQLJ) サポートは、SQLJ ANSI 規格に基づいています。 DB2 SQLJ サポートは、IBM Developer Kit for Java に含まれています。 DB2 SQLJ サポートによって、Java アプリケーションの組み込み SQL を作成、構築、および実行することができます。

IBM Developer Kit for Java に備わっている SQLJ サポートには SQLJ ランタイム・クラスが含まれていて、それは /QIBM/ProdData/Java400/ext/runtime.zip から入手できます。

SQLJ のセットアップ

サーバー上の Java アプリケーションで SQLJ を使用するためには、まずサーバーで SQLJ を使用するための準備を行う必要があります。詳細は、「SQLJ セットアップ」のトピックを参照してください。

SQLJ ツール

以下のツールも、IBM Developer Kit for Java に備わっている SQLJ サポートに含まれています。

- SQLJ 変換プログラム `sqlj` は、SQLJ プログラムの組み込み SQL ステートメントを Java ソース・ステートメントに置き換えて、SQLJ プログラム内に見つかった SQLJ 操作に関する情報を含む逐次化プロファイルを生成します。
- DB2 SQLJ プロファイル・カスタマイザー `db2profc` は、生成されたプロファイルに格納されている SQL ステートメントをプリコンパイルし、DB2 データベース内にパッケージを生成します。
- DB2 SQLJ プロファイル・プリンター `db2profp` は、DB2 のカスタマイズ済みプロファイルの内容を通常のテキスト形式で印刷します。
- SQLJ プロファイル監査プログラム・インストーラー `profdb` は、デバッグ・クラス監査プログラムをバイナリー・プロファイルの既存のセット内にインストール、およびアンインストールします。

- SQLJ プロファイル変換ツール `profconv` は、逐次化プロファイルのインスタンスを Java クラス形式に変換します。

注: これらのツールは、Qshell インタープリターで実行しなければなりません。

DB2 SQLJ の制約事項

SQLJ を使用して DB2 アプリケーションを作成する場合、以下の制約事項に注意してください。

- DB2 SQLJ サポートは、SQL ステートメントを発行するための標準 DB2 Universal Database™ 制約に従っています。
- DB2 SQLJ プロファイル・カスタマイザーを実行できるのは、ローカル・データベースへの接続に関連したプロファイル上だけです。
- SQLJ Reference Implementation では、JDK 1.1 以降が必要です。複数のバージョンの Java Development Kit の実行について詳しくは、『複数の Java Development Kit (JDK) のサポート』を参照してください。

関連概念

『Structured Query Language for Java のプロファイル』
プロファイルは、SQLJ ソース・ファイルを変換するときに、SQLJ 変換プログラム sqlj によって生成されます。 プロファイルは、一連のバイナリー・ファイルです。 そのため、これらのファイルには .ser 拡張子があります。 これらのファイルには、関連した SQLJ ソース・ファイルからの SQL ステートメントが含まれます。

6 ページの『複数の Java Development Kit (JDK) のサポート』
IBM i プラットフォームでは、複数のバージョンの Java Development Kit (JDK) と Java 2 Platform, Standard Edition がサポートされています。

182 ページの『SQL ステートメントを Java アプリケーションに組み込む』
SQLJ 内の静的 SQL ステートメントは、SQLJ 文節に含まれています。 SQLJ 文節は、`#sql` で開始して、セミコロン (;) 文字で終了します。

関連タスク

189 ページの『SQLJ を使用するためのシステムのセットアップ』
組み込み SQLJ ステートメントを含む Java プログラムを実行する前に、SQLJ をサポートするようサーバーを設定してください。 SQLJ サポートのためには、サーバーの CLASSPATH 環境変数を変更する必要があります。

186 ページの『SQLJ プログラムのコンパイルおよび実行』
Java プログラムに組み込み SQLJ ステートメントがある場合、それをコンパイルおよび実行するためには特別の手順に従う必要があります。

Structured Query Language for Java のプロファイル

プロファイルは、SQLJ ソース・ファイルを変換するときに、SQLJ 変換プログラム sqlj によって生成されます。プロファイルは、一連のバイナリー・ファイルです。そのため、これらのファイルには .ser 拡張子があります。これらのファイルには、関連した SQLJ ソース・ファイルからの SQL ステートメントが含まれます。

SQLJ ソース・コードからプロファイルを生成するには、 177 ページの『Structured Query Language for Java (SQLJ) 変換プログラム (sqlj)』 を .sqlj ファイル上で実行します。

詳しくは、 186 ページの『SQLJ プログラムのコンパイルおよび実行』を参照してください。

Structured Query Language for Java (SQLJ) 変換プログラム (sqlj)

SQLJ 変換プログラム、sqlj は、SQLJ プログラム内で見つかった SQL 操作に関する情報を含む逐次化プロファイルを生成します。 SQLJ 変換プログラムは、/QIBM/ProdData/Java400/ext/translator.zip ファイルを使用します。

プロファイルの詳細については、プロファイルを参照してください。

DB2 SQLJ プロファイル・カスタマイザー、db2profc を使用するプロファイル内での SQL ステートメントのプリコンパイル

DB2 SQLJ プロファイル・カスタマイザー、db2profc を使用して、Java アプリケーションがデータベース内でより効率的に作動するようにすることができます。

DB2 SQLJ プロファイル・カスタマイザーは、以下の事柄を行います。

- プロファイル内に格納された SQL ステートメントをプリコンパイルして、DB2 データベース内にパッケージを生成する。
- 作成されたパッケージ内の関連したステートメントを参照する SQL ステートメントを置き換えることにより、SQLJ プロファイルをカスタマイズする。

プロファイル内の SQL ステートメントをプリコンパイルするためには、Qshell コマンド・プロンプトに以下を入力します。

```
db2profc MyClass_SJProfile0.ser
```

ここで、*MyClass_SJProfile0.ser* はプリコンパイルしたいプロファイルの名前です。

DB2 SQLJ プロファイル・カスタマイザーの使用法および構文

db2profc[*options*] <*SQLJ_profile_name*>

ここで、*SQLJ_profile_name* は印刷するプロファイル名、*options* は使用したいオプションのリストです。

db2profp で使用可能なオプションは、以下のとおりです。

- `-URL=<JDBC_URL>`
- `-user=<username>`
- `-password=<password>`
- `-package=<library_name/package_name>`
- `-commitctrl=<commitment_control>`
- `-datefmt=<date_format>`
- `-datesep=<date_separator>`
- `-timefmt=<time_format>`
- `-timesep=<time_separator>`
- `-decimalpt=<decimal_point>`
- `-stmtCCSID=<CCSID>`
- `-sorttbl=<library_name/sort_sequence_table_name>`
- `-langID=<language_identifier>`

以下は、これらのオプションに関する説明です。

-URL=<*JDBC_URL*>

ここで、*JDBC_URL* は JDBC 接続の URL です。 URL の構文は、次のとおりです。

```
"jdbc:db2:systemName"
```

詳しくは、 29 ページの『Java JDBC ドライバーを使用して IBM i データベースにアクセスする』を参照してください。

-user=<*username*>

ここで、*username* はユーザー名です。デフォルト値は、ローカル接続にサインオンした現行ユーザーのユーザー ID です。

-password=<*password*>

ここで、*password* はパスワードです。デフォルト値は、ローカル接続にサインオンした現行ユーザーのパスワードです。

-package=<*library name/package name*>

ここで、*library name* はパッケージを入れるライブラリー、*package name* は生成されるパッケージの名前です。デフォルトのライブラリー名は、QUSRSYS です。デフォルトのパッケージ名は、プロファイルの名前から生成されます。パッケージ名の最大長は、10 文字です。 SQLJ プロファイル名は常に 10 文字よりも長いので、作成されるデフォルトのパッケージ名はプロファイル名とは異なるものとなります。デフォルトのパッケージ名は、プロファイル名の最初の数文字とプロファイル・キー番号とを連結して作成されます。プロファイル・キー番号が 10 文字を超える長さである場合、プロファイル・キー番号の最後の 10 文字がデフォルトのパッケージ名に使用されます。たとえば、以下の図表は一部のプロファイル名とデフォルトのパッケージ名とを示しています。

プロファイル名	デフォルトのパッケージ名
App_SJProfile0	App_SJPro0
App_SJProfile01234	App_S01234
App_SJProfile012345678	A012345678
App_SJProfile01234567891	1234567891

-commitctrl=<*commitment_control*>

ここで、*commitment_control* は必要なコミットメント制御のレベルです。コミットメント制御は、以下の文字値の 1 つを持つことができます。

値	定義
C	*CHG。ダーティー・リード、繰り返し不可の読み取り、およびファントム・リードが可能。
S	*CS。ダーティー・リードは不可。繰り返し不可の読み取り、およびファントム・リードは可能。
A	*ALL。ダーティー・リードおよび繰り返し不可の読み取りは不可。ファントム・リードは可能。
N	*NONE。ダーティー・リード、繰り返し不可の読み取り、およびファントム・リードは不可。これはデフォルトです。

-datefmt=<*date_format*>

ここで、*date_format* は使用したい日付形式のタイプです。日付形式には、以下の値の 1 つを指定できます。

値	定義
USA	IBM USA 標準規格 (mm.dd.yyyy, hh:mm a.m., hh:mm p.m.)
ISO	国際標準化機構 (yyyy-mm-dd, hh.mm.ss)。これがデフォルトです。
EUR	IBM 欧州標準規格 (dd.mm.yyyy, hh.mm.ss)
JIS	日本工業規格 (JIS) 西暦 (yyyy-mm-dd, hh:mm:ss)
MDY	月/日/年 (mm/d/yy)
DMY	日/月/年 (dd/mm/yy)
YMD	年/月/日 (yy/mm/dd)
JUL	ユリウス暦 (yy/ddd)

日付形式は、日付の結果の欄にアクセスするときに使用されます。すべての出力日付フィールドは、指定した形式で戻されます。入力日付ストリングでは、日付が有効な形式で指定されたかどうかを判別するために指定値が使用されます。デフォルト値は ISO です。

-datesep=<*date_separator*>

ここで、*date_separator* は使用したい区切り記号のタイプです。日付区切り記号は、日付の結果の欄にアクセスするときに使用されます。区切り記号には、以下の値の 1 つを使用できます。

値	定義
/	スラッシュが使用される。
.	ピリオドが使用される。
,	コンマが使用される。
-	ダッシュが使用される。これはデフォルトです。
ブランク	スペースが使用される。

-timefmt=<*time_format*>

ここで、*time_format* は時刻フィールドの表示に使用したい形式です。時刻形式は、時刻の結果の欄にアクセスするときに使用されます。入力時刻ストリングでは、時刻が有効な形式で指定されたかどうかを判別するために指定値が使用されます。時刻形式には、以下の値の 1 つを指定できます。

値	定義
USA	IBM USA 標準規格 (mm.dd.yyyy, hh:mm a.m., hh:mm p.m.)
ISO	国際標準化機構 (yyyy-mm-dd, hh.mm.ss)。これがデフォルトです。
EUR	IBM 欧州標準規格 (dd.mm.yyyy, hh.mm.ss)
JIS	日本工業規格 (JIS) 西暦 (yyyy-mm-dd, hh:mm:ss)
HMS	時/分/秒 (hh:mm:ss)

-timesep=<*time_separator*>

ここで、*time_separator* は時刻の結果の欄にアクセスするときに使用したい文字です。時刻区切り記号には、以下の値の 1 つを指定できます。

値	定義
:	コロンが使用される。
.	ピリオドが使用される。これはデフォルトです。
,	コンマが使用される。
ブランク	スペースが使用される。

-decimalpt=<*decimal_point*>
ここで、*decimal_point* は使用したい小数点です。小数点は、SQL ステートメント内で数値定数に使用されます。小数点には、以下の値の 1 つを指定できます。

値	定義
.	ピリオドが使用される。これはデフォルトです。
,	コンマが使用される。

-stmtCCSID=<*CCSID*>
ここで、*CCSID* はパッケージ内に備わっている SQL ステートメントのためのコード化文字セット ID です。カスタマイズ時間中のジョブの値がデフォルト値となります。

-sorttbl=<*library_name/sort_sequence_table_name*>
ここで、*library_name/sort_sequence_table_name* は使用したい分類順序テーブルのロケーションおよびテーブル名です。分類順序テーブルは、SQL ステートメント内でストリングを比較するために使用されます。ライブラリー名および分類順序テーブル名は、それぞれ 10 文字の制限があります。デフォルト値は、カスタマイズ時間中のジョブから取得されます。

-langID=<*language_identifier*>
ここで、*language identifier* は使用したい言語 ID です。言語 ID のデフォルト値は、カスタマイズ時間中の現行ジョブから取得されます。言語 ID は、分類順序テーブルと一緒に使用します。

関連情報

SQL プログラミング

DB2 SQLJ プロファイル (db2profp および profp) の内容の印刷

DB2 SQLJ プロファイル・プリンター である db2profp は、DB2 のカスタマイズ済みプロファイルの内容を通常のテキスト形式で印刷します。プロファイル・プリンター、profp は、SQLJ 変換プログラムによって生成されたプロファイルの内容を通常のテキスト形式で印刷します。

SQLJ 変換プログラムによって生成されたプロファイルの内容を通常のテキスト形式で印刷するには、以下のように profp ユーティリティーを使用します。

```
profp MyClass_SJProfile0.ser
```

ここで、*MyClass_SJProfile0.ser* は印刷したいプロファイルの名前です。

プロファイルの DB2 カスタマイズ済みバージョンの内容を通常のテキスト形式で印刷するには、以下のように db2profp ユーティリティーを使用します。

```
db2profp MyClass_SJProfile0.ser
```

ここで、*MyClass_SJProfile0.ser* は印刷したいプロファイルの名前です。

注: db2profp をカスタマイズしていないプロファイル上で実行すると、プロファイルがカスタマイズされていないことが通知されます。 profp をカスタマイズ済みプロファイル上で実行すると、カスタマイズされていないプロファイルの内容が表示されます。

DB2 SQLJ プロファイル・プリンターの使用法および構文

```
db2profp [options] <SQLJ_profile_name>
```

ここで、*SQLJ_profile_name* は印刷するプロファイル名、*options* は使用したいオプションのリストです。

db2profp で使用可能なオプションは、以下のとおりです。

-URL=<*JDBC_URL*>
: ここで、*JDBC_URL* は接続したい URL です。詳しくは、 29 ページの『Java JDBC ドライバーを使用して IBM i データベースにアクセスする』を参照してください。

-user=<*username*>
: ここで、*username* はユーザー・プロファイル内のユーザー名です。

-password=<*password*>
: ここで、*password* はユーザー・プロファイルのパスワードです。

SQLJ プロファイル監査プログラム・インストーラー (profdb)

SQLJ プロファイル監査プログラム・インストーラー (profdb) は、デバッグ・クラス監査プログラムをインストール、およびアンインストールします。デバッグ・クラス監査プログラムは、バイナリー・プロファイルの既存のセットにインストールされます。デバッグ・クラス監査プログラムがインストールされた後は、アプリケーションの実行時間中に呼び出されるすべての RTStatement および RTResultSet 呼び出しがログに記録されます。それらはファイルまたは標準出力にログ記録することができます。その後、ログを検査してアプリケーションの動作の検証、およびエラーのトレースを行うことができます。実行時の基礎となる RTStatement および RTResultSetcall インターフェースに対する呼び出しだけが監査されることに注意してください。

デバッグ・クラス監査プログラムをインストールするには、Qshell コマンド・プロンプトに以下を入力します。

```
profdb MyClass_SJProfile0.ser
```

ここで、*MyClass_SJProfile0.ser* は SQLJ 変換プログラムによって生成されたプロファイルの名前です。

デバッグ・クラス監査プログラムをアンインストールするには、Qshell コマンド・プロンプトに以下を入力します。

```
profdb -Cuninstall MyClass_SJProfile.ser
```

ここで、*MyClass_SJProfile0.ser* は SQLJ 変換プログラムによって生成されたプロファイルの名前です。

SQLJ プロファイル変換ツール (profconv) を使用して、逐次化プロファイルのインスタンスを Java クラス形式に変換する

SQLJ プロファイル変換ツール (profconv) は、逐次化プロファイルのインスタンスを Java クラス形式に変換します。一部のブラウザーはアプレットに関連したリソース・ファイルから逐次化オブジェクトをロードすることをサポートしていないため、profconv ツールが必要になります。 profconv ユーティリティーを実行して、変換を行います。

profconv ユーティリティーを実行するには、Qshell コマンド行に以下を入力します。

```
profconv MyApp_SJProfile0.ser
```

ここで、*MyApp_SJProfile0.ser* は変換したいプロファイル・インスタンスの名前です。

profconv ツールは、sqlj -ser2class を起動します。コマンド行オプションについては、sqljを参照してください。

SQL ステートメントを Java アプリケーションに組み込む

SQLJ 内の静的 SQL ステートメントは、SQLJ 文節に含まれています。 SQLJ 文節は、#sql で開始して、セミコロン (;) 文字で終了します。

Java アプリケーション内に SQLJ 文節を作成する前に、以下のパッケージをインポートしてください。

- import java.sql.*;
- import sqlj.runtime.*;
- import sqlj.runtime.ref.*;

最も簡単な SQLJ 文節は、処理できる文節であり、トークン #sql およびそれに続く中括弧で囲まれた SQL ステートメントで構成されます。たとえば、以下の SQLJ 文節は Java ステートメントが適正に存在できる位置であればどこにでも存在できます。

```
#sql { DELETE FROM TAB };
```

上記の例は、TAB という名前のすべての行を削除します。

SQLJ プロセス文節で中括弧の内側にあるトークンは、SQL トークンまたはホスト変数です。すべてのホスト変数は、コロン (:) 文字によって識別されます。 SQL トークンが SQLJ プロセス文節の中括弧外側に存在することはありません。たとえば、以下の Java メソッドは引数を SQL テーブルに挿入します。

```
public void insertIntoTAB1 (int x, String y, float z) throws SQLException
{
    #sql { INSERT INTO TAB1 VALUES (:x, :y, :z) };
}
```

メソッド本体は、ホスト変数 x、y、および z を含む SQLJ プロセス文節で構成されます。

一般に、SQL トークンは大文字小文字を区別しない (二重引用符で囲まれた ID を除く) ので、大文字、小文字、またはそれらの混合による記述が可能です。しかし、*Java* トークンは大文字小文字を区別します。例の中で明白にするために、大文字小文字を区別しない SQL トークンは大文字で示し、Java トークンは小文字または大文字小文字混合で示します。このトピックを通して、小文字の null は Java "null" 値を表すために使用され、大文字の NULL は SQL "null" 値を表すために使用されます。

以下のタイプの SQL 構成が、 SQLJ プログラム内に存在することがあります。
- 照会: SELECT ステートメントおよび式など。
- SQL データ変更ステートメント (DML): INSERT、UPDATE、DELETE、など。
- データ・ステートメント: FETCH、SELECT..INTO、など。
- トランザクション制御ステートメント: COMMIT、ROLLBACK、など。
- データ定義言語 (DDL、スキーマ操作言語とも呼ばれる) ステートメント: CREATE、DROP、ALTER など。
- ストアード・プロシージャーへの呼び出し: CALL MYPROC(:x, :y, :z) など
- ストアード関数の呼び出し: VALUES(MYFUN(:x)) など

Structured Query Language for Java 内のホスト変数:

組み込み SQL ステートメントへの引数は、ホスト変数を介して渡されます。ホスト変数は、ホスト言語による変数であり、SQL ステートメントに含めることができます。

ホスト変数は、以下に示す 3 つまでの部分から構成されます。

- コロン (:) 接頭部。
- パラメーター、変数、またはフィールドの Java ID である、Java ホスト変数。
- 任意指定のパラメーター・モード ID。

  このモード ID には、以下の 1 つを使用できます。
  IN、OUT、または INOUT。

Java ID を評価しても Java プログラムには副次作用がありません。そのためそれは、SQLJ 文節を置き換えるために生成された Java コード内に何回も出現することがあります。

以下の QUERY にはホスト変数 :x が含まれています。このホスト変数は、照会を含むスコープ内で可視の Java 変数、フィールド、またはパラメーター :x です。

```
SELECT COL1, COL2 FROM TABLE1 WHERE :x > COL3
```

例: SQL ステートメントを Java アプリケーションに組み込む:

以下の SQLJ アプリケーション例、App.sqlj は、静的 SQL を使用して更新データを DB2 サンプル・データベースの EMPLOYEE テーブルから検索します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

#sql iterator App_Cursor1 (String empno, String firstnme) ; // 1
#sql iterator App_Cursor2 (String) ;

class App
{

  /**********************
   **  Register Driver **
   **********************/

 static
 {
   try
   {
     Class.forName("com.ibm.db2.jdbc.app.DB2Driver").newInstance();
   }
   catch (Exception e)
   {
     e.printStackTrace();
   }
 }

  /********************
   **      Main      **
   ********************/

 public static void main(String argv[])
 {
   try
   {
```

```
App_Cursor1 cursor1;
App_Cursor2 cursor2;

String str1 = null;
String str2 = null;
long   count1;

// URL is jdbc:db2:dbname
String url = "jdbc:db2:sample";

DefaultContext ctx = DefaultContext.getDefaultContext();
if (ctx == null)
{
  try
  {
    // connect with default id/password
    Connection con = DriverManager.getConnection(url);
    con.setAutoCommit(false);
    ctx = new DefaultContext(con);
  }
  catch (SQLException e)
  {
    System.out.println("Error: could not get a default context");
    System.err.println(e) ;
    System.exit(1);
  }
  DefaultContext.setDefaultContext(ctx);
}

// retrieve data from the database
System.out.println("Retrieve some data from the database.");
#sql cursor1 = {SELECT empno, firstnme FROM employee}; // 2

// display the result set
// cursor1.next() returns false when there are no more rows
System.out.println("Received results:");
while (cursor1.next()) // 3
{
  str1 = cursor1.empno(); // 4
  str2 = cursor1.firstnme();

  System.out.print (" empno= " + str1);
  System.out.print (" firstname= " + str2);
  System.out.println("");
}
cursor1.close(); // 9

// retrieve number of employee from the database
#sql { SELECT count(*) into :count1 FROM employee }; // 5
if (1 == count1)
  System.out.println ("There is 1 row in employee table");
else
  System.out.println ("There are " + count1
                      + " rows in employee table");

// update the database
System.out.println("Update the database.");
#sql { UPDATE employee SET firstnme = 'SHILI' WHERE empno = '000010' };

// retrieve the updated data from the database
System.out.println("Retrieve the updated data from the database.");
str1 = "000010";
#sql cursor2 = {SELECT firstnme FROM employee WHERE empno = :str1}; // 6

// display the result set
// cursor2.next() returns false when there are no more rows
System.out.println("Received results:");
```

```
    while (true)
    {
      #sql { FETCH :cursor2 INTO :str2 }; // 7
      if (cursor2.endFetch()) break; // 8

      System.out.print (" empno= " + str1);
      System.out.print (" firstname= " + str2);
      System.out.println("");
    }
    cursor2.close(); // 9

    // rollback the update
    System.out.println("Rollback the update.");
    #sql { ROLLBACK work };
    System.out.println("Rollback done.");
  }
  catch( Exception e )
  {
    e.printStackTrace();
  }
 }
}
```

[1] 反復子を宣言する。このセクションでは、次の 2 種類の反復子を宣言します。

- App_Cursor1: 列データのタイプおよび名前を宣言して、列名 (列に結び付けられた名前) に応じた列の値を戻します。
- App_Cursor2: 列データのタイプを宣言して、列位置 (列に結び付けられた定位置) に応じた列の値を戻します。

[2] 反復子を初期設定する。反復子オブジェクト cursor1 が照会の結果を使用して初期設定されます。照会は結果を cursor1 に格納します。

[3] 反復子を次の行に進める。 cursor1.next() メソッドは、検索する行がなくなった場合にブール値の偽を戻します。

[4] データを移動する。名前付きアクセス機構メソッド empno() は、現在の行にある empno という名前の列の値を戻します。名前付きアクセス機構メソッド firstnme() は、現在の行にある firstnme() という名前の列の値を戻します。

[5] データをホスト変数に SELECT する。 SELECT ステートメントは、テーブル内の行数をホスト変数 count1 に渡します。

[6] 反復子を初期設定する。反復子オブジェクト cursor2 が照会の結果を使用して初期設定されます。照会は結果を cursor2 に格納します。

[7] データを検索する。 FETCH ステートメントは、結果テーブルから ByPos カーソル内で宣言された最初の列の現行値を、ホスト変数 str2 に戻します。

[8] FETCH.INTO ステートメントが成功したかを検査する。 endFetch() メソッドは、反復子が行に位置していない場合、つまり行を取り出す前回の試行が失敗した場合に、ブール値の真を戻します。 endFetch() メソッドは、行を取り出す前回の試行が成功した場合に、偽を戻します。 DB2 は next() メソッドが呼び出されたときに行の取り出しを試行します。 FETCH...INTO ステートメントは、暗黙的に next() メソッドを呼び出します。

[9] 反復子をクローズする。 close() メソッドは、反復子が保持しているリソースを解放します。反復子を明示的にクローズして、システム・リソースが適時に解放されるようにしてください。

SQLJ プログラムのコンパイルおよび実行

Java プログラムに組み込み SQLJ ステートメントがある場合、それをコンパイルおよび実行するためには特別の手順に従う必要があります。

Java プログラムに組み込み SQLJ ステートメントがある場合、それをコンパイルおよび実行するためには特別の手順に従う必要があります。

1. SQLJ を使用するためのサーバーのセットアップを行います。
2. SQLJ 変換プログラム、sqlj を Java ソース・コード上で組み込み SQL と共に使用して、Java ソース・コードおよび関連したプロファイルを生成します。接続ごとに、1 つのプロファイルが生成されます。

   たとえば、以下のコマンドを入力します。

   ```
   sqlj MyClass.sqlj
   ```

   ここで、*MyClass.sqlj* は SQLJ ファイルの名前です。

   この例では、SQLJ 変換プログラムは MyClass.java ソース・コード・ファイル、および関連したプロファイルを生成します。関連したプロファイルの名前は、MyClass_SJProfile0.ser、MyClass_SJProfile1.ser、MyClass_SJProfile2.ser、以下同様となります。

   注: SQLJ 変換プログラムは、-compile=false 文節によってコンパイル・オプションを明示的にオフにしなければ、変換済み Java ソース・コードを自動的にコンパイルしてクラス・ファイルを生成します。
3. SQLJ プロファイル・カスタマイザー・ツール、db2profc を使用して、DB2 SQLJ Customizers を生成されたプロファイルにインストールして、DB2 パッケージをローカル・システム上に作成します。

   たとえば、以下のコマンドを入力します。

   ```
   db2profc MyClass_SJProfile0.ser
   ```

   ここで *MyClass_SJProfile0.ser* は DB2 SQLJ Customizer が実行されるプロファイルの名前です。

   注: この手順はオプションですが、実行時パフォーマンスを向上させるために勧められています。
4. Java クラス・ファイルを他の Java クラス・ファイルと同じ方法で実行します。

   たとえば、以下のコマンドを入力します。

   ```
   java MyClass
   ```

   ここで、*MyClass* は Java クラス・ファイルの名前です。

関連概念

182 ページの『SQL ステートメントを Java アプリケーションに組み込む』
SQLJ 内の静的 SQL ステートメントは、SQLJ 文節に含まれています。 SQLJ 文節は、#sql で開始して、セミコロン (;) 文字で終了します。

Java SQL ルーチン

ご使用のシステムには、SQL ステートメントやプログラムから Java プログラムにアクセスする機能があります。これは、Java ストアード・プロシージャーおよび Java ユーザー定義関数 (UDF) を使用して行います。 IBM i は、Java ストアード・プロシージャーと Java UDF を呼び出すための DB2 の規則と SQLJ の規則を両方サポートしています。 Java ストアード・プロシージャーと Java UDF は両方とも、JAR フ

ァイルに保管されている Java クラスを使用できます。 IBM i は、JAR ファイルのデータベースへの登録に、*SQLJ Part 1* 規格で定義されたストアード・プロシージャーを使用します。

Java SQL ルーチンの使用

Java プログラムには、SQL ステートメントおよびプログラムからアクセスできます。これは、Java ストアード・プロシージャーおよび Java ユーザー定義関数 (UDF) を使用して行います。

Java SQL ルーチンを使用するには、以下の操作を行ってください。

1. SQLJ を使用可能にする。

   Java SQL ルーチンはどれも SQLJ を使用する可能性があるので、Java 2 Platform, Standard Edition (J2SE) の実行時には、SQLJ ランタイム・サポートを常に使用可能にしておいてください。 J2SE で SQLJ のランタイム・サポートを使用可能にするには、拡張ディレクトリーから SQLJ runtime.zip ファイルへのリンクを追加します。詳しくは、『SQLJ を使用するためのシステムのセットアップ』を参照してください。

2. ルーチン用の Java メソッドを作成する。

   Java SQL ルーチンは、Java メソッドを SQL から処理します。このメソッドは、DB2 for i パラメーター引き渡し規則または SQLJ パラメーター引き渡し規則を使用して作成する必要があります。Java SQL ルーチンで使用するメソッドのコーディングについて詳しくは、Java ストアード・プロシージャー、Java ユーザー定義関数、および Java ユーザー定義表関数を参照してください。

3. Java クラスをコンパイルする。

   Java パラメーター・スタイルを使用して作成された Java SQL ルーチンは、追加のセットアップなしでコンパイルできます。ただし、DB2GENERAL パラメーター・スタイルを使用した Java SQL ルーチンの場合は、com.ibm.db2.app.UDF クラスまたは com.ibm.db2.app.StoredProc クラスを拡張する必要があります。これらのクラスは、/QIBM/ProdData/Java400/ext/db2routines_classes.jar という JAR ファイルに入っています。これらのルーチンをコンパイルするために javac を使用する場合は、この JAR ファイルが CLASSPATH の中になければなりません。たとえば、次のコマンドは、DB2GENERAL パラメーター・スタイルを使用するルーチンが入った Java ソース・ファイルをコンパイルします。

   ```
   javac -DCLASSPATH=/QIBM/ProdData/Java400/ext/db2routines_classes.jar
   source.java
   ```

4. データベースによって使用される Java 仮想マシン (JVM) が、コンパイル済みクラスを使用できるようにする。

   データベース JVM によって使用されるユーザー定義クラスは、/QIBM/UserData/OS400/SQLLib/Function ディレクトリーか、データベースに登録されている JAR ファイルに置くことができます。

   IBM i では、他のプラットフォームで DB2 for i が Java ストアード・プロシージャーや Java UDF の保管に使用する /sqllib/function ディレクトリーに相当するディレクトリーとして、/QIBM/UserData/OS400/SQLLib/Function が使用されます。クラスが Java パッケージの一部である場合は、クラスは適切なサブディレクトリーになければなりません。たとえば、runit クラスが foo.bar パッケージの一部として作成される場合、ファイル runnit.class は統合ファイル・システムの /QIBM/ProdData/OS400/SQLLib/Function/foo/bar というディレクトリーになければなりません。

   クラス・ファイルは、データベースに登録されている JAR ファイルに置くこともできます。 JAR ファイルは、SQLJ.INSTALL_JAR ストアード・プロシージャーを使用して登録します。このストアード・プロシージャーは、JAR ID を JAR ファイルに割り当てるために使用されます。この JAR ID は、クラス・ファイルが存在する JAR ファイルを識別するために使用されます。 SQLJ.INSTALL_JAR と、

JAR ファイルを操作するための他のストアード・プロシージャーの詳細については、JAR ファイルを操作する SQLJ プロシージャーを参照してください。

5. ルーチンをデータベースに登録する。

   Java SQL ルーチンは、CREATE PROCEDURE および CREATE FUNCTION SQL ステートメントを使用してデータベースに登録します。これらのステートメントには、以下の要素が含まれます。

   CREATE キーワード
   Java SQL ルーチンを作成するための SQL ステートメントは、CREATE PROCEDURE または CREATE STATEMENT で始まります。

   ルーチンの名前
   次いで、SQL ステートメントは、データベースに認識されているルーチンの名前を識別します。これは、SQL から Java ルーチンにアクセスするために使用される名前です。

   パラメーターおよび戻り値
   次いで、SQL ステートメントは、Java ルーチン用のパラメーターおよび戻り値 (該当する場合) を識別します。

   LANGUAGE JAVA
   SQL ステートメントは、ルーチンが Java で作成されたことを示すために、キーワード LANGUAGE JAVA を使用します。

   PARAMETER STYLE KEYWORDS
   次いで、SQL ステートメントは、キーワード PARAMETER STYLE JAVA または PARAMETER STYLE DB2GENERAL を使用して、パラメーター・スタイルを識別します。

   外部名 次いで、SQL ステートメントは、Java SQL ルーチンとして処理される Java メソッドを識別します。外部名の形式は、以下のどちらかになります。

   - メソッドが /QIBM/UserData/OS400/SQLLib/Function ディレクトリーの下のクラス・ファイルに存在する場合、メソッドは *classname.methodname* という形式で識別されます。ここで、*classname* はクラスの完全修飾名で、*methodname* はメソッドの名前です。
   - メソッドがデータベースに登録されている JAR ファイルに存在する場合、メソッドは *jarid*:*classname.methodname* という形式で識別されます。ここで、*jarid* は登録されている JAR ファイルの JAR ID、*classname* はクラスの名前、*methodname* はメソッドの名前です。

   System i Navigatorを使用して、Java パラメーター・スタイルを使用するストアード・プロシージャーやユーザー定義関数を作成することができます。

6. Java プロシージャーを使用する。

   Java ストアード・プロシージャーは、SQL CALL ステートメントを使用して呼び出します。 Java UDF は、別の SQL ステートメントの一部として呼び出される関数です。

189 ページの『SQLJ を使用するためのシステムのセットアップ』
組み込み SQLJ ステートメントを含む Java プログラムを実行する前に、SQLJ をサポートするようサーバーを設定してください。 SQLJ サポートのためには、サーバーの CLASSPATH 環境変数を変更する必要があります。

189 ページの『Java ストアード・プロシージャー』
Java を使用してストアード・プロシージャーを作成する場合、パラメーターを渡すスタイルを 2 つ使用できます。

194 ページの『Java ユーザー定義スカラー関数』
Java スカラー関数は、Java プログラムから 1 つの値をデータベースに戻します。 たとえば、2 つの数の合計を戻す、スカラー関数を作成できます。

199 ページの『Java ユーザー定義テーブル関数』
DB2 は、テーブルを戻す機能を関数に提供します。 この機能は、データベースの外部からデータベースにテーブル形式で情報を公開するのに役立ちます。 たとえば、Java ストアード・プロシージャーと Java UDF (テーブルとスカラーの両方) で使用される、Java 仮想マシン (JVM) 中のプロパティー設定を公開するテーブルを作成できます。

201 ページの『JAR ファイルを操作する SQLJ プロシージャー』
Java ストアード・プロシージャーと Java UDF は両方とも、Java JAR ファイルに保管されている Java クラスを使用できます。

SQLJ を使用するためのシステムのセットアップ:

組み込み SQLJ ステートメントを含む Java プログラムを実行する前に、SQLJ をサポートするようサーバーを設定してください。 SQLJ サポートのためには、サーバーの CLASSPATH 環境変数を変更する必要があります。

Java クラスパスの処理について詳しくは、以下のページを参照してください。

Java クラスパス

SQLJ と J2SE の使用

サポートされている任意のバージョンの J2SE を実行しているサーバーで SQLJ を使用するには、以下のステップを実行してください。

1. 次のファイルをサーバーの CLASSPATH 環境変数に追加します。
   - /QIBM/ProdData/Os400/Java400/ext/sqlj_classes.jar
   - /QIBM/ProdData/Os400/Java400/ext/translator.zip

   注: translator.zip は、SQLJ 変換プログラム (sqlj コマンド) を実行する場合にのみ、追加する必要があります。SQLJ を使用するコンパイル済みの Java プログラムだけを実行する場合は translator.zip を追加する必要はありません。詳しくは、『SQLJ 変換プログラム (sqlj)』を参照してください。
2. IBM i コマンド・プロンプトで、以下のコマンドを使用して、拡張機能ディレクトリーから runtime.zip へのリンクを追加します。コマンドを 1 行で入力してから、**Enter** を押します。

   ```
   ADDLNK OBJ('/QIBM/ProdData/Os400/Java400/ext/runtime.zip')
   NEWLNK('/QIBM/UserData/Java400/ext/runtime.zip')
   ```

   拡張機能のインストールについて詳しくは、以下のページを参照してください。

   IBM Developer Kit for Java の拡張機能をインストールする

Java ストアード・プロシージャー

Java を使用してストアード・プロシージャーを作成する場合、パラメーターを渡すスタイルを 2 つ使用できます。

推奨されているスタイルは JAVA パラメーター・スタイルです。これは、SQLj (SQL ルーチンの標準) で指定されているパラメーター・スタイルに一致します。 2 番目のスタイルは DB2GENERAL です。これは、DB2 UDB によって定義されているパラメーター・スタイルです。パラメーター・スタイルでは、Java ストアード・プロシージャーのコーディング時に使用しなければならない規則についても決定します。

さらに、Java ストアード・プロシージャーに関するいくつかの制限も意識していなければなりません。

JAVA パラメーター・スタイル:

JAVA パラメーター・スタイルを使用する Java ストアード・プロシージャーをコーディングするときは、以下の規則に従う必要があります。

- Java メソッドは public void static (インスタンスではない) メソッドである必要がある。
- Java メソッドのパラメーターは、SQL 互換タイプである必要がある。
- パラメーターがヌル互換タイプ (String など) の場合、Java メソッドは SQL NULL 値をテストできる。
- 出力パラメーターは、単一エレメント配列の使用により戻される。
- Java メソッドは getConnection メソッドを使って、現行データベースにアクセスできる。

JAVA パラメーター・スタイルを使用する Java ストアード・プロシージャーは、public static メソッドです。このクラスの中では、ストアード・プロシージャーはそのメソッド名とシグニチャーによって識別されます。ストアード・プロシージャーを呼び出すときは、シグニチャーは CREATE PROCEDURE ステートメントによって定義された変数タイプに基づいて、動的に生成されます。

ヌル値を許可する Java タイプでパラメーターが渡される場合、Java メソッドは、そのパラメーターをヌルと比較して、入力パラメーターが SQL NULL かどうかを判別できます。

以下の Java タイプはヌル値をサポートしていません。

- short
- int
- long
- float
- double

ヌル値をサポートしていない Java タイプにヌル値が渡されると、エラー・コード -20205 で SQL 例外が戻されます。

出力パラメーターは、1 つのエレメントを含む配列として渡されます。 Java ストアード・プロシージャーは配列の最初のエレメントを設定して、出力パラメーターを設定できます。

組み込みアプリケーション・コンテキストへの接続には、以下の Java Database Connectivity (JDBC) 呼び出しを使用してアクセスされます。

```
connection=DriverManager.getConnection("jdbc:default:connection");
```

その後、この接続は JDBC API を使って SQL ステートメントを実行します。

以下に示すのは、1 つの入力と 2 つの出力を持った小さなストアード・プロシージャーです。これは与えられた SQL 照会を実行し、結果内の行数と SQLSTATE の両方を戻します。

例: 1 つの入力と 2 つの出力を持つストアード・プロシージャー

注: サンプル・コードをご使用の場合は、560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
package mystuff;

import java.sql.*;
```

```
public class sample2 {
  public static void donut(String query, int[] rowCount,
    String[] sqlstate) throws Exception {
  try {
    Connection c=DriverManager.getConnection("jdbc:default:connection");
    Statement s=c.createStatement();
    ResultSet r=s.executeQuery(query);
    int counter=0;
    while(r.next()){
      counter++;
    }
    r.close(); s.close();
    rowCount[0] = counter;
    }catch(SQLException x){
    sqlstate[0]= x.getSQLState();
   }
  }
 }
```

SQLj 標準では、JAVA パラメーター・スタイルを使用するルーチン内の ResultSet を戻すために、ResultSet を明示的に設定しなければなりません。 ResultSet を戻すプロシージャーが作成される場合、ResultSet パラメーターがパラメーター・リストの最後に追加されます。たとえば、以下のステートメントの場合、

```
CREATE PROCEDURE RETURNTWO()
DYNAMIC RESULT SETS 2
LANGUAGE JAVA
PARAMETER STYLE JAVA
EXTERNAL NAME 'javaClass!returnTwoResultSets'
```

シグニチャー `public static void returnTwoResultSets(ResultSet[] rs1, ResultSet[] rs2)` で Java メソッドを呼び出します。

ResultSet の出力パラメーターは、以下の例で示されているように、明示的に設定しなければなりません。DB2GENERAL スタイルのように、ResultSet および対応するステートメントをクローズしてはなりません。

例: 2 つの ResultSet を戻すストアード・プロシージャー

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;
public class javaClass {
  /**
    * Java stored procedure, with JAVA style parameters,
    * that processes two predefined sentences
    * and returns two result sets
    *
    * @param ResultSet[] rs1    first ResultSet
    * @param ResultSet[] rs2    second ResultSet
    */
public static void returnTwoResultSets (ResultSet[] rs1, ResultSet[] rs2) throws Exception
{
  //get caller's connection to the database; inherited from StoredProc
  Connection con = DriverManager.getConnection("jdbc:default:connection");

  //define and process the first select statement
  Statement stmt1 = con.createStatement();
  String sql1 = "select value from table01 where index=1";
  rs1[0] = stmt1.executeQuery(sql1);

  //define and process the second select statement
```

```
  Statement stmt2 = con.createStatement();
  Stringsql2 = "select value from table01 where index=2";
  rs2[0] = stmt2.executeQuery(sql2);
  }
}
```

サーバーで、ResultSet の順序付けを判別するために、追加の ResultSet パラメーターが調べられることはありません。サーバー上の ResultSet は、オープンされた順序で戻されます。 SQLj 標準との互換性を確保するために、前述のように、オープンされる順序で結果が割り当てられなければなりません。

DB2GENERAL パラメーター・スタイル:

DB2GENERAL パラメーター・スタイルを使った Java ストアード・プロシージャーをコーディングするときは、これらの規則に従う必要があります。

- Java ストアード・プロシージャーで定義されるクラスは、Java com.ibm.db2.app.StoredProc クラスの *extend*、またはサブクラスである必要がある。
- Java メソッドは、public void インスタンス・メソッドである必要がある。
- Java メソッドのパラメーターは、SQL 互換タイプである必要がある。
- Java メソッドは、isNull メソッドを使って SQL NULL 値をテストすることができる。
- Java メソッドは、set メソッドを使って、明示的にパラメーターを設定および戻す必要がある。
- Java メソッドは getConnection メソッドを使って、現行データベースにアクセスできる。

Java ストアード・プロシージャーを含むクラスは、com.ibm.db2.app.StoredProc を拡張したクラスである必要があります。 Java ストアード・プロシージャーは public インスタンス・メソッドです。このクラスの中では、ストアード・プロシージャーはそのメソッド名とシグニチャーによって識別されます。ストアード・プロシージャーを呼び出すときは、シグニチャーは CREATE PROCEDURE ステートメントによって定義された変数タイプに基づいて、動的に生成されます。

com.ibm.db2.app.StoredProc クラスは、isNull メソッドを提供しており、Java メソッドは入力パラメーターが SQL NULL であった場合に判別できます。 com.ibm.db2.app.StoredProc クラスには、出力パラメーターを設定するための set...() メソッドも提供されています。出力パラメーターを設定するには、これらのメソッドを使用しなければなりません。出力パラメーターを設定しない場合は、出力パラメーターは SQL NULL 値を戻します。

com.ibm.db2.app.StoredProc クラスは、組み込みアプリケーション・コンテキストへの JDBC 接続をフェッチするための以下のルーチンを提供しています。組み込みアプリケーション・コンテキストへの接続は、以下の JDBC 呼び出しを使ってアクセスされます。

```
public Java.sql.Connection getConnection( )
```

その後、この接続は JDBC API を使って SQL ステートメントを実行します。

以下に示すのは、1 つの入力と 2 つの出力を持った小さなストアード・プロシージャーです。これは与えられた SQL 照会を処理し、結果内の行の数と SQLSTATE の両方を戻します。

例: 1 つの入力と 2 つの出力を持つストアード・プロシージャー

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
package mystuff;

import com.ibm.db2.app.*;
```

```
import java.sql.*;
public class sample2 extends StoredProc {
  public void donut(String query, int rowCount,
    String sqlstate) throws Exception {
  try {
    Statement s=getConnection().createStatement();
    ResultSet r=s.executeQuery(query);
    int counter=0;
    while(r.next()){
      counter++;
    }
    r.close(); s.close();
    set(2, counter);
    }catch(SQLException x){
    set(3, x.getSQLState());
    }
  }
}
```

DB2GENERAL パラメーター・スタイルを使ったプロシージャー内の ResultSet を戻すには、プロシージャーの終了時に、ResultSet および応答するステートメントが開かれたままになっている必要があります。戻される ResultSet は、クライアント・アプリケーションによってクローズされる必要があります。複数の ResultSet が戻される場合は、ResultSet が開かれた順序で戻されます。たとえば、以下のストアード・プロシージャーは 2 つの ResultSet を戻します。

例: 2 つの ResultSet を戻すストアード・プロシージャー

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
public void returnTwoResultSets() throws Exception
{
  // get caller's connection to the database; inherited from StoredProc
  Connection con = getConnection ();
  Statement stmt1 = con.createStatement ();
  String sql1 = "select value from table01 where index=1";
  ResultSet rs1 = stmt1.executeQuery(sql1);
  Statement stmt2 = con.createStatement();
  String sql2 = "select value from table01 where index=2";
  ResultSet rs2 = stmt2.executeQuery(sql2);
}
```

Java ストアード・プロシージャーの制限:

これらの制限は、Java ストアード・プロシージャーに適用されます。

- Java ストアード・プロシージャーは追加スレッドを作成できません。追加のスレッドを作成できるのは、ジョブがマルチスレッド可能な場合だけです。 SQL ストアード・プロシージャーを呼び出すジョブで、マルチスレッドが可能であるとは限らないので、Java ストアード・プロシージャーは、追加スレッドを作成できません。
- Java クラス・ファイルにアクセスするために、借用権限を使用することはできません。
- Java ストアード・プロシージャーは、`java` コマンドと同じデフォルト・バージョンの JDK を使用します。必要であれば、Java ストアード・プロシージャーで使用する JDK のバージョンは SystemDefault.properties ファイルを使用して変更することができます。
- Blob クラスと Clob クラスは java.sql パッケージと com.ibm.db2.app パッケージの両方にあるので、同一のプログラム中でこれらの両方のクラスを使用する場合は、プログラマーはこれらのクラスの名前全

体を使用しなければならない。 com.ibm.db2.app 中の Blob クラスと Clob クラスが、ストアード・プロシージャーに渡されるパラメーターとして使用されていることを、プログラムが確認しなければなりません。

- Java ストアード・プロシージャーが作成される場合、システムはライブラリー内にサービス・プログラムを生成します。このプログラムは、プロシージャー定義を保管するのに使用されます。プログラムには、システムによって生成された名前が付けられています。この名前は、ストアード・プロシージャーを作成したジョブのジョブ・ログを調べることにより、取得できます。プログラムを保管してから復元すると、プロシージャー定義が復元されます。 Java ストアード・プロシージャーをシステム間で移動する場合は、Java クラスが含まれる統合ファイル・システムに加えて、関数の定義が含まれるサービス・プログラムも移動する必要があります。
- Java ストアード・プロシージャーは、データベースへの接続に使用される、JDBC 接続のプロパティー (たとえば、システムの命名など) を設定できません。事前取り出しが使用不可の場合を除き、デフォルトの JDBC 接続プロパティーが常に使用されます。

Java ユーザー定義スカラー関数

Java スカラー関数は、Java プログラムから 1 つの値をデータベースに戻します。たとえば、2 つの数の合計を戻す、スカラー関数を作成できます。

Java ストアード・プロシージャーのように、Java スカラー関数は、Java と DB2GENERAL という 2 つのパラメーター・スタイルのいずれかを使用します。 Java ユーザー定義関数 (UDF) をコード化する場合、Java スカラー関数に関する制限に注意しなければなりません。

パラメーター・スタイル Java

Java パラメーター・スタイルは、*SQLJ Part 1: SQL Routines* 標準で指定されているスタイルです。 Java UDF をコード化する場合、以下の規則を使用してください。

- Java メソッドは public static メソッドである必要がある。
- Java メソッドは SQL 互換タイプを戻す必要がある。この戻り値がこのメソッドの結果です。
- Java メソッドのパラメーターは SQL 互換タイプである必要がある。
- ヌル値が許可されている Java タイプの場合、Java メソッドは SQL NULL 値をテストすることができる。

たとえば、INTEGER を戻し、タイプ CHAR(5)、BLOB(10K)、および DATE の引数を取る、sample!test3 という UDF の場合、DB2 では、UDF の Java インプリメンテーションで以下のシグニチャーが必要です。

```
import com.ibm.db2.app.*;
public class sample {
   public static int test3(String arg1, Blob arg2, Date arg3) { ... }
}
```

Java メソッドのパラメーターは SQL 互換タイプでなければなりません。たとえば、UDF が SQL タイプ t1、t2、および t3 の引数を取り、タイプ t4 を戻すように宣言されている場合、以下のように、必要な Java シグニチャーを持った Java メソッドとして呼び出されます。

```
public static T4 name (T1 a, T2 b, T3 c)  { .....}
```

ここで、各パラメーターは次のように定義されます。

- *name* はメソッド名
- T1 から T4 は、SQL タイプ t1 から t4 に対応する Java タイプ。

- *a*、*b*、および *c* は入力引数用の任意の変数名。

SQL タイプと Java タイプの相互関連については、ストアード・プロシージャーおよび UDF のパラメーター引き渡し規則を参照してください。

SQL NULL 値は、初期化されていない Java 変数として表現されます。これらの変数がオブジェクト・タイプの場合は、Java ヌル値を持ちます。 SQL NULL が int などの Java スカラー・データ・タイプに渡されると、例外条件が発生します。

Java パラメーター・スタイルを使っているときに Java UDF から結果を戻すには、単純にメソッドから結果を戻します。

```
{ ....
  return value;
}
```

UDF およびストアード・プロシージャーで使用される C モジュールと同様に、Java UDF では Java 標準入出力ストリーム (System.in、System.out、および System.err) は使用できません。

パラメーター・スタイル DB2GENERAL

パラメーター・スタイル DB2GENERAL は Java UDF によって使用されます。このパラメーター・スタイルでは、戻り値は関数の最後のパラメーターとして渡され、com.ibm.db2.app.UDF クラスの *set* メソッドを使って設定される必要があります。

Java UDF をコーディングする際は、以下の規則に従う必要があります。

- Java UDF を含むクラスは、Java com.ibm.db2.app.UDF クラスの *extend*、またはサブクラスである必要がある。
- DB2GENERAL パラメーター・スタイルでは、Java メソッドは public void インスタンス・メソッドである必要がある。
- Java メソッドのパラメーターは、SQL 互換タイプである必要がある。
- Java メソッドは、isNull メソッドを使って SQL NULL 値をテストすることができる。
- DB2GENERAL パラメーター・スタイルでは、Java メソッドは set() メソッドを使って、明示的にパラメーターを設定、および戻す必要がある。

Java UDF を含むクラスは、Java クラス com.ibm.db2.app.UDF を拡張する必要があります。
DB2GENERAL パラメーター・スタイルを使った Java UDF は、Java クラスの void インスタンス・メソッドとなる必要があります。たとえば、INTEGER を戻し、タイプ CHAR(5)、BLOB(10K)、および DATE の引数を取る、sample!test3 という UDF の場合、DB2 では、UDF の Java インプリメンテーションで以下のシグニチャーが必要です。

```
import com.ibm.db2.app.*;
public class sample extends UDF {
   public void test3(String arg1, Blob arg2, String arg3, int result) { ... }
}
```

Java メソッドのパラメーターは、SQL タイプでなければなりません。たとえば、SQL タイプ t1、t2、および t3 の引数を取り、タイプ t4 を戻すものとして宣言された UDF は、次の Java シグニチャーを持った Java メソッドとして呼び出されます。

```
public void name (T1 a, T2 b, T3 c, T4 d)  { .....}
```

ここで、各パラメーターは次のように定義されます。

- *name* はメソッド名

- T1 から T4 は、SQL タイプ t1 から t4 に対応する Java タイプ。
- *a*、*b*、および *c* は入力引数用の任意の変数名。
- *d* は、算出対象の UDF 結果を表す任意の変数名。

SQL タイプと Java タイプの相互関連については、ストアード・プロシージャーおよび UDF のパラメーター引き渡し規則のセクションに記載されています。

SQL NULL 値は、初期化されていない Java 変数として表現されます。 Java 規則に従って、これらの変数は、プリミティブ・タイプの場合にはゼロの値を持ち、オブジェクト・タイプの場合には Java ヌルになります。 SQL NULL を通常のゼロと区別するために、入力引数について isNull メソッドを呼び出すことができます。

```
{ ....
  if (isNull(1)) { /* argument #1 was a SQL NULL */ }
  else           { /* not NULL */ }
}
```

前述の例では、引数の番号は 1 から始まります。 isNull() 関数は、それに続く他の関数と同様、com.ibm.db2.app.UDF クラスから継承します。 DB2GENERAL パラメーター・スタイルを使っているときに Java UDF から結果を戻すには、次のように、set() メソッドを UDF 内で使用します。

```
{ ....
  set(2, value);
}
```

ここで 2 は出力引数の指標で、*value* は互換タイプのリテラルまたは変数です。引数番号は、選択された出力の引数リストの指標です。このセクションの最初の例では、int 結果変数に指標 4 が含まれています。 UDF が戻る前に設定されない出力引数には NULL 値が入ります。

UDF およびストアード・プロシージャーで使用される C モジュールと同様に、Java UDF では Java 標準入出力ストリーム (System.in、System.out、および System.err) は使用できません。

通常、DB2 は UDF を、照会の入力または ResultSet の各行に対して 1 回ずつ、何回も呼び出します。UDF の CREATE FUNCTION ステートメントで SCRATCHPAD が指定されている場合、DB2 は、連続する UDF の呼び出しの間にいくらかの「継続性」が必要であると認識します。したがって、DB2GENERAL パラメーター・スタイルの機能の場合、Java クラスのインプリメントのインスタンスは各呼び出しのたびには作成されず、一般にはステートメントごとの UDF 参照ごとに 1 回作成されます。しかし UDF に NO SCRATCHPAD が指定されている場合、クラス・コンストラクターの呼び出しによって、UDF の呼び出しのたびに白紙のインスタンスが作成されます。

スクラッチパッドはすべての UDF の呼び出しの情報を保管するのに役立ちます。 Java UDF はインスタンス変数を使用するか、あるいは、スクラッチパッドを設定して、呼び出し間に継続性を持たせることができます。 Java UDF は com.ibm.db2.app.UDF で使用可能な getScratchPad および setScratchPad メソッドによってスクラッチパッドにアクセスします。 CREATE FUNCTION ステートメントに FINAL CALL オプションを指定している場合は、照会の終了で、オブジェクトの public void close() メソッドが呼び出されます (DB2GENERAL パラメーター・スタイルの関数の場合)。このメソッドを定義していないと、stub 関数がその後を引き継ぎ、イベントは無視されます。 com.ibm.db2.app.UDF クラスには、DB2GENERAL パラメーター・スタイルの UDF で使用できる有用な変数とメソッドが含まれています。これらの変数とメソッドについては、以下の表で説明します。

変数およびメソッド	説明
• public static final int SQLUDF_FIRST_CALL = -1; • public static final int SQLUDF_NORMAL_CALL = 0; • public static final int SQLUDF_TF_FIRST = -2; • public static final int SQLUDF_TF_OPEN = -1; • public static final int SQLUDF_TF_FETCH = 0; • public static final int SQLUDF_TF_CLOSE = 1; • public static final int SQLUDF_TF_FINAL = 2;	スカラー UDF の場合、これらは、呼び出しが最初の呼び出しか、通常の呼び出しかを判別するための定数です。テーブル UDF の場合は、呼び出しが、最初の呼び出し、オープン呼び出し、フェッチ呼び出し、クローズ呼び出し、あるいは最終呼び出しのいずれであるかを判別するための定数です。
public Connection getConnection();	このメソッドは、このストアード・プロシージャー呼び出しのための JDBC 接続ハンドルを取得し、呼び出し側アプリケーションのデータベースへの接続を表す JDBC オブジェクトを戻します。それは、C ストアード・プロシージャーにおけるヌルの SQLConnect() 呼び出しの結果に似ています。
public void close();	UDF が FINAL CALL オプションを指定して作成された場合、このメソッドは UDF 評価の終わりに、データベースにより呼び出されます。それは、C の UDF の場合の最終呼び出しに似ています。 Java UDF クラスがこのメソッドを実装していない場合には、このイベントは無視されます。
public boolean isNull(int i)	このメソッドは、与えられた指標の入力引数が SQL NULL かどうかをテストします。
• public void set(int i, short s); • public void set(int i, int j); • public void set(int i, long j); • public void set(int i, double d); • public void set(int i, float f); • public void set(int i, BigDecimal bigDecimal); • public void set(int i, String string); • public void set(int i, Blob blob); • public void set(int i, Clob clob); • public boolean needToSet(int i);	このメソッドは、出力引数を、与えられた値に設定します。次のような何らかの問題が生じると、例外が投じられます。 • UDF 呼び出しが進行していない。 • 指標が有効な出力引数を参照していない。 • データ・タイプが一致していない。 • データ長が一致していない。 • コード・ページ変換のエラーが発生した。
public void setSQLstate(String string);	このメソッドは UDF により呼び出され、この呼び出しから SQLSTATE が戻されるように設定します。 string が SQLSTATE として許容外の場合は、例外がスローされます。ユーザーは SQLSTATE を外部プログラムで設定し、関数からエラーまたは警告が戻されるようにすることができます。この場合、SQLSTATE には以下のいずれかが入ります。 • '00000'。これは成功を示します。 • '01Hxx'。ここで、xx は任意の 2 桁の数または英大文字で、警告を示します。 • '38yxx'。ここで、y は 'I' から 'Z' の英大文字、xx は任意の 2 桁の数または英大文字で、エラーを示します。

変数およびメソッド	説明
public void setSQLmessage(String string);	このメソッドは setSQLstate メソッドに類似しています。これは SQL メッセージ結果を設定します。 string が許容外 (たとえば 70 文字を超えている) の場合は、例外がスローされます。
public String getFunctionName();	このメソッドは、処理中の UDF の名前を戻します。
public String getSpecificName();	このメソッドは、処理中の UDF の特定名を戻します。
public byte[] getDBinfo();	このメソッドは、処理中の UDF についての未加工・未処理の DBINFO 構造を、バイト配列として戻します。 CREATE FUNCTION を使用し、DBINFO オプションを付けて、UDF を登録しておく必要があります。
• public String getDBname(); • public String getDBauthid(); • public String getDBver_rel(); • public String getDBplatform(); • public String getDBapplid(); • public String getDBapplid(); • public String getDBtbschema(); • public String getDBtbname(); • public String getDBcolname();	これらのメソッドは、処理中の UDF の DBINFO 構造から、該当するフィールドの値を戻します。 CREATE FUNCTION を使用し、DBINFO オプションを付けて、UDF を登録しておく必要があります。 getDBtbschema()、getDBtbname()、getDBcolname() の各メソッドは、UPDATE ステートメントの SET 文節の右辺にユーザー定義関数が指定されている場合のみ、意味のある情報を戻します。
public int getCCSID();	このメソッドは、ジョブの CCSID を戻します。
public byte[] getScratchpad();	このメソッドは、現在処理中の UDF のスクラッチパッドのコピーを戻します。まず、SCRATCHPAD オプションを付けて UDF を宣言する必要があります。
public void setScratchpad(byte ab[]);	このメソッドは、現在処理中の UDF のスクラッチパッドを、与えられたバイト配列の内容で上書きします。まず、SCRATCHPAD オプションを付けて UDF を宣言する必要があります。バイト配列は、getScratchpad() の戻りと同じサイズでなければなりません。
public int getCallType();	このメソッドは、現在行われている呼び出しのタイプを戻します。これらの値は、sqludf.h に定義されている C の値に対応しています。戻される可能性のある値は以下のとおりです。 • SQLUDF_FIRST_CALL • SQLUDF_NORMAL_CALL • SQLUDF_TF_FIRST • SQLUDF_TF_OPEN • SQLUDF_TF_FETCH • SQLUDF_TF_CLOSE • SQLUDF_TF_FINAL

Java ユーザー定義関数に関する制約事項:

これらの制限は、Java ユーザー定義関数 (UDF) に適用されます。

- Java UDF が追加のスレッドを作成しないようにする必要がある。追加のスレッドを作成できるのは、ジョブがマルチスレッド可能な場合だけです。 SQL ストアード・プロシージャーを呼び出すジョブがマルチスレッド可能かどうかは保証できないので、Java ストアード・プロシージャーは追加のスレッドを作成できません。
- データベースに定義する Java ストアード・プロシージャーの完全名は、279 文字に限定されている。この制限は EXTERNAL_NAME 列に基づきます。この列の最大幅は 279 文字です。
- 借用権限を使用して Java クラス・ファイルにアクセスできない。
- Java UDF は、システムにインストールされている最新バージョンの JDK を常に使用する。
- Blob クラスと Clob クラスは java.sql パッケージと com.ibm.db2.app パッケージの両方にあるので、同一のプログラム中でこれらの両方のクラスを使用する場合は、プログラマーはこれらのクラスの名前全体を使用しなければならない。 com.ibm.db2.app 中の Blob クラスと Clob クラスが、ストアード・プロシージャーに渡されるパラメーターとして使用されていることを、プログラムが確認しなければなりません。
- Java UDF を作成する際には、ソース関数の場合と同様に、ライブラリー中のサービス・プログラムを使用して関数の定義を格納する。サービス・プログラムの名前は、システムによって生成され、関数を作成したジョブのジョブ・ログ中にあります。このオブジェクトを保管してから別のシステムに復元すると、関数の定義も復元されます。 Java UDF をシステム間で移動する場合は、Java クラスが含まれる統合ファイル・システムに加えて、関数の定義が含まれるサービス・プログラムも移動する必要があります。
- データベースへの接続に使用する JDBC 接続のプロパティー (システムの命名など) を、Java UDF で設定できない。事前取り出しが使用不可の場合を除き、デフォルトの JDBC 接続プロパティーが常に使用されます。

Java ユーザー定義テーブル関数:

DB2 は、テーブルを戻す機能を関数に提供します。この機能は、データベースの外部からデータベースにテーブル形式で情報を公開するのに役立ちます。たとえば、Java ストアード・プロシージャーと Java UDF (テーブルとスカラーの両方) で使用される、Java 仮想マシン (JVM) 中のプロパティー設定を公開するテーブルを作成できます。

SQLJ Part 1: SQL Routines 規格はテーブル関数をサポートします。したがって、テーブル関数を使用できるのは、パラメーター・スタイル DB2GENERAL を使用する場合だけです。

テーブル関数に対して 5 種類の呼び出しが行われます。以下の表にこれらの呼び出しが説明されています。以下の内容は、関数作成 SQL ステートメント上でスクラッチパッドが指定されていることを前提にしています。

スキャンする時点	NO FINAL CALL LANGUAGE JAVA SCRATCHPAD	FINAL CALL LANGUAGE JAVA SCRATCHPAD
テーブル関数を初めて OPEN する前	呼び出しなし	クラス・コンストラクターが呼び出されます (新しいスクラッチパッドを意味します)。 FIRST 呼び出しで UDF メソッドが呼び出されます。
毎回のテーブル関数の OPEN 時	クラス・コンストラクターが呼び出されます (新しいスクラッチパッドを意味します)。 OPEN 呼び出しで UDF メソッドが呼び出されます。	OPEN 呼び出しで UDF メソッドが呼び出されます。

スキャンする時点	NO FINAL CALL LANGUAGE JAVA SCRATCHPAD	FINAL CALL LANGUAGE JAVA SCRATCHPAD
毎回のテーブル関数データの新規行の FETCH 時	FETCH 呼び出しで UDF メソッドが呼び出されます。	FETCH 呼び出しで UDF メソッドが呼び出されます。
毎回のテーブル関数の CLOSE 時	CLOSE 呼び出しで UDF メソッドが呼び出されます。 close() メソッドがある場合は呼び出されます。	CLOSE 呼び出しで UDF メソッドが呼び出されます。
テーブル関数を最後に CLOSE した後	呼び出しなし	FINAL 呼び出しで UDF メソッドが呼び出されます。 close() メソッドがある場合は呼び出されます。

例: Java テーブル関数

Java ユーザー定義テーブル関数の実行時に JVM 中のどのプロパティー設定を使用するかを判別する、Java テーブル関数の例を以下に示します。

注: 法律上の重要な情報に関しては、コードの特記事項情報をお読みください。

```
import com.ibm.db2.app.*;
import java.util.*;

public class JVMProperties extends UDF {
 Enumeration propertyNames;
 Properties properties ;

 public void dump (String property, String value) throws Exception
 {
   int callType = getCallType();
   switch(callType) {
     case SQLUDF_TF_FIRST:
       break;
     case SQLUDF_TF_OPEN:
       properties = System.getProperties();
       propertyNames = properties.propertyNames();
       break;
     case SQLUDF_TF_FETCH:
       if (propertyNames.hasMoreElements()) {
           property = (String) propertyNames.nextElement();
           value = properties.getProperty(property);
           set(1, property);
           set(2, value);
       } else {
          setSQLstate("02000");
       }
       break;
     case SQLUDF_TF_CLOSE:
       break;
     case SQLUDF_TF_FINAL:
       break;
     default:
       throw new Exception("UNEXPECT call type of "+callType);
   }
 }
}
```

テーブル関数をコンパイルして、そのクラス・ファイルを /QIBM/UserData/OS400/SQLLib/Function にコピーした後に、以下の SQL ステートメントを使用してその関数をデータベースに登録できます。

```
create function properties()
returns table (property varchar(500), value varchar(500))
external name 'JVMProperties.dump' language java
parameter style db2general fenced no sql
disallow parallel scratchpad
```

関数を登録した後で、SQL ステートメントの一部として使用できます。たとえば、以下の SELECT ステートメントは、テーブル関数によって生成されたテーブルを戻します。

```
SELECT * FROM TABLE(PROPERTIES())
```

JAR ファイルを操作する SQLJ プロシージャー

Java ストアード・プロシージャーと Java UDF は両方とも、Java JAR ファイルに保管されている Java クラスを使用できます。

JAR ファイルを使用するには、*jar-id* を JAR ファイルに関連付ける必要があります。 *jar-ids* および JAR ファイルの操作を可能にする、SQLJ スキーマによるストアード・プロシージャーが用意されています。それらのプロシージャーにより、JAR ファイルのインストール、置換、および除去が可能です。また、JAR ファイルに関連した SQL カタログの使用と更新を行う機能もあります。

SQLJ.INSTALL_JAR:

SQLJ.INSTALL_JAR ストアード・プロシージャーは、JAR ファイルをデータベース・システム中にインストールします。後続の CREATE FUNCTION および CREATE PROCEDURE ステートメント中で、この JAR ファイルを使用できます。

権限

SYSJAROBJECTS および SYSJARCONTENTS カタログ・テーブルに関する以下の特権のうち 1 つ以上が、CALL ステートメントの許可 ID で保持されていなければなりません。

- 以下のシステム権限:
  - テーブルに対する INSERT および SELECT 特権
  - ライブラリー QSYS2 に対するシステム権限 *EXECUTE
- 管理権限

以下の特権も、CALL ステートメントの許可 ID で保持されていなければなりません。

- *jar-url* パラメーターで指定されている、インストール対象の JAR ファイルに対する読み取り (*R) アクセス権。
- JAR ファイルのインストール・ディレクトリーに対する書き込み、実行、および読み取り (*RWX) アクセス権。このディレクトリーは /QIBM/UserData/OS400/SQLLib/Function/jar/*schema* (*schema* は *jar-id* のスキーマ) です。

これらの権限には、借用権限を使用できません。

SQL 構文

```
>>-CALL--SQLJ.INSTALL_JAR-- (--'jar-url'--,--'jar-id'--,--deploy--)-->

>--------------------------------------------------------------><
```

説明

jar-url インストールまたは置換対象の JAR ファイルを含む URL。サポートされている URL スキームは 'file:' だけです。

jar-id *jar-url* によって指定されたファイルに関連した、データベース中の JAR ID。 *jar-id* には SQL 命名構文が使用され、JAR ファイルは暗黙または明示の修飾子によって指定されたスキーマやライブラリーにインストールされます。

deploy 配置記述子ファイルの install_action の説明に使用される値。この整数がゼロ以外の値の場合は、install_jar プロシージャーの終了時に、配置記述子ファイルの install_actions が実行される必要があります。ゼロの値をサポートしているのは現行バージョンの DB2 for i だけです。

使用上の注意

JAR ファイルのインストール時に、DB2 for i は SYSJAROBJECTS システム・カタログ中にその JAR ファイルを登録します。さらに、JAR ファイルから Java クラス・ファイルの名前を抽出し、個々のクラスを SYSJARCONTENTS システム・カタログ中に登録します。 DB2 for i は、JAR ファイルを /QIBM/UserData/OS400/SQLLib/Function ディレクトリーの jar/schema サブディレクトリーにコピーします。 DB2 for i は、*jar-id* 文節で指定されている名前を新しい JAR ファイルのコピーに付けます。 DB2 for i によって /QIBM/UserData/OS400/SQLLib/Function/jar のサブディレクトリーにインストールされた JAR ファイルには、変更を加えることができません。その代わりに、CALL SQLJ.REMOVE_JAR コマンドと CALL SQLJ.REPLACE_JAR SQL コマンドを使用して、インストールされている JAR ファイルを除去するか置き換える必要があります。

例

以下のコマンドは、SQL 対話式セッションから発行されます。

```
CALL SQLJ.INSTALL_JAR('file:/home/db2inst/classes/Proc.jar' , 'myproc_jar', 0)
```

file:/home/db2inst/classes/ ディレクトリーにある Proc.jar ファイルは、DB2 for i 中に myproc_jar という名前でインストールされます。 Procedure.jar ファイルを使用するそれ以降の SQL コマンドは、名前 myproc_jar を使用して参照します。

SQLJ.REMOVE_JAR:

SQLJ.REMOVE_JAR ストアード・プロシージャーは、JAR ファイルをデータベース・システムから除去します。

権限

SYSJARCONTENTS および SYSJAROBJECTS カタログ・テーブルに関する以下の特権のうち 1 つ以上が、CALL ステートメントの許可 ID で保持されていなければなりません。

- 以下のシステム権限:
  - テーブルに対する SELECT および DELETE 特権
  - ライブラリー QSYS2 に対するシステム権限 *EXECUTE
- 管理権限

以下の特権も、CALL ステートメントの許可 ID で保持されていなければなりません。

- 除去される JAR ファイルに対する *OBJMGT 権限。この JAR ファイルの名前は /QIBM/UserData/OS400/SQLLib/Function/jar/schema/jarfile になります。

この権限には、借用権限を使用できません。

構文

```
>>-CALL--SQLJ.REMOVE_JAR--(--'jar-id'--,--undeploy--)----------><
```

説明

jar-id データベースから除去される JAR ファイルの JAR ID。

undeploy
配置記述子ファイルの remove_action の説明に使用される値。この整数がゼロ以外の値の場合は、install_jar プロシージャーの終了時に、配置記述子ファイルの remove_actions が実行される必要があります。ゼロの値をサポートしているのは現行バージョンの DB2 for i だけです。

例

以下のコマンドは、SQL 対話式セッションから発行されます。

```
CALL SQLJ.REMOVE_JAR('myProc_jar', 0)
```

JAR ファイル myProc_jar がデータベースから除去されます。

SQLJ.REPLACE_JAR:

SQLJ.REPLACE_JAR ストアード・プロシージャーは、データベース・システム中の JAR ファイルを置換します。

権限

SYSJAROBJECTS および SYSJARCONTENTS カタログ・テーブルに関する以下の特権のうち 1 つ以上が、CALL ステートメントの許可 ID で保持されていなければなりません。

- 以下のシステム権限:
  - テーブルに対する SELECT、INSERT、および DELETE 特権
  - ライブラリー QSYS2 に対するシステム権限 *EXECUTE
- 管理権限

以下の特権も、CALL ステートメントの許可 ID で保持されていなければなりません。

- *jar-url* パラメーターで指定されている、インストール対象の JAR ファイルに対する読み取り (*R) アクセス権。
- 除去される JAR ファイルに対する *OBJMGT 権限。この JAR ファイルの名前は /QIBM/UserData/OS400/SQLLib/Function/jar/schema/jarfile になります。

これらの権限には、借用権限を使用できません。

構文

```
>>-CALL--SQLJ.REPLACE_JAR--(--'jar-url'--,--'jar-id'--)--------><
```

説明

jar-url 置換対象の JAR ファイルを含む URL。サポートされている URL スキームは 'file:' だけです。

jar-id *jar-url* によって指定されたファイルに関連した、データベース中の JAR ID。 *jar-id* には SQL 命名構文が使用され、JAR ファイルは暗黙または明示の修飾子によって指定されたスキーマやライブラリーにインストールされます。

使用上の注意

SQLJ.REPLACE_JAR ストアード・プロシージャーは、以前に SQLJ.INSTALL_JAR を使用してデータベースにインストールされた JAR ファイルを置換します。

例

以下のコマンドは、SQL 対話式セッションから発行されます。

```
CALL SQLJ.REPLACE_JAR('file:/home/db2inst/classes/Proc.jar' , 'myproc_jar')
```

jar-id myproc_jar によって参照される現行の JAR ファイルが、file:/home/db2inst/classes/ ディレクトリー中の Proc.jar ファイルに置換されます。

SQLJ.UPDATEJARINFO:

SQLJ.UPDATEJARINFO は、SYSJARCONTENTS カタログ・テーブルの CLASS_SOURCE 列を更新します。このプロシージャーは、SQLJ 規格の一部ではありませんが、DB2 for i ストアード・プロシージャー・ビルダーによって使用されます。

権限

SYSJARCONTENTS カタログ・テーブルに関する以下の特権のうち 1 つ以上が、CALL ステートメントの許可 ID で保持されていなければなりません。

- 以下のシステム権限:
  - テーブルに対する SELECT および UPDATEINSERT 特権
  - ライブラリー QSYS2 に対するシステム権限 *EXECUTE
- 管理権限

CALL ステートメントを実行するユーザーに、以下の権限もなければなりません。

- *jar-url* パラメーターで指定されている JAR ファイルに対する読み取り (*R) アクセス権。インストール対象の JAR ファイルに対する読み取り (*R) アクセス権。
- JAR ファイルのインストール・ディレクトリーに対する書き込み、実行、および読み取り (*RWX) アクセス権。このディレクトリーは /QIBM/UserData/OS400/SQLLib/Function/jar/*schema* (*schema* は *jar-id* のスキーマ) です。

これらの権限には、借用権限を使用できません。

構文

```
>>-CALL--SQLJ.UPDATEJARINFO--(--'jar-id'--,--'class-id'--,--'jar-url'--)-->

>--------------------------------------------------------------><
```

説明

jar-id
データベース中の、更新される JAR ID。

class-id
更新されるクラスのパッケージ修飾クラス名。

jar-url
JAR ファイルの更新に使用するクラス・ファイルを含む URL。サポートされている URL スキームは 'file:' だけです。

例

以下のコマンドは、SQL 対話式セッションから発行されます。

```
CALL SQLJ.UPDATEJARINFO('myproc_jar', 'mypackage.myclass',
                        'file:/home/user/mypackage/myclass.class')
```

jar-id myproc_jar に関連した JAR ファイルが、新しいバージョンの mypackage.myclass クラスによって更新されます。新しいバージョンのクラスは、file:/home/user/mypackage/myclass.class から入手されます。

SQLJ.RECOVERJAR:

SQLJ.RECOVERJAR プロシージャーは、SYSJAROBJECTS カタログに格納されている JAR ファイルを取り出し、/QIBM/UserData/OS400/SQLLib/Function/jar/*jarschema*/*jar_id*.jar ファイルに復元します。

権限

SYSJAROBJECTS カタログ・テーブルに関する以下の特権のうち 1 つ以上が、CALL ステートメントの許可 ID で保持されていなければなりません。

- 以下のシステム権限:
  - テーブルに対する SELECT および UPDATEINSERT 特権
  - ライブラリー QSYS2 に対するシステム権限 *EXECUTE
- 管理権限

CALL ステートメントを実行するユーザーに、以下の権限もなければなりません。

- JAR ファイルのインストール・ディレクトリーに対する書き込み、実行、および読み取り (*RWX) アクセス権。このディレクトリーは /QIBM/UserData/OS400/SQLLib/Function/jar/*schema* (*schema* は *jar-id* のスキーマ) です。
- 除去される JAR ファイルに対する *OBJMGT 権限。この JAR ファイルの名前は /QIBM/UserData/OS400/SQLLib/Function/jar/schema/jarfile になります。

構文

```
>>-CALL--SQLJ.RECOVERJAR--(--'jar-id'--)-----------------------><
```

説明

jar-id データベース中の、リカバリーされる JAR ID。

例

以下のコマンドは、SQL 対話式セッションから発行されます。

```
CALL SQLJ.UPDATEJARINFO('myproc_jar')
```

myproc_jar と関連した JAR ファイルは、SYSJARCONTENT テーブルの内容で更新されます。このファイルは、/QIBM/UserData/OS400/SQLLib/Function/jar/jar_schema myproc_jar.jar にコピーされます。

SQLJ.REFRESH_CLASSES:

SQLJ.REFRESH_CLASSES ストアード・プロシージャーを使用すると、現行のデータベース接続で Java ストアード・プロシージャーまたは Java UDF が使用しているユーザー定義クラスが再ロードされます。SQLJ.REPLACE_JAR ストアード・プロシージャーの呼び出しによって加えられた変更を取得するためには、既存のデータベース接続によってこのストアード・プロシージャーが呼び出されなければなりません。

権限

なし

構文

```
>>-CALL--SQLJ.REFRESH_CLASSES-- ()-->
>--------------------------------------------------------------><
```

例

MYJAR jarid で登録された JAR ファイル内のクラスを使用する、Java ストアード・プロシージャー、MYPROCEDURE を呼び出します。

```
CALL MYPROCEDURE()
```

以下の呼び出しを使用して JAR ファイルを置き換えます。

```
CALL SQLJ.REPLACE_JAR('MYJAR', '/tmp/newjarfile.jar')
```

後の MYPROCEDURE ストアード・プロシージャーの呼び出しで、更新された JAR ファイルを使用するためには、SQLJ.REFRESH_CLASSES が呼び出されなければなりません。

```
CALL SQLJ.REFRESH_CLASSES()
```

ストアード・プロシージャーを再び呼び出します。プロシージャーが呼び出されると、更新されたクラス・ファイルが使用されます。

```
CALL MYPROCEDURE()
```

Java ストアード・プロシージャーおよび UDF 用のパラメーター引き渡し規則

以下の表には、Java ストアード・プロシージャーと UDF 中で SQL データ・タイプが表される方法がリストされています。

SQL データ・タイプ	Java パラメーター・スタイル JAVA	Java パラメーター・スタイル DB2GENERAL
SMALLINT	short	short
INTEGER	int	int
BIGINT	long	long
DECIMAL(p,s)	BigDecimal	BigDecimal
NUMERIC(p,s)	BigDecimal	BigDecimal
REAL または FLOAT(p)	float	float
DOUBLE PRECISION、FLOAT、または FLOAT(p)	double	double
CHARACTER(n)	String	String
CHARACTER(n) FOR BIT DATA	byte[]	com.ibm.db2.app.Blob
VARCHAR(n)	String	String
VARCHAR(n) FOR BIT DATA	byte[]	com.ibm.db2.app.Blob
GRAPHIC(n)	String	String
VARGRAPHIC(n)	String	String
DATE	Date	String
TIME	Time	String
TIMESTAMP	Timestamp	String
標識変数	-	-

SQL データ・タイプ	Java パラメーター・スタイル JAVA	Java パラメーター・スタイル DB2GENERAL
CLOB	java.sql.Clob	com.ibm.db2.app.Clob
BLOB	java.sql.Blob	com.ibm.db2.app.Blob
DBCLOB	java.sql.Clob	com.ibm.db2.app.Clob
DataLink	-	-
ARRAY	java.sql.Array	-

Java と他のプログラム言語

Java には、Java 以外の言語で作成されたコードを呼び出す方法がいくつかあります。

IBM i Java 環境は、Integrated Language Environment (ILE) とは異なります。 Java は ILE 言語ではないので、プログラムやサービス・プログラムを作成するために ILE オブジェクト・モジュールとバインドすることはできません。以下の表は、ILE ベースのプログラムと Java プログラムとの違いの一部を示しています。

ILE	Java
IBM i サーバー上のライブラリーまたはファイル構造のメンバーにソース・コードを格納する。	統合ファイル・システム内のストリーム・ファイルにソース・コードを格納する。
EBCDIC (拡張 2 進化 10 進コード) 形式のソース・ファイルを原始ステートメント入力ユーティリティー (SEU) で編集する。	通常は、ASCII 形式のソース・ファイルをワークステーションのエディターで編集する。
ソース・ファイルをコンパイルしてオブジェクト・コード・モジュールを生成し、IBM i サーバー上のライブラリーに格納する。	ソース・コードをコンパイルしてクラス・ファイルを生成し、統合ファイル・システムに格納する。
オブジェクト・モジュールは、プログラムまたはサービス・プログラム内で静的にバインドされる。	クラスは、実行時に必要に応じて動的にロードされる。
他の ILE プログラミング言語で記述された関数を直接呼び出すことができる。	Java から他の言語を呼び出すには、Java ネイティブ・インターフェースを使用しなければならない。
ILE 言語は、常に機械命令としてコンパイルされ、実行される。	Java プログラムは、インタープリター形式で実行することも、コンパイラー形式で実行することもできる。

注: 移植性が心配される場合は、「ピュア」Java ではないソリューションの使用を避けてください。

関連概念

220 ページの『Java 呼び出し API』
Java ネイティブ・インターフェース (JNI) の一部である呼び出し API を使用すると、Java 以外のコードで Java 仮想マシン (JVM) を作成し、Java クラスをロードおよび使用することができます。 この機能により、マルチスレッド化されたプログラムは、1 つの Java 仮想マシンで実行されている Java クラスを複数のスレッドで使用できるようになります。

228 ページの『プロセス間通信にソケットを使用する』
ソケット・ストリームは、異なるプロセス内で実行しているプログラム間での通信を行います。

232 ページの『プロセス間通信に入出力ストリームを使用する』
入出力ストリームは、別々のプロセスで実行されているプログラムの間で通信を行います。

関連資料

233 ページの『例: ILE C から Java を呼び出す』
次に示すのは、`system()` 関数を使用して Java Hello プログラムを呼び出す Integrated Language Environment (ILE) C プログラムの例です。

233 ページの『例: RPG から Java を呼び出す』
次に示すのは、QCMDEXC API を使用して Java Hello プログラムを呼び出す RPG プログラムの例です。

関連情報

IBM Toolbox for Java

ネイティブ・メソッドおよび Java ネイティブ・インターフェース

ネイティブ・メソッドとは、Java 以外の言語で開始される Java メソッドです。ネイティブ・メソッドは、Java では直接使用できないシステム固有の機能や API にアクセスできます。

ネイティブ・メソッドにはシステム固有のコードがあるので、ネイティブ・メソッドを使用するとアプリケーションのポータビリティーが制限されます。ネイティブ・メソッドは、新しいネイティブ・コード・ステートメントまたは既存のネイティブ・コードを呼び出すネイティブ・コード・ステートメントのいずれかです。

ネイティブ・メソッドが必要な場合、ネイティブ・メソッドとそれを実行する Java 仮想マシンとの間の相互操作性が必要になります。 Java ネイティブ・インターフェース (JNI) を使用すると、どのプラットフォームにも依存せずに、この相互運用性を容易に実現できます。

JNI は、一連のインターフェースであり、JNI を使うとネイティブ・メソッドと Java 仮想マシンとのさまざまな相互操作性を実現できます。たとえば、JNI には、新しいオブジェクトを作成してメソッドを呼び出すインターフェース、フィールドの取得と設定を行うインターフェース、例外を処理するインターフェース、ストリングおよび配列の操作を行うインターフェースなどが含まれています。

JNI の完全な説明は、Sun Microsystems, Inc. が提供している「Java Native Interface」を参照してください。

関連情報

Sun Microsystems, Inc. による「Java Native Interface」

Java ネイティブ・メソッド入門

ネイティブ・メソッドは、Pure Java ではプログラミングの要件を満たすことができない場合にのみ使用してください。

ネイティブ・メソッドの使用を制限し、それらを次の状況でのみ使用するようにします。

- Pure Java では使用できないシステム機能にアクセスする場合。
- パフォーマンスへの依存度が高く、ネイティブ実装によるメリットが大きいメソッドを実装する場合。
- Java が別の API を呼び出すことを可能にする既存のアプリケーション・プログラミング・インターフェース (API) とインターフェースする場合。

以下の指示は、C 言語での Java Native Interface (JNI) の使用に当てはまります。 RPG 言語での JNI の使用について詳しくは、「WebSphere Development Studio: ILE RPG プログラマーの手引き (SD88-5042-04)」の第 11 章を参照してください。

注: ネイティブ・ライブラリーまたはネイティブ・メソッド・ライブラリーという用語は、ILE ネイティブ・メソッドのコンテキストで使用される場合は、Integrated Language Environment (ILE) サービス・プログラムを指し、PASE for i ネイティブ・メソッドのコンテキストで使用される場合は、AIX 静的または共有ライブラリーを指します。

Java ネイティブ・メソッドを作成するには、以下のステップに従ってください。

1. Java クラスを作成し、標準の Java 言語構文を使用して、どのメソッドがネイティブ・メソッドであるかを指定します。

   クラスの静的イニシャライザーで、ネイティブ・メソッドの C インプリメンテーションを含むネイティブ・ライブラリーをロードするコードを追加する必要があります。ネイティブ・ライブラリーをロードするためには、System.load() または System.loadLibrary() Java メソッドを使用できます。 System.load() メソッドでは、パラメーターとしてネイティブ・ライブラリーへの絶対パスを指定し、指定されたネイティブ・ライブラリーをロードします。 System.loadLibrary() では、パラメーターとしてライブラリー名を指定し、その名前に一致するネイティブ・ライブラリーを検出して、ネイティブ・ライブラリーをロードします。 System.loadLibrary() メソッドによってネイティブ・ライブラリーの場所を探索する方法について詳しくは、 216 ページの『ネイティブ・メソッド・ライブラリーの管理』を参照してください。

   以下のライブラリー命名規則に留意する必要があります。

   - ネイティブ・メソッドが ILE ネイティブ・メソッドであり、Java コードが Sample という名前のライブラリーをロードする場合、対応する実行可能ファイルは SAMPLE という名前の ILE サービス・プログラムである必要があります。以下は、ILE ネイティブ・ライブラリーをロードする方法を示しています。

     ```
     System.loadLibrary("Sample");

     System.load("/qsys.lib/mylib.lib/Sample.srvpgm");
     ```

     注: サービス・プログラムへのシンボリック・リンクを、これらのライブラリー・ロード・メソッドで使用できます。

   - ネイティブ・メソッドが PASE for i ネイティブ・メソッドであり、Java コードが Sample という名前のライブラリーをロードする場合、対応する実行可能ファイルは libSample.a または libSample.so という名前の AIX ライブラリーである必要があります。以下は、PASE for i ネイティブ・ライブラリーをロードする方法を示しています。

     ```
     System.loadLibrary("Sample");

     System.load("/somedir/libSample.so");
     ```

2. javac ツールを使用して Java ソースをコンパイルすることにより、クラス・ファイルを作成します。

3. javah ツールを使用して、ヘッダー・ファイル (.h) を作成します。このヘッダー・ファイルには、ネイティブ・メソッドの実装を作成するための正確なプロトタイプが含まれます。ヘッダー・ファイルを作成するディレクトリーは、-d オプションで指定します。
4. ネイティブ・メソッドの C インプリメンテーション・コードを作成します。ネイティブ・メソッドのために使用される言語および関数の詳細については、 218 ページの『Java ネイティブ・メソッドおよびスレッドに関する考慮事項』 トピックを参照してください。
   a. 前のステップで作成したヘッダー・ファイルを組み込みます。
   b. ヘッダー・ファイル内のプロトタイプと正確に一致させます。
   c. ネイティブ・メソッドが Java 仮想マシンと対話しなければならない場合は、JNI によって提供される関数を使用します。
   d. ILE ネイティブ・メソッドの場合にのみ、ストリングを Java 仮想マシンに渡す場合は、ASCII コードに変換します。詳しくは、 213 ページの『ILE ネイティブ・メソッドのストリング』を参照してください。
5. ネイティブ・メソッドの C インプリメンテーション・コードをネイティブ・ライブラリーにコンパイルします。
   - ILE ネイティブ・メソッドの場合、「C モジュール作成」(CRTCMOD) コマンドを使用して、ソース・ファイルをモジュール・オブジェクトにコンパイルします。次に「サービス・プログラム作成」(CRTSRVPGM) コマンドを使用して、1 つ以上のモジュール・オブジェクトをサービス・プログラムにバインドします。このサービス・プログラムの名前は、`System.load()` または `System.loadLibrary()` Java メソッド呼び出しの Java コードで指定した名前と同じでなければなりません。

     注: ネイティブ・メソッドのインプリメンテーション・コードは、Teraspace ストレージを使用可能にしてコンパイルする必要があります。 Teraspace およびネイティブ・メソッドについて詳しくは、 212 ページの『Java 用 Teraspace ストレージ・モデル・ネイティブ・メソッド』を参照してください。
   - PASE for i ネイティブ・メソッドの場合は、`xlc` または `xlc_r` コマンドを使用して、AIX ライブラリーをコンパイルおよびビルドします。 PASE for i のライブラリーのコンパイルおよびビルドについて詳しくは、『AIX ソースのコンパイル』のトピックを参照してください。
6. Java コードで `System.loadLibrary()` 呼び出しを使用してネイティブ・ライブラリーをロードした場合は、以下のいずれかのタスクを実行します。
   - 必要なネイティブ・ライブラリー・パスのリストを `LIBPATH` 環境変数に組み込みます。 `LIBPATH` 環境変数は、QShell で、および IBM i コマンド行から変更できます。
     – Qshell コマンド・プロンプトから、次のように入力します。

     ```
     export LIBPATH=/QSYS.LIB/MYLIB.LIB
     java  myclass
     ```
     – あるいは、コマンド行から、次のように入力します。

     ```
     ADDENVVAR LIBPATH '/QSYS.LIB/MYLIB.LIB'
     JAVA  myclass
     ```
   - あるいは、**java.library.path** プロパティーでリストを提供します。 `java.library.path` プロパティーは、QShell で、および IBM i コマンド行から変更することができます。
     – Qshell コマンド・プロンプトから、次のように入力します。

     ```
     java -Djava.library.path=/QSYS.LIB/MYLIB.LIB  myclass
     ```

– あるいは、IBM i コマンド行から、次のように入力します。

```
JAVA PROP((java.library.path '/QSYS.LIB/MYLIB.LIB')) myclass
```

ここで、*/QSYS.LIB/MYLIB.LIB* は、System.loadLibrary() 呼び出しを使用してロードするネイティブ・ライブラリーを含むパスで、*myclass* は Java アプリケーションの名前です。

System.loadLibrary() メソッドによってネイティブ・ライブラリーの場所を探索する方法について詳しくは、 216 ページの『ネイティブ・メソッド・ライブラリーの管理』を参照してください。

ILE ネイティブ・メソッドの例については、 214 ページの『例: Java 用の ILE ネイティブ・メソッド』を参照してください。 PASE for i ネイティブ・メソッドの例については、 216 ページの『例: Java 用の IBM PASE for i ネイティブ・メソッド』を参照してください。

「Websphere Development Studio: ILE RPG プログラマーの手引き (SD88-5042-04)」。

218 ページの『Java ネイティブ・メソッドおよびスレッドに関する考慮事項』
ネイティブ・メソッドは、Java では使用できない関数を利用する場合に使用できます。 ネイティブ・メソッド付きの Java を使いこなすには、以下のような概念を理解する必要があります。

Sun Microsystems, Inc. による「Java Native Interface」

214 ページの『例: Java 用の ILE ネイティブ・メソッド』
この Java 用の Integrated Language Environment (ILE) ネイティブ・メソッドの例では、ネイティブ C メソッドのインスタンスを呼び出します。 次いでそれは Java ネイティブ・インターフェース (JNI) を使用して Java コードにコールバックを行い、Java ストリング変数の値を設定します。 その後、Java ストリング変数は、Java コードによって標準出力に書き込まれます。

213 ページの『ILE ネイティブ・メソッドのストリング』
ほとんどの Java ネイティブ・インターフェース (JNI) 関数は、パラメーターとして C 言語形式のストリングを受け入れます。 たとえば、JNI 関数の FindClass() は、クラス・ファイルの完全修飾名を指定するストリング・パラメーターを受け入れます。 クラス・ファイルが検出されると、このクラス・ファイルは FindClass() によってロードされ、その参照が FindClass()の呼び出し元に戻ります。

22 ページの『Java 文字のエンコード』
Java プログラムは、他のフォーマットのデータを変換して、アプリケーションが多様な国際文字セットの情報を転送および使用できるようにします。

Java 用の ILE ネイティブ・メソッド

IBM i Java 仮想マシン (JVM) は、Integrated Language Environment (ILE) で実行するネイティブ・メソッドの使用をサポートしています。

ILE ネイティブ・メソッドのサポートには以下が含まれます。

- ILE ネイティブ・メソッドからのネイティブ IBM i Java Native Interface (JNI) の全面使用
- ネイティブ IBM i JVM から ILE ネイティブ・メソッドを呼び出す機能

ILE ネイティブ・メソッドを使用している場合、以下の内容を考慮する必要があります。

- JNI 関数を使用する ILE プログラムまたはサービス・プログラムは、Teraspace ストレージを使用可能にしてコンパイルする必要があります。これが必要なのは、Teraspace ストレージのトップにマップされ、Teraspace ストレージのポインターが戻される Java オブジェクトが PASE for i ストレージの中にあるためです。また、GetxxxArrayRegion などの JNI 関数には、データが置かれているバッファーへのパラメーターがあります。PASE for i の JNI 関数がデータを Teraspace ストレージにコピーできるよう

にするためには、このポインターは Teraspace ストレージを指していなければなりません。 Teraspace ストレージを使用可能にしてプログラムをコンパイルしていない場合は、エスケープ・メッセージ MCH4443 (ターゲット・プログラム LOADLIB で無効なストレージ・モデル) が戻されます。

- JNI 関数のストリング・パラメーターは、UTF-8 でエンコードする必要があります。ストリングおよび JNI 関数について詳しくは、213 ページの『ILE ネイティブ・メソッドのストリング』 を参照してください。

Java 用 Teraspace ストレージ・モデル・ネイティブ・メソッド:

IBM i Java 仮想マシン (JVM) は、Teraspace ストレージ・モデル・ネイティブ・メソッドの使用をサポートしています。 Teraspace ストレージ・モデルは、ILE プログラム用の大規模処理のローカル・アドレス環境を提供します。 Teraspace ストレージ・モデルを使用すれば、ソース・コードをほとんどあるいは全く変更しないで、ネイティブ・メソッド・コードを他のオペレーティング・システムから IBM i に移植することができます。

注: Teraspace ストレージ・モデルは、静的ストレージ、ローカル変数、およびヒープ割り振りが Teraspace ストレージに自動的に配置される環境を提供します。Teraspace ストレージへのアクセスを使用可能にするだけでよい場合は、Teraspace ストレージ・モデルを使用する必要は**ありません**。Teraspace でネイティブ・メソッド・コードを使用可能にするには、それで十分です。Teraspace でネイティブ・メソッドを使用可能にする場合は、「C モジュール作成 (CRTCMOD)」コマンド、「C++ モジュール作成 (CRTCPPMOD)」コマンド、または他のモジュール作成コマンドで TERASPACE(*YES) パラメーターを使用します。

Teraspace ストレージ・モデルを使用したプログラミングについて詳しくは、以下の情報を参照してください。

- 「ILE 概念」の第 4 章
- 「WebSphere Development Studio ILE C/C++ Programmer's Guide」の第 17 章

Teraspace ストレージ・モデル用に作成された Java ネイティブ・メソッドの概念は、単一レベルのストレージを使用するネイティブ・メソッドの概念によく似ています。 JVM は Teraspace ストレージ・モデル・ネイティブ・メソッドに、このメソッドが JNI 関数を呼び出すために使用できる Java Native Interface (JNI) 環境へのポインターを渡します。

Teraspace ストレージ・モデル・ネイティブ・メソッド用に、JVM は、Teraspace ストレージ・モデルと 8 バイトのポインターを使用する JNI 関数インプリメンテーションを用意しています。

Teraspace ストレージ・モデル・ネイティブ・メソッドの作成

Teraspace ストレージ・モデル・ネイティブ・メソッドを正常に作成するには、Teraspace ストレージ・モデル・モジュール作成コマンドで、以下のオプションを使用する必要があります。

```
TERASPACE(*YES) STGMDL(*TERASPACE) DTAMDL(*LLP64)
```

Teraspace ストレージ機能を使用するための以下のオプション (`*TSIFC`) は、オプショナルです。

```
TERASPACE(*YES *TSIFC)
```

注: Teraspace ストレージ・モデル Java ネイティブ・メソッドを使用するときに `DTAMDL(*LLP64)` を使用しない場合、ネイティブ・メソッドを呼び出すと、実行時例外がスローされます。

ネイティブ・メソッドを実装する Teraspace ストレージ・モデル・サービス・プログラムの作成

Teraspace ストレージ・モデル・サービス・プログラムを作成するためには、「サービス・プログラムの作成 (CRTSRVPGM)」制御言語 (CL) コマンドで以下のオプションを使用します。

```
CRTSRVPGM STGMDL(*TERASPACE)
```

さらに、ACTGRP(*CALLER) オプションを使用する必要があります。このオプションを使用すると、JVM がすべての Teraspace ストレージ・モデル・ネイティブ・メソッド・サービス・プログラムを同一の Teraspace 活動化グループ内に活動化できるようになります。ネイティブ・メソッドが効率的に例外を処理するためには、このようにして Teraspace 活動化グループを作成することが重要になる場合があります。

プログラムの活動化と活動化グループについて詳しくは、「ILE 概念」の第 3 章を参照してください。

Teraspace ストレージ・モデル・ネイティブ・メソッドとの Java 呼び出し API の使用

JNI 環境ポインターがサービス・プログラムのストレージ・モデルと合わない場合は、呼び出し API GetEnv 関数を使用してください。呼び出し API GetEnv 関数は、常に正しい JNI 環境ポインターを戻します。

JVM は単一レベルと Teraspace ストレージ・モデルのネイティブ・メソッドをサポートしていますが、これらの 2 つのストレージ・モデルは異なる JNI 環境を使用します。 2 つのストレージ・モデルは異なる JNI 環境を使用するので、2 つのストレージ・モデルのネイティブ・メソッド間で JNI 環境ポインターをパラメーターとして渡すことはしないでください。

関連概念

220 ページの『Java 呼び出し API』
Java ネイティブ・インターフェース (JNI) の一部である呼び出し API を使用すると、Java 以外のコードで Java 仮想マシン (JVM) を作成し、Java クラスをロードおよび使用することができます。 この機能により、マルチスレッド化されたプログラムは、1 つの Java 仮想マシンで実行されている Java クラスを複数のスレッドで使用できるようになります。

関連情報

Sun Microsystems, Inc. による「Java Native Interface」

「C モジュール作成 (CRTCMOD)」CL コマンド

「C++ モジュール作成 (CRTCPPMOD)」CL コマンド

ILE 概念

Websphere Development Studio ILE C/C++ Programmer's Guide

ILE ネイティブ・メソッドのストリング:

ほとんどの Java ネイティブ・インターフェース (JNI) 関数は、パラメーターとして C 言語形式のストリングを受け入れます。たとえば、JNI 関数の FindClass() は、クラス・ファイルの完全修飾名を指定するストリング・パラメーターを受け入れます。クラス・ファイルが検出されると、このクラス・ファイルは FindClass() によってロードされ、その参照が FindClass()の呼び出し元に戻ります。

JNI 関数のストリング・パラメーターは、UTF-8 でエンコードする必要があります。 UTF-8 の詳細については、JNI 仕様に記載されていますが、通常は、7 ビット情報交換用米国標準コード (ASCII) 文字が

UTF-8 表示と等価であることを確認するだけで十分です。 7 ビット ASCII 文字は実際には 8 ビット文字ですが、先頭ビットは常に 0 になっています。ほとんどの C ストリングはすでに UTF-8 形式になっています。

サーバー上の統合言語処理環境 (ILE) C コンパイラーはデフォルトで拡張 2 進化 10 進コード (EBCDIC) で作動するので、JNI 関数に渡されるストリングは UTF-8 に変換する必要があります。これには、リテラル・ストリングと動的ストリングという 2 つの方法があります。 『リテラル・ストリング』 とは、ソース・コードのコンパイル時に値が分かっているストリングです。 動的ストリングとは、コンパイル時には値が不明ですが、実行時に実際の計算が行われるストリングです。

リテラル・ストリング

大部分のストリングは ASCII で表現することができ、そのようなストリングは、コンパイラーの現行のコード・ページを変更する pragma ステートメントで囲むことができます。そうすると、コンパイラーは、JNI によって必要とされる UTF-8 形式でストリングを内部的に格納します。ストリングが ASCII で表現できない場合には、元の拡張 2 進化 10 進コード (EBCDIC) ストリングを動的ストリングとして扱い、それを JNI に渡す前に iconv() によって処理すると簡単です。

たとえば、java/lang/String という名前のクラスを検索する場合、コードは次のようになります。

```
#pragma convert(819)
myClass = (*env)->FindClass(env,"java/lang/String");
#pragma convert(0)
```

番号 819 が指定された最初のプラグマは、コンパイラーに、後続のすべての二重引用符付きストリング (リテラル・ストリング) を ASCII で格納するよう指示します。番号 0 が指定された 2 番目のプラグマは、コンパイラーに、二重引用符付きストリングについてのコンパイラーのデフォルト・コード・ページ (通常、EBCDIC コード・ページ 37) に戻るよう指示します。このように、呼び出しをプラグマで囲むことによって、ストリング・パラメーターが UTF-8 でエンコードされるという JNI の要件を満たします。

注意: テキストの置換は慎重に行ってください。たとえば、コードが次のようになっている場合、

```
#pragma convert(819)
#define MyString "java/lang/String"
#pragma convert(0)
myClass = (*env)->FindClass(env,MyString);
```

結果のストリングは EBCDIC になります。これは、コンパイル時に MyString の値が FindClass() 呼び出しの中で置換されるためです。この置換の時点で、番号 819 のプラグマは有効ではありません。したがって、リテラル・ストリングは ASCII で格納されません。

動的ストリングを EBCDIC、Unicode、および UTF-8 に変換する

実行時に計算されるストリング変数を処理するために、ストリングを EBCDIC、Unicode および UTF-8 へ変換したり、この逆の変換を行う必要がある場合があります。変換は、iconv() API を使用して実行できます。 ネイティブ・メソッドのために Java ネイティブ・インターフェースを使用する例 3は、iconv() 変換記述子を作成、使用、および削除します。この方式は、iconv_t 記述子をマルチスレッド式に使用することで問題を回避しますが、パフォーマンス依存コードの場合は、静的ストレージ内に変換記述子を作成し、相互除外 (mutex) やその他の同期機能を使用することにより、複数のアクセスを制限してください。

例: Java 用の ILE ネイティブ・メソッド:

この Java 用の Integrated Language Environment (ILE) ネイティブ・メソッドの例では、ネイティブ C メソッドのインスタンスを呼び出します。次いでそれは Java ネイティブ・インターフェース (JNI) を使用し

て Java コードにコールバックを行い、Java ストリング変数の値を設定します。その後、Java ストリング変数は、Java コードによって標準出力に書き込まれます。

このソース・ファイル例の HTML 版を表示するには、以下のリンクを使用してください。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

- 537 ページの『例: NativeHello.java』
- 538 ページの『例: NativeHello.c』

ILE ネイティブ・メソッドの例を実行するためには、まず以下のトピックのタスクを完了する必要があります。

1. 541 ページの『例: ILE ネイティブ・メソッド・ソース・コードを準備する』
2. 542 ページの『例: ILE ネイティブ・メソッド・プログラム・オブジェクトを作成する』

Java 用の ILE ネイティブ・メソッドの例を実行する

上記のタスクを完了したら、この例を実行することができます。このプログラム例を実行するには、以下のコマンドのいずれかを使用します。

- IBM i コマンド・プロンプトから:

```
JAVA CLASS(NativeHello) CLASSPATH('/ileexample')
```

- Qshell コマンド・プロンプトから:

```
cd /ileexample
java NativeHello
```

Java 用の PASE for i ネイティブ・メソッド

IBM i Java 仮想マシン (JVM) は、PASE for i 環境で実行するネイティブ・メソッドの使用をサポートしています。

PASE for i ネイティブ・メソッドのサポートには以下が含まれます。

- PASE for i ネイティブ・メソッドからのネイティブ IBM i Java Native Interface (JNI) の全面使用
- ネイティブ PASE for i JVM から IBM i ネイティブ・メソッドを呼び出す機能

このサポートにより、AIX で実行する Java アプリケーションをご使用のサーバーに容易にポーティングできるようになります。クラス・ファイルと AIX ネイティブ・メソッド・ライブラリーをサーバー上の統合ファイル・システムにコピーし、制御言語 (CL)、Qshell、または PASE for i のいずれかの端末セッション・コマンド・プロンプトからそれらを実行できます。

PASE for i ネイティブ・メソッドを使用している場合、以下の内容を考慮する必要があります。

- ネイティブ・コードのアーキテクチャーが、JVM のアーキテクチャーに一致する必要があります。つまり、オブジェクト・バイナリーは、32 ビット JVM の場合は 32 ビット・バイナリーとして、64 ビット JVM の場合は 64 ビット・バイナリーとしてコンパイルする必要があります。これはユーザー提供の JVMTI エージェントなどのエージェントにも当てはまります。
- PASE for i 環境で実行中のコードで例外が発生した際には、Classic JVM とは異なる動作が予期されます。 Classic JVM であれば、エラーを検出してこれを適切な Java 例外に変換できました。しかし、IBM Technology for Java の場合は、これによって通常、JVM が終了します。

関連情報

PASE for i
この資料は、読者がすでに PASE for i に精通していることを前提としています。 PASE for i にまだ精通していない場合には、このトピックから Java における PASE for i ネイティブ・メソッドの使用についてさらに学習してください。

例: Java 用の IBM PASE for i ネイティブ・メソッド:

この Java 用の PASE for i ネイティブ・メソッドの例では、Java ネイティブ・インターフェース (JNI) を使用して Java コードにコールバックを行うネイティブ C メソッドのインスタンスを呼び出します。この例では、Java コードから直接ストリングにアクセスするのではなく、 JNI を通して Java にコールバックを行ってストリング値を取得するネイティブ・メソッドを呼び出します。

このソース・ファイル例の HTML 版を表示するには、以下のリンクを使用してください。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

- 532 ページの『例: PaseExample1.java』
- 533 ページの『例: PaseExample1.c』

PASE for i ネイティブ・メソッドの例を実行するためには、まず以下のトピックのタスクを完了する必要があります。

1. 534 ページの『例: 使用している AIX ワークステーションへのソース・コード例のダウンロード』
2. 534 ページの『例: ソース・コード例の準備』
3. 535 ページの『例: Java 用の PASE for i ネイティブ・メソッドの例を実行するために IBM i サーバーを準備する』

Java 用の PASE for i ネイティブ・メソッドの例を実行する

上記のタスクを完了したら、この例を実行することができます。このプログラム例を実行するには、以下のコマンドのいずれかを使用します。

- IBM i コマンド・プロンプトから:

```
JAVA CLASS(PaseExample1) CLASSPATH('/home/example')
```

- Qshell コマンド・プロンプトまたは PASE for i 端末セッションから:

```
cd /home/example
java PaseExample1
```

ネイティブ・メソッド・ライブラリーの管理

ネイティブ・メソッド・ライブラリーを使用する場合、とりわけ複数のバージョンのネイティブ・メソッド・ライブラリーを IBM i サーバーで管理するときには、Java ライブラリーの命名規則とライブラリー検索アルゴリズムの両方を理解している必要があります。

注: ネイティブ・ライブラリーまたはネイティブ・メソッド・ライブラリーという用語は、ILE ネイティブ・メソッドのコンテキストで使用される場合は、Integrated Language Environment (ILE) サービス・プログラムを指し、PASE for i ネイティブ・メソッドのコンテキストで使用される場合は、AIX 静的または共有ライブラリーを指します。

Java メソッド System.loadLibrary() は、指定されたライブラリー名のネイティブ・ライブラリーをロードするために使用されます。 IBM i は、Java 仮想マシン (JVM) がロードするライブラリーの名前と一致する最初のネイティブ・メソッドを使用します。 IBM i が正しいネイティブ・メソッドを見つけるようにするためには、ライブラリー名の競合や、JVM がどのネイティブ・メソッド・ライブラリーを使用するのかに関する混乱を避ける必要があります。

ネイティブ・ライブラリーの命名規則

以下のライブラリー命名規則に留意する必要があります。

- ネイティブ・メソッドが ILE ネイティブ・メソッドであり、Java コードが Sample という名前のライブラリーをロードする場合、対応する実行可能ファイルは SAMPLE という名前の ILE サービス・プログラムである必要があります。
- ネイティブ・メソッドが PASE for i ネイティブ・メソッドであり、Java コードが Sample という名前のライブラリーをロードする場合、対応する実行可能ファイルは libSample.a または libSample.so という名前の AIX ライブラリーである必要があります。

Java ライブラリーの検索順序

ネイティブ・ライブラリーを見つけるために、Java は java.library.path プロパティーを使用して、検索パスを判別します。デフォルトでは、java.library.path プロパティーは、2 つのリストを (以下の順序で) 連結した結果である値に設定されます。

1. IBM i ライブラリー・リスト
2. LIBPATH 環境変数の値。

検索を実行するため、IBM i はライブラリー・リストを統合ファイル・システムでの形式に変換します。 QSYS ファイル・システム・オブジェクトには、統合ファイル・システムでの等価の名前がありますが、一部の統合ファイル・システム・オブジェクトには、それと等価な QSYS ファイル・システムでの名前が存在しません。ライブラリー・ローダーは QSYS ファイル・システムと統合ファイル・システムの両方でオブジェクトを検索するため、IBM i は統合ファイル・システムでの形式を使用してネイティブ・メソッド・ライブラリーを検索します。

以下の表は、IBM i がどのようにライブラリー・リスト内のエントリーを統合ファイル・システムでの形式に変換するのかを示しています。

ライブラリー・リスト・エントリー	統合ファイルシステムでの形式
QSYS	/qsys.lib
QSYS2	/qsys.lib/qsys2.lib
QGPL	/qsys.lib/qgpl.lib
QTEMP	/qsys.lib/qtemp.lib

例: Sample2 ライブラリーの検索

この例では、LIBPATH 環境変数が /home/user1/lib32:/samples/lib32 に設定されます。以下の表は、上から下に向かって読むと、フル検索パスを表します。

ソース	統合ファイル・システムのディレクトリー
ライブラリー・リスト	/qsys.lib /qsys.lib/qsys2.lib /qsys.lib/qgpl.lib /qsys.lib/qtemp.lib
LIBPATH	/home/user1/lib32 /samples/lib32

注: 大文字と小文字は /QOpenSys パスでのみ区別されます。

ライブラリー Sample2 を検索するため、Java ライブラリー・ローダーは以下の順序でファイル候補を検索します。

1. /qsys.lib/sample2.srvpgm
2. /qsys.lib/libSample2.a
3. /qsys.lib/libSample2.so
4. /qsys.lib/qsys2.lib/sample2.srvpgm
5. /qsys.lib/qsys2.lib/libSample2.a
6. /qsys.lib/qsys2.lib/libSample2.so
7. /qsys.lib/qgpl.lib/sample2.srvpgm
8. /qsys.lib/qgpl.lib/libSample2.a
9. /qsys.lib/qgpl.lib/libSample2.so
10. /qsys.lib/qtemp.lib/sample2.srvpgm
11. /qsys.lib/qtemp.lib/libSample2.a
12. /qsys.lib/qtemp.lib/libSample2.so
13. /home/user1/lib32/sample2.srvpgm
14. /home/user1/lib32/libSample2.a
15. /home/user1/lib32/libSample2.so
16. /samples/lib32/sample2.srvpgm
17. /samples/lib32/libSample2.a
18. /samples/lib32/libSample2.so

IBM i は、実際に存在するリスト内の最初の候補を、ネイティブ・メソッド・ライブラリーとして JVM にロードします。

注: 統合ファイル・システム・ディレクトリーから QSYS ファイル・システムの中の IBM i オブジェクトへの、任意のシンボリック・リンクを作成することはできます。このため、有効なファイル候補には、/home/user1/lib32/sample2.srvpgm などのファイルも含まれる可能性があります。

Java ネイティブ・メソッドおよびスレッドに関する考慮事項

ネイティブ・メソッドは、Java では使用できない関数を利用する場合に使用できます。ネイティブ・メソッド付きの Java を使いこなすには、以下のような概念を理解する必要があります。

- Java または付加されたネイティブ・スレッドで作成されたかどうかにかかわりなく、Java スレッドでは、浮動小数点例外がすべて使用不可になっています。スレッドが浮動小数点例外を再度使用可能にするネイティブ・メソッドを実行する場合には、Java がそのメソッドを 2 度目にオフにすることはありません。ユーザー・アプリケーションが戻されて Java コードを実行する前にそれらのメソッドを使用不可

にしなければ、浮動小数点例外が起きる際に Java コードは正常に動作しないかもしれません。ネイティブ・スレッドが Java 仮想マシンから切り離される場合、そのスレッドの浮動小数点例外マスクは、付加されていた時の値に戻されます。

- ネイティブ・スレッドが Java 仮想マシンに付加される際には、必要に応じて Java 仮想マシンはスレッドの優先順位を変更して、Java が定義する 1 から 10 のスレッド優先順位体系に準拠します。スレッドが切り離されると、優先順位が復元されます。スレッドが付加されると、スレッドはネイティブ・メソッド・インターフェースを使用して (たとえば、POSIX API) スレッド優先順位を変更できます。 Java は、Java 仮想マシンへと移行する際には、スレッド優先順位を変更しません。
- Java ネイティブ・インターフェース (JNI) の呼び出し API 構成要素は、ユーザーが Java 仮想マシンをアプリケーション内に組み込むことを許可します。アプリケーションによって Java 仮想マシンが作成され、Java 仮想マシンが異常終了する場合、Java 仮想マシンが終了した際に初期スレッドが Java 仮想マシンに付加されたならば、MCH74A5 "Java Virtual Machine Terminated" IBM i 例外がプロセスの初期スレッドにシグナルされます。 Java 仮想マシンは、以下のいずれかの理由で異常終了することがあります。
  - ユーザーが java.lang.System.exit() メソッドを呼び出す。
  - Java 仮想マシンが必要なスレッドが終了する。
  - Java 仮想マシン内で内部エラーが生じる。

  この動作は、他のほとんどの Java プラットフォームとは異なります。他のほとんどのプラットフォームでは、Java 仮想マシンが終了すると、Java 仮想マシンを自動的に作成するプロセスは異常終了します。アプリケーションによって、シグナルが出された MCH74A5 例外のモニターおよび処理が行われると、そのプロセスは実行を継続するかもしれません。そうではない場合には、例外が処理されない時にプロセスは終了します。 IBM i システム固有の MCH74A5 例外を扱うコードを追加すると、他のプラットフォームへのアプリケーションの可搬性を低下させることがあります。

ネイティブ・メソッドは常にマルチスレッド・プロセスで実行されるので、ネイティブ・メソッドのコードはスレッド・セーフなものでなければなりません。このため、ネイティブ・メソッドで使用される言語および関数には次の制約があります。

- ILE CL はネイティブ・メソッドには使用しないでください。なぜなら、この言語はスレッド・セーフなものではないからです。スレッド・セーフな CL コマンドを実行するには、C 言語の system() 関数か java.lang.Runtime.exec() メソッドを使用できます。
  - C または C++ ネイティブ・メソッドでスレッド・セーフな CL コマンドを実行するには、C 言語の system() 関数を使用します。
  - スレッド・セーフな CL コマンドを Java から直接実行するには、java.lang.Runtime.exec() メソッドを使用します。
- AIX C/C++、 ILE C、ILE C++、ILE COBOL および ILE RPG を使用してネイティブ・メソッドを作成できますが、ネイティブ・メソッド内から呼び出す関数はすべてスレッド・セーフでなければなりません。

注: ネイティブ・メソッドを作成するためのコンパイル時サポートは、現時点では C、C++、および RPG 言語のみでしか提供されていません。他の言語でネイティブ・メソッドを作成することは可能ですが、ずっと複雑なものになります。

注意: ***標準 C、C++、COBOL、または RPG 関数のすべてがスレッド・セーフなものであるとは限りません。***

- C および C++ の exit() と abort() 関数は、ネイティブ・メソッド内で使用されるべきではありません。これらの関数は、Java 仮想マシンを実行するプロセス全体を停止させます。これには、プロセス内のすべてのスレッドが含まれています (Java のものであるかどうか関係なく)。

注: 参照されている exit() 関数は C および C++ 関数であり、java.lang.Runtime.exit() メソッドとは異なります。

サーバーでのスレッドの詳細については、マルチスレッド・アプリケーションを参照してください。

Java 呼び出し API

Java ネイティブ・インターフェース (JNI) の一部である呼び出し API を使用すると、Java 以外のコードで Java 仮想マシン (JVM) を作成し、Java クラスをロードおよび使用することができます。この機能により、マルチスレッド化されたプログラムは、1 つの Java 仮想マシンで実行されている Java クラスを複数のスレッドで使用できるようになります。

IBM Developer Kit for Java は、以下のタイプの呼び出し元の Java 呼び出し API をサポートしています。

- Teraspace ストレージを処理するために使用可能となる ILE プログラムまたはサービス・プログラム。ストレージ・モデルは、単一レベル・ストレージと Teraspace ストレージの場合があります。 JNI および Teraspace ストレージについて詳しくは、212 ページの『Java 用 Teraspace ストレージ・モデル・ネイティブ・メソッド』を参照してください。
- 32 ビットまたは 64 ビット AIX 用に作成された PASE for i 実行可能プログラム。

| **注:** LIBPATH および LDR_CNTRL 環境変数を、PASE for i 実行可能プログラムを実行する際に適切に設定
| しなければならない場合があります。

Java 仮想マシンは、アプリケーションによって制御されます。アプリケーションでは、Java 仮想マシンを作成し、Java メソッドを呼び出し (アプリケーションがサブルーチンを呼び出すのと類似した方法で)、Java 仮想マシンを破棄することができます。 Java 仮想マシンを作成すると、それは、アプリケーションによって明示的に破棄されるまで、プロセス内で実行可能な状態のまま残ります。 Java 仮想マシンの破棄時には、ファイナライザーの実行、Java 仮想マシン・スレッドの終了、および Java 仮想マシン・リソースの解放などの終結処理が行われます。

実行可能な状態の Java 仮想マシンがあれば、C や RPG などの ILE 言語で書かれたアプリケーションは、その Java 仮想マシンを呼び出して関数を実行することができます。また、Java 仮想マシンから C アプリケーションに戻ったり、Java 仮想マシンを再び呼び出したりすることもできます。一度 Java 仮想マシンが作成されたならば、Java コードを実行するために Java 仮想マシンを呼び出す前に、それを再作成する必要はありません。

呼び出し API を使用して Java プログラムを実行する場合、STDOUT および STDERR の宛先は、QIBM_USE_DESCRIPTOR_STDIO と呼ばれる環境変数によって制御されます。この環境変数が Y または I に設定されると (たとえば、QIBM_USE_DESCRIPTOR_STDIO=Y)、Java 仮想マシンは、STDIN (fd 0)、STDOUT (fd 1)、および STDERR (fd 2) のファイル記述子を使用します。この場合、プログラムでは、これらのファイルをこのジョブの最初の 3 つのファイルまたはパイプとしてオープンすることによって、それらのファイル記述子を有効な値に設定しなければなりません。ジョブで最初にオープンされたファイルには 0 の fd が与えられ、2 番目は 1 の fd、3 番目は 2 の fd になります。spawn API によって開始されるジョブの場合、これらの記述子は、ファイル記述子マップを使用して事前に割り当てることができます (spawn API に関する資料を参照)。環境変数 QIBM_USE_DESCRIPTOR_STDIO が設定されないか、またはその他の値に

設定されると、STDIN、STDOUT、または STDERR についてファイル記述子は使用されません。その代わりに、STDOUT および STDERR は、現行ジョブによって所有されているスプール・ファイルに送られ、STDIN を使用すると、入出力例外が起こります。

注: STDIN、STDOUT、および STDERR のファイル記述子が未設定であり、設定が必要であることが判明した場合は、メッセージ「CPFB9C8 (File descriptors 0, 1, and 2 must be open to run the PASE for i program. (ファイル記述子 0、1、および 2 を開いて、PASE for i プログラムを実行する必要があります))」が発行されます。

呼び出し API 関数

IBM Developer Kit for Java は、以下の呼び出し API 関数がサポートされます。

注: この API を使用する前に、 ジョブがマルチスレッド対応であることを確認しなければなりません。マルチスレッド対応ジョブの詳細については、マルチスレッド・アプリケーションを参照してください。

- **JNI_GetCreatedJavaVMs**

  作成済みのすべての Java 仮想マシンに関する情報を戻します。この API は複数の Java 仮想マシン (JVM) の情報を戻すように設計されていますが、1 つのプロセスに存在する JVM は 1 つだけです。したがって、この API は、最大で 1 つの JVM を戻します。

  シグニチャー:

```
jint JNI_GetCreatedJavaVMs(JavaVM **vmBuf,
                           jsize bufLen,
                           jsize *nVMs);
```

  vmBuf は出力域であり、そのサイズは bufLen (ポインターの数) によって判別されます。それぞれの Java 仮想マシンは、java.h で定義された、関連する JavaVM 構造を持ちます。この API は、作成済みのそれぞれの Java 仮想マシンに関連する JavaVM 構造へのポインターを vmBuf に格納します (vmBuf が 0 でない限り)。JavaVM 構造へのポインターは、作成された対応する Java 仮想マシンの順序で格納されます。 nVMs は、現在作成されている仮想マシンの数を戻します。ご使用のサーバーで、複数の Java 仮想マシンの作成がサポートされるため、1 よりも大きい値が予期されることもあります。この情報と、vmBuf のサイズにより、作成済みの各 Java 仮想マシンの JavaVM 構造へのポインターが戻されているかどうかが判別されます。

- **JNI_CreateJavaVM**

  Java 仮想マシンを作成し、後でそれをアプリケーション内で使用することを可能にします。

  シグニチャー:

```
jint JNI_CreateJavaVM(JavaVM **p_vm,
                      void **p_env,
                      void *vm_args);
```

  p_vm は、新たに作成された Java 仮想マシンを指す JavaVM ポインターのアドレスです。他のいくつかの JNI 呼び出し API では、p_vm を使用して Java 仮想マシンを識別します。 p_env は、新たに作成された Java 仮想マシンへの JNI 環境ポインターのアドレスです。このポインターは、JNI 関数を開始する、関数のテーブルを指します。 vm_args は、Java 仮想マシンの初期設定パラメーターを含む構造です。

  「Java プログラムの実行 (RUNJVA)」コマンドまたは JAVA コマンドを開始する場合に、同等のコマンド・パラメーターがあるプロパティーを指定すると、コマンド・パラメーターが優先され、プロパティーは無視されます。

JNI_CreateJavaVM API によってサポートされる固有のプロパティーのリストについては、 14 ページの『Java システム・プロパティー』を参照してください。

注: IBM i 上の Java は、単一のジョブまたはプロセス内で 1 つの Java 仮想マシン (JVM) の作成しかサポートしません。詳しくは、『複数の Java 仮想マシンのサポート』を参照してください。

- **DestroyJavaVM**

  Java 仮想マシンを破棄します。

  シグニチャー:

  ```
  jint DestroyJavaVM(JavaVM *vm)
  ```

  Java 仮想マシンの作成時には、vm が JavaVM ポインターとして戻されます。

- **AttachCurrentThread**

  Java 仮想マシンにスレッドを付加して、それが Java 仮想マシン・サービスを使用できるようにします。

  シグニチャー:

  ```
  jint AttachCurrentThread(JavaVM *vm,
                   void **p_env,
                   void *thr_args);
  ```

  JavaVM ポインター vm は、スレッドが付加される Java 仮想マシンを識別します。 p_env は、現行スレッドの JNI インターフェース・ポインターが置かれる場所へのポインターです。 thr_args には、VM 固有のスレッド付加引数が含まれます。

- **DetachCurrentThread**

  シグニチャー:

  ```
  jint DetachCurrentThread(JavaVM *vm);
  ```

  vm は、スレッドが切り離される Java 仮想マシンを識別します。

  Sun Microsystems, Inc. による「Java Native Interface」

複数の Java 仮想マシンのサポート

IBM i プラットフォーム上の Java は、単一のジョブまたはプロセス内での複数の Java 仮想マシン (JVM) の作成をサポートしなくなりました。この制約事項の影響を受けるのは、Java Native Interface Invocation (JNI) API を使用して JVM を作成するユーザーのみです。サポートにおけるこの変更は、Java コマンドを使用して Java プログラムを実行する方法には影響しません。

1 つのジョブで JNI_CreateJavaVM() を正常に複数回呼び出すことはできず、JNI_GetCreatedJavaVMs() は結果のリストに複数の JVM を戻すことができません。

単一のジョブまたはプロセス内での単一の JVM のみの作成のサポートは、Sun Microsystems, Inc. の Java の参照インプリメンテーションの標準に従ったものです。

例: Java 呼び出し API

この Integrated Language Environment (ILE) C の例は、標準の呼び出し API パラダイムに従っています。

これは以下を実行します。

- JNI_CreateJavaVM() を使用して Java 仮想マシンを作成する。
- Java 仮想マシンを使用して、実行したいクラス・ファイルを検索する。
- クラスの main メソッドの methodID を検索する。
- クラスの main メソッドを呼び出す。
- 例外が発生した場合に、エラーを報告する。

プログラムを作成する際は、QJVAJNI または QJVAJNI64 サービス・プログラムが、JNI_CreateJavaVM() API 機能を提供します。 JNI_CreateJavaVM() は Java 仮想マシンを作成します。

注: QJVAJNI64 は、teraspace/LLP64 ネイティブ・メソッドと呼び出し API のサポートのための新しいサービス・プログラムです。

これらのサービス・プログラムは、システム・バインディング・ディレクトリーにあり、制御言語 (CL) 作成コマンドで明示的に示す必要はありません。たとえば、前述のサービス・プログラムを「プログラム作成 (CRTPGM)」コマンドや「サービス・プログラムの作成 (CRTSRVPGM)」コマンドを使用する際に明示的に示すことはしません。

このプログラムを実行する方法の 1 つは、以下の制御言語 (CL) コマンドを使用することです。

```
SBMJOB CMD(CALL PGM(YOURLIB/PGMNAME)) ALWMLTTHD(*YES)
```

Java 仮想マシンを作成するジョブは、マルチスレッド対応でなければなりません。主プログラムからの出力と、プログラムからのすべての出力は、最終的に QPRINT スプール・ファイルに送られます。「投入されたジョブの処理 (WRKSBMJOB)」制御言語 (CL) コマンドを使用し、「ジョブの投入 (SBMJOB)」CL コマンドで開始したジョブを表示すると、これらのスプール・ファイルを見ることができます。

例: ILE C で Java 呼び出し API を使用する

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
#define OS400_JVM_12
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <jni.h>

/* Specify the pragma that causes all literal strings in the
 * source code to be stored in ASCII (which, for the strings
 * used, is equivalent to UTF-8)
 */

#pragma convert(819)

/* Procedure: Oops
 *
 * Description: Helper routine that is called when a JNI function
 *              returns a zero value, indicating a serious error.
 *              This routine reports the exception to stderr and
 *              ends the JVM abruptly with a call to FatalError.
 *
 * Parameters:  env -- JNIEnv* to use for JNI calls
 *              msg -- char* pointing to error description in UTF-8
 *
 * Note:        Control does not return after the call to FatalError
 *              and it does not return from this procedure.
 */
```

```
void Oops(JNIEnv* env, char *msg) {
    if ((*env)->ExceptionOccurred(env)) {
        (*env)->ExceptionDescribe(env);
    }
    (*env)->FatalError(env, msg);
}

/* This is the program's "main" routine. */
int main (int argc, char *argv[])
{

    JavaVMInitArgs initArgs; /* Virtual Machine (VM) initialization structure, passed by
                              * reference to JNI_CreateJavaVM(). See jni.h for details
                              */
    JavaVM* myJVM;           /* JavaVM pointer set by call to JNI_CreateJavaVM */
    JNIEnv* myEnv;           /* JNIEnv pointer set by call to JNI_CreateJavaVM */
    char*   myClasspath;     /* Changeable classpath 'string' */
    jclass  myClass;         /* The class to call, 'NativeHello'. */
    jmethodID mainID;        /* The method ID of its 'main' routine. */
    jclass  stringClass;     /* Needed to create the String[] arg for main */
    jobjectArray args;       /* The String[] itself */
    JavaVMOption options[1]; /* Options array -- use options to set classpath */
    int     fd0, fd1, fd2;   /* file descriptors for IO */

    /* Open the file descriptors so that IO works. */
    fd0 = open("/dev/null", O_CREAT|O_TRUNC|O_RDWR,   S_IRUSR|S_IROTH);
    fd1 = open("/dev/null", O_CREAT|O_TRUNC|O_WRONLY, S_IWUSR|S_IWOTH);
    fd2 = open("/dev/null", O_CREAT|O_TRUNC|O_WRONLY, S_IWUSR|S_IWOTH);

    /*  Set the version field of the initialization arguments for JNI v1.5. */
    initArgs.version = 0x00010004;


    /* Now, you want to specify the directory for the class to run in the classpath.
     * with  Java2, classpath is passed in as an option.
     * Note: You must specify the directory name in UTF-8 format. So, you wrap
     *       blocks of code in #pragma convert statements.
     */
    options[0].optionString="-Djava.class.path=/CrtJvmExample";

    initArgs.options=options;  /* Pass in the classpath that has been set up. */
    initArgs.nOptions = 1;     /* Pass in classpath and version options */

    /*  Create the JVM -- a nonzero return code indicates there was
     *  an error. Drop back into EBCDIC and write a message to stderr
     *  before exiting the program.
     *  Note:  This will run the default JVM and JDK which is 32bit JDK 6.0.
     *  If you want to run a different JVM and JDK, set the JAVA_HOME environment
     *  variable to the home directory of the JVM you want to use
     *  (prior to the CreateJavaVM() call).
     */
    if (JNI_CreateJavaVM(&myJVM, (void **)&myEnv, (void *)&initArgs)) {
  #pragma convert(0)
         fprintf(stderr, "Failed to create the JVM¥n");
  #pragma convert(819)
         exit(1);
    }

    /*  Use the newly created JVM to find the example class,
     *  called 'NativeHello'.
     */
    myClass = (*myEnv)->FindClass(myEnv, "NativeHello");
    if (! myClass) {
        Oops(myEnv, "Failed to find class 'NativeHello'");
    }

    /* Now, get the method identifier for the 'main' entry point
```

```
     * of the class.
     * Note: The signature of 'main' is always the same for any
     *       class called by the following java command:
     *           "main" , "([Ljava/lang/String;)V"
     */
    mainID = (*myEnv)->GetStaticMethodID(myEnv,myClass,"main",
                                         "([Ljava/lang/String;)V");
    if (! mainID) {
        Oops(myEnv, "Failed to find jmethodID of 'main'");
    }

    /* Get the jclass for String to create the array
     * of String to pass to 'main'.
     */
    stringClass = (*myEnv)->FindClass(myEnv, "java/lang/String");
    if (! stringClass) {
        Oops(myEnv, "Failed to find java/lang/String");
    }

    /*  Now, you need to create an empty array of strings,
     *  since main requires such an array as a parameter.
     */
    args = (*myEnv)->NewObjectArray(myEnv,0,stringClass,0);
    if (! args) {
        Oops(myEnv, "Failed to create args array");
    }

    /* Now, you have the methodID of main and the class, so you can
     * call the main method.
     */
    (*myEnv)->CallStaticVoidMethod(myEnv,myClass,mainID,args);

    /* Check for errors. */
    if ((*myEnv)->ExceptionOccurred(myEnv)) {
        (*myEnv)->ExceptionDescribe(myEnv);
    }

    /* Finally, destroy the JavaVM that you created. */
    (*myJVM)->DestroyJavaVM(myJVM);

    /* All done. */
    return 0;
}
```

詳しくは、 220 ページの『Java 呼び出し API』を参照してください。

java.lang.Runtime.exec() を使用する

java.lang.Runtime.exec() メソッドを使用して、Java プログラム内からプログラムまたはコマンドを呼び出します。 java.lang.Runtime.exec() メソッドを使用すると、1 つ以上の追加のスレッド対応のジョブが作成されます。追加のジョブが、このメソッドに渡されたコマンド・ストリングを処理します。

java.lang.Runtime.exec() メソッドは別個のジョブでプログラムを実行します。これは C の system() 関数とは異なる点です。 C system() 関数は、同一のジョブでプログラムを実行します。実際に発生する処理は、java.lang.Runtime.exec() に渡すコマンドの種類によって異なります。以下の表は、java.lang.Runtime.exec() が種々のタイプのコマンドを処理する方法を示しています。

コマンドのタイプ	コマンドの処理方法
java コマンド	JVM を実行する 2 番目のジョブを開始します。 JVM は、Java アプリケーションを実行する 3 番目のジョブを開始します。

コマンドのタイプ	コマンドの処理方法
プログラム	実行可能プログラム (IBM i ILE プログラムまたは PASE for i プログラム) を実行する 2 番目のジョブを開始します。
CL コマンド	IBM i ILE プログラムを実行する 2 番目のジョブを開始します。 IBM i ILE は、2 番目のジョブで CL コマンドを実行します。

例: java.lang.Runtime.exec() を使用して別の Java プログラムを呼び出す

この例では、java.lang.Runtime.exec() を使用して別の Java プログラムを呼び出す方法を示します。このクラスは、IBM Developer Kit for Java の一部として配布される Hello プログラムを呼び出します。Hello クラスが System.out に書き込みを行うときに、このプログラムは、ストリームへのハンドルを取得し、そこから読み取りを行うことができます。

CallHelloPgm Java クラスのソース・コード

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.io.*;

public class CallHelloPgm
{
   public static void main(String args[])
   {
      Process theProcess = null;
      BufferedReader inStream = null;

      System.out.println("CallHelloPgm.main() invoked");

      // call the Hello class
      try
      {
          theProcess = Runtime.getRuntime().exec("java QIBMHello");
      }
      catch(IOException e)
      {
         System.err.println("Error on exec() method");
         e.printStackTrace();
      }

      // read from the called program's standard output stream
      try
      {
         inStream = new BufferedReader(
                              new InputStreamReader( theProcess.getInputStream() ));
         System.out.println(inStream.readLine());
      }
      catch(IOException e)
      {
         System.err.println("Error on inStream.readLine()");
         e.printStackTrace();
      }
   }
}
```

例: java.lang.Runtime.exec() を使用して CL プログラムを呼び出す

この例では、Java プログラムから CL プログラムを実行する方法を示します。この例では、Java クラス CallCLPgm が CL プログラムを実行します。

| CL プログラムは、「Java 仮想マシン・ジョブ表示」(DSPJVMJOB) CL コマンドを使用して、アクティブ
| な Java 仮想マシンを含むシステム上のジョブをすべて表示します。この例では、CL プログラムはコンパ
| イル済みであり、JAVSAMPLIB と呼ばれるライブラリーに存在していることが前提となっています。 CL
| プログラムからの出力は、QSYSPRT スプール・ファイルにあります。

Java プログラムから CL コマンドを呼び出す方法の例については、『例: java.lang.Runtime.exec() を使用して CL コマンドを呼び出す』 を参照してください。

注: JAVSAMPLIB は、IBM Developer Kit ライセンス・プログラム (LP) (番号 5761-JV1) のインストール・プロセスの一部としては作成されません。このライブラリーは明示的に作成する必要があります。

CallCLPgm Java クラスのソース・コード

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.io.*;

public class CallCLPgm
{
   public static void main(String[] args)
   {
      try
      {
         Process theProcess =
                 Runtime.getRuntime().exec("/QSYS.LIB/JAVSAMPLIB.LIB/DSPJVA.PGM");
      }
      catch(IOException e)
      {
         System.err.println("Error on exec() method");
         e.printStackTrace();
      }
   }
}
```

DSPJVA CL プログラムのソース・コード

```
| PGM
|     DSPJVMJOB OUTPUT(*PRINT)
| ENDPGM
```

例: java.lang.Runtime.exec() を使用して CL コマンドを呼び出す

この例では、Java プログラムから制御言語 (CL) コマンドを実行する方法を示します。

| この例では、Java クラスが CL コマンドを実行します。 CL コマンドは、「Java 仮想マシン・ジョブ表
| 示」(DSPJVMJOB) CL コマンドを使用して、アクティブな Java 仮想マシンを含むシステム上のジョブを
| すべて表示します。 CL コマンドからの出力は、QSYSPRT スプール・ファイルにあります。

`Runtime.getRuntime().exec()` 関数に渡す CL コマンドは次のフォーマットになります。

```
Runtime.getRuntime().exec("system CLCOMMAND");
```

ここで、*CLCOMMAND* は、実行しようとしている CL コマンドです。

CL コマンドを呼び出すための Java クラスのソース・コード

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.io.*;

public class CallCLCom
{
   public static void main(String[] args)
   {
      try
      {
         Process theProcess =
            Runtime.getRuntime().exec("system DSPJVMJOB OUTPUT(*PRINT)");
      }
      catch(IOException e)
      {
         System.err.println("Error on exec() method");
         e.printStackTrace();
      }
   }
}
```

関連概念

225 ページの『java.lang.Runtime.exec() を使用する』
`java.lang.Runtime.exec()` メソッドを使用して、Java プログラム内からプログラムまたはコマンドを呼び出します。 `java.lang.Runtime.exec()` メソッドを使用すると、1 つ以上の追加のスレッド対応のジョブが作成されます。 追加のジョブが、このメソッドに渡されたコマンド・ストリングを処理します。

15 ページの『Java システム・プロパティーのリスト』
Java システム・プロパティーにより、Java プログラムのランタイム環境が決まります。 Java システム・プロパティーは、IBM i のシステム値や環境変数と似ています。

プロセス間通信

別のプロセスで実行されているプログラムと通信する際には、いくつかのオプションがあります。

オプションの 1 つは、プロセス間通信にソケットを使用することです。一方のプログラムは、サーバー・プログラムの役割を果たし、ソケット接続上で、クライアント・プログラムからの入力がないか listen します。クライアント・プログラムは、ソケットを使用してサーバーに接続します。ソケット接続が確立されると、どちらのプログラムでも情報を送受信することができます。

別のオプションは、プログラム間の通信にストリーム・ファイルを使用することです。これを行うには、System.in、System.out、および System.err クラスを使用します。

3 つ目のオプションは、IBM Toolbox for Java を使用する通信方法です。この方法では、データ待ち行列と IBM i メッセージ・オブジェクトを使用します。

さらに、この後の例で示すように、他の言語から Java を呼び出すこともできます。

関連情報

IBM Toolbox for Java

プロセス間通信にソケットを使用する

ソケット・ストリームは、異なるプロセス内で実行しているプログラム間での通信を行います。

プログラムは別個に開始することもできますし、メインの Java プログラム内から java.lang.Runtime.exec() メソッドを使用して開始することもできます。プログラムが Java 以外の言語で記述されている場合、情報交換用米国標準コード (ASCII) または拡張 2 進化 10 進コード (EBCDIC) 変換が確実に行われるようにしなければなりません。詳細については、『Java 文字のエンコード』のトピックを参照してください。

関連概念

225 ページの『java.lang.Runtime.exec() を使用する』
java.lang.Runtime.exec() メソッドを使用して、Java プログラム内からプログラムまたはコマンドを呼び出します。 java.lang.Runtime.exec() メソッドを使用すると、1 つ以上の追加のスレッド対応のジョブが作成されます。 追加のジョブが、このメソッドに渡されたコマンド・ストリングを処理します。

22 ページの『Java 文字のエンコード』
Java プログラムは、他のフォーマットのデータを変換して、アプリケーションが多様な国際文字セットの情報を転送および使用できるようにします。

例: プロセス間通信のためにソケットを使用する:

この例では、ソケットを使用して Java プログラムと C プログラムとの間で通信します。

最初に、ソケット上で聴取する C プログラムを開始してください。 Java プログラムがソケットに接続された後は、ソケット接続を使用して C プログラムがそれにストリングを送ります。 C プログラムから送られるストリングは、コード・ページ 819 の ASCII コードのストリングです。

Qshell インタープリターのコマンド行か、または他の Java プラットフォームで、コマンド `java TalkToC xxxxx nnnn` を使用して Java プログラムを開始しなくてはなりません。または IBM i コマンド行に `JAVA TALKTOC PARM(xxxxx nnnn)` を入力することにより、Java プログラムを開始します。 `xxxxx` は、C プログラムが実行されているシステムのドメイン・ネームまたはインターネット・プロトコル (IP) アドレスです。 `nnnn` は、C プログラムが使用するソケットのポート番号です。このポート番号は、C プログラムを呼び出すときに最初に渡すパラメーターとして指定する必要があります。

TalkToC クライアント Java クラスのソース・コード

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.net.*;
import java.io.*;

class TalkToC
{
   private String host = null;
   private int port = -999;
   private Socket socket = null;
   private BufferedReader inStream = null;

   public static void main(String[] args)
   {
      TalkToC caller = new TalkToC();
      caller.host = args[0];
      caller.port = new Integer(args[1]).intValue();
      caller.setUp();
      caller.converse();
      caller.cleanUp();
   }

   public void setUp()
   {
      System.out.println("TalkToC.setUp() invoked");
```

```
      try
      {
         socket = new Socket(host, port);
         inStream = new BufferedReader(new InputStreamReader(
                                       socket.getInputStream()));
      }
      catch(UnknownHostException e)
      {
         System.err.println("Cannot find host called: " + host);
         e.printStackTrace();
         System.exit(-1);
      }
      catch(IOException e)
      {
         System.err.println("Could not establish connection for " + host);
         e.printStackTrace();
         System.exit(-1);
      }
   }

   public void converse()
   {
      System.out.println("TalkToC.converse() invoked");

      if (socket != null && inStream != null)
      {
         try
         {
            System.out.println(inStream.readLine());
         }
         catch(IOException e)
         {
            System.err.println("Conversation error with host " + host);
            e.printStackTrace();
         }
      }
   }

   public void cleanUp()
   {
      try
      {
         if (inStream != null)
            inStream.close();
         if (socket != null)
            socket.close();
      }
      catch(IOException e)
      {
         System.err.println("Error in cleanup");
         e.printStackTrace();
         System.exit(-1);
      }
   }
}
```

SockServ.C は、ポート番号のパラメーターを渡すことによって開始します。たとえば、CALL SockServ '2001' とします。

SockServ.C サーバー・プログラムのソース・コード

注: 法律上の重要な情報に関しては、コードの特記事項情報をお読みください。

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
```

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <unistd.h>
#include <sys/time.h>

void main(int argc, char* argv[])
{
   int    portNum = atoi(argv[1]);
   int    server;
   int    client;
   int    address_len;
   int    sendrc;
   int    bndrc;
   char*  greeting;
   struct sockaddr_in  local_Address;
   address_len = sizeof(local_Address);

   memset(&local_Address,0x00,sizeof(local_Address));
   local_Address.sin_family = AF_INET;
   local_Address.sin_port = htons(portNum);
   local_Address.sin_addr.s_addr = htonl(INADDR_ANY);

   #pragma convert (819)
   greeting = "This is a message from the C socket server.";
   #pragma convert (0)

   /*  allocate socket    */
   if((server = socket(AF_INET, SOCK_STREAM, 0))<0)
   {
      printf("failure on socket allocation¥n");
      perror(NULL);
      exit(-1);
   }

   /* do bind   */
   if((bndrc=bind(server,(struct sockaddr*)&local_Address, address_len))<0)
   {
     printf("Bind failed¥n");
     perror(NULL);
     exit(-1);
   }

   /* invoke listen   */
   listen(server, 1);

   /* wait for client request */
   if((client = accept(server,(struct sockaddr*)NULL, 0))<0)
   {
     printf("accept failed¥n");
     perror(NULL);
     exit(-1);
   }

   /* send greeting to client    */
   if((sendrc = send(client, greeting, strlen(greeting),0))<0)
   {
      printf("Send failed¥n");
      perror(NULL);
      exit(-1);
   }

   close(client);
   close(server);
}
```

プロセス間通信に入出力ストリームを使用する

入出力ストリームは、別々のプロセスで実行されているプログラムの間で通信を行います。

java.lang.Runtime.exec() メソッドはプログラムを実行します。親プログラムは、子プロセスの入出力ストリームへのハンドルを取得し、それらのストリームに対する書き込みおよび読み取りを行うことができます。子プログラムが Java 以外の言語で作成されている場合は、情報交換用米国標準コード (ASCII) コードまたは拡張 2 進化 10 進コード (EBCDIC) の変換が行われるようにしなければなりません。詳細は、『Java 文字のエンコード』を参照してください。

関連概念

225 ページの『java.lang.Runtime.exec() を使用する』
`java.lang.Runtime.exec()` メソッドを使用して、Java プログラム内からプログラムまたはコマンドを呼び出します。 `java.lang.Runtime.exec()` メソッドを使用すると、1 つ以上の追加のスレッド対応のジョブが作成されます。 追加のジョブが、このメソッドに渡されたコマンド・ストリングを処理します。

22 ページの『Java 文字のエンコード』
Java プログラムは、他のフォーマットのデータを変換して、アプリケーションが多様な国際文字セットの情報を転送および使用できるようにします。

例: プロセス間通信に入出力ストリームを使用する:

この例では、Java から C プログラムを呼び出し、プロセス間通信に入出力ストリームを使用する方法を示します。

C プログラムは、その標準出力ストリームにストリングを書き込み、Java プログラムは、このストリングを読み取り、表示します。この例では、JAVSAMPLIB というライブラリーが作成されていることと、その中で CSAMP1 プログラムが作成されていることを前提としています。

注: JAVSAMPLIB は、IBM Developer Kit ライセンス・プログラム (LP) (番号 5761-JV1) のインストール・プロセスの一部としては作成されません。明示的にそれを作成しなければなりません。

CallPgm Java クラスのソース・コード

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.io.*;

public class CallPgm
{
   public static void main(String args[])
   {
      Process theProcess = null;
      BufferedReader inStream = null;

      System.out.println("CallPgm.main() invoked");

      // call the CSAMP1 program
      try
      {
          theProcess = Runtime.getRuntime().exec(
                      "/QSYS.LIB/JAVSAMPLIB.LIB/CSAMP1.PGM");
      }
      catch(IOException e)
      {
         System.err.println("Error on exec() method");
         e.printStackTrace();
      }
```

```
      // read from the called program's standard output stream
      try
      {
         inStream = new BufferedReader(new InputStreamReader
                          (theProcess.getInputStream()));
         System.out.println(inStream.readLine());
      }
      catch(IOException e)
      {
         System.err.println("Error on inStream.readLine()");
         e.printStackTrace();
      }
   }
 }
```

CSAMP1 C プログラムのソース・コード

注: 法律上の重要な情報に関しては、コードの特記事項情報をお読みください。

```
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char* args[])
{
     /* Convert the string to ASCII at compile time */
#pragma convert(819)
     printf("Program JAVSAMPLIB/CSAMP1 was invoked¥n");
#pragma convert(0)
     /* Stdout may be buffered, so flush the buffer */

     fflush(stdout);
}
```

例: ILE C から Java を呼び出す

次に示すのは、system() 関数を使用して Java Hello プログラムを呼び出す Integrated Language Environment (ILE) C プログラムの例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
#include <stdlib.h>

 int main(void)
 {
   int result;

   /* The system function passes the given string
    * to the CL command processor for processing.
    */

   result = system("JAVA CLASS('QIBMHello')");
 }
```

例: RPG から Java を呼び出す

次に示すのは、QCMDEXC API を使用して Java Hello プログラムを呼び出す RPG プログラムの例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
D*             DEFINE  THE PARAMETERS FOR THE QCMDEXC API
D*
DCMDSTRING        S            25    INZ('JAVA CLASS(''QIBMHello'')')
DCMDLENGTH        S            15P 5 INZ(23)
D*             NOW THE CALL TO QCMDEXC WITH THE 'JAVA' CL COMMAND
C                   CALL      'QCMDEXC'
C                   PARM                    CMDSTRING
C                   PARM                    CMDLENGTH
C*       This next line displays 'DID IT' after you exit the
C*       Java Shell via F3 or F12.
C     'DID IT'      DSPLY
C*       Set On LR to exit the RPG program
C                   SETON                                        LR
C
```

Java プラットフォーム

Java プラットフォームは、Java アプレットおよびアプリケーションを開発して管理するための環境です。これは、3 つの主要なコンポーネント (Java 言語、Java パッケージ、および Java 仮想マシン) で構成されます。

Java 言語およびパッケージは、C++ およびそのクラス・ライブラリーと類似しています。 Java パッケージにはクラスが含まれていて、どの準拠 Java 実装でも利用できます。アプリケーション・プログラミング・インターフェース (API) は、Java をサポートするどのシステムでも同じはずです。

Java が C++ のような従来型の言語と異なる点は、コンパイルして実行する方法です。従来型のプログラミング環境では、プログラムのソース・コードを作成してコンパイルし、特定ハードウェアおよびオペレーティング・システムのオブジェクト・コードにします。このオブジェクト・コードを別のオブジェクト・コード・モジュールにバインドして、実行プログラムを作成します。このコードは、特定のコンピューター・ハードウェア・セットに固有なものですから、変更を加えることなしには、別のシステムでは稼働しません。

Java アプレットおよびアプリケーション

アプレットは、HTML Web 文書に含めるよう設計された Java プログラムです。 Java アプレットを作成し、イメージを組み込むのとほとんど同じ方法で、HTML ページに組み込むことができます。 Java を利用できるブラウザーを使用して、アプレットを含む HTML ページを表示すると、アプレットのコードがシステムに転送され、ブラウザーの Java 仮想マシンで実行されます。

HTML 文書には、Java アプレットの名前と、その URL が指定されたタグが含まれます。 URL は、そのアプレットのバイトコードが存在するインターネット上の場所を示します。 Java アプレットのタグが含まれた HTML 文書が表示されると、Java を使用できる Web ブラウザーはインターネットから Java バイトコードをダウンロードし、Web 文書内のコードを処理するために Java 仮想マシンを使用します。これらの Java アプレットを使って、Web ページにグラフィックスのアニメーション表示や、対話式のコンテンツを含めることができます。

また、Web ブラウザーを使用しない Java アプリケーションを作成することも可能です。

詳しくは、Sun Microsystems の Java アプレットのチュートリアル Writing Applets を参照してください。このページには、アプレットの概要やアプレットの記述方法、およびアプレットに関連した一般的な問題が含まれています。

アプリケーションは、ブラウザーを使用せずに実行できる、スタンドアロン・プログラムです。 Java アプリケーションは、コマンド入力行から Java インタープリターを開始することによって、またコンパイルさ

れたアプリケーションのファイルを指定することによって実行できます。通常、アプリケーションは配置されたシステム上にあります。アプリケーションはシステム上のリソースにアクセスしますが、そのアクセスは Java セキュリティー・モデルによって制限されます。

Java 仮想マシン

Java 仮想マシンは、Web ブラウザーまたは任意のオペレーティング・システム (IBM i など) に追加できるランタイム環境です。 Java 仮想マシンは Java コンパイラーが生成する命令を実行します。これは、バイトコード・インタープリターとランタイムで構成されています。このランタイムでは、もともと開発されたプラットフォームに関係なく、任意のプラットフォームで Java クラス・ファイルを実行できます。

クラス・ローダーおよびセキュリティー・マネージャーは、Java ランタイムの一部で、別のプラットフォームからのコードを隔離します。ロードされるクラスごとに、アクセスを許可するシステム・リソースを制限することも可能です。

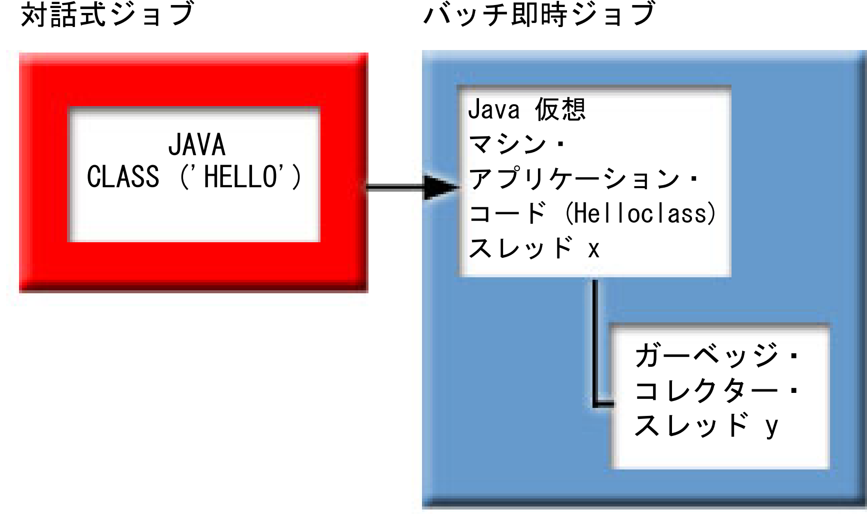
注: Java アプリケーションは制限されません。 制限されるのはアプレットだけです。アプリケーションは自由にシステム・リソースにアクセスして、ネイティブ・メソッドを使用できます。ほとんどの IBM Developer Kit for Java プログラムはアプリケーションです。

バイトコードのロードおよび実行のほかに、Java 仮想マシンには、メモリーを管理するガーベッジ・コレクターが含まれています。 401 ページの『Java ガーベッジ・コレクション』は、バイトコードのロードおよび解釈と同時に実行されます。

Java ランタイム環境

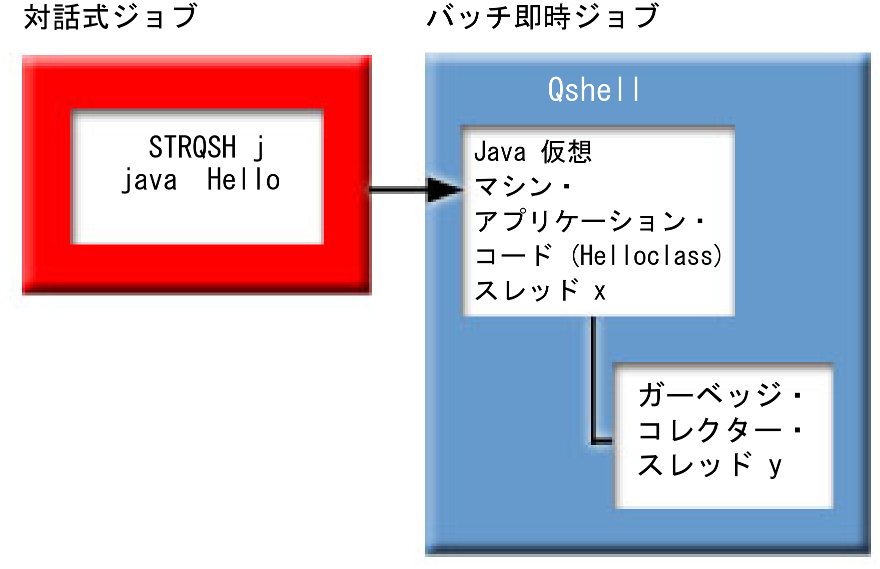
IBM i のコマンド行に「Java プログラムの実行 (RUNJVA)」コマンドまたは JAVA コマンドを入力すると、Java ランタイム環境が開始されます。 Java 環境はマルチスレッドをサポートするので、バッチ即時 (BCI) ジョブなどのスレッドをサポートするジョブで Java 仮想マシンを実行する必要があります。以下の図で示されているように、Java 仮想マシンが起動されると、ガーベッジ・コレクターが実行するジョブで追加のスレッドを開始されます。

図 1: RUNJVA または JAVA CL コマンドを使用する場合の標準的な Java 環境



また、Qshell インタープリターから Qshell 内で java コマンドを使って Java ランタイム環境を開始することもできます。この環境では、Qshell インタープリターは対話式ジョブと関連付けられた BCI ジョブで実行されます。 Java ランタイム環境は、Qshell インタープリターが実行されているジョブで開始されます。

図 2: Qshell で Java コマンドを使用する場合の Java 環境



Java ランタイム環境を対話式ジョブから開始すると、Java シェル画面が表示されます。この画面の入力行を使って、System.in ストリームにデータを入力することができます。 また、System.out ストリームや System.err ストリームに書き込まれたデータも表示されます。

Java インタープリター

Java インタープリターは、特定のハードウェア・プラットフォームで Java クラス・ファイルを解釈する、Java 仮想マシンの一部です。 Java インタープリターは各バイトコードをデコードし、対応する操作を実行します。

QP2TERM() での PASE for i プログラムの実行

221 ページの『呼び出し API 関数』

IBM Developer Kit for Java は、以下の呼び出し API 関数がサポートされます。

Java JAR とクラス・ファイル

Java ARchive (JAR) ファイルは、複数のファイルを 1 つに結合したファイル・フォーマットです。 Java 環境と他のプログラミング環境の異なる点は、Java コンパイラーでは、ハードウェア固有の命令セット用にマシン・コードを生成しないということです。代わりに、Java コンパイラーは、Java ソース・コードを Java 仮想マシンの命令に変換し、それらを Java クラス・ファイルに保管します。 JAR ファイルを使用して、クラス・ファイルを保管することができます。クラス・ファイルは特定のハードウェア・プラットフォームを対象にすることはありませんが、Java 仮想マシン・アーキテクチャーを対象とします。

JAR は一般的なアーカイブ・ツールとして使用でき、すべてのタイプ (アプレットも含む) の Java プログラムを配布できます。 Java アプレットは、1 つずつ新しい接続をオープンするのではなく、一度の

Hypertext Transfer Protocol (HTTP) トランザクションで、ブラウザーにダウンロードします。この方式でのダウンロードでは、Web ページ上でアプレットがロードして機能を開始する速度が向上します。

JAR フォーマットは圧縮もサポートしています。これは、ファイルのサイズを減らし、ダウンロード時刻を短縮します。さらにアプレット作成者は、作成元を認証するために、JAR ファイル内のそれぞれの項目にディジタルで署名できます。

JAR ファイル内のクラスを更新する場合は、`jar` ツールを使用します。

Java クラス・ファイルは、Java コンパイラーがソース・ファイルをコンパイルするときに作成される、ストリーム・ファイルです。クラス・ファイルには、クラスの各フィールドおよびメソッドを記述するテーブルが含まれています。またこのファイルには、各メソッドのバイトコード、静的データ、および Java オブジェクトを表すのに使用される記述も含まれています。

関連情報

Sun Microsytems, Inc. による「Java jar tool」

Java スレッド

スレッドとは、プログラム内で実行される、単独の独立したストリームのことを言います。 Java はマルチスレッド・プログラミング言語であるため、Java 仮想マシン内では、一度の複数のスレッドを実行することができます。 Java スレッドは、Java プログラムが同時に複数のタスクを実行するための手段として使用されます。スレッドは、本質的にプログラム内の制御のフローです。

スレッドは、並行プログラムをサポートし、アプリケーションのパフォーマンスとスケーラビリティーを向上させるのに使用される、現代的なプログラミング構成要素です。ほとんどのプログラミング言語では、アドイン・プログラミング・ライブラリーを使用することによってスレッドをサポートします。 Java の場合は、組み込みアプリケーション・プログラミング・インターフェース (API) として、スレッドをサポートしています。

注: スレッドを使用すると、より多くのタスクが並行して実行されるため、 対話性の向上、つまりキーボードでの待機時間の短縮がサポートされます。ただし、プログラムの対話機能は、必ずしもスレッドがあるというだけで向上するとは限りません。

スレッドは、実行時間の長い対話で待機しながら、プログラムがなおその他の作業も処理できるようにするためのメカニズムです。スレッドを使用すると、同じコード・ストリームの中で複数のフローをサポートすることができます。これは、**軽量プロセス**と呼ばれることもあります。 Java 言語には、スレッドの直接サポートも組み込まれています。しかし、設計上、割り込みや複数の待ちがある非同期で非ブロッキングの入出力は、サポートされていません。

スレッドを使用すると、マシンに複数のプロセッサーがある環境に適した、並列プログラムを作成できます。これは、適切に構成されれば、複数のトランザクションやユーザーの処理のためのモデルともなります。

Java プログラムのスレッドは、さまざまな状況で使用できます。プログラムの中には、複数のアクティビティーに携わることができなければならず、なおかつユーザーからのさらに別の入力にも応答できなければならないものがあります。たとえば、Web ブラウザーには、音声を再生しながらユーザーの入力に応答する能力が求められるでしょう。

スレッドでは、非同期メソッドを使用することもできます。 2 つ目のメソッドを呼び出したときに、1 つ目のメソッドが完了するまで 2 つ目のメソッドが自身のアクティビティーを続けるのを待つ必要はありません。

ただし、スレッドを使用しないほうが良い場合もたくさんあります。階層的な順次の論理が使用されるプログラムでは、1 つのスレッドでシーケンス全体を完了させることができます。このようなケースでは、複数のスレッドを使用してもプログラムが複雑になるだけで、何の益もありません。スレッドの作成と開始には、かなりの作業が伴います。操作に関係するステートメントが 2 つか 3 つしかないのであれば、それは 1 つのスレッドで扱った方が速いでしょう。これは、その操作が概念的に非同期である場合でもそういえます。複数のスレッドがオブジェクトを共用すると、オブジェクトには、スレッド・アクセスを調整し、整合性を保守するための同期化が必要になります。同期化を行うとなれば、プログラムはそれだけ複雑になり、パフォーマンスを最適化するための調整を難しくしたり、プログラミングのソースにエラーを引き起こしてしまう可能性があります。

スレッドについての詳細は、マルチスレッド・アプリケーションの作成を参照してください。

Java Development Kit

Java Development Kit (JDK) は、Java 開発者用のソフトウェアです。このソフトウェアには、Java インタープリター、Java クラス、および Java 開発ツール (コンパイラー、 デバッガー、逆アセンブラー、appletviewer、スタブ・ファイル・ジェネレーター、および文書ジェネレーター) が含まれています。

JDK では、一度開発されたアプリケーションを作成し、任意の Java 仮想マシン上の任意の場所で実行することができます。ある 1 つのシステムで JDK を使用して開発された Java アプリケーションを、コードの変更や再コンパイルを行うことなく、他のシステムでも使用することが可能です。 Java クラス・ファイルは、標準の Java 仮想マシンであれば、そのマシンにでも移植できます。

現在の JDK に関する詳細な情報を得るには、ご使用のサーバーにインストールされている IBM Developer Kit for Java のバージョンを確認してください。

ご使用のサーバーのデフォルトの IBM Developer Kit for Java Java 仮想マシンのバージョンは、以下のいずれかのコマンドを入力することによって確認できます。

- `java -version` (Qshell コマンド・プロンプトの場合)
- `RUNJVA CLASS(*VERSION)` (CL コマンド行の場合)

次に、The Source for Java Technology java.sun.com のページで同じバージョンの Sun Microsystems, Inc. JDK を探し、具体的な資料を見つけてください。 IBM Developer Kit for Java は、Sun Microsystems, Inc. の Java Technology と互換性のある製品であるため、その JDK 資料に精通しておくことは必要でしょう。

Java パッケージ

Java パッケージは、Java に関連するクラスとインターフェースをグループ化する 1 つの方法です。 Java パッケージは、他の言語で利用可能なクラス・ライブラリーと同様のものです。

Java API を同梱する Java パッケージは、Sun Microsystems, Inc. Java Development Kit (JDK) の一部として入手できます。 Java パッケージと、Java API の情報の完全なリストは、「Java 2 Platform Packages」を参照してください。

Java ツール

Sun Microsystems, Inc. Java Development Kit で提供されているツールの完全なリストは、Sun Microsystems, Inc. の「Tools Reference」を参照してください。IBM Developer Kit for Java でサポートされている個々のツールについての詳細は、「IBM Developer Kit for Java がサポートする Java ツール」を参照してください。

6 ページの『複数の Java Development Kit (JDK) のサポート』
IBM i プラットフォームでは、複数のバージョンの Java Development Kit (JDK) と Java 2 Platform, Standard Edition がサポートされています。

208 ページの『ネイティブ・メソッドおよび Java ネイティブ・インターフェース』
ネイティブ・メソッドとは、Java 以外の言語で開始される Java メソッドです。 ネイティブ・メソッドは、Java では直接使用できないシステム固有の機能や API にアクセスできます。

404 ページの『Java ツールおよびユーティリティー』
Qshell 環境には、プログラム開発で一般的に必要とされる Java 開発ツールが組み込まれています。

Java 2 Platform Packages

Sun Microsystems, Inc. による「Tools Reference」

高度なトピック

このトピックでは、バッチ・ジョブで Java を実行する方法を説明し、Java プログラムを表示、実行、またはデバッグするために、統合ファイル・システムで必要な Java ファイル権限について説明します。

Java クラス、パッケージ、およびディレクトリー

Java のクラスはそれぞれ、あるパッケージに属しています。どのクラスがどのパッケージに含まれるかは、Java ソース・ファイルの最初のステートメントに記述されます。ソース・ファイルにパッケージ・ステートメントがない場合、そのクラスは名前のないデフォルトのパッケージに含まれると見なされます。

パッケージ名は、クラスの位置するディレクトリー構造と関連があります。統合ファイル・システムでは、多くの PC システムや UNIX® システムと同様に、階層ファイル構造で Java クラスを格納できます。Java クラスを格納するディレクトリーの相対ディレクトリー・パスは、そのクラスが属するパッケージの名前と一致していなければなりません。たとえば、次の Java クラスで考えてみます。

```
package classes.geometry;
import java.awt.Dimension;

public class Shape {

Dimension metrics;

   // The implementation for the Shape class would be coded here ...

}
```

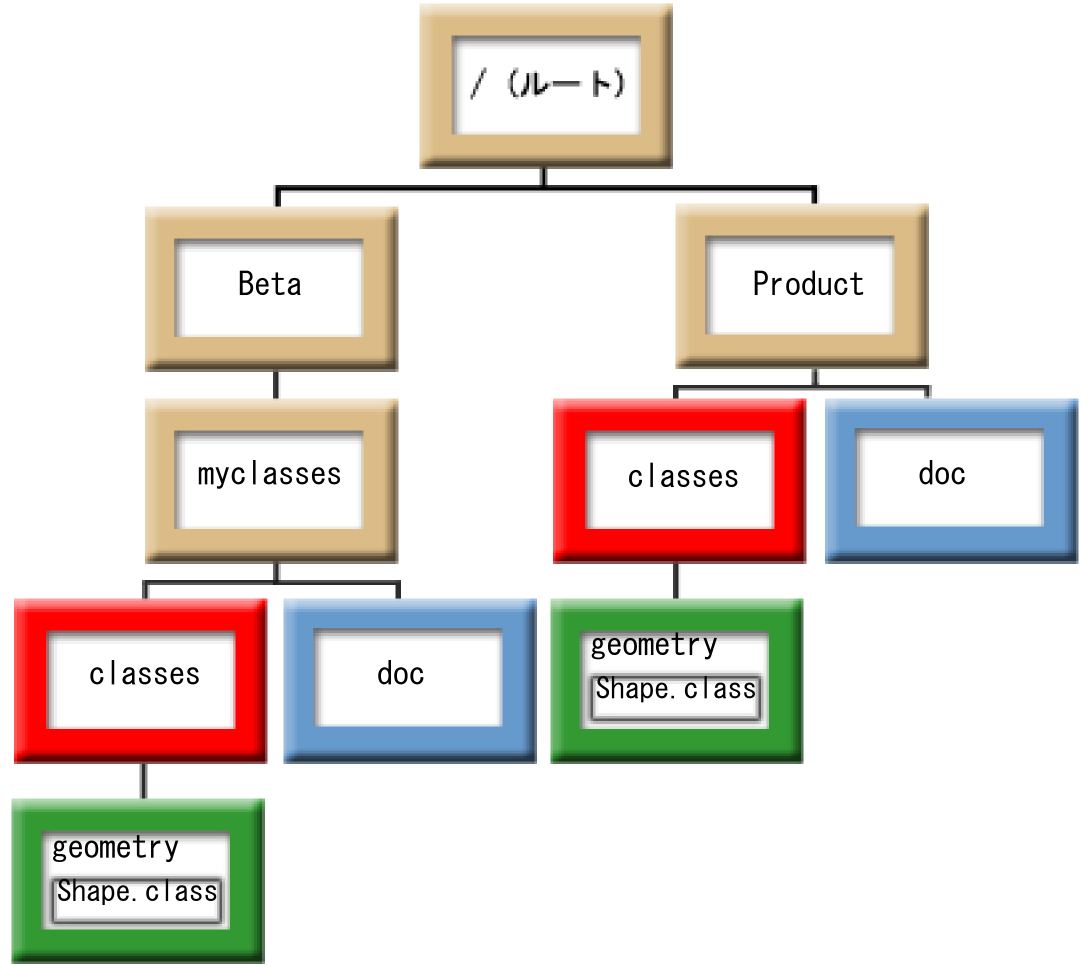
上記のコードのパッケージ・ステートメントは、Shape クラスが classes.geometry パッケージに属していることを示しています。したがって、Java ランタイムが Shape クラスを検出するためには、Shape クラスが相対ディレクトリー構造の classes/geometry に格納されていなければなりません。

注: パッケージ名は、クラスが格納されているディレクトリーの相対ディレクトリー名に対応しています。Java 仮想マシンのクラス・ローダーは、クラスパスで指定された各ディレクトリーに相対パス名を追

加してクラスを探します。また、Java 仮想マシンのクラス・ローダーは、クラスパスで指定された ZIP ファイルまたは JAR ファイルを検索してクラスを検出することもできます。

たとえば、Shape クラスが「ルート」(/) ファイル・システムの /Product/classes/geometry ディレクトリーに格納されている場合は、クラスパスに /Product を指定する必要があります。

図 1: 異なるパッケージ内にある同じ名前の Java クラスのディレクトリー構造の例



注: Shape クラスの複数のバージョンをディレクトリー構造に格納することができます。 Shape クラスのベータ版を使用するには、CLASSPATH で、Shape クラスが格納されている他のディレクトリーや ZIP ファイルの前に /Beta/myclasses を指定します。

Java コンパイラーは、Java ソース・コードをコンパイルするときに、Java クラスパス、パッケージ名、およびディレクトリー構造を使ってパッケージとクラスを探します。詳しくは、 12 ページの『Java クラスパス』を参照してください。

IFS の Java 関連ファイル

統合ファイル・システム (IFS) には、Java 関連のクラス、ソース、ZIP、および JAR ファイルが、階層ファイル構造で格納されます。 IBM Developer Kit for Java は、IFS 内のスレッド・セーフ・ファイル・システムを使用して Java 関連のクラス・ファイル、ソース・ファイル、および JAR ファイルを格納および処理するための支援をします。

関連情報

マルチスレッド・プログラミングでのファイル・システムについての考慮事項

ファイル・システムの比較

統合ファイル・システム内の Java ファイル権限

Java プログラムを実行およびデバッグするには、クラス・ファイル、JAR ファイル、および ZIP ファイルに読み取り権限 (*R) が必要です。ディレクトリーには読み取りおよび実行の権限 (*RX) が必要です。

注: 実行権限 (*X) がないファイルとディレクトリーは、常に QSECOFR 権限があるユーザーに対する実行権限 (*X) があるように表示されます。ユーザーの両方が同じファイルに同じアクセスをしていても、特定の状況で、異なるユーザーが異なる結果を得ることがあるかもしれません。このことは、Qshell インタープリターまたは java.Runtime.exec() を使用してシェル・スクリプトを実行するときに知っておく必要があります。

たとえば、一人のユーザーがシェル・スクリプトを呼び出すために java.Runtime.exec() を使用する Java を作成して、それを、QSECOFR 権限のあるユーザー ID を使用してテストします。シェル・スクリプトのファイル・モードに読み取りおよび書き込み権限が (*RW) ある場合、統合ファイル・システムはそれを、QSECOFR 権限のあるユーザー ID が実行することを許可します。しかし、非 QSECOFR 権限ユーザーは同じ Java プログラムを実行しようとすることができますが、統合ファイル・システムは、*X が欠落しているため、java.Runtime.exec() コードにシェル・スクリプトは実行できないことを知らせることができるでしょう。この場合、java.Runtime.exec() は入出力の例外をスローします。

Java プログラムによって統合ファイル・システム内に作成された新規ファイルに対して権限を割り当てることもできます。ファイルは os400.file.create.auth システム・プロパティー、ディレクトリーは os400.dir.create.auth を使用して、読み取り、書き込み、および実行権限の組み合わせを設定することができます。

詳しくは、プログラムおよび CL コマンド APIまたは統合ファイル・システムを参照してください。

バッチ・ジョブで Java を実行する

「ジョブ投入 (SBMJOB)」コマンドを使うと、Java プログラムはバッチ・ジョブ内で実行します。このモードでは、Java Qshell コマンド入力画面を、System.in、System.out、System.err ストリームを処理するために使用することはできません。

これらのストリームは、他のファイルに転送することができます。デフォルトの処理では、System.out および System.err ストリームはスプール・ファイルに送信されます。 System.in からの読み取り要求で入出力例外を出すバッチ・ジョブは、スプール・ファイルを所有します。 Java プログラム内で System.in、System.out、および System.err の転送を行うことができます。また、os400.stdin、os400.stdout、および os400.stderr システム・プロパティーを使用して、System.in、System.out、および System.err を転送することもできます。

注: SBMJOB コマンドを実行すると、ユーザー・プロファイルで指定された HOME ディレクトリーが現行作業ディレクトリー (CWD) に設定されます。

例: バッチ・ジョブで Java を実行する

```
| SBMJOB CMD(JAVA QIBMHello OPTION(*VERBOSE)) CPYENVVAR(*YES)
```

上記の例で JAVA コマンドを実行すると、2 番目のジョブが作成されます。ですから、バッチ・ジョブが実行されるサブシステムは、複数のジョブを実行できなければなりません。

バッチ・ジョブが複数のジョブを実行できることを、以下のステップに従うことによって検証できます。

1. CL コマンド行で `DSPSBSD(MYSBSD)` と入力する。ここで、`MYSBSD` は、バッチ・ジョブのサブシステム記述を表します。
2. オプション 6 のジョブ待ち行列項目を選ぶ。
3. ジョブ待ち行列の Max Active フィールドを参照する。

GUI を使用しないホスト上で Java アプリケーションを実行する

Java アプリケーションを、IBM i サーバーなどのグラフィカル・ユーザー・インターフェース (GUI) のないホスト上で実行したい場合は、 Native Abstract Windowing Toolkit (NAWT) を使用することができます。

NAWT を使用して、Java アプリケーションおよびサーブレットに、Java 2 Platform, Standard Edition (J2SE) AWT のグラフィックス機能の全機能を提供することができます。

Native Abstract Windowing Toolkit

Native Abstract Windowing Toolkit (NAWT) は、実際のツールキットではなく、Java アプリケーションおよびサーブレットで Java 2 Platform, Standard Edition (J2SE) の Abstract Windowing Toolkit (AWT) グラフィック機能を使用できるようにするためのネイティブ IBM i サポートを表すために発達した用語です。

AWT の使用に必要な情報の大半は、以下のリンクから入手することができます。

- Abstract Window Toolkit
- Java Internationalization FAQ

AWT モードを選択する

AWT を使用する際は、通常モードとヘッドレス・モードの 2 つのモードが選択できます。どちらのモードを選択するかを決める 1 つの要素となるのは、Heavyweight AWT コンポーネントを使用する必要があるかどうか、という点です。

通常モード

Java AWT API を使用してウィンドウ、フレーム、ダイアログ・ボックス、または同様の Heavyweight コンポーネントを表示するアプリケーションでは、通常モードを使用する必要があります。アプリケーションでマウス操作イベントやキーボード入力の受信が予期される場合は、通常モードを使用してください。通常モードはデフォルト・モードであり、これを使用可能にするために何も指定する必要はありません。

ヘッドレス・モード

ヘッドレス・モードは、Java アプリケーションがユーザーと直接対話しない場合に使用できます。つまり、ウィンドウやダイアログ・ボックスが表示されず、キーボードやマウスによる入力がなく、Heavyweight AWT コンポーネントを使用しない Java アプリケーションの場合です。このモードは、Java の起動時に Java プロパティー *java.awt.headless=true* を指定することによって選択できます。ヘッドレス・モードを使用する場合は、VNC/X サーバーを使用する必要はありません。

ヘッドレス・モードを使用できるアプリケーションの例として、以下が挙げられます。

- リモート・ユーザーに戻されるデータ・ストリームに組み込むイメージを作成するためだけに AWT API を使用する、サーブレットまたは他のサーバー・ベースのプログラム
- Heavyweight AWT コンポーネントによる実際の表示は行わず、イメージまたはイメージ・ファイルの作成や操作のみを行うプログラム

Java プロパティー *java.awt.headless* のデフォルト値は false です。

Heavyweight AWT コンポーネント

以下の項目は、Heavyweight AWT コンポーネントと見なされます。これらの項目を必要とするアプリケーションでは、通常モードを使用してください。

表 8. Heavyweight AWT コンポーネント

Heavyweight AWT コンポーネント			
アプレット	フレーム	リスト	ロボット
ボタン	JApplet	メニュー	スクロール・バー
チェック・ボックス	JDialog	メニュー・バー	スクロール・ペイン
選択項目	JFrame	メニュー・コンポーネント	テキスト域
ダイアログ	JWindow	メニュー項目	テキスト・コンポーネント
ファイル・ダイアログ	ラベル	ポップアップ・メニュー	ウィンドウ

完全なグラフィカル・ユーザー・インターフェース・サポートを備えた通常モードで AWT を使用する:

グラフィカル・ユーザー・インターフェースをサポートするためには、ウィンドウ操作システムが必要です。IBM i Java でサポートされている選択肢は、Virtual Network Computing (VNC) サーバーです。VNC サーバーは専用のマウス、キーボード、およびグラフィックス機能付きモニターを必要としないため、このシステムに適しています。IBM は、PASE for i で稼働するバージョンの VNC サーバーを提供しています。以下の指示に従って、VNC をインストールおよび開始し、VNC を使用するように Java セッションを構成します。

AWT をテストまたは使用開始するためには、まず、以下の必須およびオプショナル・ステップを実行する必要があります。

- VNC パスワードを作成します。これは、VNC サーバーの開始に使用される各ユーザー・プロファイルにつき 1 回行う必要があります。
- VNC サーバーを開始します。通常は、各システム IPL の後に行います。
- AWT 環境変数を構成します。これは、各セッションで最初に Java を実行し、AWT API を使用する前に 1 回行います。
- Java システム・プロパティーを構成します。これは、Java を実行する度に行う必要があります。
- 対話式で使用する場合のオプショナル・ステップ: iceWM ウィンドウ・マネージャーを構成します。
- ユーザーと直接対話する場合のオプショナル・ステップ: VNC ビューアーまたは Web ブラウザーを使用して VNC に接続します。
- オプショナル・ステップ: ご使用の AWT 構成を検証します。

VNC パスワード・ファイルの作成:

Virtual Network Computing (VNC) サーバーで Native Abstract Windowing Toolkit (NAWT) を使用するためには、VNC パスワード・ファイルを作成する必要があります。

VNC サーバーのデフォルトの設定においては、無許可ユーザーのアクセスから VNC の表示を保護するためにこれが使用するパスワード・ファイルが必要です。 VNC パスワード・ファイルは、VNC サーバーを開始するために使用するプロファイルの下に作成する必要があります。 IBM iコマンド・プロンプトで以下を入力します。

1. `MKDIR DIR('/home/VNCprofile/.vnc')`
2. `QAPTL/VNCPASSWD USEHOME(*NO) PWDFILE('/home/VNCprofile/.vnc/passwd')` ここで *VNCprofile* は、VNC サーバーを開始したプロファイルを示します。

リモート・システムから VNCviewer または Web ブラウザーを使用して VNC サーバーに対話式にアクセスするには、このステップで指定したパスワードを使用する必要があります。

VNC サーバーの開始:

Virtual Network Computing (VNC) サーバーを開始するには、以下のステップを実行します。

n は使用するディスプレイ番号です。ディスプレイ番号は、1 から 99 の範囲の任意の整数にすることができます。

.Xauthority ファイル

VNC サーバーの開始プロセスでは、新規の .Xauthority ファイルが作成されるか、あるいは既存の .Xauthority ファイルが変更されます。 X サーバー権限は、暗号化されたキー情報を含む .Xauthority ファイルを使用して、他のユーザーのアプリケーションが X サーバー要求を傍受することがないようにします。 Java 仮想マシン (JVM) と VNC の間のセキュア通信のためには、JVM と VNC の両方が .Xauthority ファイル内の暗号化されたキー情報にアクセス**できなければなりません。**

.Xauthority ファイルは VNC を開始したプロファイルに属します。 JVM と VNC の両方が共に .Xauthority ファイルにアクセスできるようにするための最も容易な方法は、同一のユーザー・プロファイルで VNC サーバーと JVM を実行することです。 VNC サーバーと JVM の両方を同一のユーザー・プロファイルで実行できない場合は、XAUTHORITY 環境変数を構成して正しい .Xauthority ファイルを指し示すことができます。

Virtual Network Computing (VNC) サーバーを開始するには、コマンド行で `CALL PGM(QSYS/QP2SHELL) PARM('/QOpenSys/QIBM/ProdData/DeveloperTools/vnc/vncserver_java' ':n'))` というコマンドを入力し (*n* は使用するディスプレイ番号)、**ENTER** を押します。 ディスプレイ番号は、1 から 99 の範囲の任意の整数にすることができます。

VNC サーバーを開始すると、IBM i サーバーのシステム名とディスプレイ番号を示すメッセージ (例えば、`New 'X'desktop is systemname:1` など) が表示されます。このシステム名とディスプレイ番号は、後で AWT を使用する Java アプリケーションを実行する際に、DISPLAY 環境変数を構成するために必要になります。忘れないように書き留めておいてください。

同時に複数の VNC サーバーを実行する場合は、各 VNC サーバーごとに固有なディスプレイ番号が必要になります。ここで示すように、VNC サーバーを開始する際に明示的にディスプレイ値を指定するなら、各アプリケーションがどのディスプレイ番号を使用するかを制御できます。あるいは、ディスプレイ番号を指定したくない場合は、前述のコマンドから `':n'` を除去して vncserver_java プログラムに使用可能なディスプレイ番号を検出させ、そのディスプレイ番号を書き留めておくこともできます。

.Xauthority ファイル

VNC サーバーの開始プロセスでは、新規の .Xauthority ファイルが作成されるか、あるいはサーバーを開始するユーザーのホーム・ディレクトリーにある既存の .Xauthority ファイルが変更されます。.Xauthority ファイルには、X サーバー要求が他のユーザーのアプリケーションによって解釈されないようにするための、暗号鍵の権限情報が含まれています。Java 仮想マシン (JVM) と VNC の間で安全な通信を行うためには、JVM と VNC の両方が同じ .Xauthority ファイルにアクセス**できなければなりません**。

.Xauthority ファイルは VNC を開始したプロファイルに属します。 JVM と VNC サーバーの両方がアクセスを共用できるようにするためには、VNC サーバーと JVM を同じユーザー・プロファイルで実行します。これができない場合は、XAUTHORITY 環境変数を構成して、この環境変数が正しい .Xauthority ファイルを指すようにすることができます。

NAWT 環境変数の構成:

Java を実行する場合で、完全な AWT グラフィカル・ユーザー・インターフェースのサポートを希望する場合は、DISPLAY 環境変数と XAUTHORITY 環境変数が、使用する X サーバー・ディスプレイと正しい .Xauthority ファイルを検索する場所を Java に伝えるように定義されている必要があります。

DISPLAY 環境変数

Java プログラムを実行するセッションで、DISPLAY 環境変数をシステム名およびディスプレイ番号に設定します。 IBM i コマンド・プロンプトで以下のコマンドを入力して ENTER を押します。

```
ADDENVVAR ENVVAR(DISPLAY) VALUE('systemname:n')
```

ここで、*systemname* はシステムのホスト名または IP アドレスを示し、*n* は使用する VNC サーバーのディスプレイ番号を示します。

XAUTHORITY 環境変数

Java プログラムを実行するセッションの中で、XAUTHORITY 環境変数を `/home/VNCprofile/.Xauthority` に設定します。ここで、*VNCprofile* は VNC サーバーを開始したプロファイルを示します。IBM i コマンド・プロンプトからコマンド

```
ADDENVVAR ENVVAR(XAUTHORITY) VALUE('/home/VNCprofile/.Xauthority')
```

を実行します。なお、*VNCprofile* の部分は該当するプロファイル名に置き換えてください。

iceWM ウィンドウ・マネージャーの構成:

Virtual Network Computing (VNC) サーバーを対話式に使用したい場合は、NAWT をセットアップする際のオプショナル・ステップとして、iceWM ウィンドウ・マネージャー (iceWM) を構成してください。これには、たとえば、グラフィカル・ユーザー・インターフェース (GUI) を備えた Java アプリケーションを実行したい場合などがあります。 iceWM は、IBM i Tools For Developers PRPQ に組み込まれている、小さいけれども強力なウィンドウ・マネージャーです。

iceWM はバックグラウンドで実行して、VNC サーバーの X Window 環境内で実行しているウィンドウの外観を制御します。 iceWM は、多くのポピュラーなウィンドウ・マネージャーに似たインターフェースおよびフィーチャーのセットを提供します。組み込みの vncserver_java スクリプトのデフォルト動作では、VNC サーバーが開始され、iceWM が実行されます。

このステップを完了すると、iceWM が必要とするいくつかの構成ファイルが作成されます。 iceWM は必要に応じて使用不可にすることもできます。

iceWM の構成

iceWM ウィンドウ・マネージャーを構成するには、IBM i コマンド・プロンプトで以下のステップを実行してください。これらのステップは、必ず VNC サーバーを開始するために使用するプロファイルで実行してください。

1. 以下のコマンドを入力し、**ENTER** を押してインストールを開始します。

   ```
   STRPTL CLIENT(IGNORE)
   ```

   IGNORE 値は、NAWT が必要とする STRPTL の構成フィーチャーのみをコマンドが活動化するようにするためのプレースホルダーとして機能します。

2. 以下のコマンドを入力し、**ENTER** を押してサインオフします。

   ```
   SIGNOFF
   ```

   サインオフすると、STRPTL コマンドのセッション固有の結果が、NAWT を使用または構成するためにその後実行する操作に影響しなくなります。

注: STRPTL コマンドは、VNC サーバーを開始する各プロファイルごとに 1 回のみ実行してください。NAWT では、コマンドの使用可能なオプションの引数はいずれも必須ではありません。この記述は、5799-PTL IBM i Tools For Developers PRPQ に関連した STRPTL のセットアップ手順に優先します。

iceWM を使用不可にする

VNC サーバーを開始すると、iceWM を実行するためのコマンドが含まれた、xstartup_java というスクリプト・ファイルが作成されるか、あるいはこの名前の既存のファイルが変更されます。 xstartup_java スクリプト・ファイルは、以下の統合ファイル・システム・ディレクトリーにあります。

```
/home/VNCprofile/.vnc/
```

VNCprofile は VNC サーバーを開始したプロファイルの名前です。

iceWM を完全に使用不可にする場合は、テキスト・エディターを使用して、iceWM を開始するスクリプトの中の行をコメント化するか、あるいは除去してください。行をコメント化するには、行の先頭にポンド記号 (#) を挿入します。

VNCviewer または Web ブラウザーを使用する:

IBM i サーバーでグラフィカル・ユーザー・インターフェース (GUI) を備えたアプリケーションを実行する場合は、VNCviewer または Web ブラウザーを使用して Virtual Network Computing (VNC) サーバーに接続しなければなりません。VNCviewer または Web ブラウザーは、パーソナル・コンピューターなどの、グラフィックスを処理できるプラットフォームで実行する必要があります。

注: 以下のステップを実行するには、ディスプレイ番号と VNC パスワードが分かっていなければなりません。Virtual Network Computing (VNC) サーバーを開始すると、ディスプレイ番号の値が決まります。VNC パスワード・ファイルを作成すると、VNC パスワードが設定されます。

VNCviewer を使用した VNC サーバーへのアクセス

VNCviewer を使用して VNC サーバーに接続するには、以下のステップを実行します。

1. VNCviewer アプリケーションをダウンロードしてインストールします。
   - ほとんどのプラットフォーム用の VNCviewer は、RealVNC Web サイトから入手できます。
2. ダウンロードした VNCviewer を開始します。プロンプトで、システム名とディスプレイ番号を入力し、「**OK**」をクリックします。

3. パスワード・プロンプトで、VNC パスワードを入力して VNC サーバー・ディスプレイにアクセスします。

Web ブラウザーを使用した VNC サーバーへのアクセス

Web ブラウザー を使用して VNC サーバーに接続するには、以下のステップを実行します。

1. ブラウザーを開始し、以下の URL にアクセスします。

   ```
   http://systemname:58nn
   ```

   ここで、各パラメーターは次のように定義されます。

   - *systemname* は、VNC サーバーを実行しているシステムの名前または IP アドレスです。
   - *nn* は 2 桁表示の VNC サーバー・ディスプレイ番号です。

     たとえば、システム名が system_one で、ディスプレイ番号が 2 の場合、URL は次のようになります。

     ```
     http://system_one:5802
     ```

2. URL に正常にアクセスすると、VNC サーバー・パスワードを求めるプロンプトが表示されます。パスワード・プロンプトで、VNC パスワードを入力して VNC サーバー・ディスプレイにアクセスします。

VNC の使用上のヒント:

IBM i 制御言語 (CL) コマンドを使用して Virtual Network Computing (VNC) サーバーを開始および停止し、現行で実行している VNC サーバーに関する情報を表示します。

CL プログラムからの VNC ディスプレイ・サーバーの開始

以下の例は、DISPLAY 環境変数を設定し、制御言語 (CL) コマンドを使用して自動的に VNC を開始するための 1 つの方法を示しています。

```
CALL QP2SHELL PARM('/QOpenSys/QIBM/ProdData/DeveloperTools/vnc/vncserver_java' ':n')
ADDENVVAR ENVVAR(DISPLAY) VALUE('systemname:n')
```

ここで、各パラメーターは次のように定義されます。

- *systemname* は、VNC が実行されるシステムのホスト名または IP アドレスです。
- ここで、*n* は開始したいディスプレイ番号を表す数値です。

注: この例では、まだディスプレイ *n* を実行しておらず、必要な VNC パスワードを正常に作成してあることを想定しています。パスワード・ファイルの作成について詳しくは、VNC パスワード・ファイルの作成を参照してください。

CL プログラムからの VNC ディスプレイ・サーバーの停止

以下のコードは、CL プログラムから VNC サーバーを停止するための 1 つの方法を表しています。

```
CALL QP2SHELL PARM('/QOpenSys/QIBM/ProdData/DeveloperTools/vnc/vncserver_java' '-kill' ':n')
```

n は、終了するディスプレイ番号を表す数値です。

実行中の VNC ディスプレイ・サーバーの検査

現行でどの (存在する場合) VNC サーバーがシステム上で実行しているのかを判別するには、以下のステップを実行してください。

1. IBM i コマンド行から PASE for i シェルを開始します。

```
CALL QP2TERM
```

2. PASE for i シェル・プロンプトから、 ps コマンドを使用して VNC サーバーをリストします。

```
ps gaxuw | grep Xvnc
```

このコマンドの結果の出力では、実行中の VNC サーバーが以下のフォーマットで表示されます。

```
john  418  0.9  0.0  5020  0  - A  Jan 31 222:26
  /QOpenSys/QIBM/ProdData/DeveloperTools/vnc/Xvnc :1 -desktop X -httpd
jane   96  0.2  0.0   384  0  - A  Jan 30 83:54
  /QOpenSys/QIBM/ProdData/DeveloperTools/vnc/Xvnc :2 -desktop X -httpd
```

ここで、

- 最初の列は、サーバーを開始したプロファイルです。
- 2 番目の列はサーバーのプロセス ID です。
- */QOpensys/* で始まる情報は、VNC サーバーを開始したコマンド (引数を含む) です。通常ディスプレイ番号は、Xvnc コマンドの引数リストの中の最初の項目です。

注: 上記の出力例で表されている Xvnc プロセスは、実際の VNC サーバー・プログラムの名前です。vncserver_java スクリプトを実行する際は Xvnc を開始してください。これは Xvnc のための環境とパラメーターを準備してから、Xvnc を開始します。

AWT を WebSphere Application Server と共に使用するためのヒント:

WebSphere ベースのアプリケーションをヘッドレス・モードではなくフル GUI モードで実行する必要がある場合は、ここで説明する、WebSphere と VNC サーバーの接続に伴う問題を避けるためのヒントを活用してください。

セキュア通信の確保

VNC サーバーは、WebSphere などの使用するアプリケーションとの接続が安全であることを確認するために、X 権限検査というメソッドを使用します。

VNC サーバーの開始プロセスでは、暗号化されたキー情報を含む .Xauthority ファイルが作成されます。WebSphere Application Server が VNC にアクセスするためには、VNC サーバーが使用しているのと同じ .Xauthority ファイルにアクセスし、同じファイルを使用することが**必要** です。

これを実現するためには、次のいずれかの方法を使用します。

同一のプロファイルを使用して WebSphere Application Server と VNC を実行する

WebSphere Application Server と VNC サーバーの両方を同じユーザー・プロファイルを使用して開始すると、両者はデフォルトで同じ .Xauthority ファイルを使用します。このためには、WebSphere のデフォルト・ユーザー (QEJBSVR) から VNC サーバーを開始するか、WebSphere のデフォルト・ユーザーを VNC サーバーの開始に使用するプロファイルに変更する必要があります。

アプリケーション・サーバーのユーザー・プロファイルを、デフォルトのユーザー (QEJBSVR) から別のプロファイルに切り替えるには、以下の操作を行う必要があります。

1. WebSphere Application Server 管理コンソールを使用して、アプリケーション・サーバーの構成を変更します。
2. System i Navigatorを使用して新しいプロファイルを使用可能にします

別のプロファイルを使用して WebSphere Application Server と VNC を実行する

この場合は、ある特定のユーザー・プロファイルで WebSphere Application Server を開始し、.Xauthority ファイルが別のユーザー・プロファイルによって所有されるようにします。WebSphere Application Server が VNC サーバーを開始できるようにするには、以下のステップを実行します。

1. 希望するユーザー・プロファイルから VNC サーバーを開始して、新規の .Xauthority ファイルを作成 (または既存の .Xauthority ファイルを更新) します。例えば、IBM i 制御言語 (CL) コマンド行から以下のコマンドを入力して ENTER を押します。

   ```
   CALL QP2SHELL PARM('/QOpenSys/QIBM/ProdData/DeveloperTools/vnc/vncserver_java' ':n')
   ```

   n はディスプレイ番号 (1 から 99 の範囲の数値) です。

   注: .Xauthority ファイルは、VNC サーバーを実行しているプロファイルのディレクトリーにあります。

2. 以下の CL コマンドを使用して、WebSphere Application Server を実行しているプロファイルに対し、.Xauthority ファイルの読み取り権限を付与します。

   ```
   CHGAUT OBJ('/home') USER(WASprofile) DTAAUT(*RX)
   CHGAUT OBJ('/home/VNCprofile') USER(WASprofile) DTAAUT(*RX)
   CHGAUT OBJ('/home/VNCprofile/.Xauthority') USER(WASprofile) DTAAUT(*R)
   ```

   VNCprofile および *WASprofile* は、VNC サーバーと WebSphere Application Server を実行している該当のプロファイルです。

   注: これらのステップに従う必要があるのは、*VNCprofile* と *WASprofile* とが異なるプロファイルである場合だけです。 *VNCprofile* と *WASprofile* とが同じプロファイルの場合にこれらのステップに従うと、VNC が正しく機能しない原因となる場合があります。

3. WebSphere Application Server 管理コンソールから、アプリケーション用の DISPLAY および XAUTHORITY 環境変数を定義します。
   - DISPLAY に対しては、`system:n` または `localhost:n` を使用してください。

     system はシステムの名前または IP アドレスであり、*n* は VNC サーバーを開始するために使用したディスプレイ番号です。
   - XAUTHORITY に対しては、`/home/VNCprofile/.Xauthority` を使用してください。

     VNCprofile は VNC サーバーを開始したプロファイルです。
4. WebSphere Application Server を再始動して、構成の変更をピックアップします。

   マネージメント・セントラルによるユーザーおよびグループの管理

   WebSphere Application Server for IBM i

AWT 構成の検査:

Java テスト・プログラムを実行することによって、ご使用の AWT 構成を検査できます。

IBM i コマンド行からテスト・プログラムを実行する場合は、テストするモードに合わせて以下のいずれかのコマンドを入力します。

```
JAVA CLASS(NAWTtest) CLASSPATH('/QIBM/ProdData/Java400')
```

または

```
JAVA CLASS (NAWTtest) CLASSPATH('/QIBM/ProdData/Java400') PROP((java.awt.headless true))
```

テスト・プログラムは JPEG でコード化されたイメージを作成し、それを統合ファイル・システム内の以下のパスに保管します。

```
/tmp/NAWTtest.jpg
```

テスト・プログラムを実行したら、テスト・プログラムによってこのファイルが作成されており、Java 例外が生成されていないことを確認してください。イメージを表示する場合は、バイナリー・モードを使用して、グラフィックスを処理できるシステムにイメージ・ファイルをアップロードし、それをブラウザーやペイント・プログラム、その他の同様のツールで表示します。

Java セキュリティー

このトピックでは、借用権限について詳述し、SSL を使用して Java アプリケーション中のソケット・ストリームを保護する方法を説明します。

Java アプリケーションは、IBM i プラットフォーム上の他のすべてのプログラムと同じセキュリティー上の制限を受けます。Java プログラムを IBM i サーバー上で実行するには、統合ファイル・システム内のクラス・ファイルに対する権限が必要です。プログラムが開始されると、それはユーザーの権限の下で実行されます。

IBM i サーバー上で実行する Java プログラムのほとんどはアプレットではなくアプリケーションなので、"sandbox" セキュリティー・モデルによる制限を受けません。

注: JAAS、JCE、JGSS、および JSSE は基本 JDK の一部で、拡張とは見なされません。

IBM i 7.1 での借用権限に関する変更点

| Java プログラムを通してのユーザー・プロファイル権限の借用は、IBM i 7.1 ではサポートされません。
| このトピックでは、ご使用のアプリケーションが借用権限を使用しているかどうかを判別する方法、および
| この変更に合わせてアプリケーションを変更する方法について説明します。

| IBM i 7.1 では、Java アプリケーションは、Java プログラムを通してユーザー・プロファイル権限を借用
| することができなくなります。

これらのトピックでは、Java 借用権限が使用されるいくつかの一般的な状況と、Java 借用権限の依存関係を除去するために Java アプリケーションをどのように変更できるかについて説明します。

アプリケーションが借用権限を使用しているかどうかを判別する

借用権限に対する変更の影響を受ける Java アプリケーションがあるかどうかを判別するのに役立つ、IBM i 5.3、5.4、および 6.1 のツールが使用できます。このツールは JVM と連動して、JVM が借用権限の使用をログに記録した Java プログラムを報告します。このツールは以下の PTF から入手可能です。

- **IBM i 5.3**

  このツールは PTF SI27769 で提供されています。JVM のログ機能が Java Group PTF SF99269、レベル 15 によって提供されており、JDK 5.0 を使用する Java プログラムに限定されています。

- **IBM i 5.4**

  このツールは PTF SI27772 で提供されています。JVM のログ機能はすべての JDK に対して使用可能で、追加の PTF は必要ありません。

- **IBM i 6.1**

追加の PTF は必要ありません。 JVM のログ機能はすべての JDK に対して使用可能です。

このツールは、借用権限のサポートを使用して作られた Java プログラムのシステムをスキャンして、借用権限に依存している可能性のある Java アプリケーションの識別も助けます。

デフォルトで、このツールは JVM のログに記録された借用権限の使用を表示し、さらにシステム全体をスキャンします。ただし、このツールはいくつかの Qshell オプションもサポートしています。

```
usage: /qsys.lib/qjava.lib/qjvaadptl.pgm [option]...
        Valid options include
          -h            : Show this usage statement.
          -o <file>      : Write output to the specified file.
          -d <directory> : Scan only the specified directory tree.
          -noscan        : Do not scan system. Report only logged uses.
```

ツールからの出力は、システム上のどの Java アプリケーションが借用権限を使用しているかを判別する助けになります。この情報を使用して、以下を行う必要があります。

- usage が購入したコードの中にある場合は、借用権限に関連した計画をベンダーに問い合わせてください。
- usage がユーザーのコードの中にある場合は、この資料に概説されている解決策をすべて読み、借用権限を使用しないようにコードを変更する意思と能力が自分にあるかどうかを確認してください。

借用権限の使用

借用権限は IBM i オブジェクトやデータベース・レコードに対する操作の実行、またはネイティブ・メソッドへのアクセスでのみ役に立つため、このトピック・コレクションの例はこれらの分野に焦点を当てています。借用権限についての基本的な説明は、「機密保護解説書」の『所有者の権限を借用するオブジェクト』のトピックを参照してください。

借用権限を使用することで、メソッドは、いくつかの操作を遂行するために、プログラムを実行したユーザーの権限の代わりにプログラム所有者の権限を借用することができます。概念的に、これは UNIX の Set UID や Set GID、およびその標準的な使用の例である Change Password ((UNIX に組み込まれている) と非常によく似ています。それぞれのユーザーがパスワード・ファイルを変更する権限を持つようにするのは得策ではありませんが、プログラムを信頼してプログラムがそのユーザーのパスワードを変更できるようにしておくなら、そのプログラムは機能しやすくなります。

System i では、JVM がシステム・ライセンス内部コード (SLIC) の承認コンピューティング・ベースの一部であったため、Java プログラムに借用権限のフィーチャーを提供することができました。IBM i では、Java のインプリメンテーションが変更され、JVM がユーザー・レベルのプログラムとなったため、Java はこのフィーチャーを提供することができなくなりました。

同様の機能を実現する最も一般的なアプローチは、同等のユーザー・プロファイル権限を借用して必要な操作を実行する Java プログラムに、ILE ネイティブ・メソッドを追加する方法です。また、ネイティブ・メソッドを追加しなくても、より大きな権限を持つ別のプロセスで機能を実行し、必要に応じてそのプログラムに要 求を送信することによって、借用権限と同様の効果を 実現することも できます。

例: 借用権限の代替策

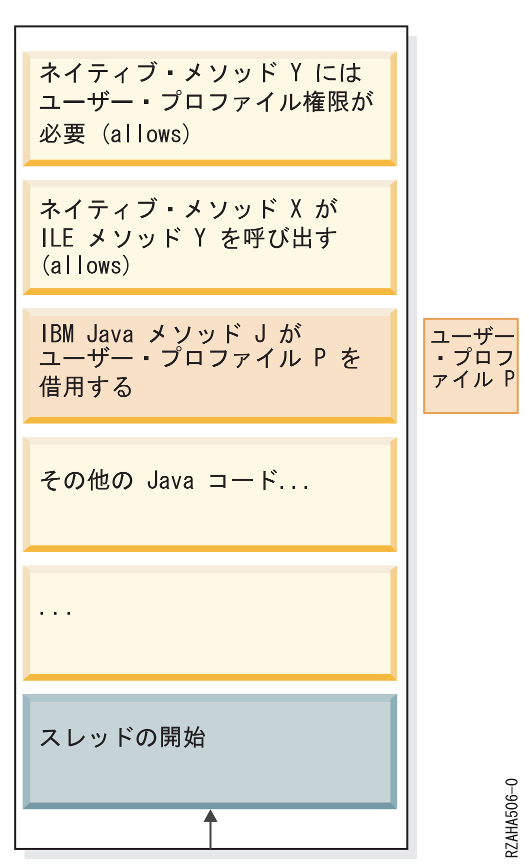
このトピックでは、いくつかの Java 借用権限の使用例と、提案されているいくつかの代替策を取り上げます。これらの代替策では、ILE サービス・プログラムのネイティブ・メソッドを使用することにより、Java の例と似た手法で権限を借用します。

特定の環境においては、他の代替策が可能であり、その方法の方が優れている場合もあります。1 つには、追加の権限を獲得するためにプロセスのユーザー・プロファイルをスワップする方法も使用できる場合があります。このトピックでは、ユーザー・プロファイルのスワッピングについては説明しません。この方法には、方法そのものに一連の問題とリスクが存在します。このトピック・コレクションの例では、Java 借用権限が使用される 2 つの一般的な事例について説明し、それに対して使用可能な代替策を説明します。

- 『例 1: ネイティブ・メソッドを呼び出す直前に権限を借用する Java メソッド』
- 255 ページの『代替策 1A: ネイティブ・メソッド X の再パッケージ化』
- 257 ページの『代替策 1B: 新しいネイティブ・メソッド N』
- 259 ページの『例 2: ネイティブ・メソッドを呼び出す前に権限を借用し、他の Java メソッドを呼び出す Java メソッド』
- 261 ページの『代替策 2: 新しいネイティブ・メソッド N』
- 263 ページの『サンプル・コマンドのコンパイル』

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

例 1: ネイティブ・メソッドを呼び出す直前に権限を借用する Java メソッド



この例では、Java プログラムに IBM Java メソッド **J** が含まれており、このメソッドがユーザー・プロファイル **P** を借用してネイティブ・メソッド **X** を直接呼び出します。ネイティブ・メソッド **X** は ILE メソッド **Y** を呼び出します。この ILE メソッドには借用権限が必要です。

JA61Example1.java

```
public class JA61Example1 {
    public static void main(String args[]) {
 int returnVal = J();
 if (returnVal > 0) {
     System.out.println("Adopted authority successfully.");
 }
 else {
     System.out.println("ERROR: Unable to adopt authority.");
 }
    }

    static int J() {
 return X();
    }

    // Returns: 1 if able to successfully access *DTAARA JADOPT61/DATAAREA
    static native int X();

    static {
 System.loadLibrary("EX1");
    }
}
```

JA61Example1.h

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class JA61Example1 */

#ifndef _Included_JA61Example1
#define _Included_JA61Example1
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     JA61Example1
 * Method:    X
 * Signature: ()I
 */
JNIEXPORT jint JNICALL Java_JA61Example1_X
  (JNIEnv *, jclass);

#ifdef __cplusplus
}
#endif
#endif
```

JA61Example1.c

```
/* This contains the source code for native method Java_JA61Example1_X. This
   module is bound into service program JADOPT61/EX1. */

#include "JA61Example1.h"
#include "JA61ModuleY.h"

/*
 * Class:     JA61Example1
 * Method:    X
 * Signature: ()I
 */
JNIEXPORT jint JNICALL Java_JA61Example1_X(JNIEnv* env, jclass klass) {
   return methodY();
}
```

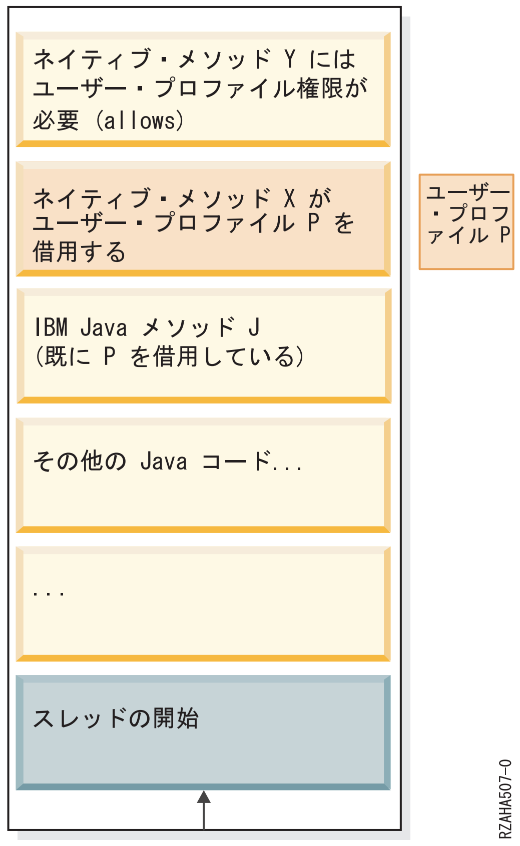
JA61ModuleY.h

```
/* This method tries to change *DTAARA JADOPT61/DATAAREA. This
   will only be possible if the JADOPT61UP user profile has been adopted. */
int methodY(void);
```

JA61ModuleY.c

```
#include <except.h>
#include <stdio.h>
#include "JA61ModuleY.h"
#include <xxdtaa.h>

#define START 1
#define LENGTH 8


/* This method tries to operate on *DTAARA JADOPT61/DATAAREA. This
 will only be possible if the JADOPT61UP user profile has been adopted. */
int methodY(void) {
   int returnValue;
   volatile int com_area;
   char newdata[LENGTH] = "new data";
   _DTAA_NAME_T dtaname = {"DATAAREA  ", "JADOPT61  "};

   /* Monitor for exception in this range */
#pragma exception_handler(ChangeFailed, 0, _C1_ALL, _C2_MH_ESCAPE)
   /* change the *DTAARA JADOPT61/DATAAREA */
   QXXCHGDA(dtaname, START, LENGTH, newdata);
#pragma disable_handler

ChangeCompleted:
   printf("Successfully updated data area\n");
   returnValue = 1;
   goto TestComplete;

ChangeFailed:      /* Control goes here for an exception */
   printf("Got an exception.\n");
   returnValue = 0;

TestComplete:
   printf("methodY completed\n");
   return returnValue;
}
```

代替策 1A: ネイティブ・メソッド X の再パッケージ化



権限の借用を保持する 1 つの方法は、ネイティブ・メソッド **X** を新しいサービス・プログラムに分割します。次いで、この新しいサービス・プログラムは、Java メソッド **J** がその前に借用したユーザー・プロファイル **P** を借用することができます。

JA61Alternative1A.java

```
public class JA61Alternative1A {
    public static void main(String args[]) {
 int returnVal = J();
 if (returnVal > 0) {
     System.out.println("Adopted authority successfully.");
 }
 else {
     System.out.println("ERROR: Unable to adopt authority.");
 }
    }

    static int J() {
 return X();
    }

    // Returns: 1 if able to successfully access *DTAARA JADOPT61/DATAAREA
    static native int X();
```

```
    static {
 System.loadLibrary("ALT1A");
    }
}
```

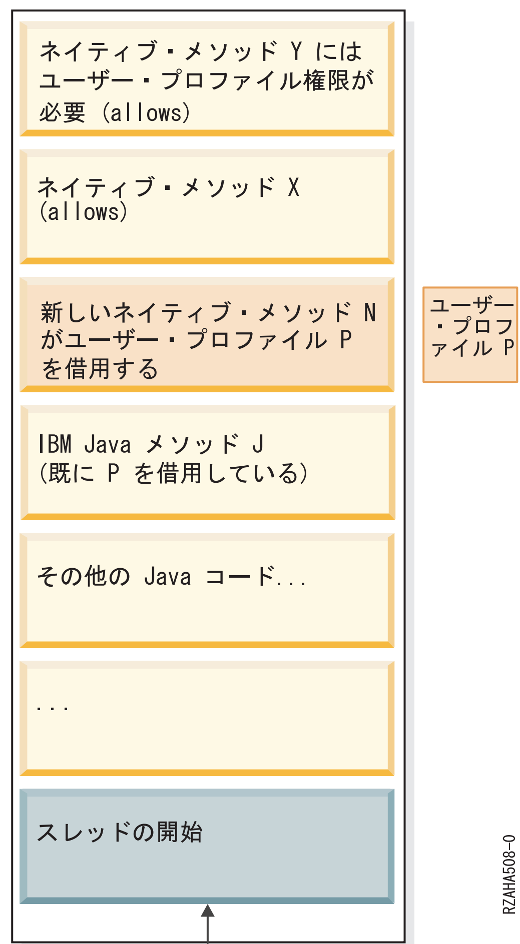
JA61Alternative1A.h

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class JA61Alternative1A */

#ifndef _Included_JA61Alternative1A
#define _Included_JA61Alternative1A
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     JA61Alternative1A
 * Method:    X
 * Signature: ()I
 */
JNIEXPORT jint JNICALL Java_JA61Alternative1A_X
  (JNIEnv *, jclass);

#ifdef __cplusplus
}
#endif
#endif
```

JA61Alternative1A.c

```
/* This contains the source code for native method Java_JA61Alternative1A_X. This
   module is bound into service program JADOPT61/ALT1A.*/

#include "JA61Alternative1A.h"
#include "JA61ModuleY.h"

/*
 * Class:     JA61Alternative1A
 * Method:    X
 * Signature: ()I
 */
JNIEXPORT jint JNICALL Java_JA61Alternative1A_X(JNIEnv* env, jclass klass) {
   return methodY();
}
```

代替策 1B: 新しいネイティブ・メソッド N



ユーザー・プロファイルの借用を保持する別の方法は、ユーザー・プロファイル **P** を借用するサービス・プログラムの中に全く新しいネイティブ・メソッド **N** を作成します。この新しいメソッドは Java メソッド **J** で呼び出されて、ネイティブ・メソッド **X** を呼び出します。Java メソッド**J** では **X** の代わりに **N** を呼び出すように変更が必要になりますが、ネイティブ・メソッド **X** では変更や再パッケージ化は必要ありません。

JA61Alternative1B.java

```
public class JA61Alternative1B {
    public static void main(String args[]) {
 int returnVal = J();
 if (returnVal > 0) {
     System.out.println("Adopted authority successfully.");
 }
 else {
     System.out.println("ERROR: Unable to adopt authority.");
 }
    }

    static int J() {
 return N();
    }
```

```
    // Returns: 1 if able to successfully access *DTAARA JADOPT61/DATAAREA
    static native int N();

    static {
 System.loadLibrary("ALT1B");
    }
}
```

JA61Alternative1B.h

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class JA61Alternative1B */

#ifndef _Included_JA61Alternative1B
#define _Included_JA61Alternative1B
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     JA61Alternative1B
 * Method:    N
 * Signature: ()I
 */
JNIEXPORT jint JNICALL Java_JA61Alternative1B_N
  (JNIEnv *, jclass);

#ifdef __cplusplus
}
#endif
#endif
```

JA61Alternative1B.c

```
/* This contains the source code for native method Java_JA61Alternative1B_N. This
   module is bound into service program JADOPT61/ALT1B. */

#include "JA61Alternative1B.h"
#include "JA61Example1.h"

/*
 * Class:     JA61Alternative1B
 * Method:    N
 * Signature: ()I
 */
JNIEXPORT jint JNICALL Java_JA61Alternative1B_N(JNIEnv* env, jclass klass) {
   return Java_JA61Example1_X(env, klass);  /* from JA61Example1.h */
}
```

例 2: ネイティブ・メソッドを呼び出す前に権限を借用し、他の Java メソッドを呼び出す Java メソッド

IBM Java メソッド **J1** は、ユーザー・プロファイル **P** を借用する Java プログラムの中にあります。**J1** は Java メソッド **J2** を呼び出し、**J2** は **J3** を呼び出します。次いで **J3** はネイティブ・メソッド **X** を呼び出します。ネイティブ・メソッド **X** は ILE メソッド **Y** を呼び出します。この ILE メソッドには借用権限が必要です。

JA61Example2.java

```
public class JA61Example2 {
    public static void main(String args[]) {
 int returnVal = J1();
 if (returnVal > 0) {
     System.out.println("Adopted authority successfully.");
 }
 else {
     System.out.println("ERROR: Unable to adopt authority.");
```

```
 }
    }

    static int J1() {
 return JA61Example2Allow.J2();
    }

}
```

JA61Example2Allow.java

```
public class JA61Example2Allow {
    public static int J2() {
 return J3();
    }

    static int J3() {
 return X();
    }

    // Returns: 1 if able to successfully access *DTAARA JADOPT61/DATAAREA
    static native int X();

    static {
 System.loadLibrary("EX2ALLOW");
    }
}
```

JA61Example2Allow.h

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class JA61Example2Allow */

#ifndef _Included_JA61Example2Allow
#define _Included_JA61Example2Allow
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     JA61Example2Allow
 * Method:    X
 * Signature: ()I
 */
JNIEXPORT jint JNICALL Java_JA61Example2Allow_X
  (JNIEnv *, jclass);

#ifdef __cplusplus
}
#endif
#endif
```

JA61Example2Allow.c

```
/* This contains the source code for native method Java_JA61Example2Allow_X. This
   module is bound into service program JADOPT61/EX2ALLOW. */

#include "JA61Example2Allow.h"
#include "JA61ModuleY.h"

/*
 * Class:     JA61Example2Allow
 * Method:    X
 * Signature: ()I
```

```
 */
JNIEXPORT jint JNICALL Java_JA61Example2Allow_X(JNIEnv* env, jclass klass) {
   return methodY();
}
```

代替策 2: 新しいネイティブ・メソッド N

このケースで借用権限を保持するためには、新しいネイティブ・メソッド **N** を作成できます。このネイティブ・メソッドは、ユーザー・プロファイル **P** を借用するサービス・プログラムの中に置かれます。

次いで、ネイティブ・メソッド **N** は JNI を使用して Java メソッド **J2** を呼び出します。このメソッドは変更されていません。Java メソッド **J1** では、Java メソッド **J2** の代わりにネイティブ・メソッド **N** を呼び出すように変更が必要です。

JA61Alternative2.java

```
public class JA61Alternative2 {
    public static void main(String args[]) {
 int returnVal = J1();
 if (returnVal > 0) {
     System.out.println("Adopted authority successfully.");
 }
 else {
     System.out.println("ERROR: Unable to adopt authority.");
 }
    }

    static native int N();

    static int J1() {
 return N();
    }

    static {
 System.loadLibrary("ALT2");
    }

}
```

JA61Alternative2.h

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class JA61Alternative2 */

#ifndef _Included_JA61Alternative2
#define _Included_JA61Alternative2
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     JA61Alternative2
 * Method:    N
 * Signature: ()I
 */
JNIEXPORT jint JNICALL Java_JA61Alternative2_N
  (JNIEnv *, jclass);

#ifdef __cplusplus
}
#endif
#endif
```

JA61Alternative2.C

```
include "JA61Alternative2.h"

/*
 * Class:     JA61Alternative2
 * Method:    N
 * Signature: ()I
 */
JNIEXPORT jint JNICALL Java_JA61Alternative2_N(JNIEnv* env, jclass klass) {
#pragma convert(819)
    char* className = "JA61Example2Allow";
    char* methodName = "J2";
    char* methodSig = "()I";
#pragma convert(0)

    // Locate class JA61Example2Allow
    jclass cls = env->FindClass(className);
```

```
    // Get the method id for J2()I and call it.
    jmethodID methodID = env->GetStaticMethodID(cls, methodName, methodSig);
    int result = env->CallStaticIntMethod(cls, methodID);
    return result;
}
```

サンプル・コマンドのコンパイル

以下の指示はすべて、IBM i サーバー上の /home/javatests/adoptup/v6r1mig というディレクトリーにあるソース・コードに基づいています。

Qshell で、次のようにします。

```
> cd /home/javatests/adoptup/v6r1mig > javac -g *.java
```

CL から、次のようにします。

```
> CRTLIB JADOPT61
> CRTUSRPRF USRPRF(JADOPT61UP) STATUS(*DISABLED)
> CRTUSRPRF USRPRF(JADOPT61) PASSWORD(j61adopt) INLPGM(QSYS/QCMD) SPCAUT(*NONE)

> CRTDTAARA DTAARA(JADOPT61/DATAAREA) TYPE(*CHAR) LEN(50) VALUE('Initial value')
> GRTOBJAUT OBJ(JADOPT61/DATAAREA) OBJTYPE(*DTAARA) USER(JADOPT61UP) AUT(*ALL)
> RVKOBJAUT OBJ(JADOPT61/DATAAREA) OBJTYPE(*DTAARA) USER(*PUBLIC) AUT(*ALL)
> RVKOBJAUT OBJ(JADOPT61/DATAAREA) OBJTYPE(*DTAARA) USER(YOUR_USER_ID) AUT(*ALL)
```

すべての例で使用される SRVPGMY を作成します。

```
> CRTCMOD MODULE(JADOPT61/MODULEY) SRCSTMF('/home/javatests/adoptup/v6r1mig/JA61ModuleY.c')
DBGVIEW(*ALL)
> CRTSRVPGM SRVPGM(JADOPT61/SRVPGMY) MODULE(JADOPT61/MODULEY) EXPORT(*ALL)
```

252 ページの『例 1: ネイティブ・メソッドを呼び出す直前に権限を借用する Java メソッド』を作成します。

```
> CRTCMOD MODULE(JADOPT61/EX1) SRCSTMF('/home/javatests/adoptup/v6r1mig/JA61Example1.c')
INCDIR('/home/javatests/adoptup/v6r1mig') DBGVIEW(*ALL)
> CRTSRVPGM SRVPGM(JADOPT61/EX1) EXPORT(*ALL) BNDSRVPGM(JADOPT61/SRVPGMY)
> QSH CMD('chown JADOPT61UP /home/javatests/adoptup/v6r1mig/JA61Example1.class')
> CRTJVAPGM CLSF('/home/javatests/adoptup/v6r1mig/JA61Example1.class') USRPRF(*OWNER)
```

255 ページの『代替策 1A: ネイティブ・メソッド X の再パッケージ化』を作成します。

```
> CRTCMOD MODULE(JADOPT61/ALT1A) SRCSTMF('/home/javatests/adoptup/v6r1mig/JA61Alternative1A.c')
INCDIR('/home/javatests/adoptup/v6r1mig') DBGVIEW(*ALL)
> CRTSRVPGM SRVPGM(JADOPT61/ALT1A) EXPORT(*ALL) BNDSRVPGM(JADOPT61/SRVPGMY) USRPRF(*OWNER)
> CHGOBJOWN OBJ(JADOPT61/ALT1A) OBJTYPE(*SRVPGM) NEWOWN(JADOPT61UP)
> CRTJVAPGM CLSF('/home/javatests/adoptup/v6r1mig/JA61Alternative1A.class')
```

257 ページの『代替策 1B: 新しいネイティブ・メソッド N』を作成します。

```
> CRTCMOD MODULE(JADOPT61/ALT1B) SRCSTMF('/home/javatests/adoptup/v6r1mig/JA61Alternative1B.c')
INCDIR('/home/javatests/adoptup/v6r1mig') DBGVIEW(*ALL)
> CRTSRVPGM SRVPGM(JADOPT61/ALT1B) EXPORT(*ALL) BNDSRVPGM(JADOPT61/EX1) USRPRF(*OWNER)
> CHGOBJOWN OBJ(JADOPT61/ALT1B) OBJTYPE(*SRVPGM) NEWOWN(JADOPT61UP)
> CRTJVAPGM CLSF('/home/javatests/adoptup/v6r1mig/JA61Alternative1B.class')
```

259 ページの『例 2: ネイティブ・メソッドを呼び出す前に権限を借用し、他の Java メソッドを呼び出す Java メソッド』を作成します。

```
> CRTCMOD MODULE(JADOPT61/EX2ALLOW) SRCSTMF('/home/javatests/adoptup/v6r1mig/JA61Example2Allow.c')
INCDIR('/home/javatests/adoptup/v6r1mig') DBGVIEW(*ALL)
> CRTSRVPGM SRVPGM(JADOPT61/EX2ALLOW) EXPORT(*ALL) BNDSRVPGM(JADOPT61/SRVPGMY)
> QSH CMD('chown JADOPT61UP /home/javatests/adoptup/v6r1mig/JA61Example2.class')
> CRTJVAPGM CLSF('/home/javatests/adoptup/v6r1mig/JA61Example2.class') USRPRF(*OWNER)
> CRTJVAPGM CLSF('/home/javatests/adoptup/v6r1mig/JA61Example2Allow.class') USEADPAUT(*YES)
```

261 ページの『代替策 2: 新しいネイティブ・メソッド N』を作成します。

```
> CRTCPPMOD MODULE(JADOPT61/ALT2) SRCSTMF('/home/javatests/adoptup/v6r1mig/JA61Alternative2.C')
INCDIR('/home/javatests/adoptup/v6r1mig') DBGVIEW(*ALL)
> CRTSRVPGM SRVPGM(JADOPT61/ALT2) EXPORT(*ALL) USRPRF(*OWNER)
> CHGOBJOWN OBJ(JADOPT61/ALT2) OBJTYPE(*SRVPGM) NEWOWN(JADOPT61UP)
> CRTJVAPGM CLSF('/home/javatests/adoptup/v6r1mig/JA61Alternative2.class')
```

サンプルを実行するには、以下のステップを実行します。

```
> sign on as JADOPT61
> ADDLIBLE JADOPT61
> ADDENVVAR ENVVAR(CLASSPATH) VALUE('/home/javatests/adoptup/v6r1mig')
> JAVA JA61Example1
> JAVA JA61Alternative1A
> JAVA JA61Alternative1B
> JAVA JA61Example2
> JAVA JA61Alternative2
```

Java セキュリティー・モデル

Java アプレットはどのシステムからでもダウンロードできます。そのため、悪質なアプレットから保護するためのセキュリティー機構が Java 仮想マシンに組み込まれています。 Java ランタイム・システムは、Java 仮想マシンがバイトコードをロードするときにそれを検査します。これにより、それらが適正なバイトコードであること、および Java 仮想マシンが Java アプレットに課しているどの制限にも違反しないことが確認されます。

アプレットの場合と同じく、バイトコード・ローダーおよび検査装置はバイトコードが有効であるか、およびデータ・タイプが適切に使用されているかどうかを検査します。それらはさらに、レジスターおよびメモリーが正しくアクセスされているか、およびスタックがオーバーフローするまたはアンダーフローしていないかどうかを検査します。これらの検査によって、Java 仮想マシンがシステムの保全性を妨げることなくクラスを実行できることが保証されます。

Java アプレットは、実行可能な操作、メモリーへのアクセス方法、および Java 仮想マシンを使用する方法に関して制限を受けます。その制限は、Java アプレットが基礎となるオペレーティング・システムまたはシステム上のデータにアクセスすることを防ぎます。これは、"sandbox" セキュリティー・モデルと呼ばれます。 Java アプレットが自分のサンドボックス (砂箱) 内でのみ「遊ぶ」ことができるからです。

"sandbox" セキュリティー・モデルは、クラス・ローダー、クラス・ファイル・ベリファイヤー、および java.lang.SecurityManager クラスの組み合わせで実現されています。

SSL を使用するセキュア・アプリケーション

Sun Microsystems, Inc. による「Security」

Java Cryptography Extension

Java Cryptography Extension (JCE) は、暗号化、鍵生成、鍵合意のフレームワークおよびインプリメンテーション、ならびにメッセージ認証コード (MAC) アルゴリズムを備えています。暗号化のサポートには、対称、非対称、ブロック、およびストリーム暗号が含まれます。さらに、セキュア・ストリームおよびシールされたオブジェクトもサポートされています。 JCE は、メッセージ要約およびデジタル署名のインターフェースおよびインプリメンテーションをすでに含む Java 2 プラットフォームを補足します。

JCE の概要については、Sun の JCE 資料 (英語) を参照してください。この Web サイトのドキュメントには、他の多くの Web ベースの情報源へのリンクが含まれています。

| IBM は、IBM i に関して、以下の JCE プロバイダーを提供しています。

| **IBMJCE**
| デフォルトの JCE プロバイダー。

| **IBMJCEFIPS**
| FIPS 140 に対して評価済みの JCE プロバイダー・インプリメンテーション。 IBMJCEFIPS JCE
| プロバイダーについて詳しくは、IBM developerWorks® Web サイト上のセキュリティー情報 (英
| 語) を参照してください。

| **IBMJCECCAI5OS**
| JCE を拡張し、IBM 共通暗号化アーキテクチャー・インターフェースを通して暗号ハードウェア
| を使用する JCE プロバイダー・インプリメンテーション。 IBMJCECCAI5OS JCE プロバイダー
| について詳しくは、『ハードウェア暗号化機能の使用』を参照してください。

ハードウェア暗号化機能の使用

IBMJCECCAI5OS インプリメンテーションは、Java Cryptography Extension (JCE) と Java Cryptography Architecture (JCA) を拡張して、IBM Common Cryptographic Architecture (CCA) インターフェースからハードウェア暗号化機能を使用できるようにします。

IBMJCECCAI5OS プロバイダーは、既存の JCE 体系内でハードウェア暗号化機能を活用することにより、Java 2 プログラマーが既存の Java アプリケーションに極力変更を加えることなく、ハードウェア暗号化機能が持つセキュリティー面とパフォーマンス面での大きな利点を利用できるようにしています。ハードウェア暗号化機能の複雑な部分は通常の JCE の中で処理されるため、ハードウェア暗号化装置を使用した高度なセキュリティーとパフォーマンスが簡単に使用できるようになります。IBMJCECCAI5OS プロバイダーは、現行のプロバイダーと同じ方法で JCE フレームワークに接続します。ハードウェア要求に対しては、ネイティブ・メソッドを介して CCA API が呼び出されます。IBMJCECCAI5OS プロバイダーは、JCECCAI5OSKS Java 鍵ストア・タイプに CCA RSA 鍵ラベルを保管します。

ハードウェア暗号化機能の要件

ハードウェア暗号化機能を使用するためには、以下がシステムにインストールされている必要があります。

- モデル 4764 暗号化コプロセッサー
- IBM i (5770-SS1) オプション 35 - CCA Cryptographic Service Provider
- | ライセンス・プログラム・オファリング (LPO) 5733-CY3 - IBM Cryptographic Device Manager

IBM ハードウェア暗号化機能プロバイダーのフィーチャー

IBMJCECCAI5OS プロバイダーは、以下のアルゴリズムをサポートしています。

表 9. IBMJCECCAI5OS プロバイダーでサポートされているアルゴリズム

署名アルゴリズム	暗号アルゴリズム	メッセージ確認コード	メッセージ要約
SHA1withRSA MD2WithRSA MD5WithRSA	RSA	HmacMD2 HmacMD5 HmacSHA1	MD2 MD5 SHA-1

また、IBMJCECCAI5OS プロバイダーには、強力な疑似乱数発生ルーチン (PRNG)、鍵ファクトリーによる鍵生成、`keytool` アプリケーションによる鍵/証明書生成と鍵/証明書管理も組み込まれています。

ハードウェア暗号アクセス・プロバイダーは、`hwkeytool` アプリケーションで使用可能です。

注: insertProviderAt() メソッドや addProviderAt() メソッドを使用して JVM に IBMJCECCAI5OS プロバイダーを追加することはできません。

暗号化のシステム・プロパティー

暗号化装置を扱うための以下のシステム・プロパティーが使用できます。

i5os.crypto.device
使用する暗号化装置を指定します。このプロパティーが設定されていない場合は、デフォルト装置の CRP01 が使用されます。

i5os.crypto.keystore
使用する CCA 鍵ストア・ファイルを指定します。このプロパティーが設定されていない場合は、暗号化装置の記述内で指定されている鍵ストア・ファイルが使用されます。

4764 暗号化コプロセッサー

408 ページの『Java hwkeytool』
hwkeytool アプリケーションは、Java Cryptography Extension (JCE) および Java Cryptography Architecture (JCA) でモデル 4764 暗号化コプロセッサーの暗号化機能を使用可能にします。

15 ページの『Java システム・プロパティーのリスト』
Java システム・プロパティーにより、Java プログラムのランタイム環境が決まります。 Java システム・プロパティーは、IBM i のシステム値や環境変数と似ています。

鍵ペアとハードウェアの使用:

ハードウェア暗号化環境では、RETAINED と公開鍵データ・セット (PKDS) という 2 つのタイプの鍵ペアを使用して暗号化コプロセッサーを利用できます。

IBMJCECAI5OS プロバイダーでサポートされているハードウェア鍵ペアは、どちらのタイプもすべてのアプリケーションで利用できます。RETAINED および PKDS 鍵ペアは、それぞれラベルを戻します。このラベルは、IBMJCECAI5OS プロバイダーで鍵と同様に扱われます。アプリケーションがラベルを保管するメカニズムは選択可能です。

IBMJCECAI5OS プロバイダーは、4764 暗号化コプロセッサー内に格納されているマスター・キーで暗号化された共通暗号化アーキテクチャー (CCA) 鍵ストア・ファイルに RSA 鍵を保管します。
JCECCAI5OSKS 鍵ストアは、鍵レコードのラベルを CCA 鍵ストアに保管します。

RETAINED ハードウェア鍵ペア
IBMJCECAI5OS プロバイダーでサポートされている最も安全な暗号化のインプリメンテーションは、現実のハードウェア暗号化装置に鍵を保管して、鍵ペアの機密部分である秘密鍵を取り出したり表示したりできないようにする方法です。これは、秘密鍵がハードウェア装置に保存され、暗号化される前の鍵を表示したり取り出したりすることができないことから、*RETAINED* 鍵ペアと呼ばれます。鍵ペアの生成時には、ラベル と呼ばれる秘密鍵の参照だけがアプリケーションに戻され、あるいは鍵ストアに保管されます。

鍵が必要な時は、鍵が保存されているハードウェア・カードに要求と鍵のラベルが送信されます。ハードウェア・カード上で、保存されている鍵を使用して暗号化操作が実行され、結果が戻されます。RETAINED 鍵は、最も安全な鍵タイプです。RETAINED 鍵の欠点は、鍵のバックアップと回復ができないという点です。カードに問題が発生すると、鍵は失われてしまいます。

公開鍵データ・セット (PKDS) ハードウェア鍵ペア
RSA 鍵を利用する別の選択肢として、PKDS 鍵ペアを使用する方法があります。このタイプの鍵ペアを生成すると、秘密鍵はコプロセッサーのマスター・キーで暗号化され、暗号化される前の鍵

のテキストは表示したり取り出したりすることができなくなります。この鍵ペアは、DB2 データベース・ファイル内に保管されます。鍵ペアの生成時には、*ラベル* と呼ばれる秘密鍵の参照だけがアプリケーションに戻され、あるいは鍵ストアに保管されます。鍵ペアをファイルに保管するため、鍵をバックアップしておき、カードに問題が発生した場合でも鍵を回復させることが可能です。

Java Secure Socket Extension

Java Secure Socket Extension (JSSE) は、Secure Sockets Layer (SSL) と Transport Layer Security (TLS) の両方の基礎となるメカニズムを要約するフレームワークと似ています。基礎となっているプロトコルの複雑さと特色を要約することによって、JSSE はプログラマーが安全で暗号化された通信を使用できるようにすると同時に、セキュリティーのぜい弱性を最小限に抑えます。 Java Secure Socket Extension (JSSE) は、SSL プロトコルと TLS プロトコルの両方を使用して、クライアントとサーバーの間に安全で暗号化された通信を提供します。

SSL/TLS は、サーバーおよびクライアントを認証してプライバシーおよびデータ保全性を備えることを可能にします。すべての SSL/TLS 通信は、サーバーとクライアントとの間の「ハンドシェーク」から始まります。ハンドシェークの際、SSL/TLS はクライアントとサーバーが互いに通信するために使用する暗号の組を取り交わします。この暗号の組は、SSL/TLS で使用可能な種々のセキュリティー機能の組み合わせです。

JSSE は、以下の方法によりアプリケーションのセキュリティーを向上させます。

- 暗号化により通信データを保護する。
- リモート・ユーザー ID を認証する。
- リモート・システム名を認証する。

注: JSSE では、ディジタル証明書を使用して、Java アプリケーションのソケット通信を暗号化します。ディジタル証明書は、保護システム、ユーザー、およびアプリケーションを識別するためのインターネット標準です。 IBM ディジタル証明書マネージャーを使用すると、ディジタル証明書を制御できます。詳しくは、IBM ディジタル認証マネージャーを参照してください。

JSSE を使用して Java アプリケーションの保護を向上させるには、以下のようにします。

- IBM i で JSSE をサポートできるように準備する。
- 以下のようにして、JSSE を使用するように Java アプリケーションを設計する。
  - ソケット・ファクトリーをまだ使用していない場合は、ソケット・ファクトリーを使用するように Java ソケット・コードを変更する。
  - JSSE を使用するように Java コードを変更する。
- 以下のようにして、ディジタル証明書を使用して Java アプリケーションの保護を向上させる。
  1. 使用するディジタル証明書のタイプを選択します。
  2. アプリケーション実行時にディジタル証明書を使用します。

QsyRegisterAppForCertUse API を使用して、ご使用の Java アプリケーションを保護アプリケーションとして登録することもできます。

システムで Secure Sockets Layer をサポートできるように準備する

ご使用の IBM i サーバーで Secure Sockets Layer (SSL) を使用できるように準備するには、ディジタル証明書マネージャーの LP をインストールする必要があります。

ディジタル証明書マネージャーの LP のインストール、5770-SS1 IBM i - ディジタル証明書マネージャー

また、システム上のディジタル証明書にアクセスできるか作成できることを確認する必要もあります。

関連情報

ディジタル証明書マネージャー

ソケット・ファクトリーを使用するように Java コードを変更する

既存のコードで Secure Sockets Layer (SSL) を使用するには、まず最初にソケット・ファクトリーを使用するようにコードを変更しなければなりません。

ソケット・ファクトリーを使用するようにコードを変更するには、以下のステップを実行します。

1. 以下の行をご使用のプログラムに追加して、SocketFactory クラスをインポートします。

   ```
   import javax.net.*;
   ```

2. SocketFactory オブジェクトのインスタンスを宣言する行を追加します。以下に例を示します。

   ```
   SocketFactory socketFactory
   ```

3. SocketFactory インスタンスを、メソッド SocketFactory.getDefault() と同等の値に設定して、初期設定します。以下に例を示します。

   ```
   socketFactory = SocketFactory.getDefault();
   ```

   SocketFactory の宣言全体は、以下のようになるはずです。

   ```
   SocketFactory socketFactory = SocketFactory.getDefault();
   ```

4. 既存のソケットを初期設定します。宣言するソケットごとに、ソケット・ファクトリー上の SocketFactory メソッド createSocket(host,port) を呼び出します。

   この時点で、ソケットの宣言は以下のようになるはずです。

   ```
   Socket s = socketFactory.createSocket(host,port);
   ```

   ここで、

   - *s* は、作成するソケットです。
   - *socketFactory* は、ステップ 2 で作成した SocketFactory です。
   - *host* は、ホスト・サーバーの名前を表すストリング変数です。
   - *port* は、ソケット接続のポート番号を表す整変数です。

上記のステップをすべて完了すると、コードでソケット・ファクトリーが使用されます。コードにこれ以外の変更を加える必要はありません。呼び出されるメソッドとソケットの構文は、依然としてすべて稼働します。

例: サーバーのソケット・ファクトリーを使用するように Java コードを変更する:

以下の例は、simpleSocketServer という単純なソケット・クラスを変更し、ソケット・ファクトリーを使用してすべてのソケットを作成できるようにする方法を示しています。 1 つ目の例は、ソケット・ファクトリーのない simpleSocketServer クラスを示しています。 2 つ目の例は、ソケット・ファクトリーのある simpleSocketServer クラスを示しています。 2 つ目の例では、simpleSocketServer が factorySocketServer に名前変更されています。

例 1: ソケット・ファクトリーのないソケット・サーバー・プログラム

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
/* File simpleSocketServer.java*/

import java.net.*;
import java.io.*;

public class simpleSocketServer {
  public static void main (String args[]) throws IOException {

    int serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java simpleSocketServer serverPort");
      System.out.println("Defaulting to port 3000 since serverPort not specified.");
    }
    else
        serverPort = new Integer(args[0]).intValue();

    System.out.println("Establishing server socket at port " + serverPort);

    ServerSocket    serverSocket =
      new ServerSocket(serverPort);

    // a real server would handle more than just one client like this...

    Socket s = serverSocket.accept();
    BufferedInputStream is = new BufferedInputStream(s.getInputStream());
    BufferedOutputStream os = new BufferedOutputStream(s.getOutputStream());

    // This server just echoes back what you send it...

    byte buffer[] = new byte[4096];

    int bytesRead;

    // read until "eof" returned
    while ((bytesRead = is.read(buffer)) > 0) {
      os.write(buffer, 0, bytesRead); // write it back
      os.flush();    // flush the output buffer
    }


    s.close();
    serverSocket.close();
  }       // end main()

}       // end class definition
```

例 2: ソケット・ファクトリーのある単純なソケット・サーバー・プログラム

```
/* File factorySocketServer.java */

// need to import javax.net to pick up the ServerSocketFactory class
import javax.net.*;
import java.net.*;
import java.io.*;

public class factorySocketServer {
  public static void main (String args[]) throws IOException {

    int serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java simpleSocketServer serverPort");
      System.out.println("Defaulting to port 3000 since serverPort not specified.");
```

```
    }
    else
        serverPort = new Integer(args[0]).intValue();

    System.out.println("Establishing server socket at port " + serverPort);

    // Change the original simpleSocketServer to use a
    // ServerSocketFactory to create server sockets.
    ServerSocketFactory serverSocketFactory =
      ServerSocketFactory.getDefault();
    // Now have the factory create the server socket.  This is the last
    // change from the original program.
    ServerSocket    serverSocket =
      serverSocketFactory.createServerSocket(serverPort);

    // a real server would handle more than just one client like this...

    Socket s = serverSocket.accept();
    BufferedInputStream is = new BufferedInputStream(s.getInputStream());
    BufferedOutputStream os = new BufferedOutputStream(s.getOutputStream());

    // This server just echoes back what you send it...

    byte buffer[] = new byte[4096];

    int bytesRead;

    while ((bytesRead = is.read(buffer)) > 0) {
      os.write(buffer, 0, bytesRead);
      os.flush();
    }


    s.close();
    serverSocket.close();
  }

}
```

例: クライアントのソケット・ファクトリーを使用するように Java コードを変更する:

以下の例は、simpleSocketClient という単純なソケット・クラスを変更し、ソケット・ファクトリーを使用してすべてのソケットを作成できるようにする方法を示しています。 1 つ目の例は、ソケット・ファクトリーのない simpleSocketClient クラスを示しています。 2 つ目の例は、ソケット・ファクトリーのある simpleSocketClient クラスを示しています。 2 つ目の例では、simpleSocketClient が factorySocketClient に名前変更されています。

例 1: ソケット・ファクトリーのないソケット・クライアント・プログラム

注: サンプル・コードをご使用の場合は、560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
/* Simple Socket Client Program */

import java.net.*;
import java.io.*;

public class simpleSocketClient {
  public static void main (String args[]) throws IOException {

    int serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java simpleSocketClient serverHost serverPort");
```

```
      System.out.println("serverPort defaults to 3000 if not specified.");
      return;
    }
    if (args.length == 2)
      serverPort = new Integer(args[1]).intValue();

    System.out.println("Connecting to host " + args[0] + " at port " +
                       serverPort);

    // Create the socket and connect to the server.
    Socket  s = new Socket(args[0], serverPort);
    .
    .
    .

    // The rest of the program continues on from here.
```

例 2: ソケット・ファクトリーのある単純なソケット・クライアント・プログラム

```
/* Simple Socket Factory Client Program */

// Notice that javax.net.* is imported to pick up the SocketFactory class.
import javax.net.*;
import java.net.*;
import java.io.*;

public class factorySocketClient {
  public static void main (String args[]) throws IOException {

    int serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java factorySocketClient serverHost serverPort");
      System.out.println("serverPort defaults to 3000 if not specified.");
      return;
    }
    if (args.length == 2)
      serverPort = new Integer(args[1]).intValue();

    System.out.println("Connecting to host " + args[0] + " at port " +
                       serverPort);

    // Change the original simpleSocketClient program to create a
    // SocketFactory and then use the socket factory to create sockets.

    SocketFactory socketFactory = SocketFactory.getDefault();

    // Now the factory creates the socket.  This is the last change
    // to the original simpleSocketClient program.

    Socket  s = socketFactory.createSocket(args[0], serverPort);
    .
    .
    .

    // The rest of the program continues on from here.
```

Secure Sockets Layer を使用するように Java コードを変更する

すでにコード中でソケット・ファクトリーを使用してソケットを作成している場合は、ご使用のプログラムに Secure Sockets Layer (SSL) サポートを追加できます。

まだコード中でソケット・ファクトリーを使用していない場合は、ソケット・ファクトリーを使用するように Java コードを変更するを参照してください。

SSL を使用するようにコードを変更するには、以下のステップを実行します。

1. javax.net.ssl.* をインポートして、SSL サポートを追加する。
   ```
   import javax.net.ssl.*;
   ```
2. SSLSocketFactory を使用して SocketFactory を初期設定することにより、SocketFactory を宣言する。
   ```
   SocketFactory newSF = SSLSocketFactory.getDefault();
   ```
3. 新しい SocketFactory を使用して、以前の SocketFactory の場合と同じ方法でソケットを初期設定する。
   ```
   Socket s = newSF.createSocket(args[0], serverPort);
   ```

これで、コード中で SSL サポートが使用されるようになりました。コードにこれ以外の変更を加える必要はありません。

例: Secure Sockets Layer を使用するように Java サーバーを変更する:

以下の例は、factorySocketServer という 1 つのクラスを変更して、Secure Sockets Layer (SSL) を使用できるようにする方法を示しています。

1 つ目の例は、SSL を使用しない factorySocketServer クラスを示しています。 2 つ目の例は、同じクラスで SSL を使用するものを示しており、名前が factorySSLSocketServer に変更されています。

例 1: SSL を使用しない単純な factorySocketServer クラス

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
/* File factorySocketServer.java */
// need to import javax.net to pick up the ServerSocketFactory class
import javax.net.*;
import java.net.*;
import java.io.*;

public class factorySocketServer {
  public static void main (String args[]) throws IOException {

    int serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java simpleSocketServer serverPort");
      System.out.println("Defaulting to port 3000 since serverPort not specified.");
    }
    else
        serverPort = new Integer(args[0]).intValue();

    System.out.println("Establishing server socket at port " + serverPort);

    // Change the original simpleSocketServer to use a
    // ServerSocketFactory to create server sockets.
    ServerSocketFactory serverSocketFactory =
      ServerSocketFactory.getDefault();
    // Now have the factory create the server socket.  This is the last
    // change from the original program.
    ServerSocket    serverSocket =
      serverSocketFactory.createServerSocket(serverPort);

    // a real server would handle more than just one client like this...

    Socket s = serverSocket.accept();
    BufferedInputStream is = new BufferedInputStream(s.getInputStream());
    BufferedOutputStream os = new BufferedOutputStream(s.getOutputStream());
```

```
    // This server just echoes back what you send it.

    byte buffer[] = new byte[4096];

    int bytesRead;

    while ((bytesRead = is.read(buffer)) > 0) {
      os.write(buffer, 0, bytesRead);
      os.flush();
    }

    s.close();
    serverSocket.close();
  }
}
```

例 2: SSL を使用する単純な factorySocketServer クラス

```
/* File factorySocketServer.java */

// need to import javax.net to pick up the ServerSocketFactory class
import javax.net.*;
import java.net.*;
import java.io.*;

public class factorySocketServer {
  public static void main (String args[]) throws IOException {

    int serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java simpleSocketServer serverPort");
      System.out.println("Defaulting to port 3000 since serverPort not specified.");
    }
    else
        serverPort = new Integer(args[0]).intValue();

    System.out.println("Establishing server socket at port " + serverPort);

    // Change the original simpleSocketServer to use a
    // ServerSocketFactory to create server sockets.
    ServerSocketFactory serverSocketFactory =
      ServerSocketFactory.getDefault();
    // Now have the factory create the server socket.  This is the last
    // change from the original program.
    ServerSocket    serverSocket =
      serverSocketFactory.createServerSocket(serverPort);

    // a real server would handle more than just one client like this...

    Socket s = serverSocket.accept();
    BufferedInputStream is = new BufferedInputStream(s.getInputStream());
    BufferedOutputStream os = new BufferedOutputStream(s.getOutputStream());

    // This server just echoes back what you send it.

    byte buffer[] = new byte[4096];

    int bytesRead;

    while ((bytesRead = is.read(buffer)) > 0) {
      os.write(buffer, 0, bytesRead);
      os.flush();
    }
```

```
    s.close();
    serverSocket.close();
  }
}
```

例: Secure Sockets Layer を使用するように Java クライアントを変更する:

以下の例は、factorySocketClient という 1 つのクラスを変更して、Secure Sockets Layer (SSL) を使用できるようにする方法を示しています。 1 つ目の例は、SSL を使用しない factorySocketClient クラスを示しています。 2 つ目の例は、同じクラスで SSL を使用するものを示しており、名前が factorySSLSocketClient に変更されています。

例 1: SSL を使用しない単純な factorySocketClient クラス

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
/* Simple Socket Factory Client Program */

import javax.net.*;
import java.net.*;
import java.io.*;

public class factorySocketClient {
  public static void main (String args[]) throws IOException {

    int serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java factorySocketClient serverHost serverPort");
      System.out.println("serverPort defaults to 3000 if not specified.");
      return;
    }
    if (args.length == 2)
      serverPort = new Integer(args[1]).intValue();

    System.out.println("Connecting to host " + args[0] + " at port " +
                       serverPort);

    SocketFactory socketFactory = SocketFactory.getDefault();

    Socket  s = socketFactory.createSocket(args[0], serverPort);
    .
    .
    .

    // The rest of the program continues on from here.
```

例 2: SSL を使用する単純な factorySocketClient クラス

```
// Notice that we import javax.net.ssl.* to pick up SSL support
import javax.net.ssl.*;
import javax.net.*;
import java.net.*;
import java.io.*;

public class factorySSLSocketClient {
  public static void main (String args[]) throws IOException {

    int serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java factorySSLSocketClient serverHost serverPort");
      System.out.println("serverPort defaults to 3000 if not specified.");
```

```
    return;
  }
  if (args.length == 2)
    serverPort = new Integer(args[1]).intValue();

  System.out.println("Connecting to host " + args[0] + " at port " +
                     serverPort);

  // Change this to create an SSLSocketFactory instead of a SocketFactory.
  SocketFactory socketFactory = SSLSocketFactory.getDefault();

  // We do not need to change anything else.
  // That's the beauty of using factories!
  Socket  s = socketFactory.createSocket(args[0], serverPort);
  .
  .
  .

  // The rest of the program continues on from here.
```

ディジタル証明書の選択

どのディジタル証明書を使用するか決める際には、複数の要素を考慮する必要があります。ご使用のシステムのデフォルト証明書を使用することもできますし、別の証明書を指定して使用することもできます。

以下の場合は、システムのデフォルト証明書を使用することもできます。

- ご使用の Java アプリケーションに特定のセキュリティー要件がない。
- ご使用の Java アプリケーションに必要なセキュリティーの種類が分からない。
- システムのデフォルト証明書が、ご使用の Java アプリケーションのセキュリティー要件を満たしている。

注: システムのデフォルト証明書を使用することに決めた場合は、システム管理者に問い合わせて、デフォルトのシステム証明書が作成されていることを確認してください。

システムのデフォルト証明書を使用しない場合は、別の使用する証明書を選択する必要があります。 2 種類の証明書を選択できます。

- **ユーザー証明書**。この証明書は、アプリケーションのユーザーを識別します。
- **システム証明書**。この証明書は、アプリケーションが実行されているシステムを識別します。

以下の場合は、ユーザー証明書が必要です。

- アプリケーションがクライアント・アプリケーションとして実行されている。
- どのユーザーがアプリケーションを使って作業しているかを識別する証明書が必要である。

以下の場合は、システム証明書が必要です。

- アプリケーションがサーバー・アプリケーションとして実行されている。
- アプリケーションが実行されているシステムを識別する証明書が必要である。

必要な種類の証明書を判別し終えたならば、アクセス可能な証明書コンテナーの中から該当するディジタル証明書を選択できます。

関連情報

ディジタル証明書マネージャー

Java アプリケーション実行時にディジタル証明書を使用する

Secure Sockets Layer (SSL) を使用するには、ディジタル証明書を使用して Java アプリケーションを実行する必要があります。

使用するディジタル証明書を指定するには、以下のプロパティーを使用してください。

- os400.certificateContainer
- os400.certificateLabel

たとえば、ディジタル証明書 MYCERTIFICATE を使用して Java アプリケーション MyClass.class を実行したい場合に、MYCERTIFICATE がディジタル証明書コンテナー YOURDCC 中にあると、`java` コマンドは以下のようになります。

```
java -Dos400.certificateContainer=YOURDCC
     -Dos400.certificateLabel=MYCERTIFICATE MyClass
```

使用するディジタル証明書をまだ決めていない場合は、275 ページの『ディジタル証明書の選択』を参照してください。システムのデフォルト証明書を使用するように決めることもできます。この証明書は、システムのデフォルトの証明書コンテナー中に保管されています。

システムのデフォルトのディジタル証明書を使用するには、証明書や証明書コンテナーをどこにも指定する必要はありません。ご使用の Java アプリケーションで自動的にシステムのデフォルトのディジタル証明書が使用されます。

ディジタル証明書と -os400.certificateLabel プロパティー

ディジタル証明書は、保護システム、ユーザー、およびアプリケーションを識別するためのインターネット標準です。ディジタル証明書は、ディジタル証明書コンテナー中に保管されています。ディジタル証明書コンテナーのデフォルトの証明書を使用したい場合は、証明書のラベルを指定する必要はありません。特定のディジタル証明書を使用したい場合は、以下のプロパティーを使用して `java` コマンド中に証明書のラベルを指定しなければなりません。

```
os400.certificateLabel=
```

たとえば、MYCERTIFICATE という名前の証明書を使用したい場合は、以下のような `java` コマンドを入力します。

```
java -Dos400.certificateLabel=MYCERTIFICATE MyClass
```

この例では、Java アプリケーション MyClass により証明書 MYCERTIFICATE が使用されます。MYCERTIFICATE が MyClass に使用されるためには、システムのデフォルトの証明書コンテナー中になければなりません。

ディジタル証明書コンテナーと -os400.certificateContainer プロパティー

ディジタル証明書コンテナーにはディジタル証明書が保管されています。 IBM i のシステム・デフォルト証明書コンテナーを使用したい場合は、証明書コンテナーを指定する必要はありません。特定のディジタル証明書コンテナーを使用するには、以下のプロパティーを使用して `java` コマンド中にディジタル証明書コンテナーを指定する必要があります。

```
os400.certificateContainer=
```

たとえば、MYDCC という名前の、使用したいディジタル証明書を含む証明書コンテナーを使用したい場合は、以下のような java コマンドを入力します。

```
java -Dos400.certificateContainer=MYDCC MyClass
```

この例では、MyClass.class という名前の Java アプリケーションが、MYDCC という名前のディジタル証明書コンテナー中にあるデフォルトのディジタル証明書を使用して、システム上で実行されます。アプリケーション中に作成したソケットによって、MYDCC 中のデフォルト証明書が使用されて、自己識別が行われ、すべての通信保護が行われます。

ディジタル証明書コンテナー中のディジタル証明書 MYCERTIFICATE を使用したい場合は、以下のような java コマンドを入力します。

```
java -Dos400.certificateContainer=MYDCC
        -Dos400.certificateLabel=MYCERTIFICATE MyClass
```

関連情報

ディジタル証明書マネージャー

Java Secure Socket Extension 1.4 の使用

この情報は、J2SDK バージョン 1.4 が稼働する IBM i サーバー上で JSSE を使用する場合にのみ適用されます。JSSE は、SSL と TLS の両方の基礎となるメカニズムを要約するフレームワークと似ています。基礎となっているプロトコルの複雑さと特色を要約することによって、JSSE はプログラマーが安全で暗号化された通信を使用できるようにすると同時に、セキュリティーのぜい弱性を最小限に抑えます。 Java Secure Socket Extension (JSSE) は、Secure Sockets Layer (SSL) プロトコルと Transport Layer Security (TLS) プロトコルの両方を使用して、クライアントとサーバーの間に安全で暗号化された通信を提供します。

IBM JSSE インプリメンテーションは IBM JSSE と呼ばれています。 IBM JSSE には、ネイティブ IBM i JSSE プロバイダーと純正の Java JSSE プロバイダーが組み込まれています。

JSSE 1.4 をサポートするようにシステムを構成する:

IBM JSSE を使用するようにシステムを構成する。このトピックには、ソフトウェア要件、JSSE プロバイダーの変更方法、および必要なセキュリティー・プロパティーとシステム・プロパティーに関する情報もあります。

| ご使用の IBM i サーバー上で Java を使用している場合、JSSE は既に構成済みです。デフォルトの構成
| では、IBMJSSE2 という IBM 純正の Java JSSE プロバイダーを使用します。

JSSE プロバイダーの変更

純粋の Java JSSE プロバイダーの代わりにネイティブ IBM i JSSE プロバイダーを使用するように、JSSE を構成することができます。いくつかの特定の JSSE セキュリティー・プロパティーと Java システム・プロパティーを変更することによって、この 2 つのプロバイダーの間の切り替えができます。

セキュリティー・マネージャー

Java セキュリティー・マネージャーを使用可能にして JSSE アプリケーションを実行している場合、使用可能なネットワーク許可を設定する必要がある可能性があります。詳しくは、Permissions in the Java 2 SDK の SSL Permission の説明を参照してください。

JSSE 1.4 プロバイダー:

IBM JSSE には、ネイティブ IBM i JSSE プロバイダーと 2 つの純正 Java JSSE プロバイダーが組み込まれています。どのプロバイダーを選択して使用するかは、アプリケーションの必要に応じて異なります。

3 つのプロバイダーはすべて JSSE インターフェースの仕様に準拠します。これらは双方向通信が可能であったり、任意の他の SSL または TLS インプリメンテーション (非 Java インプリメンテーションでも可) と通信することができます。

ピュア Java JSSE プロバイダー

ピュア Java JSSE プロバイダーは、以下のフィーチャーを備えています。

- ディジタル証明書を制御および構成するためのあらゆるタイプの KeyStore オブジェクト (たとえば、JKS および PKCS12 など) を処理します。
- 複数のインプリメンテーションの JSSE コンポーネントを組み合わせて一緒に使用することができます。

IBMJSSEProvider2 はピュア Java 実装のプロバイダー名です。このプロバイダー名は、適切なケースを使用して java.security.Security.getProvider() メソッドか、あるいは、いくつかの JSSE クラス用の種々の getInstance() メソッドに渡す必要があります。

注: IBMJSSE と呼ばれる第 2 のピュア Java JSSE プロバイダーが出荷されています。この古いプロバイダーは、IBMJSSEProvider2 では推奨されなくなりました。

ピュア Java JSSE FIPS 140-2 プロバイダー

ピュア Java JSSE FIPS 140-2 プロバイダーは、以下のフィーチャーを備えています。

- 暗号モジュール用の連邦情報処理標準 (FIPS) 140-2 を使用してコンパイルします。
- ディジタル証明書を制御および構成するためのあらゆるタイプの KeyStore オブジェクトを処理します。

注: ピュア Java JSSE FIPS 140-2 プロバイダーは、任意の他の実装のコンポーネントがその実装にプラグインされることを許可しません。

IBMJSSEFIPS はピュア Java JSSE FIPS 140-2 実装のプロバイダー名です。このプロバイダー名は、適切なケースを使用して java.security.Securirty.getProvider() メソッドか、あるいは、いくつかの JSSE クラス用の種々の getInstance() メソッドに渡す必要があります。

ネイティブ IBM i JSSE プロバイダー

ネイティブ IBM i JSSE プロバイダーには、以下の機能があります。

- ネイティブ IBM i SSL サポートを使用します。
- ディジタル証明書を構成、制御するために、ディジタル証明書マネージャーの使用を許可します。これは、固有な IBM i タイプの KeyStore (IbmISeriesKeyStore) を介して提供されます。
- 最善のパフォーマンスを提供します。
- 複数のインプリメンテーションの JSSE コンポーネントを組み合わせて一緒に使用することができます。ただし、最良のパフォーマンスを得るには、JSSE ネイティブ IBM i コンポーネントのみを使用してください。

IBMi5OSJSSEProvider は、ネイティブ IBM i 実装の名前です。このプロバイダー名は、適切なケースを使用して java.security.Security.getProvider() メソッドか、あるいは、いくつかの JSSE クラス用の種々の getInstance() メソッドに渡す必要があります。

デフォルト JSSE プロバイダーの変更

セキュリティー・プロパティーに適切な変更を加えることによって、デフォルトの JSSE プロバイダーを変更することができます。詳細は、『JSSE セキュリティー・プロパティー』を参照してください。

JSSE プロバイダーを変更した後は、システム・プロパティーが、新しいプロバイダーが必要とするディジタル証明書情報 (鍵ストア) 用の適切な構成を指定していることを確認します。詳細は、『Java システム・プロパティー』を参照してください。

JSSE 1.4 セキュリティー・プロパティー:

Java 仮想マシン (JVM) は、多数の重要なセキュリティー・プロパティーを使用します。これらのセキュリティー・プロパティーは、Java マスター・セキュリティー・プロパティー・ファイルを編集することによって設定されます。

| これは java.security というファイルで、通常は IBM i サーバー上の `/QOpenSys/QIBM/ProdData/JavaVM/`
| `jdk14/64bit/jre/lib/security` ディレクトリーにあります。

以下のリストは、JSSE を使用するための、関連するいくつかのセキュリティー・プロパティーを示しています。この説明は、java.security ファイルを編集するためのガイドとして使用してください。

security.provider.<integer>

使用する JSSE プロバイダー。これは静的に暗号プロバイダー・クラスの登録も行います。以下の例に示すように、異なる JSSE プロバイダーを指定してください。

```
security.provider.5=com.ibm.jsse2.IBMJSSEProvider2
security.provider.6=com.ibm.i5os.jsse.JSSEProvider
security.provider.7=com.ibm.jsse.IBMJSSEProvider
security.provider.8=com.ibm.fips.jsse.IBMJSSEFIPSProvider
```

| **注:** これらの 4 つのプロバイダー・エントリーはすべて必須なのではありません。エントリーは、そのプロバイダーが使用される場合にのみ必要です。デフォルトでは、これらのプロバイダーのうち、2 つのみが java.security にリストされています。プロバイダーの順序が変更された場合は、プロバイダーの番号も変更して、順序を保持する必要があります。

ssl.KeyManagerFactory.algorithm

デフォルトの KeyManagerFactory アルゴリズムを指定します。ネイティブ IBM i JSSE プロバイダーの場合は、以下を使用します。

```
ssl.KeyManagerFactory.algorithm=IbmISeriesX509
```

純正の Java JSSE プロバイダーの場合は、以下を使用します。

```
ssl.KeyManagerFactory.algorithm=IbmX509
```

詳しくは、javax.net.ssl.KeyManagerFactory の javadoc を参照してください。

ssl.TrustManagerFactory.algorithm

デフォルトの TrustManagerFactory アルゴリズムを指定します。ネイティブ IBM i JSSE プロバイダーの場合は、以下を使用します。

```
ssl.TrustManagerFactory.algorithm=IbmISeriesX509
```

純正の Java JSSE プロバイダーの場合は、以下を使用します。

```
| ssl.TrustManagerFactory.algorithm=IbmPKIX
```

詳しくは、javax.net.ssl.TrustManagerFactory の javadoc を参照してください。

ssl.SocketFactory.provider

デフォルトの SSL ソケット・ファクトリーを指定します。ネイティブ IBM i JSSE プロバイダーの場合は、以下を使用します。

```
| ssl.SocketFactory.provider=com.ibm.i5os.jsse.JSSESocketFactory
```

純正の Java JSSE プロバイダーの場合は、以下を使用します。

```
| ssl.SocketFactory.provider=com.ibm.jsse2.SSLSocketFactoryImpl
```

詳しくは、javax.net.ssl.SSLSocketFactory の javadoc を参照してください。

ssl.ServerSocketFactory.provider

デフォルトの SSL サーバー・ソケット・ファクトリーを指定します。ネイティブ IBM i JSSE プロバイダーの場合は、以下を使用します。

```
| ssl.ServerSocketFactory.provider=com.ibm.i5os.jsse.JSSEServerSocketFactory
```

純正の Java JSSE プロバイダーの場合は、以下を使用します。

```
| ssl.ServerSocketFactory.provider=com.ibm.jsse2.SSLServerSocketFactoryImpl
```

詳しくは、javax.net.ssl.SSLServerSocketFactory の javadoc を参照してください。

JSSE for 1.4 Java システム・プロパティー:

アプリケーションで JSSE を使用するには、デフォルトの SSLContext オブジェクトが構成の確認を行うために必要な、いくつかのシステム・プロパティーを指定する必要があります。いくつかのプロパティーは両方のプロバイダーに適用され、その他はネイティブ IBM i プロバイダーにだけ適用されます。

ネイティブ IBM i JSSE プロバイダーを使用するとき、プロパティーを指定しない場合、os400.certificateContainer はデフォルトの *SYSTEM になりますが、それは、JSSE がシステム証明書ストア内のデフォルトのエントリーを使用することを意味します。

両方にプロバイダーに作用するプロパティー

以下のプロパティーは両方の JSSE プロバイダーに適用されます。それぞれの説明では、該当する場合には、デフォルトのプロパティーも示しています。

javax.net.ssl.trustStore

デフォルトの TrustManager が使用する KeyStore オブジェクトが入っているファイルの名前。デフォルト値は jssecacerts または (jssecacerets が存在しない場合は) cacerts です。

javax.net.ssl.trustStoreType

デフォルトの TrustManager が使用する KeyStore オブジェクトのタイプ。デフォルト値は KeyStore.getDefaultType メソッドによって戻される値です。

javax.net.ssl.trustStorePassword

デフォルトの TrustManager が使用する KeyStore オブジェクトのパスワード。

javax.net.ssl.keyStore

デフォルトの KeyManager が使用する KeyStore オブジェクトが入っているファイルの名前。

javax.net.ssl.keyStoreType

デフォルトの KeyManager が使用する KeyStore オブジェクトのタイプ。デフォルト値は KeyStore.getDefaultType メソッドによって戻される値です。

javax.net.ssl.keyStorePassword

デフォルトの KeyManager が使用する KeyStore オブジェクトのパスワード。

ネイティブ IBM i JSSE プロバイダーにのみ適用されるプロパティー

以下のプロパティーは、ネイティブ IBM i JSSE プロバイダーにのみ適用されます。

os400.secureApplication

アプリケーション ID。 JSSE は、以下のいずれかのプロパティーが指定されていない場合にのみ、このプロパティーを使用します。

- javax.net.ssl.keyStore
- javax.net.ssl.keyStorePassword
- javax.net.ssl.keyStoreType
- javax.net.ssl.trustStore
- javax.net.ssl.trustStorePassword
- javax.ssl.net.trustStoreType

os400.certificateContainer

使用する鍵リングの名前。 JSSE は、以下のいずれかのプロパティーが指定されていない場合にのみ、このプロパティーを使用します。

- javax.net.ssl.keyStore
- javax.net.ssl.keyStorePassword
- javax.net.ssl.keyStoreType
- javax.net.ssl.trustStore
- javax.net.ssl.trustStorePassword
- javax.ssl.net.trustStoreType
- os400.secureApplication

os400.certificateLabel

使用する鍵リング・ラベル。 JSSE は、以下のいずれかのプロパティーが指定されていない場合にのみ、このプロパティーを使用します。

- javax.net.ssl.keyStore
- javax.net.ssl.keyStorePassword
- javax.net.ssl.trustStore

- javax.net.ssl.trustStorePassword
- javax.ssl.net.trustStoreType
- os400.secureApplication

関連概念

15 ページの『Java システム・プロパティーのリスト』
Java システム・プロパティーにより、Java プログラムのランタイム環境が決まります。 Java システム・プロパティーは、IBM i のシステム値や環境変数と似ています。

関連情報

Sun Java Web サイトの「System Properties」

ネイティブ IBM i JSSE 1.4 プロバイダーの使用:

ネイティブ IBM i JSSE プロバイダーは、JSSE クラスおよびインターフェースの一式を備えています。これには JSSE KeyStore クラスおよび SSLConfiguration クラスの実装が含まれています。

ネイティブ IBM i プロバイダーを効果的に使用するには、このトピックの情報を使い、さらに JSSE 1.4 の SSLConfiguration Javadoc 情報も参照してください。

SSLContext.getInstance メソッドのプロトコル値

以下の表では、ネイティブ IBM i JSSE プロバイダーの SSLContext.getInstance メソッドのプロトコル値を示し、それを説明しています。

サポートされる SSL プロトコルは、システムに設定されているシステム値によって制限を受けることがあります。詳細は、『システム管理』情報の「Security system values: Secure Sockets Layer protocols」サブトピックを参照してください。

プロトコル値	サポートされている SSL プロトコル
SSL	SSL バージョン 2、SSL バージョン 3、および TLS バージョン 1。SSLv2 形式の Hello にカプセル化された SSLv3 または TLSv1 の Hello を受け入れます。
SSLv2	SSL バージョン 2
SSLv3	SSL バージョン 3 プロトコル。SSLv2 形式の Hello にカプセル化された SSLv3 Hello を受け入れます。
TLS	SSL バージョン 2、SSL バージョン 3、および TLS バージョン 1。SSLv2 形式の Hello にカプセル化された SSLv3 または TLSv1 の Hello を受け入れます。
TLSv1	TLS バージョン 1 プロトコル。Request for Comments (RFC) 2246 で定義されています。 SSLv2 形式の Hello にカプセル化された TLSv1 Hello を受け入れます。
SSL_TLS	SSL バージョン 2、SSL バージョン 3、および TLS バージョン 1。SSLv2 形式の Hello にカプセル化された SSLv3 または TLSv1 の Hello を受け入れます。

ネイティブ IBM i KeyStore 実装

ネイティブ IBM i プロバイダーは、IbmISeriesKeyStore タイプの KeyStore クラスの実装を備えています。鍵ストアの実装は、ディジタル証明書マネージャー・サポートのまわりのラッパーを提供します。鍵ス

トアの内容は、特定のアプリケーション ID または鍵リング・ファイル、パスワード、およびラベルに応じて異なります。 JSSE はディジタル証明書マネージャーから鍵ストア・エントリーをロードします。エントリーをロードするため、JSSE はアプリケーションが最初に鍵ストア・エントリーまたは鍵ストア情報にアクセスしようとするときに、適切なアプリケーション ID または鍵リング情報を使用します。鍵ストアは変更できず、構成の変更はすべてディジタル証明書マネージャーを使用して行わなければなりません。

ネイティブ IBM i プロバイダーを使用する際の推奨事項

| JKS KeyStore または IbmX509 TrustManagerFactory などのネイティブでないコンポーネントを使用できる
| ようにするために、JAR ファイル `ibmjsseprovider2.jar` を次のようにブート・クラスパスに追加する必
| 要があります。

```
| -Xbootclasspath/p:/QOpenSys/QIBM/ProdData/JavaVM/jdk14/64bit/jre/lib/ext/ibmjsseprovider2.jar
```

| **関連資料**

『JSSE 1.4 の SSLConfiguration Javadoc 情報』

関連情報

ディジタル証明書マネージャー

Security system values: Secure Sockets Layer protocols

JSSE 1.4 の SSLConfiguration Javadoc 情報:

com.ibm.as400

クラス SSLConfiguration

```
| java.lang.Object
|   |
|   +--com.ibm.i5os.jsse.SSLConfiguration
```

インプリメントされているすべてのインターフェース:

java.lang.Cloneable, javax.net.ssl.ManagerFactoryParameters

public final class **SSLConfiguration**
extends java.lang.Object
implements javax.net.ssl.ManagerFactoryParameters, java.lang.Cloneable

このクラスは、ネイティブ IBM i JSSE インプリメンテーションによって必要とされる構成の仕様を提供します。

ネイティブ IBM i JSSE インプリメンテーションは、タイプが "IbmISeriesKeyStore" の `KeyStore` オブジェクトを使用すると、最も効率的に作動します。このタイプの `KeyStore` オブジェクトには、ディジタル証明書マネージャー (DCM) に登録されたアプリケーション ID かまたは鍵リング・ファイル (ディジタル証明書コンテナー) に基づいた、キー項目および信頼できる証明書項目が含まれます。また、このタイプの `KeyStore` オブジェクトを使用して、"IbmISeriesSslProvider" `Provider` から `X509KeyManger` および `X509TrustManager` オブジェクトを初期化することができます。さらに、`X509KeyManager` および `X509TrustManager` オブジェクトを使用して、"IbmISeriesSslProvider" から `SSLContext` オブジェクトを初期化することができます。次に `SSLContext` オブジェクトは、`KeyStore` オブジェクトに指定された構成情報に基づいて、ネイティブ IBM i JSSE インプリメンテーションへのアクセスを提供します。"IbmISeriesKeyStore" `KeyStore` のロードが実行されるたびに、アプリケーション ID または鍵リング・ファイルに指定された現行の構成に基づいて、`KeyStore` が初期化されます。

また、このクラスを使用して、任意の有効なタイプの KeyStore オブジェクトを生成することもできます。KeyStore は、アプリケーション ID または鍵リング・ファイルに指定された現行の構成に基づいて初期化されます。アプリケーション ID または鍵リング・ファイルによって指定された構成に何らかの変更を加えた場合、その変更を反映するために、KeyStore オブジェクトを再生成する必要があります。"IbmISeriesKeyStore" 以外のタイプの KeyStore を正常に作成できるようにするには、鍵リング・パスワードを (アプリケーション ID を使用する場合は、*SYSTEM 証明書ストアに) 指定する必要があることに注意してください。作成される "IbmISeriesKeyStore" タイプの KeyStore 用の秘密鍵に正常にアクセスできるようにするには、鍵リング・パスワードを指定する必要があります。

対象: SDK 1.4 以降

参照: KeyStore, X509KeyManager, X509TrustManager, SSLContext

コンストラクターの要約

SSLConfiguration() 新規の SSLConfiguration を作成します。詳しくは、 285 ページの『コンストラクターの詳細』を参照してください。

表 10. **メソッドの要約**

void	288 ページの『clear』() すべての get メソッドがヌルを戻すように、オブジェクト内のすべての情報を消去します。
java.lang.Object	289 ページの『clone』() この SSL 構成の新規コピーを生成します。
boolean	289 ページの『equals』(java.lang.Objectobj) 他の何らかのオブジェクトがこのオブジェクトに「相当する」かどうかを示します。
protected void	288 ページの『finalize』() ガーベッジ・コレクションが、オブジェクトの参照がこれ以上ないと判断するときに、ガーベッジ・コレクターによってオブジェクト上で呼び出されます。
java.lang.String	287 ページの『getApplicationId』() アプリケーション ID を戻します。
java.lang.String	288 ページの『getKeyringLabel』() 鍵リング・ラベルを戻します。
java.lang.String	287 ページの『getKeyringName』() 鍵リング名を戻します。
char[]	288 ページの『getKeyringPassword』() 鍵リング・パスワードを戻します。
java.security.KeyStore	290 ページの『getKeyStore』(char[]password) 与えられたパスワードを使用して、タイプが "IbmISeriesKeyStore" の鍵ストアを戻します。
java.security.KeyStore	290 ページの『 getKeyStore』(java.lang.Stringtype, char[]password) 与えられたパスワードを使用して、要求されたタイプの鍵ストアを戻します。
int	289 ページの『hashCode』() オブジェクトのハッシュ・コード値を戻します。
staticvoid	(java.lang.String[]args) SSLConfiguration 関数を実行します。
void	(java.lang.String[]args, java.io.PrintStreamout) SSLConfiguration 関数を実行します。
void	289 ページの『setApplicationId』(java.lang.StringapplicationId) アプリケーション ID を設定します。
void	289 ページの『setApplicationId』(java.lang.StringapplicationId, char[]password) アプリケーション ID および鍵リング・パスワードを設定します。
void	288 ページの『setKeyring』(java.lang.Stringname,java.lang.Stringlabel, char[]password) 鍵リング情報を設定します。

クラス java.lang.Object から継承されるメソッド
getClass, notify, notifyAll, toString, wait, wait, wait

コンストラクターの詳細

SSLConfiguration

`public SSLConfiguration()`

新規の SSLConfiguration を作成します。アプリケーション ID および鍵リング情報はデフォルト値に初期設定されます。

アプリケーション ID のデフォルト値は、"os400.secureApplication" プロパティーに指定された値です。

鍵リング情報のデフォルト値は、"os400.secureApplication" プロパティーが指定されている場合はヌルです。 "os400.secureApplication" プロパティーが指定されていない場合、鍵リング名のデフォルト値は、"os400.certificateContainer" プロパティーに指定された値です。 "os400.secureApplication" プロパティーが指定されていない場合、鍵リング・ラベルは "os400.certificateLabel" プロパティーに値に初期設定されます。 "os400.secureApplication" プロパティーと "os400.certificateContainer" プロパティーのどちらも設定されていない場合、鍵リング名は "*SYSTEM" に初期設定されます。

メソッドの詳細

----

main

`public static void main(java.lang.String[]args)`

SSLConfiguration 関数を実行します。実行できるコマンドは、-help、-create、-display、および -update の 4 つです。コマンドは、指定される最初のパラメーターでなければなりません。

指定できるオプションは以下のとおりです (任意の順序)。

-keystore **keystore-file-name**
: 作成、更新、または表示される鍵ストア・ファイルの名前を指定します。このオプションは、すべてのコマンドに必須です。

-storepass **keystore-file-password**
: 作成、更新、または表示される鍵ストア・ファイルに関連したパスワードを指定します。このオプションは、すべてのコマンドに必須です。

-storetype **keystore-type**
: 作成、更新、または表示される鍵ストア・ファイルのタイプを指定します。このオプションは、どのコマンドにも指定できます。このオプションが指定されない場合、"IbmISeriesKeyStore" の値が使用されます。

-appid **application-identifier**
: 作成または更新される鍵ストア・ファイルを初期化するために使用されるアプリケーション ID を指定します。このオプションは、*-create* および *-update* コマンドには任意指定です。 *-appid*、*keyring*、および *-systemdefault* オプションの 1 つだけを指定できます。

-keyring **keyring-file-name**
: 作成または更新される鍵ストア・ファイルを初期化するために使用される鍵リング・ファイル名を指定します。このオプションは、*-create* および *-update* コマンドには任意指定です。 *-appid*、*keyring*、および *-systemdefault* オプションの 1 つだけを指定できます。

-keyringpass **keyring-file-password**
: 作成または更新される鍵ストア・ファイルを初期化するために使用される鍵リング・ファイル・パスワードを指定します。このオプションは、*-create* および *-update* コマンドに指定できます。また、鍵ストア・タイプに "IbmISeriesKeyStore" 以外が指定されているときには、このオプションは必須です。このオプションが指定されない場合、隠しておく鍵リング・パスワードが使用されます。

-keyringlabel **keyring-file-label**
: 作成または更新される鍵ストア・ファイルを初期化するために使用される鍵リング・ラベルを指定します。このオプションは、*-keyring* オプションも指定されている場合に限り指定できます。*keyring* オプションが指定されているときにこのオプションが指定されない場合、鍵リング内のデフォルト・ラベルが使用されます。

-systemdefault
: 作成または更新される鍵ストア・ファイルを初期化するために使用されるシステム・デフォルト値を指定します。このオプションは、*-create* および *-update* コマンドには任意指定です。 *-appid*、*keyring*、および *-systemdefault* オプションの 1 つだけを指定できます。

-v
: 冗長出力が作成されることを指定します。このオプションは、どのコマンドにも指定できます。

ヘルプ・コマンドは、パラメーターをこのメソッドに指定するための使用法情報を表示します。ヘルプ機能を呼び出すためのパラメーターは、以下のように指定されます。

```
-help
```

作成コマンドは、新規の鍵ストア・ファイルを指定します。作成コマンドには 3 つのバリエーションがあります。 1 つめのバリエーションは、特定のアプリケーション ID に基づいて鍵ストアを作成します。2 つめのバリエーションは、鍵リングの名前、ラベル、およびパスワードに基づいて鍵ストアを作成します。3 つめのバリエーションは、システム・デフォルト構成に基づいて鍵ストアを作成します。

特定のアプリケーション ID に基づいて鍵ストアを作成するには、*-appid* オプションを指定する必要があります。次のパラメーターは、名前が "keystore.file"、パスワードが "keypass" であるタイプ "IbmISeriesKeyStore" の鍵ストア・ファイルを作成します。これはアプリケーション ID "APPID" に基づいて初期設定されます。

```
-create -keystore keystore.file -storepass keypass -storetype IbmISeriesKeyStore
       -appid APPID
```

特定の鍵リング・ファイルに基づいて鍵ストアを作成するには、*-keyring* オプションを指定する必要があります。また、*-keyringpass* および *keyringlabel* オプションを指定することもできます。次のパラメーターは、名前が "keystore.file"、パスワードが "keypass" であるタイプ "IbmISeriesKeyStore" の鍵ストア・ファイルを作成します。これは、名前が "keyring.file"、鍵リング・パスワードが"ringpass"、鍵リング・ラベルが "keylabel" の鍵リング・ファイルに基づいて初期設定されます。

```
-create -keystore keystore.file -storepass keypass -storetype IbmISeriesKeyStore
       -keyring keyring.file -keyringpass ringpass -keyringlabel keylabel
```

システム・デフォルト構成に基づいて鍵ストアを作成するには、*-systemdefault* オプションを指定する必要があります。次のパラメーターは、名前が "keystore.file"、パスワードが "keypass" であるタイプ "IbmISeriesKeyStore" の鍵ストア・ファイルを作成します。これはシステム・デフォルト構成に基づいて初期設定されます。

```
-create -keystore keystore.file -storepass keypass -systemdefault
```

更新コマンドは、タイプが "IbmISeriesKeyStore" の既存の鍵ストア・ファイルを更新します。更新コマンドには、作成コマンドのバリエーションと同一の 3 つのバリエーションがあります。更新コマンドのオプションは、作成コマンドに使用したオプションと同一です。表示コマンドは、既存の鍵ストア・ファイルに指定された構成を表示します。次のパラメーターは、名前が "keystore.file"、パスワードが "keypass" であるタイプ "IbmISeriesKeyStore" の鍵ストア・ファイルで指定された構成を表示します。

```
-display -keystore keystore.file -storepass keypass -storetype IbmISeriesKeyStore
```

パラメーター:

args - コマンド行引数

----

run

```
public void run(java.lang.String[]args,
                java.io.PrintStreamout)
```

SSLConfiguration 関数を実行します。このメソッドのパラメーターおよび機能は main() メソッドと同一です。

パラメーター:

args - コマンド引数

out - 結果が書き込まれる出力ストリーム

参照: com.ibm.as400.SSLConfiguration.main()

----

getApplicationId

```
public java.lang.String getApplicationId()
```

アプリケーション ID を戻します。

戻されるもの:

アプリケーション ID。

----

getKeyringName

```
public java.lang.String getKeyringName()
```

鍵リング名を戻します。

戻されるもの:

鍵リング名。

----

getKeyringLabel

```
public java.lang.String getKeyringLabel()
```

鍵リング・ラベルを戻します。

戻されるもの:
鍵リング・ラベル。

----

getKeyringPassword

```
public final char[] getKeyringPassword()
```

鍵リング・パスワードを戻します。

戻されるもの:
鍵リング・パスワード。

----

finalize

```
protected void finalize()
                 throws java.lang.Throwable
```

ガーベッジ・コレクションが、オブジェクトの参照がこれ以上ないと判断するときに、ガーベッジ・コレクターによってオブジェクト上で呼び出されます。

オーバーライド:
クラス java.lang.Object の finalize

スロー:
java.lang.Throwable - このメソッドによって出される例外。

----

clear

```
public void clear()
```

すべての get メソッドがヌルを戻すように、オブジェクト内のすべての情報を消去します。

----

setKeyring

```
public void setKeyring(java.lang.Stringname,
                       java.lang.Stringlabel,
                       char[]password)
```

鍵リング情報を設定します。

パラメーター:
name - 鍵リング名

label - 鍵リング・ラベル。または、デフォルトの鍵リング項目が使用される場合はヌル。

password - 鍵リング・パスワード。または隠しておくパスワードが使用される場合はヌル。

----

setApplicationId

```
public void setApplicationId(java.lang.StringapplicationId)
```

アプリケーション ID を設定します。

パラメーター:
  applicationId - アプリケーション ID。

----

setApplicationId

```
public void setApplicationId(java.lang.StringapplicationId,
                             char[]password)
```

アプリケーション ID および鍵リング・パスワードを設定します。鍵リング・パスワードを指定すると、作成される鍵ストアが秘密鍵にアクセスできるようになります。

パラメーター:
  applicationId - アプリケーション ID。

  password - 鍵リング・パスワード。

----

equals

```
public boolean equals(java.lang.Objectobj)
```

他の何らかのオブジェクトがこのオブジェクトに「相当する」かどうかを示します。

オーバーライド:
  クラス java.lang.Object の equals

パラメーター:
  obj - 比較されるオブジェクト

戻されるもの:
  オブジェクトが同じ構成情報を指定するかどうかを示す標識

----

hashCode

```
public int hashCode()
```

オブジェクトのハッシュ・コード値を戻します。

オーバーライド:
  クラス java.lang.Object の hashCode

戻されるもの:
  このオブジェクトのハッシュ・コード値。

----

clone

```
public java.lang.Object clone()
```

この SSL 構成の新規コピーを生成します。この SSL 構成のコンポーネントに対するこれ以降の変更は新規コピーに影響を与えません。逆の場合も同じです。

オーバーライド:
クラス java.lang.Object の clone

戻されるもの:
この SSL 構成のコピー

----

getKeyStore

```
public java.security.KeyStore getKeyStore(char[]password)
                                   throws java.security.KeyStoreException
```

与えられたパスワードを使用して、タイプが "IbmISeriesKeyStore" の鍵ストアを戻します。鍵ストアは、オブジェクトに現在保管されている構成情報に基づいて初期設定されます。

パラメーター:
password - 鍵ストアの初期設定に使用されます

戻されるもの:
オブジェクトに現在保管されている構成情報に基づいて初期設定される KeyStore 鍵ストア

スロー:
java.security.KeyStoreException - 鍵ストアを作成できなかった場合

----

getKeyStore

```
public java.security.KeyStore getKeyStore(java.lang.Stringtype,
                                          char[]password)
                                   throws java.security.KeyStoreException
```

与えられたパスワードを使用して、要求されたタイプの鍵ストアを戻します。鍵ストアは、オブジェクトに現在保管されている構成情報に基づいて初期設定されます。

パラメーター:
type - 戻される鍵ストアのタイプ

password - 鍵ストアの初期設定に使用されます

戻されるもの:
オブジェクトに現在保管されている構成情報に基づいて初期設定される KeyStore 鍵ストア

スロー:
java.security.KeyStoreException - 鍵ストアを作成できなかった場合

例: IBM Java Secure Sockets Extension 1.4:

JSSE の例では、クライアントおよびサーバーがネイティブ IBM i JSSE プロバイダーを使用して、安全な通信を可能にするコンテキストを作成する方法を示しています。

注: いずれの例でも、java.security ファイルの指定するプロパティーにかかわず、ネイティブ IBM i JSSE プロバイダーを使用します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

例: バージョン 1.4 の SSLContext オブジェクトを使用する SSL クライアント:

このクライアント・プログラムの例では、"MY_CLIENT_APP" アプリケーション ID を使用するために初期化を行う、SSLContext オブジェクトを使用します。このプログラムでは、java.security ファイルにおける指定の有無にかかわらず、ネイティブ IBM i インプリメンテーションを使用します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
////////////////////////////////////////////////////////////////////////////////
//
// This example client program utilizes an SSLContext object, which it initializes
// to use the "MY_CLIENT_APP" application ID.
//
// The example uses the native IBM i JSSE provider, regardless of the
// properties specified by the java.security file.
//
// Command syntax:
//    java SslClient
//
//////////////////////////////////////////////////////////////////////////////////

import java.io.*;
import javax.net.ssl.*;
import java.security.*;
import import com.ibm.i5os.jsse.SSLConfiguration;

/**
 * SSL Client Program.
 */
public class SslClient {

   /**
    * SslClient main method.
    *
    * @param args the command line arguments (not used)
    */
   public static void main(String args[]) {
       /*
        * Set up to catch any exceptions thrown.
        */
       try {
           /*
            * Initialize an SSLConfiguration object to specify an application
            * ID.  "MY_CLIENT_APP" must be registered and configured
            * correctly with the Digital Certificate Manager (DCM).
            */
           SSLConfiguration config = new SSLConfiguration();
           config.setApplicationId("MY_CLIENT_APP");
           /*
            * Get a KeyStore object from the SSLConfiguration object.
            */
           char[] password = "password".toCharArray();
           KeyStore ks = config.getKeyStore(password);
           /*
            * Allocate and initialize a KeyManagerFactory.
            */
           KeyManagerFactory kmf =
                             KeyManagerFactory.getInstance("IbmISeriesX509");
           kmf.init(ks, password);
           /*
            * Allocate and initialize a TrustManagerFactory.
```

```
             */
            TrustManagerFactory tmf =
                            TrustManagerFactory.getInstance("IbmISeriesX509");
            tmf.init(ks);
            /*
             * Allocate and initialize an SSLContext.
             */
            SSLContext c =
                       SLContext.getInstance("SSL", "IBMi5OSJSSEProvider");
            c.init(kmf.getKeyManagers(), tmf.getTrustManagers(), null);
            /*
             * Get the SSLSocketFactory from the SSLContext.
             */
            SSLSocketFactory sf = c.getSocketFactory();
            /*
             * Create an SSLSocket.
             *
             * Change the hard-coded IP address to the IP address or host name
             * of the server.
             */
            SSLSocket s = (SSLSocket) sf.createSocket("1.1.1.1", 13333);
            /*
             * Send a message to the server using the secure session.
             */
            String sent = "Test of java SSL write";
            OutputStream os = s.getOutputStream();
            os.write(sent.getBytes());
            /*
             * Write results to screen.
             */
            System.out.println("Wrote " + sent.length() + " bytes...");
            System.out.println(sent);
            /*
             * Receive a message from the server using the secure session.
             */
            InputStream is = s.getInputStream();
            byte[] buffer = new byte[1024];
            int bytesRead = is.read(buffer);
            if (bytesRead == -1)
                throw new IOException("Unexpected End-of-file Received");
            String received = new String(buffer, 0, bytesRead);
            /*
             * Write results to screen.
             */
            System.out.println("Read " + received.length() + " bytes...");
            System.out.println(received);
        } catch (Exception e) {
            System.out.println("Unexpected exception caught: " +
                                   e.getMessage());
            e.printStackTrace();
        }
    }

}
```

例: バージョン 1.4 の SSLContext オブジェクトを使用する SSL サーバー:

以下のサーバー・プログラムは、過去に作成された鍵ストア・ファイルによって初期化を行う、SSLContext オブジェクトを使用します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
///////////////////////////////////////////////////////////////////////////////////
//
// The following server program utilizes an SSLContext object that it
```

```
// initializes with a previously created keystore file.
//
// The keystore file has the following name and keystore password:
//   File name: /home/keystore.file
//   Password:  password
//
// The example program needs the keystore file in order to create an
// IbmISeriesKeyStore object. The KeyStore object must specify MY_SERVER_APP as
// the application identifier.
//
// To create the keystore file, you can use the following Qshell command:
//
//   java com.ibm.i5os.SSLConfiguration  -create  -keystore /home/keystore.file
//     -storepass password  -appid MY_SERVER_APP
//
// Command syntax:
//     java  JavaSslServer
//
// You can also create the keystore file by entering this command at the CL command prompt:
//
//  RUNJVA CLASS(com.ibm.i5os.SSLConfiguration) PARM('-create' '-keystore'
//      '/home/keystore.file' '-storepass' 'password' '-appid' 'MY_SERVER_APP')
//
////////////////////////////////////////////////////////////////////////////////

import java.io.*;
import javax.net.ssl.*;
import java.security.*;
/**
* Java SSL Server Program using Application ID.
*/
public class JavaSslServer {

   /**
    * JavaSslServer main method.
    *
    * @param args the command line arguments (not used)
    */
   public static void main(String args[]) {
       /*
        * Set up to catch any exceptions thrown.
        */
       try {
           /*
            * Allocate and initialize a KeyStore object.
            */
           char[] password = "password".toCharArray();
           KeyStore ks = KeyStore.getInstance("IbmISeriesKeyStore");
           FileInputStream fis = new FileInputStream("/home/keystore.file");
           ks.load(fis, password);
           /*
            * Allocate and initialize a KeyManagerFactory.
            */
           KeyManagerFactory kmf =
                            KeyManagerFactory.getInstance("IbmISeriesX509");
           kmf.init(ks, password);
           /*
            * Allocate and initialize a TrustManagerFactory.
            */
           TrustManagerFactory tmf =
                          TrustManagerFactory.getInstance("IbmISeriesX509");
           tmf.init(ks);
           /*
            * Allocate and initialize an SSLContext.
            */
           SSLContext c =
                      SSLContext.getInstance("SSL", "IBMi5OSJSSEProvider");
```

```
            c.init(kmf.getKeyManagers(), tmf.getTrustManagers(), null);
            /*
             * Get the an SSLServerSocketFactory from the SSLContext.
             */
            SSLServerSocketFactory sf = c.getServerSocketFactory();
            /*
             * Create an SSLServerSocket.
             */
            SSLServerSocket ss =
                (SSLServerSocket) sf.createServerSocket(13333);
            /*
             * Perform an accept() to create an SSLSocket.
             */
            SSLSocket s = (SSLSocket) ss.accept();
            /*
             * Receive a message from the client using the secure session.
             */
            InputStream is = s.getInputStream();
            byte[] buffer = new byte[1024];
            int bytesRead = is.read(buffer);
            if (bytesRead == -1)
                throw new IOException("Unexpected End-of-file Received");
            String received = new String(buffer, 0, bytesRead);
            /*
             * Write results to screen.
             */
            System.out.println("Read " + received.length() + " bytes...");
            System.out.println(received);
            /*
             * Echo the message back to the client using the secure session.
             */
            OutputStream os = s.getOutputStream();
            os.write(received.getBytes());
            /*
             * Write results to screen.
             */
            System.out.println("Wrote " + received.length() + " bytes...");
            System.out.println(received);
        } catch (Exception e) {
            System.out.println("Unexpected exception caught: " +
                               e.getMessage());
            e.printStackTrace();
        }
    }

}
```

Java Secure Socket Extension 1.5 の使用

この情報は、Java 2 Platform, Standard Edition (J2SE) バージョン 1.5 以降のリリースが稼働するシステムで JSSE を使用する場合にのみ適用されます。JSSE は、SSL と TLS の両方の基礎となるメカニズムを要約するフレームワークと似ています。基礎となっているプロトコルの複雑さと特色を要約することによって、JSSE はプログラマーが安全で暗号化された通信を使用できるようにすると同時に、セキュリティーのぜい弱性を最小限に抑えます。 Java Secure Socket Extension (JSSE) は、Secure Sockets Layer (SSL) プロトコルと Transport Layer Security (TLS) プロトコルの両方を使用して、クライアントとサーバーの間に安全で暗号化された通信を提供します。

IBM JSSE インプリメンテーションは IBM JSSE と呼ばれています。 IBM JSSE には、ネイティブ IBM i JSSE プロバイダーと IBM 純正の Java JSSE プロバイダーが組み込まれています。また、Sun Microsystems, Inc. JSSE は初めに IBM i サーバーに同梱で提供され、その後も継続して提供されます。

JSSE 1.5 をサポートするようにサーバーを構成する:

異なる JSSE インプリメンテーションを使用するように IBM i を構成します。このトピックには、ソフトウェア要件、JSSE プロバイダーの変更方法、および必要なセキュリティー・プロパティーとシステム・プロパティーに関する情報もあります。デフォルトの構成では、IBMJSSE2 という IBM 純正の Java JSSE プロバイダーを使用します。

| ご使用の IBM i サーバーで Java 2 Platform, Standard Edition (J2SE) バージョン 1.5 を使用している場
| 合、JSSE は既に構成済みです。デフォルトの構成では、IBM 純正の Java JSSE プロバイダーを使用しま
| す。

JSSE プロバイダーの変更

IBM 純正の Java JSSE プロバイダーの代わりにネイティブ IBM i JSSE プロバイダーや Sun Microsystems, Inc. の JSSE プロバイダーを使用するように JSSE を構成することもできます。いくつかの特定の JSSE セキュリティー・プロパティーと Java システム・プロパティーを変更することによって、プロバイダーの間の切り替えができます。

セキュリティー・マネージャー

Java セキュリティー・マネージャーを使用可能にして JSSE アプリケーションを実行している場合、使用可能なネットワーク許可を設定する必要がある可能性があります。詳しくは、Permissions in the Java 2 SDK の SSL Permission の説明を参照してください。

JSSE 1.5 プロバイダー:

IBM JSSE には、ネイティブ IBM i JSSE プロバイダーと IBM 純正の Java JSSE プロバイダーが組み込まれています。Sun Microsystems, Inc. JSSE は、IBM i サーバーにも同梱されています。どのプロバイダーを選択して使用するかは、アプリケーションの必要に応じて異なります。

プロバイダーはすべて JSSE インターフェースの仕様に準拠します。これらは双方向通信が可能であったり、任意の他の SSL または TLS インプリメンテーション (非 Java インプリメンテーションでも可) と通信することができます。

Sun Microsystems, Inc. 純正の Java JSSE プロバイダー

これは、JSSE の Sun Java インプリメンテーションです。この JSSE は初めに IBM i に同梱で提供され、その後も継続して提供されます。Sun Microsystems, Inc. JSSE について詳しくは、Sun Microsystems, Inc. による「Java Secure Socket Extension (JSSE) Reference Guide 」を参照してください。

IBM 純正の Java JSSE プロバイダー

IBM 純正の Java JSSE プロバイダーは、以下のフィーチャーを備えています。

- ディジタル証明書を制御および構成するためのあらゆるタイプの KeyStore オブジェクト (たとえば、JKS および PKCS12 など) を処理します。
- 複数のインプリメンテーションの JSSE コンポーネントを組み合わせて一緒に使用することができます。

IBMJSSEProvider2 はピュア Java 実装のプロバイダー名です。このプロバイダー名は、適切なケースを使用して java.security.Security.getProvider() メソッドか、あるいは、いくつかの JSSE クラス用の種々の getInstance() メソッドに渡す必要があります。

ネイティブ IBM i JSSE プロバイダー

ネイティブ IBM i JSSE プロバイダーには、以下の機能があります。

- ネイティブ IBM i SSL サポートを使用します。
- ディジタル証明書を構成、制御するために、ディジタル証明書マネージャーの使用を許可します。これは、固有な IBM i タイプの KeyStore (IbmISeriesKeyStore) を介して提供されます。
- 複数のインプリメンテーションの JSSE コンポーネントを組み合わせて一緒に使用することができます。

IBMi5OSJSSEProvider は、ネイティブ IBM i 実装の名前です。このプロバイダー名は、適切なケースを使用して java.security.Security.getProvider() メソッドか、あるいは、いくつかの JSSE クラス用の種々の getInstance() メソッドに渡す必要があります。

デフォルト JSSE プロバイダーの変更

セキュリティー・プロパティーに適切な変更を加えることによって、デフォルトの JSSE プロバイダーを変更することができます。詳しくは、『JSSE 1.5 セキュリティー・プロパティー』を参照してください。

JSSE プロバイダーを変更した後は、システム・プロパティーが、新しいプロバイダーが必要とするディジタル証明書情報 (鍵ストア) 用の適切な構成を指定していることを確認します。詳しくは、297 ページの『JSSE for 1.5 Java システム・プロパティー』を参照してください。

JSSE 1.5 セキュリティー・プロパティー:

Java 仮想マシン (JVM) は、多数の重要なセキュリティー・プロパティーを使用します。これらのセキュリティー・プロパティーは、Java マスター・セキュリティー・プロパティー・ファイルを編集することによって設定されます。

| java.security というこのファイルは、通常はサーバー上の /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit/
| jre/lib/security または /QOpenSys/QIBM/ProdData/JavaVM/jdk50/64bit/jre/lib/security ディレクトリ
| ーにあります。

以下のリストは、JSSE を使用するための、関連するいくつかのセキュリティー・プロパティーを示しています。この説明は、java.security ファイルを編集するためのガイドとして使用してください。

security.provider.<integer>

使用する JSSE プロバイダー。これは静的に暗号プロバイダー・クラスの登録も行います。以下の例のように、異なる JSSE プロバイダーを正確に指定してください。

```
security.provider.5=com.ibm.i5os.jsse.JSSEProvider
security.provider.6=com.ibm.jsse2.IBMJSSEProvider2
security.provider.7=com.sun.net.ssl.internal.ssl.Provider
```

ssl.KeyManagerFactory.algorithm

デフォルトの KeyManagerFactory アルゴリズムを指定します。ネイティブ IBM i JSSE プロバイダーの場合は、以下を使用します。

```
ssl.KeyManagerFactory.algorithm=IbmISeriesX509
```

IBM 純正の Java JSSE プロバイダーの場合は、以下を使用します。

```
ssl.KeyManagerFactory.algorithm=IbmX509
```

Sun Microsystems, Inc. 純正の Java JSSE プロバイダーの場合は、以下を使用します。

```
ssl.KeyManagerFactory.algorithm=SunX509
```

詳しくは、javax.net.ssl.KeyManagerFactory の Javadoc を参照してください。

ssl.TrustManagerFactory.algorithm

デフォルトの TrustManagerFactory アルゴリズムを指定します。ネイティブ IBM i JSSE プロバイダーの場合は、以下を使用します。

```
ssl.TrustManagerFactory.algorithm=IbmISeriesX509
```

IBM 純正の Java JSSE プロバイダーの場合は、以下を使用します。

```
ssl.TrustManagerFactory.algorithm=IbmX509
```

詳しくは、javax.net.ssl.TrustManagerFactory の Javadoc を参照してください。

ssl.SocketFactory.provider

デフォルトの SSL ソケット・ファクトリーを指定します。ネイティブ IBM i JSSE プロバイダーの場合は、以下を使用します。

```
ssl.SocketFactory.provider=com.ibm.i5os.jsse.JSSESocketFactory
```

IBM 純正の Java JSSE プロバイダーの場合は、以下を使用します。

```
ssl.SocketFactory.provider=com.ibm.jsse2.SSLSocketFactoryImpl
```

詳しくは、javax.net.ssl.SSLSocketFactory の Javadoc を参照してください。

ssl.ServerSocketFactory.provider

デフォルトの SSL サーバー・ソケット・ファクトリーを指定します。ネイティブ IBM i JSSE プロバイダーの場合は、以下を使用します。

```
ssl.ServerSocketFactory.provider=com.ibm.i5os.jsse.JSSEServerSocketFactory
```

純正の Java JSSE プロバイダーの場合は、以下を使用します。

```
ssl.ServerSocketFactory.provider=com.ibm.jsse2.SSLServerSocketFactoryImpl
```

詳しくは、javax.net.ssl.SSLServerSocketFactory の Javadoc を参照してください。

関連情報

javax.net.ssl.KeyManagerFactory Javadoc

javax.net.ssl.TrustManagerFactory Javadoc

javax.net.ssl.SSLSocketFactory Javadoc

javax.net.ssl.SSLServerSocketFactory Javadoc

JSSE for 1.5 Java システム・プロパティー:

アプリケーションで JSSE を使用するには、デフォルトの SSLContext オブジェクトが構成の確認を行うために必要な、いくつかのシステム・プロパティーを指定する必要があります。いくつかのプロパティーはすべてのプロバイダーに適用され、その他はネイティブ IBM i プロバイダーにだけ適用されます。

ネイティブ IBM i JSSE プロバイダーを使用するとき、プロパティーを指定しない場合は、os400.certificateContainer がデフォルトの *SYSTEM になりますが、それは、JSSE がシステム証明書ストア内のデフォルトのエントリーを使用することを意味します。

ネイティブ IBM i JSSE プロバイダーと IBM 純正 Java JSSE プロバイダーに適用されるプロパティー

以下のプロパティーは両方の JSSE プロバイダーに適用されます。それぞれの説明では、該当する場合には、デフォルトのプロパティーも示しています。

javax.net.ssl.trustStore

デフォルトの TrustManager が使用する KeyStore オブジェクトが入っているファイルの名前。デフォルト値は jssecacerts または (jssecacerets が存在しない場合は) cacerts です。

javax.net.ssl.trustStoreType

デフォルトの TrustManager が使用する KeyStore オブジェクトのタイプ。デフォルト値は KeyStore.getDefaultType メソッドによって戻される値です。

javax.net.ssl.trustStorePassword

デフォルトの TrustManager が使用する KeyStore オブジェクトのパスワード。

javax.net.ssl.keyStore

デフォルトの KeyManager が使用する KeyStore オブジェクトが入っているファイルの名前。デフォルト値は jssecacerts または (jssecacerets が存在しない場合は) cacerts です。

javax.net.ssl.keyStoreType

デフォルトの KeyManager が使用する KeyStore オブジェクトのタイプ。デフォルト値は KeyStore.getDefaultType メソッドによって戻される値です。

javax.net.ssl.keyStorePassword

デフォルトの KeyManager が使用する KeyStore オブジェクトのパスワード。

ネイティブ IBM i JSSE プロバイダーにのみ適用されるプロパティー

以下のプロパティーは、ネイティブ IBM i JSSE プロバイダーにのみ適用されます。

os400.secureApplication

アプリケーション ID。 JSSE は、以下のいずれかのプロパティーが指定されていない場合にのみ、このプロパティーを使用します。

- javax.net.ssl.keyStore
- javax.net.ssl.keyStorePassword
- javax.net.ssl.keyStoreType
- javax.net.ssl.trustStore

- javax.net.ssl.trustStorePassword
- javax.ssl.net.trustStoreType

os400.certificateContainer

使用する鍵リングの名前。 JSSE は、以下のいずれかのプロパティーが指定されていない場合にのみ、このプロパティーを使用します。

- javax.net.ssl.keyStore
- javax.net.ssl.keyStorePassword
- javax.net.ssl.keyStoreType
- javax.net.ssl.trustStore
- javax.net.ssl.trustStorePassword
- javax.ssl.net.trustStoreType
- os400.secureApplication

os400.certificateLabel

使用する鍵リング・ラベル。 JSSE は、以下のいずれかのプロパティーが指定されていない場合にのみ、このプロパティーを使用します。

- javax.net.ssl.keyStore
- javax.net.ssl.keyStorePassword
- javax.net.ssl.trustStore
- javax.net.ssl.trustStorePassword
- javax.ssl.net.trustStoreType
- os400.secureApplication

関連概念

15 ページの『Java システム・プロパティーのリスト』
Java システム・プロパティーにより、Java プログラムのランタイム環境が決まります。 Java システム・プロパティーは、IBM i のシステム値や環境変数と似ています。

関連情報

Sun Microsystems, Inc. 「System Properties」

ネイティブ IBM i JSSE 1.5 プロバイダーの使用:

ネイティブ IBM i JSSE プロバイダーは、JSSE クラスおよびインターフェースの一式を備えています。これには JSSE KeyStore クラスおよび SSLConfiguration クラスの実装が含まれています。

SSLContext.getInstance メソッドのプロトコル値

以下の表では、ネイティブ IBM i JSSE プロバイダーの SSLContext.getInstance メソッドのプロトコル値を示し、それを説明しています。

サポートされる SSL プロトコルは、システムに設定されているシステム値によって制限を受けることがあります。詳細は、『システム管理』情報の「Security system values: Secure Sockets Layer protocols」サブトピックを参照してください。

プロトコル値	サポートされている SSL プロトコル
SSL	SSL バージョン 2、SSL バージョン 3、および TLS バージョン 1。SSLv2 形式の Hello にカプセル化された SSLv3 または TLSv1 の Hello を受け入れます。
SSLv2	SSL バージョン 2
SSLv3	SSL バージョン 3 プロトコル。SSLv2 形式の Hello にカプセル化された SSLv3 Hello を受け入れます。
TLS	SSL バージョン 2、SSL バージョン 3、および TLS バージョン 1。SSLv2 形式の Hello にカプセル化された SSLv3 または TLSv1 の Hello を受け入れます。
TLSv1	TLS バージョン 1 プロトコル。Request for Comments (RFC) 2246 で定義されています。 SSLv2 形式の Hello にカプセル化された TLSv1 Hello を受け入れます。
SSL_TLS	SSL バージョン 2、SSL バージョン 3、および TLS バージョン 1。SSLv2 形式の Hello にカプセル化された SSLv3 または TLSv1 の Hello を受け入れます。

ネイティブ IBM i KeyStore のインプリメンテーション

ネイティブ IBM i プロバイダーは、IbmISeriesKeyStore と IBMi5OSKeyStore という 2 つの KeyStore クラスのインプリメンテーションを提供します。どちらの KeyStore のインプリメンテーションも、ディジタル証明書マネージャー (DCM) サポートのまわりにラッパーを提供します。

IbmISeriesKeyStore
: 鍵ストアの内容は、特定のアプリケーション ID または鍵リング・ファイル、パスワード、およびラベルに応じて異なります。 JSSE はディジタル証明書マネージャーから鍵ストア・エントリーをロードします。エントリーをロードするため、JSSE はアプリケーションが最初に鍵ストア・エントリーまたは鍵ストア情報にアクセスしようとするときに、適切なアプリケーション ID または鍵リング情報を使用します。鍵ストアは変更できず、構成の変更はすべてディジタル証明書マネージャーを使用して行わなければなりません。

IBMi5OSKeyStore
: この鍵ストアの内容は、i5OS 証明書ストア・ファイルとそのファイルにアクセスするためのパスワードに基づいています。この KeyStore クラスでは、証明書ストアの変更ができます。ディジタル証明書マネージャーを使用せずに行うことができます。

  IBMi5OSKeyStore のインプリメンテーションは、Sun Microsystems, Inc. の Java KeyStore API の仕様に準拠しています。詳細は、Sun Microsystems, Inc. の Keystore Javadoc の情報を参照してください。

DCM を使用して鍵ストアを管理する方法について詳しくは、『ディジタル証明書マネージャー』のトピックを参照してください。

関連情報

Security system values: Secure Sockets Layer protocols

i5OSLoadStoreParameter クラスの Javadoc 情報:

com.ibm.i5os.keystore

クラス i5OSLoadStoreParameter

```
java.lang.Object
  |
  +--com.ibm.i5os.keystore.i5OSLoadStoreParameter
```

インプリメントされているすべてのインターフェース:

java.security.Keystore.LoadStoreParameter

```
public class i5OSLoadStoreParameter
extends java.lang.Object
implements java.security.Keystore.LoadStoreParameter
```

このクラスは、IBM i 証明書ストアのロード/保管に使用できる KeyStore.ProtectionParameter オブジェクトを作成します。作成後、このクラスは、アクセスされる証明書ストアについての情報とその証明書ストアを保護するために使用されるパスワードの情報を提供します。

このクラスの使用の例を以下に示します。

```
//initialize the keystore
    KeyStore ks = KeyStore.getInstance("IBMi5OSKeyStore");

 //Load an existing keystore
    File kdbFile = new File("/tmp/certificateStore.kdb");
    i5OSLoadStoreParameter lsp =
   new i5OSLoadStoreParameter (kdbFile, "password".toCharArray());
    ks.load(lsp);

 //Get and Add entries to the certificate store
    ...

 //Save the certificate store
    Ks.store(lsp);
```

対象: SDK 1.5

コンストラクターの要約

i5OSLoadStoreParameter(java.io.File ksFile, char[] password)

KeyStore ファイルから ProtectionParameter のインスタンスを作成し、i5OS 証明書ストアのロード/保管に使用するパスワードを作成します。

i5OSLoadStoreParameter(java.io.File ksFile, java.security.KeyStore.PasswordProtection pwdProtParam)

KeyStore ファイルから ProtectionParameter のインスタンスを作成し、i5OS 証明書ストアのロード/保管に使用する PasswordProtection を作成します。

表 11. **メソッドの要約**

java.security.KeyStore.ProtectionParameter	*303 ページの『getProtectionParameter』*() は、この LoadStoreParameter に関連付けられている KeyStore.KeyStoreParameter を戻します。

クラス java.lang.Object から継承されるメソッド

```
clone、equals、finalize、getClass、hashCode、notify、notifyAll、toString、wait、wait、wait
```

コンストラクターの詳細

i5OSLoadStoreParameter

```
public i5OSLoadStoreParameter(java.io.File ksFile,
                              char[] password)
                       throws java.lang.IllegalArgumentException
```

KeyStore ファイルから ProtectionParameter のインスタンスを作成し、i5OS 証明書ストアのロード/保管に使用するパスワードを作成します。

パラメーター:

ksFile - KeyStore の File オブジェクト。

keystore.load() が i5OSLoadStoreParameter(ksFile = null, password) で使用された場合は、新規の鍵ストアが作成されます。

keystore.store() が i5OSLoadStoreParameter(ksFile = null, password) で使用された場合は、IllegalArgumentException がスローされます。

password - i5OS 証明書ストアにアクセスするためのパスワード。 NULL または空にすることはできません。

スロー:

java.lang.IllegalArgumentException - パスワードが NULL または空の場合

----

i5OSLoadStoreParameter

```
public i5OSLoadStoreParameter(java.io.File ksFile,
                              java.security.KeyStore.PasswordProtection pwdProtParam)
                       throws java.lang.IllegalArgumentException
```

KeyStore ファイルから ProtectionParameter のインスタンスを作成し、i5OS 証明書ストアのロード/保管に使用する PasswordProtection を作成します。

keystore.load() が i5OSLoadStoreParameter(ksFile = null, password) で使用された場合は、新規の鍵ストアが作成されます。

keystore.store() が i5OSLoadStoreParameter(ksFile = null, password) で使用された場合は、IllegalArgumentException がスローされます。

パラメーター:

ksFile - KeyStore の File オブジェクト。

pwdProtParam - パスワードの獲得に使用される PasswordProtection インスタンス。NULL にすることはできません。

スロー:

java.lang.IllegalArgumentException - KeyStore.PasswordProtection が NULL の場合、または pwdProtParam にあるパスワードが NULL か空である場合

メソッドの詳細

----

getProtectionParameter

```
public java.security.KeyStore.ProtectionParameter getProtectionParameter()
```

この LoadStoreParameter に関連付けられている KeyStore.KeyStoreParameter を戻します。

指定するもの:

インターフェース `java.security.KeyStore.LoadStoreParameter` の `getProtectionParameter`

戻されるもの:

KeyStore.KeyStoreParameter インターフェースをインプリメントするインスタンス

参照:

`java.security.KeyStore.ProtectionParameter#getProtectionParameter()`

i5OSSystemCertificateStoreFile クラスの Javadoc 情報:

com.ibm.i5os.keystore

クラス i5OSSystemCertificateStoreFile

```
java.lang.Object
   java.io.File
       com.ibm.i5os.keystore.i5OSSystemCertificateStoreFile
```

インプリメントされているすべてのインターフェース:

java.io.Serializable、java.lang.Comparable<java.io.File>

```
public class i5OSSystemCertificateStoreFile
extends java.io.File
```

このクラスは、*SYSTEM 証明書ストア・ファイルを指す新しい File インプリメンテーションを提供します。このクラスは、ユーザーが実際のストアのパスを知らなくても *SYSTEM 証明書ストアをロードできるメカニズムを備えています。

*SYSTEM 証明書ストアを鍵ストアにロードするには、まず最初に i5OSSystemCertificateStoreFile を作成します。

ここから、2 つの方法で鍵ストアへのロードを行うことができます。

- i5OSLoadStoreParameter を使用する方法:

```
//create an i5OSSystemCertificateStoreFile
     File starSystemFile = new i5OSSystemCertificateStoreFile();

  //use that file to create an i5OSLoadStoreParameter
     i5OSLoadStoreParameter lsp = new i5OSLoadStoreParameter(starSystemFile, pwd);

  //load the certificate store into a keystore
     KeyStore ks = KeyStore.getInstance("IBMi5OSKeyStore");
     ks.load(lsp);
```

- FileInputStream を使用する方法:

```
//create an i5OSSystemCertificateStoreFile
     File starSystemFile = new i5OSSystemCertificateStoreFile();

  //create an input stream to the starSystemFile
     FileInputStream fis = new FileInputStream(starSystemFile);

  //load the certificate store into a keystore
     KeyStore ks = KeyStore.getInstance("IBMi5OSKeyStore");
     ks.load(fis, pwd);
```

対象: SDK 1.5

参照: *逐次化された形式*

フィールドの要約

クラス java.io.File から継承されるフィールド
pathSeparator、pathSeparatorChar、separator、separatorChar

コンストラクターの要約

i5OSSystemCertificateStoreFile()

*SYSTEM 証明書ストア・ファイルを指す File() を作成します。

メソッドの要約

クラス java.io.File から継承されるメソッド
canRead、canWrite、compareTo、createNewFile、createTempFile、createTempFile、delete、deleteOnExit、equals、exists、getAbsoluteFile、getAbsolutePath、getCanonicalFile、getCanonicalPath、getName、getParent、getParentFile、getPath、hashCode、isAbsolute、isDirectory、isFile、isHidden、lastModified、length、list、list、listFiles、listFiles、listFiles、listRoots、mkdir、mkdirs、renameTo、setLastModified、setReadOnly、toString、toURI、toURL

クラス java.lang.Object から継承されるメソッド
clone、finalize、getClass、notify、notifyAll、wait、wait、wait

コンストラクターの詳細

i5OSSystemCertificateStoreFile

public **i5OSSystemCertificateStoreFile**()

*SYSTEM 証明書ストア・ファイルを指す File() を作成します。

バージョン 1.5 の SSLConfiguration Javadoc 情報:

com.ibm.i5os.jsse

クラス SSLConfiguration

```
java.lang.Object
  |
  +--com.ibm.i5os.jsse.SSLConfiguration
```

インプリメントされているすべてのインターフェース:

java.lang.Cloneable, javax.net.ssl.ManagerFactoryParameters

public final class **SSLConfiguration**
extends java.lang.Object
implements javax.net.ssl.ManagerFactoryParameters, java.lang.Cloneable

このクラスは、ネイティブ IBM i JSSE インプリメンテーションによって必要とされる構成の仕様を提供します。

ネイティブ IBM i JSSE インプリメンテーションは、タイプが "IbmISeriesKeyStore" の KeyStore オブジェクトを使用すると、最も効率的に作動します。このタイプの KeyStore オブジェクトには、ディジタル証明書マネージャー (DCM) に登録されたアプリケーション ID かまたは鍵リング・ファイル (ディジタル証明書コンテナー) に基づいた、キー項目および信頼できる証明書項目が含まれます。また、このタイプの KeyStore オブジェクトを使用して、"IBMi5OSJSSEProvider" Provider から X509KeyManger および X509TrustManager オブジェクトを初期化することができます。さらに、X509KeyManager および X509TrustManager オブジェクトを使用して、"IBMi5OSJSSEProvider" から SSLContext オブジェクトを初期化することができます。次に SSLContext オブジェクトは、KeyStore オブジェクトに指定された構成情報に基づいて、ネイティブ IBM i JSSE インプリメンテーションへのアクセスを提供します。"IbmISeriesKeyStore" KeyStore のロードが実行されるたびに、アプリケーション ID または鍵リング・ファイルに指定された現行の構成に基づいて、KeyStore が初期化されます。

また、このクラスを使用して、任意の有効なタイプの KeyStore オブジェクトを生成することもできます。KeyStore は、アプリケーション ID または鍵リング・ファイルに指定された現行の構成に基づいて初期化されます。アプリケーション ID または鍵リング・ファイルによって指定された構成に何らかの変更を加えた場合、その変更を反映するために、KeyStore オブジェクトを再生成する必要があります。"IbmISeriesKeyStore" 以外のタイプの KeyStore を正常に作成できるようにするには、鍵リング・パスワードを (アプリケーション ID を使用する場合は、*SYSTEM 証明書ストアに) 指定する必要があることに注意してください。作成される "IbmISeriesKeyStore" タイプの KeyStore 用の秘密鍵に正常にアクセスできるようにするには、鍵リング・パスワードを指定する必要があります。

対象: SDK 1.5

参照: KeyStore, X509KeyManager, X509TrustManager, SSLContext

コンストラクターの要約

SSLConfiguration() 新規の SSLConfiguration を作成します。詳しくは、 306 ページの『コンストラクターの詳細』を参照してください。

表 12. **メソッドの要約**

void	310 ページの『clear』() すべての get メソッドがヌルを戻すように、オブジェクト内のすべての情報を消去します。
java.lang.Object	311 ページの『clone』() この SSL 構成の新規コピーを生成します。

表 12. **メソッドの要約** *(続き)*

boolean	311 ページの『equals』(java.lang.Objectobj) 他の何らかのオブジェクトがこのオブジェクトに「相当する」かどうかを示します。
protected void	309 ページの『finalize』() ガーベッジ・コレクションが、オブジェクトの参照がこれ以上ないと判断するときに、ガーベッジ・コレクターによってオブジェクト上で呼び出されます。
java.lang.String	309 ページの『getApplicationId』() アプリケーション ID を戻します。
java.lang.String	309 ページの『getKeyringLabel』() 鍵リング・ラベルを戻します。
java.lang.String	309 ページの『getKeyringName』() 鍵リング名を戻します。
char[]	309 ページの『getKeyringPassword』() 鍵リング・パスワードを戻します。
java.security.KeyStore	311 ページの『getKeyStore』(char[]password) 与えられたパスワードを使用して、タイプが "IbmISeriesKeyStore" の鍵ストアを戻します。
java.security.KeyStore	312 ページの『 getKeyStore』(java.lang.Stringtype, char[]password) 与えられたパスワードを使用して、要求されたタイプの鍵ストアを戻します。
int	311 ページの『hashCode』() オブジェクトのハッシュ・コード値を戻します。
staticvoid	(java.lang.String[]args) SSLConfiguration 関数を実行します。
void	(java.lang.String[]args, java.io.PrintStreamout) SSLConfiguration 関数を実行します。
void	310 ページの『setApplicationId』(java.lang.StringapplicationId) アプリケーション ID を設定します。
void	310 ページの『setApplicationId』(java.lang.StringapplicationId, char[]password) アプリケーション ID および鍵リング・パスワードを設定します。
void	310 ページの『setKeyring』(java.lang.Stringname,java.lang.Stringlabel, char[]password) 鍵リング情報を設定します。

クラス java.lang.Object から継承されるメソッド
getClass, notify, notifyAll, toString, wait, wait, wait

コンストラクターの詳細

SSLConfiguration

public **SSLConfiguration**()

新規の SSLConfiguration を作成します。アプリケーション ID および鍵リング情報はデフォルト値に初期設定されます。

アプリケーション ID のデフォルト値は、"os400.secureApplication" プロパティーに指定された値です。

鍵リング情報のデフォルト値は、"os400.secureApplication" プロパティーが指定されている場合はヌルです。 "os400.secureApplication" プロパティーが指定されていない場合、鍵リング名のデフォルト値は、"os400.certificateContainer" プロパティーに指定された値です。 "os400.secureApplication" プロパティーが指定されていない場合、鍵リング・ラベルは "os400.certificateLabel" プロパティーに値に初期設定されます。

"os400.secureApplication" プロパティーと "os400.certificateContainer" プロパティーのどちらも設定されていない場合、鍵リング名は "*SYSTEM" に初期設定されます。

メソッドの詳細

----

main

```
public static void main(java.lang.String[]args)
```

SSLConfiguration 関数を実行します。実行できるコマンドは、-help、-create、-display、および -update の 4 つです。コマンドは、指定される最初のパラメーターでなければなりません。

指定できるオプションは以下のとおりです (任意の順序)。

-keystore **keystore-file-name**
作成、更新、または表示される鍵ストア・ファイルの名前を指定します。このオプションは、すべてのコマンドに必須です。

-storepass **keystore-file-password**
作成、更新、または表示される鍵ストア・ファイルに関連したパスワードを指定します。このオプションは、すべてのコマンドに必須です。

-storetype **keystore-type**
作成、更新、または表示される鍵ストア・ファイルのタイプを指定します。このオプションは、どのコマンドにも指定できます。このオプションが指定されない場合、"IbmISeriesKeyStore" の値が使用されます。

-appid **application-identifier**
作成または更新される鍵ストア・ファイルを初期化するために使用されるアプリケーション ID を指定します。このオプションは、*-create* および *-update* コマンドには任意指定です。 *-appid*、*keyring*、および *-systemdefault* オプションの 1 つだけを指定できます。

-keyring **keyring-file-name**
作成または更新される鍵ストア・ファイルを初期化するために使用される鍵リング・ファイル名を指定します。このオプションは、*-create* および *-update* コマンドには任意指定です。 *-appid*、*keyring*、および *-systemdefault* オプションの 1 つだけを指定できます。

-keyringpass **keyring-file-password**
作成または更新される鍵ストア・ファイルを初期化するために使用される鍵リング・ファイル・パスワードを指定します。このオプションは、*-create* および *-update* コマンドに指定できます。また、鍵ストア・タイプに "IbmISeriesKeyStore" 以外が指定されているときには、このオプションは必須です。このオプションが指定されない場合、隠しておく鍵リング・パスワードが使用されます。

-keyringlabel **keyring-file-label**
作成または更新される鍵ストア・ファイルを初期化するために使用される鍵リング・ラベルを指定します。このオプションは、*-keyring* オプションも指定されている場合に限り指定できます。*keyring* オプションが指定されているときにこのオプションが指定されない場合、鍵リング内のデフォルト・ラベルが使用されます。

-systemdefault
作成または更新される鍵ストア・ファイルを初期化するために使用されるシステム・デフォルト値

を指定します。このオプションは、*-create* および *-update* コマンドには任意指定です。 *-appid*、*keyring*、および *-systemdefault* オプションの 1 つだけを指定できます。

-v 冗長出力が作成されることを指定します。このオプションは、どのコマンドにも指定できます。

ヘルプ・コマンドは、パラメーターをこのメソッドに指定するための使用法情報を表示します。ヘルプ機能を呼び出すためのパラメーターは、以下のように指定されます。

```
-help
```

作成コマンドは、新規の鍵ストア・ファイルを指定します。作成コマンドには 3 つのバリエーションがあります。 1 つめのバリエーションは、特定のアプリケーション ID に基づいて鍵ストアを作成します。2 つめのバリエーションは、鍵リングの名前、ラベル、およびパスワードに基づいて鍵ストアを作成します。3 つめのバリエーションは、システム・デフォルト構成に基づいて鍵ストアを作成します。

特定のアプリケーション ID に基づいて鍵ストアを作成するには、*-appid* オプションを指定する必要があります。次のパラメーターは、名前が "keystore.file"、パスワードが "keypass" であるタイプ "IbmISeriesKeyStore" の鍵ストア・ファイルを作成します。これはアプリケーション ID "APPID" に基づいて初期設定されます。

```
-create -keystore keystore.file -storepass keypass -storetype IbmISeriesKeyStore
        -appid APPID
```

特定の鍵リング・ファイルに基づいて鍵ストアを作成するには、*-keyring* オプションを指定する必要があります。また、*-keyringpass* および *keyringlabel* オプションを指定することもできます。次のパラメーターは、名前が "keystore.file"、パスワードが "keypass" であるタイプ "IbmISeriesKeyStore" の鍵ストア・ファイルを作成します。これは、名前が "keyring.file"、鍵リング・パスワードが"ringpass"、鍵リング・ラベルが "keylabel" の鍵リング・ファイルに基づいて初期設定されます。

```
-create -keystore keystore.file -storepass keypass -storetype IbmISeriesKeyStore
        -keyring keyring.file -keyringpass ringpass -keyringlabel keylabel
```

システム・デフォルト構成に基づいて鍵ストアを作成するには、*-systemdefault* オプションを指定する必要があります。次のパラメーターは、名前が "keystore.file"、パスワードが "keypass" であるタイプ "IbmISeriesKeyStore" の鍵ストア・ファイルを作成します。これはシステム・デフォルト構成に基づいて初期設定されます。

```
-create -keystore keystore.file -storepass keypass -systemdefault
```

更新コマンドは、タイプが "IbmISeriesKeyStore" の既存の鍵ストア・ファイルを更新します。更新コマンドには、作成コマンドのバリエーションと同一の 3 つのバリエーションがあります。更新コマンドのオプションは、作成コマンドに使用したオプションと同一です。表示コマンドは、既存の鍵ストア・ファイルに指定された構成を表示します。次のパラメーターは、名前が "keystore.file"、パスワードが "keypass" であるタイプ "IbmISeriesKeyStore" の鍵ストア・ファイルで指定された構成を表示します。

```
-display -keystore keystore.file -storepass keypass -storetype IbmISeriesKeyStore
```

パラメーター:
`args` - コマンド行引数

---

run

```
public void run(java.lang.String[]args,
               java.io.PrintStreamout)
```

SSLConfiguration 関数を実行します。このメソッドのパラメーターおよび機能は main() メソッドと同一です。

パラメーター:

`args` - コマンド引数

`out` - 結果が書き込まれる出力ストリーム

参照:`com.ibm.i5os.jsse.SSLConfiguration.main()`

----

getApplicationId

```
public java.lang.String getApplicationId()
```

アプリケーション ID を戻します。

戻されるもの:

アプリケーション ID。

----

getKeyringName

```
public java.lang.String getKeyringName()
```

鍵リング名を戻します。

戻されるもの:

鍵リング名。

----

getKeyringLabel

```
public java.lang.String getKeyringLabel()
```

鍵リング・ラベルを戻します。

戻されるもの:

鍵リング・ラベル。

----

getKeyringPassword

```
public final char[] getKeyringPassword()
```

鍵リング・パスワードを戻します。

戻されるもの:

鍵リング・パスワード。

----

finalize

```
protected void finalize()
                 throws java.lang.Throwable
```

ガーベッジ・コレクションが、オブジェクトの参照がこれ以上ないと判断するときに、ガーベッジ・コレクターによってオブジェクト上で呼び出されます。

オーバーライド:

クラス java.lang.Object の finalize

スロー:

java.lang.Throwable - このメソッドによって出される例外。

----

clear

```
public void clear()
```

すべての get メソッドがヌルを戻すように、オブジェクト内のすべての情報を消去します。

----

setKeyring

```
public void setKeyring(java.lang.Stringname,
                       java.lang.Stringlabel,
                       char[]password)
```

鍵リング情報を設定します。

パラメーター:

name - 鍵リング名

label - 鍵リング・ラベル。または、デフォルトの鍵リング項目が使用される場合はヌル。

password - 鍵リング・パスワード。または隠しておくパスワードが使用される場合はヌル。

----

setApplicationId

```
public void setApplicationId(java.lang.StringapplicationId)
```

アプリケーション ID を設定します。

パラメーター:

applicationId - アプリケーション ID。

----

setApplicationId

```
public void setApplicationId(java.lang.StringapplicationId,
                             char[]password)
```

アプリケーション ID および鍵リング・パスワードを設定します。鍵リング・パスワードを指定すると、作成される鍵ストアが秘密鍵にアクセスできるようになります。

パラメーター:

applicationId - アプリケーション ID。

password - 鍵リング・パスワード。

----

equals

```
public boolean equals(java.lang.Objectobj)
```

他の何らかのオブジェクトがこのオブジェクトに「相当する」かどうかを示します。

オーバーライド:
　　クラス java.lang.Object の equals

パラメーター:
　　obj - 比較されるオブジェクト

戻されるもの:
　　オブジェクトが同じ構成情報を指定するかどうかを示す標識

----

hashCode

```
public int hashCode()
```

オブジェクトのハッシュ・コード値を戻します。

オーバーライド:
　　クラス java.lang.Object の hashCode

戻されるもの:
　　このオブジェクトのハッシュ・コード値。

----

clone

```
public java.lang.Object clone()
```

この SSL 構成の新規コピーを生成します。この SSL 構成のコンポーネントに対するこれ以降の変更は新規コピーに影響を与えません。逆の場合も同じです。

オーバーライド:
　　クラス java.lang.Object の clone

戻されるもの:
　　この SSL 構成のコピー

----

getKeyStore

```
public java.security.KeyStore getKeyStore(char[]password)
                                   throws java.security.KeyStoreException
```

与えられたパスワードを使用して、タイプが "IbmISeriesKeyStore" の鍵ストアを戻します。鍵ストアは、オブジェクトに現在保管されている構成情報に基づいて初期設定されます。

パラメーター:
　　password - 鍵ストアの初期設定に使用されます

戻されるもの:
　　オブジェクトに現在保管されている構成情報に基づいて初期設定される KeyStore 鍵ストア

スロー:

`java.security.KeyStoreException` - 鍵ストアを作成できなかった場合

----

getKeyStore

```
public java.security.KeyStore getKeyStore(java.lang.Stringtype,
                                          char[]password)
                               throws java.security.KeyStoreException
```

与えられたパスワードを使用して、要求されたタイプの鍵ストアを戻します。鍵ストアは、オブジェクトに現在保管されている構成情報に基づいて初期設定されます。

パラメーター:

`type` - 戻される鍵ストアのタイプ

`password` - 鍵ストアの初期設定に使用されます

戻されるもの:

オブジェクトに現在保管されている構成情報に基づいて初期設定される KeyStore 鍵ストア

スロー:

`java.security.KeyStoreException` - 鍵ストアを作成できなかった場合

例: IBM Java Secure Sockets Extension 1.5:

JSSE の例では、クライアントおよびサーバーがネイティブ IBM i JSSE プロバイダーを使用して、安全な通信を可能にするコンテキストを作成する方法を示しています。

注: いずれの例でも、java.security ファイルの指定するプロパティーにかかわず、ネイティブ IBM i JSSE プロバイダーを使用します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

例: バージョン 1.5 の SSLContext オブジェクトを使用する SSL クライアント:

このクライアント・プログラムの例では、"MY_CLIENT_APP" アプリケーション ID を使用するために初期化を行う、SSLContext オブジェクトを使用します。このプログラムでは、java.security ファイルにおける指定の有無にかかわらず、ネイティブ IBM i インプリメンテーションを使用します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
//////////////////////////////////////////////////////////////////////////////////
//
// This example client program utilizes an SSLContext object, which it initializes
// to use the "MY_CLIENT_APP" application ID.
//
// The example uses the native JSSE provider, regardless of the
// properties specified by the java.security file.
//
// Command syntax:
//     java  SslClient
//
//////////////////////////////////////////////////////////////////////////////////

import java.io.*;
import javax.net.ssl.*;
```

```
import java.security.*;
import com.ibm.i5os.jsse.SSLConfiguration;
/**
 * SSL Client Program.
 */
public class SslClient {

   /**
    * SslClient main method.
    *
    * @param args the command line arguments (not used)
    */
   public static void main(String args[]) {
       /*
        * Set up to catch any exceptions thrown.
        */
       try {
           /*
            * Initialize an SSLConfiguration object to specify an application
            * ID.  "MY_CLIENT_APP" must be registered and configured
            * correctly with the Digital Certificate Manager (DCM).
            */
           SSLConfiguration config = new SSLConfiguration();
           config.setApplicationId("MY_CLIENT_APP");
           /*
            * Get a KeyStore object from the SSLConfiguration object.
            */
           char[] password = "password".toCharArray();
           KeyStore ks = config.getKeyStore(password);
           /*
            * Allocate and initialize a KeyManagerFactory.
            */
           KeyManagerFactory kmf =
                             KeyManagerFactory.getInstance("IbmISeriesX509");
           kmf.init(ks, password);
           /*
            * Allocate and initialize a TrustManagerFactory.
            */
           TrustManagerFactory tmf =
                           TrustManagerFactory.getInstance("IbmISeriesX509");
           tmf.init(ks);
           /*
            * Allocate and initialize an SSLContext.
            */
           SSLContext c =
                      SSLContext.getInstance("SSL", "IBMi5OSJSSEProvider");
           c.init(kmf.getKeyManagers(), tmf.getTrustManagers(), null);
           /*
            * Get the an SSLSocketFactory from the SSLContext.
            */
           SSLSocketFactory sf = c.getSocketFactory();
           /*
            * Create an SSLSocket.
            *
            * Change the hard-coded IP address to the IP address or host name
            * of the server.
            */
           SSLSocket s = (SSLSocket) sf.createSocket("1.1.1.1", 13333);
           /*
            * Send a message to the server using the secure session.
            */
           String sent = "Test of java SSL write";
           OutputStream os = s.getOutputStream();
           os.write(sent.getBytes());
           /*
            * Write results to screen.
            */
```

```
            System.out.println("Wrote " + sent.length() + " bytes...");
            System.out.println(sent);
            /*
             * Receive a message from the server using the secure session.
             */
            InputStream is = s.getInputStream();
            byte[] buffer = new byte[1024];
            int bytesRead = is.read(buffer);
            if (bytesRead == -1)
                throw new IOException("Unexpected End-of-file Received");
            String received = new String(buffer, 0, bytesRead);
            /*
             * Write results to screen.
             */
            System.out.println("Read " + received.length() + " bytes...");
            System.out.println(received);
        } catch (Exception e) {
            System.out.println("Unexpected exception caught: " +
                               e.getMessage());
            e.printStackTrace();
        }
    }

}
```

例: バージョン 1.5 の SSLContext オブジェクトを使用する SSL サーバー:

以下のサーバー・プログラムは、過去に作成された鍵ストア・ファイルによって初期化を行う、SSLContext オブジェクトを使用します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
////////////////////////////////////////////////////////////////////////////////
//
// The following server program utilizes an SSLContext object that it
// initializes with a previously created keystore file.
//
// The keystore file has the following name and keystore password:
//   File name: /home/keystore.file
//   Password:  password
//
// The example program needs the keystore file in order to create an
// IbmISeriesKeyStore object. The KeyStore object must specify MY_SERVER_APP as
// the application identifier.
//
// To create the keystore file, you can use the following Qshell command:
//
//   java com.ibm.i5os.SSLConfiguration  -create  -keystore /home/keystore.file
//     -storepass password  -appid MY_SERVER_APP
//
// Command syntax:
//     java  JavaSslServer
//
// You can also create the keystore file by entering this command at an CL command prompt:
//
//  RUNJVA CLASS(com.ibm.i5os.SSLConfiguration) PARM('-create' '-keystore'
//      '/home/keystore.file' '-storepass' 'password' '-appid' 'MY_SERVER_APP')
//
////////////////////////////////////////////////////////////////////////////////

import java.io.*;
import javax.net.ssl.*;
import java.security.*;
/**
* Java SSL Server Program using Application ID.
```

```
*/
public class JavaSslServer {

   /**
    * JavaSslServer main method.
    *
    * @param args the command line arguments (not used)
    */
   public static void main(String args[]) {
       /*
        * Set up to catch any exceptions thrown.
        */
       try {
           /*
            * Allocate and initialize a KeyStore object.
            */
           char[] password = "password".toCharArray();
           KeyStore ks = KeyStore.getInstance("IbmISeriesKeyStore");
           FileInputStream fis = new FileInputStream("/home/keystore.file");
           ks.load(fis, password);
           /*
            * Allocate and initialize a KeyManagerFactory.
            */
           KeyManagerFactory kmf =
                             KeyManagerFactory.getInstance("IbmISeriesX509");
           kmf.init(ks, password);
           /*
            * Allocate and initialize a TrustManagerFactory.
            */
           TrustManagerFactory tmf =
                           TrustManagerFactory.getInstance("IbmISeriesX509");
           tmf.init(ks);
           /*
            * Allocate and initialize an SSLContext.
            */
           SSLContext c =
                      SSLContext.getInstance("SSL", "IBMi5OSJSSEProvider");
           c.init(kmf.getKeyManagers(), tmf.getTrustManagers(), null);
           /*
            * Get the an SSLServerSocketFactory from the SSLContext.
            */
           SSLServerSocketFactory sf = c.getServerSocketFactory();
           /*
            * Create an SSLServerSocket.
            */
           SSLServerSocket ss =
               (SSLServerSocket) sf.createServerSocket(13333);
           /*
            * Perform an accept() to create an SSLSocket.
            */
           SSLSocket s = (SSLSocket) ss.accept();
           /*
            * Receive a message from the client using the secure session.
            */
           InputStream is = s.getInputStream();
           byte[] buffer = new byte[1024];
           int bytesRead = is.read(buffer);
           if (bytesRead == -1)
               throw new IOException("Unexpected End-of-file Received");
           String received = new String(buffer, 0, bytesRead);
           /*
            * Write results to screen.
            */
           System.out.println("Read " + received.length() + " bytes...");
           System.out.println(received);
           /*
            * Echo the message back to the client using the secure session.
```

```
             */
            OutputStream os = s.getOutputStream();
            os.write(received.getBytes());
            /*
             * Write results to screen.
             */
            System.out.println("Wrote " + received.length() + " bytes...");
            System.out.println(received);
        } catch (Exception e) {
            System.out.println("Unexpected exception caught: " +
                               e.getMessage());
            e.printStackTrace();
        }
    }

}
```

Java Secure Socket Extension 6 の使用

この情報は、Java 2 Platform, Standard Edition (J2SE) バージョン 6 以降のリリースが稼働するシステムで JSSE を使用する場合にのみ適用されます。JSSE は、SSL と TLS の両方の基礎となるメカニズムを要約するフレームワークと似ています。基礎となっているプロトコルの複雑さと特色を要約することによって、JSSE はプログラマーが安全で暗号化された通信を使用できるようにすると同時に、セキュリティーのぜい弱性を最小限に抑えます。 Java Secure Socket Extension (JSSE) は、Secure Sockets Layer (SSL) プロトコルと Transport Layer Security (TLS) プロトコルの両方を使用して、クライアントとサーバーの間に安全で暗号化された通信を提供します。

IBM JSSE インプリメンテーションは IBM JSSE と呼ばれています。 IBM JSSE には、ネイティブ IBM i JSSE プロバイダーと IBM 純正の Java JSSE プロバイダーが組み込まれています。また、Sun Microsystems, Inc. JSSE は初めに IBM i サーバーに同梱で提供され、その後も継続して提供されます。

JSSE 6 をサポートするようにサーバーを構成する:

異なる JSSE インプリメンテーションを使用するように IBM i サーバーを構成します。このトピックには、ソフトウェア要件、JSSE プロバイダーの変更方法、および必要なセキュリティー・プロパティーとシステム・プロパティーに関する情報もあります。デフォルトの構成では、IBMJSSE2 という IBM 純正の Java JSSE プロバイダーを使用します。

| ご使用の IBM i サーバーで Java 2 Platform, Standard Edition (J2SE) バージョン 6 を使用している場合、
| JSSE は既に構成済みです。デフォルトの構成では、IBM 純正の Java JSSE プロバイダーを使用します。

JSSE プロバイダーの変更

IBM 純正の Java JSSE プロバイダーの代わりにネイティブ IBM i JSSE プロバイダーや Sun Microsystems, Inc. の JSSE プロバイダーを使用するように JSSE を構成することもできます。いくつかの特定の JSSE セキュリティー・プロパティーと Java システム・プロパティーを変更することによって、プロバイダーの間の切り替えができます。

セキュリティー・マネージャー

Java セキュリティー・マネージャーを使用可能にして JSSE アプリケーションを実行している場合、使用可能なネットワーク許可を設定する必要がある可能性があります。詳しくは、Permissions in the Java 2 SDK の SSL Permission の説明を参照してください。

JSSE 6 プロバイダー:

IBM JSSE には、ネイティブ IBM i JSSE プロバイダーと IBM 純正の Java JSSE プロバイダーが組み込まれています。Sun Microsystems, Inc. JSSE は、IBM i サーバーにも同梱されています。どのプロバイダーを選択して使用するかは、アプリケーションの必要に応じて異なります。

プロバイダーはすべて JSSE インターフェースの仕様に準拠します。これらは双方向通信が可能であったり、任意の他の SSL または TLS インプリメンテーション (非 Java インプリメンテーションでも可) と通信することができます。

Sun Microsystems, Inc. 純正の Java JSSE プロバイダー

これは、JSSE の Sun Java インプリメンテーションです。この JSSE は初めに IBM i サーバーに同梱で提供され、その後も継続して提供されます。Sun Microsystems, Inc. JSSE について詳しくは、Sun Microsystems, Inc. による「Java Secure Socket Extension (JSSE) Reference Guide」を参照してください。

IBM 純正の Java JSSE プロバイダー

IBM 純正の Java JSSE プロバイダーは、以下のフィーチャーを備えています。
- ディジタル証明書を制御および構成するためのあらゆるタイプの KeyStore オブジェクト (たとえば、JKS および PKCS12 など) を処理します。
- 複数のインプリメンテーションの JSSE コンポーネントを組み合わせて一緒に使用することができます。

IBMJSSEProvider2 はピュア Java 実装のプロバイダー名です。このプロバイダー名は、適切なケースを使用して java.security.Security.getProvider() メソッドか、あるいは、いくつかの JSSE クラス用の種々の getInstance() メソッドに渡す必要があります。

ネイティブ IBM i JSSE プロバイダー

ネイティブ IBM i JSSE プロバイダーには、以下の機能があります。
- ネイティブ IBM i SSL サポートを使用します。
- ディジタル証明書を構成、制御するために、ディジタル証明書マネージャーの使用を許可します。これは、固有な IBM i タイプの KeyStore (IbmISeriesKeyStore) を介して提供されます。
- 複数のインプリメンテーションの JSSE コンポーネントを組み合わせて一緒に使用することができます。

IBMi5OSJSSEProvider は、ネイティブ IBM i 実装の名前です。このプロバイダー名は、適切なケースを使用して java.security.Security.getProvider() メソッドか、あるいは、いくつかの JSSE クラス用の種々の getInstance() メソッドに渡す必要があります。

デフォルト JSSE プロバイダーの変更

セキュリティー・プロパティーに適切な変更を加えることによって、デフォルトの JSSE プロバイダーを変更することができます。

JSSE プロバイダーを変更した後は、システム・プロパティーが、新しいプロバイダーが必要とするディジタル証明書情報 (鍵ストア) 用の適切な構成を指定していることを確認します。

詳しくは、「JSSE 6 セキュリティー・プロパティー」を参照してください。

Sun Microsystems, Inc. による「JSSE Reference Guide」

『JSSE 6 セキュリティー・プロパティー』
Java 仮想マシン (JVM) は、多数の重要なセキュリティー・プロパティーを使用します。これらのセキュリティー・プロパティーは、Java マスター・セキュリティー・プロパティー・ファイルを編集することによって設定されます。

JSSE 6 セキュリティー・プロパティー:

Java 仮想マシン (JVM) は、多数の重要なセキュリティー・プロパティーを使用します。これらのセキュリティー・プロパティーは、Java マスター・セキュリティー・プロパティー・ファイルを編集することによって設定されます。

java.security というこのファイルは、通常はサーバー上の /QOpenSys/QIBM/ProdData/JavaVM/jdk60/32bit/jre/lib/security または/QOpenSys/QIBM/ProdData/JavaVM/jdk60/64bit/jre/lib/security ディレクトリーにあります。

以下のリストは、JSSE を使用するための、関連するいくつかのセキュリティー・プロパティーを示しています。この説明は、java.security ファイルを編集するためのガイドとして使用してください。

security.provider.<integer>

使用する JSSE プロバイダー。これは静的に暗号プロバイダー・クラスの登録も行います。以下の例のように、異なる JSSE プロバイダーを正確に指定してください。

```
security.provider.5=com.ibm.i5os.jsse.JSSEProvider
security.provider.6=com.ibm.jsse2.IBMJSSEProvider2
security.provider.7=com.sun.net.ssl.internal.ssl.Provider
```

ssl.KeyManagerFactory.algorithm

デフォルトの KeyManagerFactory アルゴリズムを指定します。ネイティブ IBM i JSSE プロバイダーの場合は、以下を使用します。

```
ssl.KeyManagerFactory.algorithm=IbmISeriesX509
```

IBM 純正の Java JSSE プロバイダーの場合は、以下を使用します。

```
ssl.KeyManagerFactory.algorithm=IbmX509
```

Sun Microsystems, Inc. 純正の Java JSSE プロバイダーの場合は、以下を使用します。

```
ssl.KeyManagerFactory.algorithm=SunX509
```

詳しくは、javax.net.ssl.KeyManagerFactory の Javadoc を参照してください。

ssl.TrustManagerFactory.algorithm

デフォルトの TrustManagerFactory アルゴリズムを指定します。ネイティブ IBM i JSSE プロバイダーの場合は、以下を使用します。

```
ssl.TrustManagerFactory.algorithm=IbmISeriesX509
```

IBM 純正の Java JSSE プロバイダーの場合は、以下を使用します。

```
ssl.TrustManagerFactory.algorithm=IbmX509
```

詳しくは、javax.net.ssl.TrustManagerFactory の Javadoc を参照してください。

ssl.SocketFactory.provider

デフォルトの SSL ソケット・ファクトリーを指定します。ネイティブ IBM i JSSE プロバイダーの場合は、以下を使用します。

```
ssl.SocketFactory.provider=com.ibm.i5os.jsse.JSSESocketFactory
```

IBM 純正の Java JSSE プロバイダーの場合は、以下を使用します。

```
ssl.SocketFactory.provider=com.ibm.jsse2.SSLSocketFactoryImpl
```

詳しくは、javax.net.ssl.SSLSocketFactory の Javadoc を参照してください。

ssl.ServerSocketFactory.provider

デフォルトの SSL サーバー・ソケット・ファクトリーを指定します。ネイティブ IBM i JSSE プロバイダーの場合は、以下を使用します。

```
ssl.ServerSocketFactory.provider=com.ibm.i5os.jsse.JSSEServerSocketFactory
```

純正の Java JSSE プロバイダーの場合は、以下を使用します。

```
ssl.ServerSocketFactory.provider=com.ibm.jsse2.SSLServerSocketFactoryImpl
```

詳しくは、javax.net.ssl.SSLServerSocketFactory の Javadoc を参照してください。

関連情報

javax.net.ssl.KeyManagerFactory Javadoc

javax.net.ssl.TrustManagerFactory Javadoc

javax.net.ssl.SSLSocketFactory Javadoc

javax.net.ssl.SSLServerSocketFactory Javadoc

JSSE for 6 Java システム・プロパティー:

アプリケーションで JSSE を使用するには、デフォルトの SSLContext オブジェクトが構成の確認を行うために必要な、いくつかのシステム・プロパティーを指定する必要があります。いくつかのプロパティーはすべてのプロバイダーに適用され、その他はネイティブ IBM i プロバイダーにだけ適用されます。

ネイティブ IBM i JSSE プロバイダーを使用するとき、プロパティーを指定しない場合は、os400.certificateContainer がデフォルトの *SYSTEM になりますが、それは、JSSE がシステム証明書ストア内のデフォルトのエントリーを使用することを意味します。

ネイティブ IBM i JSSE プロバイダーと IBM 純正 Java JSSE プロバイダーに適用されるプロパティー

以下のプロパティーは両方の JSSE プロバイダーに適用されます。それぞれの説明では、該当する場合には、デフォルトのプロパティーも示しています。

javax.net.ssl.trustStore

デフォルトの TrustManager が使用する KeyStore オブジェクトが入っているファイルの名前。デフォルト値は jssecacerts または (jssecacerets が存在しない場合は) cacerts です。

javax.net.ssl.trustStoreType

デフォルトの TrustManager が使用する KeyStore オブジェクトのタイプ。デフォルト値は KeyStore.getDefaultType メソッドによって戻される値です。

javax.net.ssl.trustStorePassword

デフォルトの TrustManager が使用する KeyStore オブジェクトのパスワード。

javax.net.ssl.keyStore

デフォルトの KeyManager が使用する KeyStore オブジェクトが入っているファイルの名前。デフォルト値は jssecacerts または (jssecacerets が存在しない場合は) cacerts です。

javax.net.ssl.keyStoreType

デフォルトの KeyManager が使用する KeyStore オブジェクトのタイプ。デフォルト値は KeyStore.getDefaultType メソッドによって戻される値です。

javax.net.ssl.keyStorePassword

デフォルトの KeyManager が使用する KeyStore オブジェクトのパスワード。

ネイティブ IBM i JSSE プロバイダーにのみ適用されるプロパティー

以下のプロパティーは、ネイティブ IBM i JSSE プロバイダーにのみ適用されます。

os400.secureApplication

アプリケーション ID。 JSSE は、以下のいずれかのプロパティーが指定されていない場合にのみ、このプロパティーを使用します。

- javax.net.ssl.keyStore
- javax.net.ssl.keyStorePassword
- javax.net.ssl.keyStoreType
- javax.net.ssl.trustStore
- javax.net.ssl.trustStorePassword
- javax.ssl.net.trustStoreType

os400.certificateContainer

使用する鍵リングの名前。 JSSE は、以下のいずれかのプロパティーが指定されていない場合にのみ、このプロパティーを使用します。

- javax.net.ssl.keyStore
- javax.net.ssl.keyStorePassword
- javax.net.ssl.keyStoreType
- javax.net.ssl.trustStore
- javax.net.ssl.trustStorePassword
- javax.ssl.net.trustStoreType
- os400.secureApplication

os400.certificateLabel

使用する鍵リング・ラベル。 JSSE は、以下のいずれかのプロパティーが指定されていない場合にのみ、このプロパティーを使用します。

- javax.net.ssl.keyStore
- javax.net.ssl.keyStorePassword
- javax.net.ssl.trustStore
- javax.net.ssl.trustStorePassword
- javax.ssl.net.trustStoreType
- os400.secureApplication

関連概念

15 ページの『Java システム・プロパティーのリスト』
Java システム・プロパティーにより、Java プログラムのランタイム環境が決まります。 Java システム・プロパティーは、IBM i のシステム値や環境変数と似ています。

関連情報

Sun Microsystems, Inc. 「System Properties」

ネイティブ IBM i JSSE 6 プロバイダーの使用:

ネイティブ IBM i JSSE プロバイダーは、JSSE クラスおよびインターフェースの一式を備えています。これには JSSE KeyStore クラスおよび SSLConfiguration クラスの実装が含まれています。

SSLContext.getInstance メソッドのプロトコル値

以下の表では、ネイティブ IBM i JSSE プロバイダーの SSLContext.getInstance メソッドのプロトコル値を示し、それを説明しています。

サポートされる SSL プロトコルは、システムに設定されているシステム値によって制限を受けることがあります。詳細は、『システム管理』情報の「Security system values: Secure Sockets Layer protocols」サブトピックを参照してください。

プロトコル値	サポートされている SSL プロトコル
SSL	SSL バージョン 3 および TLS バージョン 1。SSLv2 形式の Hello にカプセル化された SSLv3 または TLSv1 の Hello を受け入れます。
SSLv3	SSL バージョン 3 プロトコル。SSLv2 形式の Hello にカプセル化された SSLv3 Hello を受け入れます。
TLS	TLS バージョン 1 プロトコル。Request for Comments (RFC) 2246 で定義されています。 SSLv2 形式の Hello にカプセル化された TLSv1 Hello を受け入れます。
TLSv1	TLS バージョン 1 プロトコル。Request for Comments (RFC) 2246 で定義されています。 SSLv2 形式の Hello にカプセル化された TLSv1 Hello を受け入れます。
SSL_TLS	SSL バージョン 3 および TLS バージョン 1。SSLv2 形式の Hello にカプセル化された SSLv3 または TLSv1 の Hello を受け入れます。

ネイティブ IBM i KeyStore のインプリメンテーション

ネイティブ IBM i プロバイダーは、IbmISeriesKeyStore と IBMi5OSKeyStore という 2 つの KeyStore クラスのインプリメンテーションを提供します。どちらの KeyStore のインプリメンテーションも、ディジタル証明書マネージャー (DCM) サポートのまわりにラッパーを提供します。

IbmISeriesKeyStore
: 鍵ストアの内容は、特定のアプリケーション ID または鍵リング・ファイル、パスワード、およびラベルに応じて異なります。 JSSE はディジタル証明書マネージャーから鍵ストア・エントリーをロードします。エントリーをロードするため、JSSE はアプリケーションが最初に鍵ストア・エントリーまたは鍵ストア情報にアクセスしようとするときに、適切なアプリケーション ID または鍵リング情報を使用します。鍵ストアは変更できず、構成の変更はすべてディジタル証明書マネージャーを使用して行わなければなりません。

IBMi5OSKeyStore
: この鍵ストアの内容は、i5OS 証明書ストア・ファイルとそのファイルにアクセスするためのパスワードに基づいています。この KeyStore クラスでは、証明書ストアの変更ができます。ディジタル証明書マネージャーを使用せずに行うことができます。

  IBMi5OSKeyStore のインプリメンテーションは、Sun Microsystems, Inc. の Java KeyStore API の仕様に準拠しています。詳細は、Sun Microsystems, Inc. の Keystore Javadoc の情報を参照してください。

DCM を使用して鍵ストアを管理する方法について詳しくは、『ディジタル証明書マネージャー』のトピックを参照してください。

関連情報

Security system values: Secure Sockets Layer protocols

i5OSLoadStoreParameter クラスの Javadoc 情報:

com.ibm.i5os.keystore

クラス i5OSLoadStoreParameter

```
java.lang.Object
  |
  +--com.ibm.i5os.keystore.i5OSLoadStoreParameter
```

インプリメントされているすべてのインターフェース:

java.security.Keystore.LoadStoreParameter

```
public class i5OSLoadStoreParameter
extends java.lang.Object
implements java.security.Keystore.LoadStoreParameter
```

このクラスは、i5OS 証明書ストアのロード/保管に使用できる KeyStore.ProtectionParameter オブジェクトを作成します。作成後、このクラスは、アクセスされる証明書ストアについての情報とその証明書ストアを保護するために使用されるパスワードの情報を提供します。

このクラスの使用の例を以下に示します。

```
//initialize the keystore
    KeyStore ks = KeyStore.getInstance("IBMi5OSKeyStore");

 //Load an existing keystore
```

```
    File kdbFile = new File("/tmp/certificateStore.kdb");
    i5OSLoadStoreParameter lsp =
   new i5OSLoadStoreParameter (kdbFile, "password".toCharArray());
    ks.load(lsp);

 //Get and Add entries to the certificate store
    ...

 //Save the certificate store
    Ks.store(lsp);
```

対象: SDK 1.5

コンストラクターの要約

i5OSLoadStoreParameter(java.io.File ksFile, char[] password)

KeyStore ファイルから ProtectionParameter のインスタンスを作成し、i5OS 証明書ストアのロード/保管に使用するパスワードを作成します。

i5OSLoadStoreParameter(java.io.File ksFile, java.security.KeyStore.PasswordProtection pwdProtParam)

KeyStore ファイルから ProtectionParameter のインスタンスを作成し、i5OS 証明書ストアのロード/保管に使用する PasswordProtection を作成します。

表 13. **メソッドの要約**

java.security.KeyStore.ProtectionParameter	*324 ページの『getProtectionParameter』*() は、この LoadStoreParameter に関連付けられている KeyStore.KeyStoreParameter を戻します。

クラス java.lang.Object から継承されるメソッド
clone、equals、finalize、getClass、hashCode、notify、notifyAll、toString、wait、wait、wait

----

コンストラクターの詳細

i5OSLoadStoreParameter

```
public i5OSLoadStoreParameter(java.io.File ksFile,
                              char[] password)
                       throws java.lang.IllegalArgumentException
```

KeyStore ファイルから ProtectionParameter のインスタンスを作成し、i5OS 証明書ストアのロード/保管に使用するパスワードを作成します。

パラメーター:

ksFile - KeyStore の File オブジェクト。

keystore.load() が i5OSLoadStoreParameter(ksFile = null, password) で使用された場合は、新規の鍵ストアが作成されます。

keystore.store() が i5OSLoadStoreParameter(ksFile = null, password) で使用された場合は、IllegalArgumentException がスローされます。

password - i5OS 証明書ストアにアクセスするためのパスワード。 NULL または空にすることはできません。

スロー:

java.lang.IllegalArgumentException - パスワードが NULL または空の場合

----

i5OSLoadStoreParameter

```
public i5OSLoadStoreParameter(java.io.File ksFile,
                              java.security.KeyStore.PasswordProtection pwdProtParam)
                       throws java.lang.IllegalArgumentException
```

KeyStore ファイルから ProtectionParameter のインスタンスを作成し、i5OS 証明書ストアのロード/保管に使用する PasswordProtection を作成します。

keystore.load() が i5OSLoadStoreParameter(ksFile = null, password) で使用された場合は、新規の鍵ストアが作成されます。

keystore.store() が i5OSLoadStoreParameter(ksFile = null, password) で使用された場合は、IllegalArgumentException がスローされます。

パラメーター:

ksFile - KeyStore の File オブジェクト。

pwdProtParam - パスワードの獲得に使用される PasswordProtection インスタンス。NULL にすることはできません。

スロー:

java.lang.IllegalArgumentException - KeyStore.PasswordProtection が NULL の場合、または pwdProtParam にあるパスワードが NULL か空である場合

メソッドの詳細

----

getProtectionParameter

```
public java.security.KeyStore.ProtectionParameter getProtectionParameter()
```

この LoadStoreParameter に関連付けられている KeyStore.KeyStoreParameter を戻します。

指定するもの:

インターフェース java.security.KeyStore.LoadStoreParameter の getProtectionParameter

戻されるもの:

KeyStore.KeyStoreParameter インターフェースをインプリメントするインスタンス

参照:

java.security.KeyStore.ProtectionParameter#getProtectionParameter()

i5OSSystemCertificateStoreFile クラスの Javadoc 情報:

com.ibm.i5os.keystore

クラス i5OSSystemCertificateStoreFile

```
java.lang.Object
   java.io.File
       com.ibm.i5os.keystore.i5OSSystemCertificateStoreFile
```

インプリメントされているすべてのインターフェース:

java.io.Serializable、java.lang.Comparable<java.io.File>

```
public class i5OSSystemCertificateStoreFile
extends java.io.File
```

このクラスは、*SYSTEM 証明書ストア・ファイルを指す新しい File インプリメンテーションを提供します。このクラスは、ユーザーが実際のストアのパスを知らなくても *SYSTEM 証明書ストアをロードできるメカニズムを備えています。

*SYSTEM 証明書ストアを鍵ストアにロードするには、まず最初に i5OSSystemCertificateStoreFile を作成します。

ここから、2 つの方法で鍵ストアへのロードを行うことができます。

- i5OSLoadStoreParameter を使用する方法:

```
//create an i5OSSystemCertificateStoreFile
     File starSystemFile = new i5OSSystemCertificateStoreFile();

  //use that file to create an i5OSLoadStoreParameter
     i5OSLoadStoreParameter lsp = new i5OSLoadStoreParameter(starSystemFile, pwd);

  //load the certificate store into a keystore
     KeyStore ks = KeyStore.getInstance("IBMi5OSKeyStore");
     ks.load(lsp);
```

- FileInputStream を使用する方法:

```
//create an i5OSSystemCertificateStoreFile
     File starSystemFile = new i5OSSystemCertificateStoreFile();

  //create an input stream to the starSystemFile
     FileInputStream fis = new FileInputStream(starSystemFile);

  //load the certificate store into a keystore
     KeyStore ks = KeyStore.getInstance("IBMi5OSKeyStore");
     ks.load(fis, pwd);
```

対象: SDK 1.5

参照: *逐次化された形式*

フィールドの要約

クラス java.io.File から継承されるフィールド
pathSeparator、pathSeparatorChar、separator、separatorChar

コンストラクターの要約

i5OSSystemCertificateStoreFile()

*SYSTEM 証明書ストア・ファイルを指す File() を作成します。

メソッドの要約

クラス java.io.File から継承されるメソッド
canRead、canWrite、compareTo、createNewFile、createTempFile、createTempFile、delete、deleteOnExit、equals、exists、getAbsoluteFile、getAbsolutePath、getCanonicalFile、getCanonicalPath、getName、getParent、getParentFile、getPath、hashCode、isAbsolute、isDirectory、isFile、isHidden、lastModified、length、list、list、listFiles、listFiles、listFiles、listRoots、mkdir、mkdirs、renameTo、setLastModified、setReadOnly、toString、toURI、toURL

クラス java.lang.Object から継承されるメソッド
clone、finalize、getClass、notify、notifyAll、wait、wait、wait

コンストラクターの詳細

i5OSSystemCertificateStoreFile

```
public i5OSSystemCertificateStoreFile()
```

*SYSTEM 証明書ストア・ファイルを指す File() を作成します。

バージョン 6 の SSLConfiguration Javadoc 情報:

com.ibm.i5os.jsse

クラス SSLConfiguration

```
java.lang.Object
  |
  +--com.ibm.i5os.jsse.SSLConfiguration
```

インプリメントされているすべてのインターフェース:

java.lang.Cloneable, javax.net.ssl.ManagerFactoryParameters

public final class **SSLConfiguration**
extends java.lang.Object
implements javax.net.ssl.ManagerFactoryParameters, java.lang.Cloneable

このクラスは、ネイティブ IBM i JSSE インプリメンテーションによって必要とされる構成の仕様を提供します。

ネイティブ IBM i JSSE インプリメンテーションは、タイプが "IbmISeriesKeyStore" の KeyStore オブジェクトを使用すると、最も効率的に作動します。このタイプの KeyStore オブジェクトには、ディジタル証明書マネージャー (DCM) に登録されたアプリケーション ID かまたは鍵リング・ファイル (ディジタル証明書コンテナー) に基づいた、キー項目および信頼できる証明書項目が含まれます。また、このタイプの KeyStore オブジェクトを使用して、"IBMi5OSJSSEProvider" Provider から X509KeyManger および X509TrustManager オブジェクトを初期化することができます。さらに、X509KeyManager および X509TrustManager オブジェクトを使用して、"IBMi5OSJSSEProvider" から SSLContext オブジェクトを初

期化することができます。次に SSLContext オブジェクトは、KeyStore オブジェクトに指定された構成情報に基づいて、ネイティブ IBM i JSSE インプリメンテーションへのアクセスを提供します。"IbmISeriesKeyStore" KeyStore のロードが実行されるたびに、アプリケーション ID または鍵リング・ファイルに指定された現行の構成に基づいて、KeyStore が初期化されます。

また、このクラスを使用して、任意の有効なタイプの KeyStore オブジェクトを生成することもできます。KeyStore は、アプリケーション ID または鍵リング・ファイルに指定された現行の構成に基づいて初期化されます。アプリケーション ID または鍵リング・ファイルによって指定された構成に何らかの変更を加えた場合、その変更を反映するために、KeyStore オブジェクトを再生成する必要があります。"IbmISeriesKeyStore" 以外のタイプの KeyStore を正常に作成できるようにするには、鍵リング・パスワードを (アプリケーション ID を使用する場合は、*SYSTEM 証明書ストアに) 指定する必要があることに注意してください。作成される "IbmISeriesKeyStore" タイプの KeyStore 用の秘密鍵に正常にアクセスできるようにするには、鍵リング・パスワードを指定する必要があります。

対象: SDK 1.5

参照: KeyStore, X509KeyManager, X509TrustManager, SSLContext

コンストラクターの要約

SSLConfiguration() 新規の SSLConfiguration を作成します。詳しくは、 328 ページの『コンストラクターの詳細』を参照してください。

表 14. **メソッドの要約**

void	331 ページの『clear』() すべての get メソッドがヌルを戻すように、オブジェクト内のすべての情報を消去します。
java.lang.Object	333 ページの『clone』() この SSL 構成の新規コピーを生成します。
boolean	332 ページの『equals』(java.lang.Objectobj) 他の何らかのオブジェクトがこのオブジェクトに「相当する」かどうかを示します。
protected void	331 ページの『finalize』() ガーベッジ・コレクションが、オブジェクトの参照がこれ以上ないと判断するときに、ガーベッジ・コレクターによってオブジェクト上で呼び出されます。
java.lang.String	330 ページの『getApplicationId』() アプリケーション ID を戻します。
java.lang.String	331 ページの『getKeyringLabel』() 鍵リング・ラベルを戻します。
java.lang.String	330 ページの『getKeyringName』() 鍵リング名を戻します。
char[]	331 ページの『getKeyringPassword』() 鍵リング・パスワードを戻します。
java.security.KeyStore	333 ページの『getKeyStore』(char[]password) 与えられたパスワードを使用して、タイプが "IbmISeriesKeyStore" の鍵ストアを戻します。
java.security.KeyStore	333 ページの『 getKeyStore』(java.lang.Stringtype, char[]password) 与えられたパスワードを使用して、要求されたタイプの鍵ストアを戻します。
int	332 ページの『hashCode』() オブジェクトのハッシュ・コード値を戻します。
staticvoid	(java.lang.String[]args) SSLConfiguration 関数を実行します。
void	(java.lang.String[]args, java.io.PrintStreamout) SSLConfiguration 関数を実行します。
void	332 ページの『setApplicationId』(java.lang.StringapplicationId) アプリケーション ID を設定します。

表 14. **メソッドの要約** *(続き)*

void	332 ページの『setApplicationId』(java.lang.StringapplicationId, char[]password) アプリケーション ID および鍵リング・パスワードを設定します。
void	331 ページの『setKeyring』(java.lang.Stringname,java.lang.Stringlabel, char[]password) 鍵リング情報を設定します。

クラス java.lang.Object から継承されるメソッド
getClass, notify, notifyAll, toString, wait, wait, wait

コンストラクターの詳細

SSLConfiguration

```
public SSLConfiguration()
```

新規の SSLConfiguration を作成します。アプリケーション ID および鍵リング情報はデフォルト値に初期設定されます。

アプリケーション ID のデフォルト値は、"os400.secureApplication" プロパティーに指定された値です。

鍵リング情報のデフォルト値は、"os400.secureApplication" プロパティーが指定されている場合はヌルです。 "os400.secureApplication" プロパティーが指定されていない場合、鍵リング名のデフォルト値は、"os400.certificateContainer" プロパティーに指定された値です。 "os400.secureApplication" プロパティーが指定されていない場合、鍵リング・ラベルは "os400.certificateLabel" プロパティーに値に初期設定されます。 "os400.secureApplication" プロパティーと "os400.certificateContainer" プロパティーのどちらも設定されていない場合、鍵リング名は "*SYSTEM" に初期設定されます。

メソッドの詳細

----

main

```
public static void main(java.lang.String[]args)
```

SSLConfiguration 関数を実行します。実行できるコマンドは、-help、-create、-display、および -update の 4 つです。コマンドは、指定される最初のパラメーターでなければなりません。

指定できるオプションは以下のとおりです (任意の順序)。

-keystore **keystore-file-name**
: 作成、更新、または表示される鍵ストア・ファイルの名前を指定します。このオプションは、すべてのコマンドに必須です。

-storepass **keystore-file-password**
: 作成、更新、または表示される鍵ストア・ファイルに関連したパスワードを指定します。このオプションは、すべてのコマンドに必須です。

-storetype **keystore-type**
: 作成、更新、または表示される鍵ストア・ファイルのタイプを指定します。このオプションは、どのコマンドにも指定できます。このオプションが指定されない場合、"IbmISeriesKeyStore" の値が使用されます。

-appid **application-identifier**
: 作成または更新される鍵ストア・ファイルを初期化するために使用されるアプリケーション ID を指定します。このオプションは、*-create* および *-update* コマンドには任意指定です。 *-appid*、*keyring*、および *-systemdefault* オプションの 1 つだけを指定できます。

-keyring **keyring-file-name**
: 作成または更新される鍵ストア・ファイルを初期化するために使用される鍵リング・ファイル名を指定します。このオプションは、*-create* および *-update* コマンドには任意指定です。 *-appid*、*keyring*、および *-systemdefault* オプションの 1 つだけを指定できます。

-keyringpass **keyring-file-password**
: 作成または更新される鍵ストア・ファイルを初期化するために使用される鍵リング・ファイル・パスワードを指定します。このオプションは、*-create* および *-update* コマンドに指定できます。また、鍵ストア・タイプに "IbmISeriesKeyStore" 以外が指定されているときには、このオプションは必須です。このオプションが指定されない場合、隠しておく鍵リング・パスワードが使用されます。

-keyringlabel **keyring-file-label**
: 作成または更新される鍵ストア・ファイルを初期化するために使用される鍵リング・ラベルを指定します。このオプションは、*-keyring* オプションも指定されている場合に限り指定できます。 *keyring* オプションが指定されているときにこのオプションが指定されない場合、鍵リング内のデフォルト・ラベルが使用されます。

-systemdefault
: 作成または更新される鍵ストア・ファイルを初期化するために使用されるシステム・デフォルト値を指定します。このオプションは、*-create* および *-update* コマンドには任意指定です。 *-appid*、*keyring*、および *-systemdefault* オプションの 1 つだけを指定できます。

-v
: 冗長出力が作成されることを指定します。このオプションは、どのコマンドにも指定できます。

ヘルプ・コマンドは、パラメーターをこのメソッドに指定するための使用法情報を表示します。ヘルプ機能を呼び出すためのパラメーターは、以下のように指定されます。

```
-help
```

作成コマンドは、新規の鍵ストア・ファイルを指定します。作成コマンドには 3 つのバリエーションがあります。 1 つめのバリエーションは、特定のアプリケーション ID に基づいて鍵ストアを作成します。2 つめのバリエーションは、鍵リングの名前、ラベル、およびパスワードに基づいて鍵ストアを作成します。3 つめのバリエーションは、システム・デフォルト構成に基づいて鍵ストアを作成します。

特定のアプリケーション ID に基づいて鍵ストアを作成するには、*-appid* オプションを指定する必要があります。次のパラメーターは、名前が "keystore.file"、パスワードが "keypass" であるタイプ "IbmISeriesKeyStore" の鍵ストア・ファイルを作成します。これはアプリケーション ID "APPID" に基づいて初期設定されます。

```
-create -keystore keystore.file -storepass keypass -storetype IbmISeriesKeyStore
        -appid APPID
```

特定の鍵リング・ファイルに基づいて鍵ストアを作成するには、*-keyring* オプションを指定する必要があります。また、*-keyringpass* および *keyringlabel* オプションを指定することもできます。次のパラメーター

は、名前が "keystore.file"、パスワードが "keypass" であるタイプ "IbmISeriesKeyStore" の鍵ストア・ファイルを作成します。これは、名前が "keyring.file"、鍵リング・パスワードが"ringpass"、鍵リング・ラベルが "keylabel" の鍵リング・ファイルに基づいて初期設定されます。

```
-create -keystore keystore.file -storepass keypass -storetype IbmISeriesKeyStore
        -keyring keyring.file -keyringpass ringpass -keyringlabel keylabel
```

システム・デフォルト構成に基づいて鍵ストアを作成するには、*-systemdefault* オプションを指定する必要があります。次のパラメーターは、名前が "keystore.file"、パスワードが "keypass" であるタイプ "IbmISeriesKeyStore" の鍵ストア・ファイルを作成します。これはシステム・デフォルト構成に基づいて初期設定されます。

```
-create -keystore keystore.file -storepass keypass -systemdefault
```

更新コマンドは、タイプが "IbmISeriesKeyStore" の既存の鍵ストア・ファイルを更新します。更新コマンドには、作成コマンドのバリエーションと同一の 3 つのバリエーションがあります。更新コマンドのオプションは、作成コマンドに使用したオプションと同一です。表示コマンドは、既存の鍵ストア・ファイルに指定された構成を表示します。次のパラメーターは、名前が "keystore.file"、パスワードが "keypass" であるタイプ "IbmISeriesKeyStore" の鍵ストア・ファイルで指定された構成を表示します。

```
-display -keystore keystore.file -storepass keypass -storetype IbmISeriesKeyStore
```

パラメーター:

`args` - コマンド行引数

----

run

```
public void run(java.lang.String[]args,
                java.io.PrintStreamout)
```

SSLConfiguration 関数を実行します。このメソッドのパラメーターおよび機能は main() メソッドと同一です。

パラメーター:

`args` - コマンド引数

`out` - 結果が書き込まれる出力ストリーム

参照:`com.ibm.i5os.jsse.SSLConfiguration.main()`

----

getApplicationId

```
public java.lang.String getApplicationId()
```

アプリケーション ID を戻します。

戻されるもの:

アプリケーション ID。

----

getKeyringName

```
public java.lang.String getKeyringName()
```

鍵リング名を戻します。

戻されるもの:
鍵リング名。

----

getKeyringLabel

```
public java.lang.String getKeyringLabel()
```

鍵リング・ラベルを戻します。

戻されるもの:
鍵リング・ラベル。

----

getKeyringPassword

```
public final char[] getKeyringPassword()
```

鍵リング・パスワードを戻します。

戻されるもの:
鍵リング・パスワード。

----

finalize

```
protected void finalize()
                throws java.lang.Throwable
```

ガーベッジ・コレクションが、オブジェクトの参照がこれ以上ないと判断するときに、ガーベッジ・コレクターによってオブジェクト上で呼び出されます。

オーバーライド:
クラス java.lang.Object の finalize

スロー:
java.lang.Throwable - このメソッドによって出される例外。

----

clear

```
public void clear()
```

すべての get メソッドがヌルを戻すように、オブジェクト内のすべての情報を消去します。

----

setKeyring

```
public void setKeyring(java.lang.Stringname,
                       java.lang.Stringlabel,
                       char[]password)
```

鍵リング情報を設定します。

パラメーター:
name - 鍵リング名

label - 鍵リング・ラベル。または、デフォルトの鍵リング項目が使用される場合はヌル。

password - 鍵リング・パスワード。または隠しておくパスワードが使用される場合はヌル。

----

setApplicationId

```
public void setApplicationId(java.lang.StringapplicationId)
```

アプリケーション ID を設定します。

パラメーター:

applicationId - アプリケーション ID。

----

setApplicationId

```
public void setApplicationId(java.lang.StringapplicationId,
                             char[]password)
```

アプリケーション ID および鍵リング・パスワードを設定します。鍵リング・パスワードを指定すると、作成される鍵ストアが秘密鍵にアクセスできるようになります。

パラメーター:

applicationId - アプリケーション ID。

password - 鍵リング・パスワード。

----

equals

```
public boolean equals(java.lang.Objectobj)
```

他の何らかのオブジェクトがこのオブジェクトに「相当する」かどうかを示します。

オーバーライド:

クラス java.lang.Object の equals

パラメーター:

obj - 比較されるオブジェクト

戻されるもの:

オブジェクトが同じ構成情報を指定するかどうかを示す標識

----

hashCode

```
public int hashCode()
```

オブジェクトのハッシュ・コード値を戻します。

オーバーライド:

クラス java.lang.Object の hashCode

戻されるもの:

このオブジェクトのハッシュ・コード値。

----

clone

```
public java.lang.Object clone()
```

この SSL 構成の新規コピーを生成します。この SSL 構成のコンポーネントに対するこれ以降の変更は新規コピーに影響を与えません。逆の場合も同じです。

オーバーライド:

クラス java.lang.Object の clone

戻されるもの:

この SSL 構成のコピー

----

getKeyStore

```
public java.security.KeyStore getKeyStore(char[]password)
                                throws java.security.KeyStoreException
```

与えられたパスワードを使用して、タイプが "IbmISeriesKeyStore" の鍵ストアを戻します。鍵ストアは、オブジェクトに現在保管されている構成情報に基づいて初期設定されます。

パラメーター:

password - 鍵ストアの初期設定に使用されます

戻されるもの:

オブジェクトに現在保管されている構成情報に基づいて初期設定される KeyStore 鍵ストア

スロー:

java.security.KeyStoreException - 鍵ストアを作成できなかった場合

----

getKeyStore

```
public java.security.KeyStore getKeyStore(java.lang.Stringtype,
                                          char[]password)
                                throws java.security.KeyStoreException
```

与えられたパスワードを使用して、要求されたタイプの鍵ストアを戻します。鍵ストアは、オブジェクトに現在保管されている構成情報に基づいて初期設定されます。

パラメーター:

type - 戻される鍵ストアのタイプ

password - 鍵ストアの初期設定に使用されます

戻されるもの:

オブジェクトに現在保管されている構成情報に基づいて初期設定される KeyStore 鍵ストア

スロー:

java.security.KeyStoreException - 鍵ストアを作成できなかった場合

例: IBM Java Secure Sockets Extension 6:

JSSE の例では、クライアントおよびサーバーがネイティブ IBM i JSSE プロバイダーを使用して、安全な通信を可能にするコンテキストを作成する方法を示しています。

注: いずれの例でも、java.security ファイルの指定するプロパティーにかかわず、ネイティブ IBM i JSSE プロバイダーを使用します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

例: バージョン 6 の SSLContext オブジェクトを使用する SSL クライアント:

このクライアント・プログラムの例では、"MY_CLIENT_APP" アプリケーション ID を使用するために初期化を行う、SSLContext オブジェクトを使用します。このプログラムでは、java.security ファイルにおける指定の有無にかかわらず、ネイティブ IBM i インプリメンテーションを使用します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
///////////////////////////////////////////////////////////////////////////////////
//
// This example client program utilizes an SSLContext object, which it initializes
// to use the "MY_CLIENT_APP" application ID.
//
// The example uses the native JSSE provider, regardless of the
// properties specified by the java.security file.
//
// Command syntax:
//    java SslClient
//
///////////////////////////////////////////////////////////////////////////////////

import java.io.*;
import javax.net.ssl.*;
import java.security.*;
import com.ibm.i5os.jsse.SSLConfiguration;
/**
 * SSL Client Program.
 */
public class SslClient {

   /**
    * SslClient main method.
    *
    * @param args the command line arguments (not used)
    */
   public static void main(String args[]) {
       /*
        * Set up to catch any exceptions thrown.
        */
       try {
           /*
            * Initialize an SSLConfiguration object to specify an application
            * ID.  "MY_CLIENT_APP" must be registered and configured
            * correctly with the Digital Certificate Manager (DCM).
            */
           SSLConfiguration config = new SSLConfiguration();
           config.setApplicationId("MY_CLIENT_APP");
           /*
            * Get a KeyStore object from the SSLConfiguration object.
            */
           char[] password = "password".toCharArray();
           KeyStore ks = config.getKeyStore(password);
           /*
            * Allocate and initialize a KeyManagerFactory.
            */
           KeyManagerFactory kmf =
                            KeyManagerFactory.getInstance("IbmISeriesX509");
```

```
        kmf.init(ks, password);
        /*
         * Allocate and initialize a TrustManagerFactory.
         */
        TrustManagerFactory tmf =
                        TrustManagerFactory.getInstance("IbmISeriesX509");
        tmf.init(ks);
        /*
         * Allocate and initialize an SSLContext.
         */
        SSLContext c =
                   SSLContext.getInstance("SSL", "IBMi5OSJSSEProvider");
        c.init(kmf.getKeyManagers(), tmf.getTrustManagers(), null);
        /*
         * Get the an SSLSocketFactory from the SSLContext.
         */
        SSLSocketFactory sf = c.getSocketFactory();
        /*
         * Create an SSLSocket.
         *
         * Change the hard-coded IP address to the IP address or host name
         * of the server.
         */
        SSLSocket s = (SSLSocket) sf.createSocket("1.1.1.1", 13333);
        /*
         * Send a message to the server using the secure session.
         */
        String sent = "Test of java SSL write";
        OutputStream os = s.getOutputStream();
        os.write(sent.getBytes());
        /*
         * Write results to screen.
         */
        System.out.println("Wrote " + sent.length() + " bytes...");
        System.out.println(sent);
        /*
         * Receive a message from the server using the secure session.
         */
        InputStream is = s.getInputStream();
        byte[] buffer = new byte[1024];
        int bytesRead = is.read(buffer);
        if (bytesRead == -1)
            throw new IOException("Unexpected End-of-file Received");
        String received = new String(buffer, 0, bytesRead);
        /*
         * Write results to screen.
         */
        System.out.println("Read " + received.length() + " bytes...");
        System.out.println(received);
    } catch (Exception e) {
        System.out.println("Unexpected exception caught: " +
                           e.getMessage());
        e.printStackTrace();
    }
  }

}
```

例: バージョン 6 の SSLContext オブジェクトを使用する SSL サーバー:

以下のサーバー・プログラムは、過去に作成された鍵ストア・ファイルによって初期化を行う、SSLContext オブジェクトを使用します。

注: サンプル・コードをご使用の場合は、560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
///////////////////////////////////////////////////////////////////////////////////
//
// The following server program utilizes an SSLContext object that it
// initializes with a previously created keystore file.
//
// The keystore file has the following name and keystore password:
//   File name: /home/keystore.file
//   Password:  password
//
// The example program needs the keystore file in order to create an
// IbmISeriesKeyStore object. The KeyStore object must specify MY_SERVER_APP as
// the application identifier.
//
// To create the keystore file, you can use the following Qshell command:
//
//   java com.ibm.i5os.SSLConfiguration  -create  -keystore /home/keystore.file
//     -storepass password  -appid MY_SERVER_APP
//
// Command syntax:
//     java  JavaSslServer
//
// You can also create the keystore file by entering this command at an CL command prompt:
//
//  RUNJVA CLASS(com.ibm.i5os.SSLConfiguration) PARM('-create' '-keystore'
//      '/home/keystore.file' '-storepass' 'password' '-appid' 'MY_SERVER_APP')
//
//////////////////////////////////////////////////////////////////////////////////

import java.io.*;
import javax.net.ssl.*;
import java.security.*;
/**
* Java SSL Server Program using Application ID.
*/
public class JavaSslServer {

   /**
    * JavaSslServer main method.
    *
    * @param args the command line arguments (not used)
    */
   public static void main(String args[]) {
       /*
        * Set up to catch any exceptions thrown.
        */
       try {
           /*
            * Allocate and initialize a KeyStore object.
            */
           char[] password = "password".toCharArray();
           KeyStore ks = KeyStore.getInstance("IbmISeriesKeyStore");
           FileInputStream fis = new FileInputStream("/home/keystore.file");
           ks.load(fis, password);
           /*
            * Allocate and initialize a KeyManagerFactory.
            */
           KeyManagerFactory kmf =
                             KeyManagerFactory.getInstance("IbmISeriesX509");
           kmf.init(ks, password);
           /*
            * Allocate and initialize a TrustManagerFactory.
            */
           TrustManagerFactory tmf =
                           TrustManagerFactory.getInstance("IbmISeriesX509");
           tmf.init(ks);
           /*
            * Allocate and initialize an SSLContext.
```

```
             */
            SSLContext c =
                       SSLContext.getInstance("SSL", "IBMi5OSJSSEProvider");
            c.init(kmf.getKeyManagers(), tmf.getTrustManagers(), null);
            /*
             * Get the an SSLServerSocketFactory from the SSLContext.
             */
            SSLServerSocketFactory sf = c.getServerSocketFactory();
            /*
             * Create an SSLServerSocket.
             */
            SSLServerSocket ss =
                (SSLServerSocket) sf.createServerSocket(13333);
            /*
             * Perform an accept() to create an SSLSocket.
             */
            SSLSocket s = (SSLSocket) ss.accept();
            /*
             * Receive a message from the client using the secure session.
             */
            InputStream is = s.getInputStream();
            byte[] buffer = new byte[1024];
            int bytesRead = is.read(buffer);
            if (bytesRead == -1)
                throw new IOException("Unexpected End-of-file Received");
            String received = new String(buffer, 0, bytesRead);
            /*
             * Write results to screen.
             */
            System.out.println("Read " + received.length() + " bytes...");
            System.out.println(received);
            /*
             * Echo the message back to the client using the secure session.
             */
            OutputStream os = s.getOutputStream();
            os.write(received.getBytes());
            /*
             * Write results to screen.
             */
            System.out.println("Wrote " + received.length() + " bytes...");
            System.out.println(received);
        } catch (Exception e) {
            System.out.println("Unexpected exception caught: " +
                               e.getMessage());
            e.printStackTrace();
        }
    }

}
```

Java Authentication and Authorization Service

Java認証・承認サービス (JAAS) は、Java 2 Platform, Standard Edition (J2SE) の標準拡張機能です。 J2SE が提供するアクセス制御は、コードの発生元や署名者に基づくものです (コード・ソースをベースにしたアクセス制御)。ただし、コードの実行者に基づく追加のアクセス制御を実施する機能に欠けています。 JAAS が提供するフレームワークには、このサポートが Java 2 セキュリティー・モデルに対して追加されています。

IBM i での JAAS インプリメンテーションは、Sun Microsystems, Inc. のインプリメンテーションと互換性があります。この資料では、IBM i インプリメンテーションの固有の性質について扱います。ここでは、JAAS 拡張機能の一般資料に精通していることを前提とします。この情報と IBM i の情報を利用しやすくするために、以下のリンクが用意されています。

関連情報

JAAS API 仕様
JAAS に関する Javadoc の情報が示されています。

JAAS LoginModule
JAAS の認証に関する面に焦点を合わせています。

Java 認証・承認サービス (JAAS) 1.0

Java認証・承認サービス (JAAS) は、Java 2 Software Development Kit、標準拡張機能です。現在、Java 2 は、コード・ソースに基づくアクセス制御 (コードの*発信元* およびコードの*署名者* に基づいたアクセス制御) を提供しています。ただし、コードの*実行者* に基づく追加のアクセス制御を実施する機能に欠けています。 JAAS が提供するフレームワークでは、このサポートとともに Java 2 セキュリティー・モデルを拡大しています。

開発者用ガイド

- **概説**
- **この資料の対象読者**
- **関連資料**
- **紹介**
- **コア・クラス**
- **共通クラス**
  - **Subject**
  - **Principals**
  - **Credentials**
- **認証クラス**
- **LoginContext**
- **LoginModule**
- **CallbackHandler**
- **Callback**
- **権限クラス**
- **Policy**
- **AuthPermission**
- **PrivateCredentialPermission**

リファレンス

- インプリメンテーション
- "Hello World"、JAAS スタイル!
- 付録 A: java.security セキュリティー・プロパティー・ファイルでの JAAS 設定
- 付録 B: ログイン構成ファイル
- 付録 C: 権限ポリシー・ファイル

概説

Java認証・承認サービス (JAAS) は、Java 2 Software Development Kit、バージョン 1.3 の標準拡張機能です。現在、Java 2 は、コード・ソースに基づくアクセス制御 (コードの*発信元* およびコードの*署名者* に基づいたアクセス制御) を提供しています。ただし、コードの*実行者* に基づく追加のアクセス制御を実施する機能に欠けています。 JAAS が提供するフレームワークでは、このサポートとともに Java 2 セキュリティー・モデルを拡大しています。

この資料は、2000 年 3 月 17 日に最終更新されました。

この資料の対象読者

この資料は、コード・ソース・ベースのセキュリティー・モデルと Subject ベースのセキュリティー・モデルによって制限されたアプリケーションを作成しようとしている、経験を積んだプログラマーを対象としています。

関連資料

この資料は、読者が以下の資料をすでに読み終えていることを前提としています。

- Java 2 Software Development Kit API Specification
- JAAS API 仕様
- Security and the Java platform

このガイドの補足は、Sun Microsystems, Inc. が提供する LoginModule Developer's Guide です。

紹介

JAAS インフラストラクチャーは、2 つのメイン・コンポーネント、すなわち**認証**コンポーネントと**権限** (承認) コンポーネントに分割できます。 JAAS 認証コンポーネントは、Java がアプリケーション、アプレット、Bean、またはサーブレットとして稼働しているかどうかにかかわりなく、現在だれがそのコードを処理しているかについて確実かつ安全に判別する機能を提供します。 JAAS 権限 (承認) コンポーネントは、そのコード・ソース (Java 2 で実行される) に応じて、また認証された人物に応じて、処理中の Java コードが機密タスクを実行するのを制約することにより、既存の Java 2 セキュリティー・フレームワークを補足します。

JAAS 認証は、*プラグ可能* 形式で実行されます。これにより、Java アプリケーションは、基礎となる認証テクノロジーからの独立を保つことが可能になります。したがって、アプリケーションそのものに変更を加えなくても、新規または更新された認証テクノロジーをアプリケーションの下で接続することができます。アプリケーションは、

```
LoginContext
```

オブジェクトをインスタンス化することによって、認証プロセスを使用可能にし、次に認証テクノロジーを決定する

```
Configuration
```

を参照するか、認証の実行時に使用される

```
LoginModule
```

を参照します。標準的な LoginModules はユーザー名およびパスワードを入力するようにプロンプトを出して、それらを確認します。あるいは、音声または指紋のサンプルを読み取って、それらを確認する場合もあります。

コードを処理するユーザーが認証されると、JAAS 権限 (承認) コンポーネントが既存の Java 2 アクセス制御モデルとともに機能して、重要なリソースへのアクセスを保護します。

アクセス制御決定がコードの場所およびコードの署名者にのみ基づく (

`CodeSource`

) Java 2 とは異なり、JAAS アクセス制御決定は処理コードの

`CodeSource`

と、コードを実行するユーザーの両方、または

`Subject`

に基づきます。 JAAS ポリシーは Java 2 ポリシーを、関係のある Subject ベースの情報で拡張したものに過ぎないことに注意してください。そのため、Java 2 で認識され、理解される許可 (たとえば、

`java.io.FilePermission`

および

`java.net.SocketPermission`

) も JAAS によって理解され、認識されます。 さらに、JAAS セキュリティー・ポリシーは既存の Java 2 セキュリティー・ポリシーとは物理的に分離しているにもかかわらず、2 つのポリシーは一緒に 1 つの論理ポリシーを形成します。

コア・クラス

JAAS コア・クラスは、3 つのカテゴリーである共通、認証、および権限に分けることができます。

- 共通クラス
  - Subject、Principals、Credentials
- 認証クラス
  - LoginContext、LoginModule、CallbackHandler、Callback
- 権限クラス
  - Policy、AuthPermission、PrivateCredentialPermission

共通クラス

共通クラスは、JAAS 認証コンポーネントと権限 (承認) コンポーネントの両方に共有されます。

主要な JAAS クラスは

`Subject`

で、単一のエンティティー (個人など) に関連した情報のグループを表します。これは、エンティティーのプリンシパル、公開信任状、および秘密信任状を包含します。

JAAS は、既存の Java 2

`java.security.Principal`

インターフェースを使用して、プリンシパルを表すことに注意してください。また、JAAS は別個の信任状インターフェースまたはクラスを導入しないことにも注意してください。 JAAS によって定義される信任状は任意のオブジェクトです。

Subject

リソースへのアクセスを許可するには、アプリケーションは最初に要求のソースを認証する必要があります。 JAAS フレームワークは、要求のソースを表すために、サブジェクトという用語を定義します。サブジェクトは任意のエンティティー (個人またはサービスなど) です。サブジェクトが認証されると、そこに関連した ID、つまりプリンシパルが取り込まれます。サブジェクトは多数のプリンシパルを持つことがあります。たとえば、ある個人が名前プリンシパル ("John Doe") と、それを他のサブジェクトと区別する SSN プリンシパル ("123-45-6789") を持つことがあります。

また、

`Subject`

がセキュリティー関連の属性を持つこともあります。これを信任状と言います。特殊な保護を必要とする重要な信任状 (秘密暗号鍵など) は、秘密信任状

`Set`

に保管されます。共有されることが意図されている信任状 (公開鍵証明書または Kerberos チケットなど) は、公開信任状

`Set`

に保管されます。アクセスして変更する信任状セットが異なれば、それに応じた異なる許可が必要です。

サブジェクトは次のコンストラクターを使用して作成されます。

```
public Subject();

public Subject(boolean readOnly, Set principals,
               Set pubCredentials, Set privCredentials);
```

最初のコンストラクターは、プリンシパルおよび信任状の空 (非ヌル) のセットを持つサブジェクトを作成します。 2 番目のコンストラクターは、プリンシパルおよび信任状の指定されたセットを持つサブジェクトを作成します。また、読み取り専用サブジェクト (不変のプリンシパルおよび信任状セット) を作成できるブール引数を持っています。

これらのコンストラクターを使用しないで、認証済みサブジェクトへの参照を取得する代わりの方法が、LoginContext セクションに示されています。

サブジェクトが読み取り専用状態になるようにインスタンス化されなかった場合、次のメソッドを呼び出すことによって、読み取り専用状態に設定することができます。

```
public void setReadOnly();
```

このメソッドを呼び出すには、

`AuthPermission("setReadOnly")`

が必要です。読み取り専用状態になったら、プリンシパルまたは信任状を追加または削除しようとすると、

`IllegalStateException`

が出されます。

サブジェクトの読み取り専用状態をテストするには、次のメソッドを呼び出すことができます。

```
public boolean isReadOnly();
```

サブジェクトに関連したプリンシパルを検索するには、次の 2 つのメソッドが使用可能です。

```
public Set getPrincipals();
public Set getPrincipals(Class c);
```

最初のメソッドは、サブジェクトに含まれるすべてのプリンシパルを戻し、2 番目のメソッドは、指定されたクラス c のインスタンスまたはクラス c のサブクラスのインスタンスであるプリンシパルだけを戻します。サブジェクトに関連したプリンシパルがない場合には、空のセットが戻されます。

サブジェクトに関連した公開信任状を検索するには、次のメソッドが使用可能です。

```
public Set getPublicCredentials();
public Set getPublicCredentials(Class c);
```

これらのメソッドで観察される動作は、

getPrincipals

メソッドの動作と同一です。

サブジェクトに関連した秘密信任状を検索するには、次のメソッドが使用可能です。

```
public Set getPrivateCredentials();
public Set getPrivateCredentials(Class c);
```

これらのメソッドで観察される動作は、

getPrincipals

および

getPublicCredentials

メソッドの動作と同一です。

サブジェクトのプリンシパル・セット、公開信任状セット、または秘密信任状セットを変更するか、またはそれに対する操作を行うには、呼び出し元が

java.util.Set

クラスで定義されたメソッドを使用します。次の例は、このことを示しています。

```
Subject subject;
Principal principal;
Object credential;

// add a Principal and credential to the Subject
subject.getPrincipals().add(principal);
subject.getPublicCredentials().add(credential);
```

それぞれのセットを変更するには、

AuthPermission("modifyPrincipals")

、

AuthPermission("modifyPublicCredentials")

、または

AuthPermission("modifyPrivateCredentials")

が必要です。また、

getPrincipals

、

getPublicCredentials

、および

getPrivateCredentials

メソッドから戻されるセットだけが、サブジェクトのそれぞれの内部セットによって戻されることに注意してください。そのため、戻されたセットに対する変更は、内部セットにも影響を与えます。

getPrincipals(Class c)

、

getPublicCredentials(Class c)

、および

getPrivateCredentials(Class c)

メソッドから戻されるセットは、サブジェクトのそれぞれの内部セットによって戻されません。メソッドの呼び出しのたびに、新規セットが作成されて戻されます。これらのセットに対する変更は、サブジェクトの内部セットに影響を与えません。次のメソッドは、指定された

AccessControlContext

に関連したサブジェクトを戻すか、または指定した

AccessControlContext

に関連したサブジェクトがない場合には、ヌルを戻します。

```
public static Subject getSubject(final AccessControlContext acc);
```

Subject.getSubject

を呼び出すには、

AuthPermission("getSubject")

が必要です。

また、Subject クラスには、

java.lang.Object

から継承された次のメソッドが含まれます。

```
public boolean equals(Object o);
public String toString();
public int hashCode();
```

特定のサブジェクトとして処理を実行するために、次の静的メソッドを呼び出すことができます。

```
public static Object doAs(final Subject subject,
                     final java.security.PrivilegedAction action);

public static Object doAs(final Subject subject,
                     final java.security.PrivilegedExceptionAction action)
                     throws java.security.PrivilegedActionException;
```

どちらのメソッドも最初に、指定された ***subject*** を現行のスレッドの

AccessControlContext

に関連付け、次に ***action*** を処理します。これにより、***action*** が ***subject*** として実行されます。最初のメソッドは実行時例外をスローすることがありますが、通常処理では、このメソッドはその action引数の run() メソッドから Object を戻します。 2 番目のメソッドは同様に動作しますが、その

`PrivilegedExceptionAction`

run() メソッドからチェック済み例外をスローできる点が異なります。

`AuthPermission("doAs")`

は、

`doAs`

メソッドを呼び出すために必要です。

最初の

`doAs`

メソッドを使用する 2 つの例を示します。クラスが

`com.ibm.security.Principal`

で、"BOB" という名前のプリンシパルを持つ

`Subject`

が

`LoginContext`

"lc" によって認証されていることを想定します。また、SecurityManager がインストールされており、JAAS アクセス制御ポリシーに以下が存在していることも想定しています (JAAS ポリシー・ファイルの詳細は、Policy セクションを参照してください)。

```
   // Grant "BOB" permission to read the file "foo.txt"
   grant Principal com.ibm.security.Principal "BOB" {
      permission java.io.FilePermission "foo.txt", "read";
};
```

Subject.doAs の例 1

```
    class ExampleAction implements java.security.PrivilegedAction {
        public Object run() {
            java.io.File f = new java.io.File("foo.txt");

            // exists() invokes a security check
            if (f.exists()) {
                System.out.println("File foo.txt exists.");
            }
            return null;
        }
    }

    public class Example1 {
        public static void main(String[] args) {

            // Authenticate the subject, "BOB".
            // This process is described in the
            // LoginContext section.

            Subject bob;
            ...
```

```
            // perform "ExampleAction" as "BOB":
            Subject.doAs(bob, new ExampleAction());
        }
    }
```

処理中に、

`ExampleAction`

では、

`f.exists()`

を呼び出す際にセキュリティー検査が行われます。ただし、

`ExampleAction`

は "BOB" として実行しており、JAAS ポリシー (上述) は、必要な

`FilePermission`

を "BOB" に付与するため、

`ExampleAction`

はセキュリティー検査に合格します。

例 2 は、例 1 と同じシナリオを使用します。

Subject.doAs の例 2

```
    public class Example2 {
        // Example of using an anonymous action class.
        public static void main(String[] args) {
            // Authenticate the subject, "BOB".
            // This process is described in the
            // LoginContext section.

            Subject bob;
            ...

            // perform "ExampleAction" as "BOB":
            Subject.doAs(bob, new ExampleAction() {
                public Object run() {
                    java.io.File f = new java.io.File("foo.txt");
                    if (f.exists()) {
                        System.out.println("File foo.txt exists.");
                    }
                    return null;
                }
            });
        }
    }
```

例の permission grant ステートメントが間違って変更された場合 (正しくない CodeBase を追加したり、Principal を "MOE" に変更するなど)、どちらの例も

`SecurityException`

をスローします。 grant ブロックから Principal フィールドを除去してから、それを Java 2 ポリシー・ファイルに移動させると、

`SecurityException`

はスローされません。なぜなら、ここでは許可がより一般的になっているからです (すべての プリンシパルに使用可能)。

どちらの例も同じ関数を実行するため、一方のコードを他方より優先するには理由が必要です。例 1 は、無名クラスに精通していないプログラマーにとって、読み取りやすいでしょう。また、***action*** クラスは、固有の CodeBase を持つ個別のファイルに置くことができます。そうすると、permission grant はこの情報を使用できます。例 2 はよりコンパクトで、実行される ***action*** は見つけるのが簡単です。なぜなら、

`doAs`

の呼び出しで実行されるからです。

さらに、次のメソッドも特定のサブジェクトとして処理を実行します。

ただし、

`doAsPrivileged`

メソッドには、提供された ***action*** および ***subject*** に基づいてセキュリティー検査があります。提供されたコンテキストは、指定された ***subject*** および ***action*** に結びつけられます。ヌル・コンテキスト・オブジェクトは、現行の

`AccessControlContext`

を完全に無視します。

```
    public static Object doAsPrivileged(final Subject subject,
                         final java.security.PrivilegedAction action,
                         final java.security.AccessControlContext acc);

    public static Object doAsPrivileged(final Subject subject,
                         final java.security.PrivilegedExceptionAction action,
                         final java.security.AccessControlContext acc)
                         throws java.security.PrivilegedActionException;
```

`doAsPrivileged`

メソッドは、次の点で

`doAs`

メソッドと同様に動作します。すなわち、***subject*** はコンテキスト ***acc*** に関連付けられること、***action*** が実行されること、および実行時例外またはチェック例外がスローされることです。ただし、

`doAsPrivileged`

メソッドは、最初に既存のスレッドの

`AccessControlContext`

を空にしてから、***subject*** を提供されたコンテキストに関連付けて、***action*** を呼び出します。ヌルの ***acc*** 引数を指定すると、アクセス制御決定 (***action*** の処理中に呼び出される) は ***subject*** と ***action*** だけに基づくことになります。

`AuthPermission("doAsPrivileged")`

は、

`doAsPrivileged`

メソッドを呼び出すときに必要です。

Principals

すでに説明したとおり、プリンシパルはサブジェクトに関連付けることができます。プリンシパルは、サブジェクトの ID を表しており、

```
java.security.Principal
```

および

```
java.io.Serializable
```

インターフェースをインプリメントする必要があります。 Subject セクションでは、サブジェクトに関連したプリンシパルを更新する方法を説明しています。

Credentials

公開信任状クラスと秘密信任状クラスは、コア JAAS クラス・ライブラリーの一部ではありません。そのため、どの java クラスでも信任状を表すことができます。ただし、開発者はそれらの信任状クラスに、信任状に関連した 2 つのインターフェース、つまり Refreshable と Destroyable をインプリメントすることを選択できます。

Refreshable

この**インターフェース**は、信任状がそれ自身をリフレッシュする機能を提供します。たとえば、特定の時間制限付きの有効期間を指定した信任状は、その信任状が有効である期間を呼び出し元がリフレッシュできるようにするためにこのインターフェースをインプリメントします。インターフェースには次の 2 つの抽象メソッドがあります。

```
boolean isCurrent();
```

信任状が現行、つまり有効かどうかを判別します。

```
void refresh() throws RefreshFailedException;
```

信任状の妥当性を更新または拡張します。このメソッドのインプリメンテーションでは、

```
AuthPermission("refreshCredential")
```

セキュリティー検査を実行して、呼び出し元が信任状をリフレッシュする許可を持つようにします。

Destroyable

この**インターフェース**は、信任状内の内容を破棄する機能を提供します。インターフェースには次の 2 つの抽象メソッドがあります。

```
boolean isDestroyed();
```

信任状が破棄されているかどうかを判別します。

```
void destroy() throws DestroyFailedException;
```

この信任状に関連した情報を破棄して消去します。これ以降に、この信任状に対して特定のメソッドを呼び出すと、

```
IllegalStateException
```

がスローされます。このメソッドのインプリメンテーションでは、

```
AuthPermission("destroyCredential")
```

セキュリティー検査を実行して、呼び出し元が信任状を破棄する許可を持つようにします。

認証クラス

`Subject`

を認証するために、以下のステップが実行されます。

1. アプリケーションは、

   `LoginContext`

   をインスタンス化します。
2. `LoginContext`

   は構成を調べて、そのアプリケーション用に構成されたすべての LoginModules をロードします。
3. アプリケーションは LoginContext の *login* メソッドを呼び出します。
4. *login* メソッドは、ロードされたすべての LoginModules を呼び出します。それぞれの

   `LoginModule`

   は、

   `Subject`

   の認証を試みます。成功すると、LoginModules は関係のあるプリンシパルおよび信任状を

   `Subject`

   に関連付けます。
5. `LoginContext`

   は認証状況をアプリケーションに戻します。
6. 認証が成功すると、アプリケーションは認証済みの

   `Subject`

   を

   `LoginContext`

   から取得します。

LoginContext

`LoginContext`

クラスは、サブジェクトを認証するために使用される基本メソッドを提供し、さらに基礎となる認証テクノロジーから独立してアプリケーションを開発する方法を提供します。

`LoginContext`

は構成

`Configuration`

を調べて、特定のアプリケーション用に構成された認証サービス、すなわち LoginModules を判別します。したがって、アプリケーションそのものに変更を加えなくても、そのアプリケーションの下で様々な LoginModules を接続することができます。

`LoginContext`

は、選択できる以下の 4 つのコンストラクターを提供します。

```
public LoginContext(String name) throws LoginException;

public LoginContext(String name, Subject subject) throws LoginException;

public LoginContext(String name, CallbackHandler callbackHandler)
       throws LoginException

public LoginContext(String name, Subject subject,
       CallbackHandler callbackHandler) throws LoginException
```

すべてのコンストラクターは共通のパラメーター、***name*** を共有します。この引数は、ログイン構成に索引を付けるために、

`LoginContext`

によって使用されます。入力パラメーターとして

`Subject`

を取らないコンストラクターは、新規の

`Subject`

をインスタンス化します。すべてのコンストラクターで、ヌル入力は許可されません。呼び出し元は、

`AuthPermission("createLoginContext")`

に

`LoginContext`

をインスタンス化することを求めます。

実際の認証は、次のメソッドの呼び出しとともに行われます。

```
public void login() throws LoginException;
```

login が呼び出されると、構成された LoginModules のそれぞれの *login* メソッドがすべて、認証を実行するために呼び出されます。認証が成功すると、認証済みの

`Subject`

(この時点で、プリンシパル、公開信任状、および秘密信任状を保有可能です) を、次のメソッドを使用して取得することができます。

```
public Subject getSubject();
```

`Subject`

をログアウトし、その認証済みのプリンシパルおよび信任状を除去するために、次のメソッドが提供されています。

```
public void logout() throws LoginException;
```

アプリケーション内の次のコードの断片は、"moduleFoo" という名前の構成項目を持つ構成ファイルにアクセスした後で、"bob" と呼ばれるサブジェクトを認証します。

```
Subject bob = new Subject();
LoginContext lc = new LoginContext("moduleFoo", bob);
try {
    lc.login();
```

```
        System.out.println("authentication successful");
    } catch (LoginException le) {
        System.out.println("authentication unsuccessful"+le.printStackTrace());
    }
```

アプリケーション内の次のコードの断片は、「名前のない」サブジェクトを認証してから、getSubject メソッドを使用してそれを取得します。

```
    LoginContext lc = new LoginContext("moduleFoo");
    try {
        lc.login();
        System.out.println("authentication successful");
    } catch (LoginException le) {
        System.out.println("authentication unsuccessful"+le.printStackTrace());
    }
    Subject subject = lc.getSubject();
```

認証が失敗すると、*getSubject* はヌルを戻します。また、

`Subject.getSubject`

の場合に存在した、これを行うために必要な

`AuthPermission("getSubject")`

はありません。

LoginModule

LoginModule **インターフェース**は、アプリケーションの下で接続できる様々な種類の認証テクノロジーをインプリメントする機能を開発者に提供します。たとえば、

`LoginModule`

の 1 つのタイプでは、ユーザー名/パスワード・ベースの形式の認証を実行できます。

LoginModule Developer's Guide は、LoginModules をインプリメントするための段階的な説明を開発者に与える詳しい資料です。

`LoginModule`

をインスタンス化するためには、

`LoginContext`

は各

`LoginModule`

が、引数を取らない public コンストラクターを提供することを期待します。次に、

`LoginModule`

を関連情報とともに初期化するために、

`LoginContext`

は、LoginModule の

`initialize`

メソッドを呼び出します。提供された *subject* は、ヌル以外であることが保証されます。

```
   void initialize(Subject subject, CallbackHandler callbackHandler,
        Map sharedState, Map options);
```

次のメソッドは、認証プロセスを開始します。

```
boolean login() throws LoginException;
```

メソッド・インプリメンテーションの例では、ユーザーにユーザー名とパスワードを入力するように促し、その後 NIS または LDAP などの命名サービスに保管されたデータと照らして情報を検査することができます。代わりのインプリメンテーションでは、スマート・カードおよびバイオメトリック認証デバイスとのインターフェースをとり、単にユーザー情報を基礎となるオペレーティング・システムから抽出します。これは、JAAS 認証プロセスの***フェーズ 1*** と見なされます。

次のメソッドは、認証プロセスを完了し、終了させます。

```
boolean commit() throws LoginException;
```

認証プロセスの***フェーズ 1*** が成功した場合、このメソッドは***フェーズ 2*** を継続します。それは、プリンシパル、公開信任状、および秘密信任状をサブジェクトに関連付けることです。***フェーズ 1*** が失敗した場合、*commit* メソッドは以前に保管された認証状態 (ユーザー名およびパスワードなど) を除去します。

次のメソッドは、***フェーズ 1*** が成功しなかった場合に認証プロセスを停止します。

```
boolean abort() throws LoginException;
```

このメソッドの標準的インプリメンテーションでは、以前に保管された認証状態 (ユーザー名またはパスワードなど) をクリーンアップします。次のメソッドは、サブジェクトをログアウトします。

```
boolean logout() throws LoginException;
```

このメソッドは、元々

`Subject`

に関連付けられているプリンシパルおよび信任状を

`commit`

操作中に除去します。信任状は除去時に破棄されます。

CallbackHandler

場合によっては、LoginModule は、認証情報を取得するためにユーザーと通信しなければなりません。LoginModules はこのために CallbackHandler を使用します。アプリケーションは CallbackHandler **インターフェース**をインプリメントし、それを LoginContext に渡します。そして、基礎となる LoginModules に直接転送します。 LoginModules は、ユーザーからの入力 (パスワードまたはスマート・カード・ピン番号など) を収集することと、ユーザーに情報 (状況情報など) を提供することの両方のために、CallbackHandler を使用します。アプリケーションが CallbackHandler を指定できるようにすることにより、基礎となる LoginModules は、アプリケーションがユーザーと対話する様々な方法からは独立したままでいられます。たとえば、GUI アプリケーション用の CallbackHandler をインプリメントすると、ユーザーからの入力を送信請求するためにウィンドウが表示されます。非 GUI ツール用の CallbackHandler をインプリメントすると、ユーザーはコマンド行から直接入力するように促されます。

`CallbackHandler`

は、以下をインプリメントするための 1 つのメソッドとの**インターフェース**です。

```
void handle(Callback[] callbacks)
throws java.io.IOException, UnsupportedCallbackException;
```

Callback

javax.security.auth.callback パッケージには、Callback **インターフェース**といくつかのインプリメンテーションが含まれています。 LoginModules は、Callback の配列を CallbackHandler の *handle* メソッドに直接渡します。

使用法の詳細については、各種の Callback API を調べてください。

権限クラス

`Subject`

の認証が成功すると、Subject.doAs または Subject.doAsPrivileged メソッドを呼び出すことにより、きめ細かいアクセス制御をその

`Subject`

に対して設定することができます。その

`Subject`

に付与された許可は、JAAS

`Policy`

に構成されます。

Policy

これは、システム規模の JAAS アクセス制御を表すための**抽象**クラスです。デフォルトとして、JAAS はファイル・ベースのサブクラス・インプリメンテーション、PolicyFile を提供します。それぞれの

`Policy`

サブクラスは、以下のメソッドをインプリメントする必要があります。

```
public abstract java.security.PermissionCollection getPermissions
                (Subject subject,
                java.security.CodeSource cs);
public abstract void refresh();
```

`getPermissions`

メソッドは、指定された

`Subject`

および

`CodeSource`

に付与された許可を戻します。

`refresh`

メソッドは、ランタイム

`Policy`

を、それが永続ストア (例えば、ファイルまたはデータベース) から最後にロードされて以降に加えられた変更で更新します。

`refresh`

メソッドは、

`AuthPermission("refreshPolicy")`

を必要とします。

次のメソッドは、現行のランタイム

`Policy`

オブジェクトを取得し、呼び出し元に

`AuthPermission("getPolicy")`

を持つことを求めるセキュリティー検査で保護されています。

```
public static Policy getPolicy();
```

次のコードの例は、

`Policy`

オブジェクトを照会して、指定された

`Subject`

および

`CodeSource`

に付与された許可のセットを調べる方法を示しています。

```
policy = Policy.getPolicy();
PermissionCollection perms = policy.getPermissions(subject, codeSource);
```

Java ランタイムに新規の

`Policy`

を設定するには、

`Policy.setPolicy`

メソッドを使用します。このメソッドは、呼び出し元が

`AuthPermission("setPolicy")`

を持っていることを必要とします。

```
public static void setPolicy(Policy policy);
```

ポリシー・ファイルのサンプル項目:

これらの例は、デフォルトの PolicyFile インプリメンテーションにのみ関係があります。

`Policy`

内の各項目は、***grant*** 項目として表されます。各 ***grant*** 項目は、codebase/code-signers/Principals トリプレットを指定すると同時に、そのトリプレットに付与される許可を指定します。特に、指定された *codebase* からダウンロードされ、指定された *code signers* によって署名されたコードに、許可が付与されます。このことは、そのコードを実行する

`Subject`

が、指定されたすべての *Principals* をその

Principal

セットに持っている限り行われます。

Subject

が実行中のコードに関連付けられる様子については、Subject.doAs の例 を参照してください。

```
grant CodeBase ["URL"],
      Signedby ["signers"],
      Principal [Principal_Class] "Principal_Name",
      Principal ... {
    permission Permission_Class ["Target_Name"]
                                [, "Permission_Actions"]
                                [, signedBy "SignerName"];
};

// example grant entry
grant CodeBase "http://griffin.ibm.com", Signedby "davis",
      Principal com.ibm.security.auth.NTUserPrincipal "kent" {
    permission java.io.FilePermission "c:/kent/files/*", "read, write";
};
```

Principal 情報が JAAS

Policy

grant 項目で指定されていない場合、構文解析例外がスローされます。ただし、通常の Java 2 コード・ソース・ベースのポリシー・ファイルにすでに存在している (したがって、*Principal* 情報を持たない) grant 項目は、依然として有効です。そのような場合、*Principal* 情報は '*' (grant 項目はすべてのプリンシパルに適用される) であることが暗示されます。

grant 項目の CodeBase および Signedby コンポーネントは、JAAS

Policy

ではオプションです。それらが存在しない場合、どんなコードベースでもマッチングし、さらにどんな署名者 (符号なしコードを含む) でもマッチングします。

上記の例では、***grant*** 項目は、"http://griffin.ibm.com" からダウンロードされ、"davis" によって署名されて、NT ユーザー "kent" として実行しているコードが、1 つの

Permission

を持つことを指定します。この

Permission

は、処理コードがディレクトリー "c:¥kent¥files" にあるファイルの読み取り/書き込みを行うことを許可します。

複数のプリンシパルが 1 つの ***grant*** 項目内にリストされることがあります。コードを実行している現行の

Subject

は、その

Principal

セット内の指定されたすべてのプリンシパルが項目の許可を付与されるようにする必要があります。

```
grant Principal com.ibm.security.auth.NTUserPrincipal "kent",
      Principal com.ibm.security.auth.NTSidGroupPrincipal "S-1-1-0" {
    permission java.io.FilePermission "c:/user/kent/", "read, write";
    permission java.net.SocketPermission "griffin.ibm.com", "connect";
};
```

この項目は、NT グループ識別番号が "S-1-1-0" の NT ユーザー "kent" として実行する任意のコードに、"c:¥user¥kent" にあるファイルの読み取りおよび書き込みを行う許可と、"griffin.ibm.com" に対するソケット接続を行う許可の両方を付与します。

AuthPermission

このクラスは、JAAS に必要な基本的な許可をカプセル化します。 AuthPermission には、名前 (「ターゲット名」とも言われる) が含まれますが、アクション・リストは含まれません。名前付きの許可はあってもなくてもかまいません。 (

```
Permission
```

クラスから) 継承されたメソッドに加えて、

```
AuthPermission
```

には、次の 2 つの public コンストラクターがあります。

```
public AuthPermission(String name);
public AuthPermission(String name, String actions);
```

最初のコンストラクターは、指定された名前を持つ新規の AuthPermission を作成します。 2 番目のコンストラクターも、指定された名前を持つ新規の AuthPermission オブジェクトを作成しますが、追加の *actions* 引数 (現在は未使用で、ヌルになっている) を持っています。このコンストラクターは、新規の Permission オブジェクトをインスタンス化するために、

```
Policy
```

オブジェクト用にのみ存在します。たいていのコードの場合、最初のコンストラクターが適切です。

AuthPermission オブジェクトは、Policy、Subject、LoginContext、および Configuration オブジェクトへのアクセスを保護するために使用されます。サポートされる有効な名前のリストについては、AuthPermission Javadoc を参照してください。

PrivateCredentialPermission

このクラスは、Subject の秘密信任状へのアクセスを保護し、1 つの public コンストラクターを提供します。

```
public PrivateCredentialPermission(String name, String actions);
```

このクラスの詳細については、PrivateCredentialPermission Javadoc を参照してください。

インプリメンテーション

注: 付録 A には、ここで説明する静的プロパティーを含む、サンプルの **java.security** ファイルが記載されています。

JAAS プロバイダーおよびポリシー・ファイルにはデフォルト値が存在するため、ユーザーは JAAS をインプリメントするためにそれらのリストを静的に (java.security ファイル内で) リストすることも、動的に (コマンド行 ***-D*** オプション) リストすることも必要ありません。また、デフォルト構成およびポリシー・ファイル・プロバイダーは、ユーザー開発のプロバイダーによって置き換えられることがあります。そのた

め、このセクションでは、JAAS デフォルト・プロバイダーおよびポリシー・ファイルとともに、代わりのプロバイダーを使用可能にするプロパティーの説明を試みます。

ここで要約される事柄よりも詳しい情報については、Default Policy File API および Default Configuration File API をお読みください。

認証プロバイダー

認証プロバイダー、または構成クラスは、

```
login.configuration.provider=[class]
```

とともに *java.security* ファイルで静的に設定されます。このプロバイダーは、

```
Configuration
```

オブジェクトを作成します。

以下に例を示します。

```
login.configuration.provider=com.foo.Config
```

セキュリティー・プロパティー

```
login.configuration.provider
```

が java.security にない場合、JAAS はそれを次のデフォルト値に設定します。

```
com.ibm.security.auth.login.ConfigFile
```

```
Configuration
```

が作成される前にセキュリティー・マネージャーが設定される場合、

```
AuthPermission("getLoginConfiguration")
```

が付与される必要があります。

構成プロバイダーをコマンド行で動的に設定する方法はありません。

認証構成ファイル

認証構成ファイルは、*java.security* で

```
login.config.url.n=[URL]
```

とともに静的に設定されます。ここで、*n* は 1 から始まる連続する整数です。フォーマットは、Java セキュリティー・ポリシー・ファイル (policy.url.n=[URL]) のフォーマットと同一です。

セキュリティー・プロパティー

```
policy.allowSystemProperty
```

が java.security で "true" に設定される場合、ユーザーは次のプロパティーとともに ***-D*** オプションを使用して、コマンド行でポリシー・ファイルを動的に設定することができます。

```
java.security.auth.login.config
```

。値はパスまたは URL です。例 (NT の場合):

```
... -Djava.security.auth.login.config=c:¥config_policy¥login.config ...
または
... -Djava.security.auth.login.config=file:c:/config_policy/login.config ...
```

注: コマンド行で二重等号 (==) を使用すると、ユーザーは見つかったその他のすべてのポリシー・ファイルをオーバーライドすることができます。

構成ファイルが静的または動的に見つからない場合、JAAS は構成ファイルを次のデフォルト位置からロードしようとします。

```
${user.home}¥.java.login.config
```

ここで、*${user.home}* はシステムに依存する位置です。

権限プロバイダー

権限プロバイダー、または JAAS Policy クラスは、

```
auth.policy.provider=[class]
```

とともに *java.security* ファイルで静的に設定されます。このプロバイダーは、JAAS サブジェクト・ベースの

```
Policy
```

オブジェクトを作成します。

以下に例を示します。

```
auth.policy.provider=com.foo.Policy
```

セキュリティー・プロパティー

```
auth.policy.provider
```

が java.security にない場合、JAAS はそれを次のデフォルト値に設定します。

```
com.ibm.security.auth.PolicyFile
```

```
Configuration
```

が作成される前にセキュリティー・マネージャーが設定される場合、

```
AuthPermission("getPolicy")
```

が付与される必要があります。

権限プロバイダーをコマンド行で動的に設定する方法はありません。

権限ポリシー・ファイル

権限ポリシー・ファイルは、*java.security* で

```
auth.policy.url.n=[URL]
```

とともに静的に設定されます。ここで、*n* は 1 から始まる連続する整数です。フォーマットは、Java セキュリティー・ポリシー・ファイル (policy.url.n=[URL]) のフォーマットと同一です。

セキュリティー・プロパティー

```
policy.allowSystemProperty
```

が java.security で "true" に設定される場合、ユーザーは次のプロパティーとともに ***-D*** オプションを使用して、コマンド行でポリシー・ファイルを動的に設定することができます。

```
java.security.auth.policy
```

。値はパスまたは URL です。例 (NT の場合):

```
... -Djava.security.auth.policy=c:¥auth_policy¥java.auth.policy ...
または
... -Djava.security.auth.policy=file:c:/auth_policy/java.auth.policy ...
```

注: コマンド行で二重等号 (==) を使用すると、ユーザーは見つかったその他のすべてのポリシー・ファイルをオーバーライドすることができます。

権限ポリシーのロード元にするデフォルト位置はありません。

"Hello World"、JAAS スタイル!

黒っぽいサングラスをかけ、お気に入りのフェドーラ帽をかぶり、手にはアルト・サックスを取り ... **JAAS**-y を始めるときが来ました! これがもう一つの "Hello World!" プログラムです。このセクションでは、JAAS インストールをテストするためのプログラムを使用可能にします。

インストール: JAAS がインストールされていることが想定されています。たとえば、JAAS JAR ファイルがご使用の Development Kit の extensions ディレクトリーにコピーされています。

ファイルの取得: HelloWorld.tar をご使用のテスト・ディレクトリーにダウンロードします。 "jar xvf HelloWorld.tar" を使用して、それを解凍します。

テスト・ディレクトリーの内容を検査します。

ソース・ファイル:

- HWLoginModule.java
- HWPrincipal.java
- HelloWorld.java

クラス・ファイル

- ソース・ファイルは、classes ディレクトリーにプリコンパイルされています。

ポリシー・ファイル

- jaas.config
- java2.policy
- jaas.policy

ソース・ファイルのコンパイル: 3 つのソース・ファイル、***HWLoginModule.java***、***HWPrincipal.java***、および ***HelloWorld.java*** はすでにコンパイルされているため、コンパイルする必要はありません。

ソース・ファイルが変更される場合、それらが保管されているテスト・ディレクトリーに変更して、次のように入力します。

```
javac -d .¥classes *.java
```

クラスパスに classes ディレクトリー (.¥classes) を追加して、クラスをコンパイルできるようにする必要があります。

注:

```
HWLoginModule
```

および

`HWPrincipal`

は、

`com.ibm.security`

パッケージにあり、コンパイル時に適切なディレクトリー (>test_dir<¥classes¥com¥ibm¥security) に作成されます。

ポリシー・ファイルの調査: 構成ファイル ***jaas.config*** には次の 1 つの項目が含まれています。

```
helloWorld {
   com.ibm.security.HWLoginModule required debug=true;
};
```

1 つの

`LoginModule`

だけがテスト・ケースに提供されます。***HelloWorld*** アプリケーションを処理するときには、

`LoginModuleControlFlag`

(required、requisite、sufficient、optional) を変えたり、デバッグ・フラグを削除したりして、試してみてください。テストで利用できる LoginModule が他にもある場合、この構成を自由に変えて複数の LoginModule で試してみることもできます。

`HWLoginModule`

について、簡単に説明します。

Java 2 ポリシー・ファイル、***java2.policy*** には、1 つの許可ブロックが含まれています。

```
grant {
   permission javax.security.auth.AuthPermission "createLoginContext";
   permission javax.security.auth.AuthPermission "modifyPrincipals";
   permission javax.security.auth.AuthPermission "doAsPrivileged";
};
```

HelloWorld アプリケーションは、(1) LoginContext オブジェクトを作成し、(2) 認証された

`Subject`

のプリンシパルを変更し、(3)

`Subject`

クラスの doAsPrivileged メソッドを呼び出すため、3 つの許可が必要です。

JAAS ポリシー・ファイル ***jaas.policy*** にも、1 つの許可ブロックが含まれています。

```
grant Principal com.ibm.security.HWPrincipal "bob" {
   permission java.util.PropertyPermission "java.home", "read";
   permission java.util.PropertyPermission "user.home", "read";
   permission java.io.FilePermission "foo.txt", "read";
};
```

3 つの許可は、最初に ***bob*** という名前の

`HWPrincipal`

に認可されます。認証された

`Subject`

に追加される実際のプリンシパルは、ログイン・プロセス (さらに後) で使用されるユーザー名です。

以下に ***HelloWorld*** のアクション・コードを示します。 3 つのシステム呼び出し (許可が必要である理由) が**ボールド体**で示されています。

```
Subject.doAsPrivileged(lc.getSubject(), new PrivilegedAction() {
    public Object run() {
        System.out.println("¥nYour java.home property: "
                          +System.getProperty("java.home"));

        System.out.println("¥nYour user.home property: "
                          +System.getProperty("user.home"));

        File f = new File("foo.txt");
        System.out.print("¥nfoo.txt does ");
        if (!f.exists()) System.out.print("not ");
        System.out.println("exist in your current directory");

        System.out.println("¥nOh, by the way ...");

                try {
            Thread.currentThread().sleep(2000);
        } catch (Exception e) {
            // ignore
        }
        System.out.println("¥n¥nHello World!¥n");
        return null;
    }
}, null);
```

HelloWorld プログラムを実行するときは、さまざまなユーザー名を使用し、それに応じて ***jaas.policy*** を変更できます。 ***java2.policy*** を変更する必要はありません。また、テスト・ディレクトリー内に ***foo.txt*** というファイルを作成して、最後のシステム呼び出しをテストします。

ソース・ファイルの調査:

LoginModule、

`HWLoginModule`

は、正しいパスワード (大文字小文字の区別がある) を入力したユーザーを認証します: **Go JAAS**。

HelloWorld アプリケーションでは、ユーザーは 3 回までパスワードの入力を試みることができます。 **Go JAAS** が正しく入力されると、ユーザー名と同じ名前を持つ

`HWPrincipal`

が、認証された

`Subject`

に追加されます。

Principal クラスの

`HWPrincipal`

は、入力されたユーザー名に基づくプリンシパルを表します。この名前は、認証されたサブジェクトに許可を与える際に重要になります。

メイン・アプリケーションの

`HelloWorld`

は、**helloWorld** という名前の構成項目に基づいて、

LoginContext

をまず作成します。構成ファイルについてはすでに説明されています。ユーザー入力を検索するためにコールバックが使用されます。このプロセスを調べるには、***HelloWorld.java*** ファイルにある

MyCallbackHandler

クラスを参照してください。

```
LoginContext lc = null;
try {
    lc = new LoginContext("helloWorld", new MyCallbackHandler());
} catch (LoginException le) {
    le.printStackTrace();
    System.exit(-1);
}
```

ユーザーがユーザー名とパスワードを (最大 3 回) 入力し、**Go JAAS** がパスワードとして入力されると、サブジェクトは認証されます (

HWLoginModule

によってサブジェクトに

HWPrincipal

が追加されます)。

すでに説明したとおり、以後の作業は、認証されたサブジェクトとして実行されます。

HelloWorld テストの実行

HelloWorld プログラムを実行するには、最初にテスト・ディレクトリーに変更します。構成およびポリシー・ファイルをロードする必要があります。正しいプロパティーについてインプリメンテーションを参照して、 *java.security* かコマンド行のいずれかに設定します。後者のメソッドについて、これから説明します。

以下のコマンドは、明確にするために複数の行に分割されています。 1 つの連続するコマンドとして入力してください。

```
java -Djava.security.manager=
     -Djava.security.auth.login.config=.¥jaas.config
     -Djava.security.policy=.¥java2.policy
     -Djava.security.auth.policy=.¥jaas.policy
     HelloWorld
```

注: 各ユーザーのテスト・ディレクトリー標準パスは異なるため、ポリシー・ファイルに ".¥filename" を使用することが必要です。希望する場合は、 "." をテスト・ディレクトリーへのパスの代わりにしてください。たとえば、テスト・ディレクトリーが "c:¥test¥hello" の場合、最初のファイルは次のように変更されます。

```
     -Djava.security.auth.login.config=c:¥test¥hello¥jaas.config
```

ポリシー・ファイルが見つからない場合は、

SecurityException

がスローされます。見つかった場合は、 ***java.home*** および ***user.home*** プロパティーに関する情報が表示されます。また、テスト・ディレクトリーに ***foo.txt*** というファイルがあるかどうかが検査されます。最後に、いたるところに "Hello World" というメッセージが表示されます。

HelloWorld を楽しむ

好きなだけ ***HelloWorld*** を再実行してください。入力したユーザー名/パスワードを変え、構成ファイル項目を変更し、ポリシー・ファイル許可を変更し、さらに追加の LoginModules を ***helloWorld*** 構成項目に追加する (積み重ねる) ことまで、すでに提案されました。さらに、codebase フィールドをポリシー・ファイルに追加することができます。

最後に、セキュリティー・マネージャーなしでプログラムを実行して、問題発生時における動作を調べてみてください。

付録 A: java.security セキュリティー・プロパティー・ファイル・ファイルでの JAAS 設定

以下に示すのは、すべての Java 2 インストールに現れる

`java.security`

ファイルです。このファイルは、Java 2 ランタイムの

`lib/security`

(Windows では

`lib¥security`

) ディレクトリーに現れます。したがって、Java 2 ランタイムが

`jdk1.3`

というディレクトリーにインストールされている場合、ファイルは以下のようになります。

- `jdk1.3/lib/security/java.security`

  (Unix)
- `jdk1.3¥lib¥security¥java.security`

  (Windows)

JAAS は 4 つの新規のプロパティーを

`java.security`

に追加します。

- 認証プロパティー
  - `login.configuration.provider`
  - `login.policy.url.n`
- 権限プロパティー
  -

auth.policy.provider

–

auth.policy.url.n

新規の JAAS プロパティーは、このファイルの末尾に置かれます。

```
#
# This is the "master security properties file".
#
# In this file, various security properties are set for use by
# java.security classes. This is where users can statically register
# Cryptography Package Providers ("providers" for short). The term
# "provider" refers to a package or set of packages that supply a
# concrete implementation of a subset of the cryptography aspects of
# the Java Security API. A provider may, for example, implement one or
# more digital signature algorithms or message digest algorithms.
#
# Each provider must implement a subclass of the Provider class.
# To register a provider in this master security properties file,
# specify the Provider subclass name and priority in the format
#
#    security.provider.n=className
#
# This declares a provider, and specifies its preference
# order n. The preference order is the order in which providers are
# searched for requested algorithms (when no specific provider is
# requested). The order is 1-based; 1 is the most preferred, followed
# by 2, and so on.
#
# className must specify the subclass of the Provider class whose
# constructor sets the values of various properties that are required
# for the Java Security API to look up the algorithms or other
# facilities implemented by the provider.
#
# There must be at least one provider specification in java.security.
# There is a default provider that comes standard with the JDK. It
# is called the "SUN" provider, and its Provider subclass
# named Sun appears in the sun.security.provider package. Thus, the
# "SUN" provider is registered via the following:
#
#    security.provider.1=sun.security.provider.Sun
#
# (The number 1 is used for the default provider.)
#
# Note: Statically registered Provider subclasses are instantiated
# when the system is initialized. Providers can be dynamically
# registered instead by calls to either the addProvider or
# insertProviderAt method in the Security class.

#
# List of providers and their preference orders (see above):
#
security.provider.1=sun.security.provider.Sun

#
# Class to instantiate as the system Policy. This is the name of the class
# that will be used as the Policy object.
#
policy.provider=sun.security.provider.PolicyFile
# The default is to have a single system-wide policy file,
# and a policy file in the user's home directory.
policy.url.1=file:${java.home}/lib/security/java.policy
policy.url.2=file:${user.home}/.java.policy

# whether or not we expand properties in the policy file
# if this is set to false, properties (${...}) will not be expanded in policy
```

```
# files.
policy.expandProperties=true

# whether or not we allow an extra policy to be passed on the command line
# with -Djava.security.policy=somefile. Comment out this line to disable
# this feature.
policy.allowSystemProperty=true

# whether or not we look into the IdentityScope for trusted Identities
# when encountering a 1.1 signed JAR file. If the identity is found
# and is trusted, we grant it AllPermission.
policy.ignoreIdentityScope=false

#
# Default keystore type.
#
keystore.type=jks

#
# Class to instantiate as the system scope:
#
system.scope=sun.security.provider.IdentityDatabase

##############################################################################
#
# Java Authentication and Authorization Service (JAAS)
# properties and policy files:
#

# Class to instantiate as the system Configuration for authentication.
# This is the name of the class that will be used as the Authentication
# Configuration object.
#
login.configuration.provider=com.ibm.security.auth.login.ConfigFile
# The default is to have a system-wide login configuration file found in
# the user's home directory.  For multiple files, the format is similar to
# that of CodeSource-base policy files above, that is policy.url.n
login.config.url.1=file:${user.home}/.java.login.config

# Class to instantiate as the system Principal-based Authorization Policy.
# This is the name of the class that will be used as the Authorization
# Policy object.
#
auth.policy.provider=com.ibm.security.auth.PolicyFile
# The default is to have a system-wide Principal-based policy file found in
# the user's home directory.  For multiple files, the format is similar to
# that of CodeSource-base policy files above, that is policy.url.n and
# auth.policy.url.n
auth.policy.url.1=file:${user.home}/.java.auth.policy
```

付録 B: ログイン構成ファイル

ログイン構成ファイルには、以下の形式の 1 つ以上の

`LoginContext`

アプリケーション名が含まれています。

```
Application {
    LoginModule Flag ModuleOptions;
    > more LoginModule entries <
    LoginModule Flag ModuleOptions;
};
```

ログイン構成ファイルは、

`java.security`

ファイルにある

`login.config.url.n`

セキュリティー・プロパティーを使用して配置されます。このプロパティーおよび

`java.security`

ファイルの位置については、付録 A を参照してください。

Flag の値は、認証が回を重ねて進むときの全部的な動作を制御します。以下は、*Flag* の有効な値と、それぞれの意味の説明を表しています。

1. **Required**。

   `LoginModule`

   は成功する必要があります。成功しても失敗しても、認証は

   `LoginModule`

   リストの処理を引き続き先に進めます。
2. **Requisite**。

   `LoginModule`

   は成功する必要があります。成功すると、認証は

   `LoginModule`

   リストの処理を継続します。失敗すると、制御はただちにアプリケーションに戻ります (認証は

   `LoginModule`

   リストの処理を先に進めません)。
3. **Sufficient**。

   `LoginModule`

   は成功しなくても構いません。成功すると、制御はただちにアプリケーションに戻ります (認証は

   `LoginModule`

   リストの処理を先に進めません)。失敗すると、認証は

   `LoginModule`

   リストの処理を継続します。
4. **Optional**。

   `LoginModule`

   は成功しなくても構いません。成功しても失敗しても、認証は

   `LoginModule`

   リストの処理を引き続き先に進めます。

全体の認証は、すべての *Required* および *Requisite* の LoginModules が正常に実行される場合に限り成功します。

Sufficient の

`LoginModule`

が構成されて成功する場合、全体の認証が成功するには、その *Sufficient* の

`LoginModule`

の前に *Required* および *Requisite* の LoginModules だけが正常に実行される必要があります。 *Required* または *Requisite* の LoginModules がアプリケーション用に構成されていない場合、少なくとも 1 つの *Sufficient* または *Optional* の

`LoginModule`

が成功する必要があります。

サンプル構成ファイル:

```
/*  Sample Configuration File  */

Login1 {
   com.ibm.security.auth.module.SampleLoginModule required debug=true;
};

Login2 {
   com.ibm.security.auth.module.SampleLoginModule required;
   com.ibm.security.auth.module.NTLoginModule sufficient;
   ibm.loginModules.SmartCard requisite debug=true;
   ibm.loginModules.Kerberos optional debug=true;
};
```

注: フラグには大文字小文字の区別がありません。 *REQUISITE* = *requisite* = *Requisite* です。

Login1 は、クラス

`com.ibm.security.auth.module.SampleLoginModule`

のインスタンスである 1 つの LoginModule のみ持っています。したがって、**Login1** に関連した

`LoginContext`

は、そのただ 1 つのモジュールが正常に認証を行う場合に限り、正常に認証されます。 *Required* フラグは、この例では意味がありません。複数のモジュールが存在するときに、フラグ値は認証に関係のある影響を与えます。

Login2 は、表を使って説明する方が簡単です。

Login2 認証状況									
サンプル・ログイン・モジュール	required	pass	pass	pass	pass	fail	fail	fail	fail
NT ログイン・モジュール	sufficient	pass	fail	fail	fail	pass	fail	fail	fail
スマート・カード	requisite	*	pass	pass	fail	*	pass	pass	fail
Kerberos	optional	*	pass	fail	*	*	pass	fail	*
全体の認証		pass	pass	pass	fail	fail	fail	fail	fail

* = 以前の *REQUISITE* モジュールが失敗したか、または以前の *SUFFICIENT* モジュールが成功したため、アプリケーションに制御が戻ったことによる、意味のない値。

付録 C: 権限ポリシー・ファイル

上記のプリンシパル・ベースの JAAS ポリシーの grant ブロックの例では十分でない場合、ここにさらに説明があります。

```
// SAMPLE JAAS POLICY FILE:  java.auth.policy

// The following permissions are granted to Principal 'Pooh' and all codesource:

grant Principal com.ibm.security.Principal "Pooh" {
   permission javax.security.auth.AuthPermission "setPolicy";
   permission java.util.PropertyPermission "java.home", "read";
   permission java.util.PropertyPermission "user.home", "read";
   permission java.io.FilePermission "c:/foo/jaas.txt", "read";
};

// The following permissions are granted to Principal 'Pooh' AND 'Eyeore'
// and CodeSource signedBy "DrSecure":

grant signedBy "DrSecure"
      Principal com.ibm.security.Principal "Pooh",
      Principal com.ibm.security.Principal "Eyeore" {
   permission javax.security.auth.AuthPermission "modifyPublicCredentials";
   permission javax.security.auth.AuthPermission "modifyPrivateCredentials";
   permission java.net.SocketPermission "us.ibm.com", "connect,accept,resolve";
   permission java.net.SocketPermission "griffin.ibm.com", "accept";
};

// The following permissions are granted to Principal 'Pooh' AND 'Eyeore' AND
// 'Piglet' and CodeSource from the c:¥jaas directory signed by "kent" and "bruce":

grant codeBase "file:c:/jaas/*",
      signedBy "kent, bruce",
      Principal com.ibm.security.Principal "Pooh",
      Principal com.ibm.security.Principal "Eyeore",
      Principal com.ibm.security.Principal "Piglet"  {
   permission javax.security.auth.AuthPermission "getSubject";
   permission java.security.SecurityPermission "printIdentity";
   permission java.net.SocketPermission "guapo.ibm.com", "accept";
};
```

IBM Java Generic Security Service (JGSS)

Java Generic Security Service (JGSS) は、認証およびセキュア・メッセージング用の汎用インターフェースを提供します。このインターフェースで、秘密鍵、公開鍵、または他のセキュリティー・テクノロジーをベースにした各種のメカニズムを使用することができます。

基礎となるセキュリティー・メカニズムの複雑さや特性を標準インターフェースで一般化することにより、JGSS はセキュア・ネットワーク・アプリケーションの開発に以下の利点を提供します。

- 単一の抽象インターフェースを利用するアプリケーションを開発することができる。
- 変更を加えることなく、セキュリティー・メカニズムの異なるアプリケーションを使用することができる。

JGSS は Generic Security Service Application Programming Interface (GSS-API) 用の Java バインディングを定義しますが、その API は Internet Engineering Task Force (IETF) によって標準化され、X/Open Group によって採用されている 暗号 API です。

IBM JGSS インプリメンテーションは IBM JGSS と呼ばれています。 IBM JGSS は基礎となるデフォルト・セキュリティー・システムとして Kerberos V5 を使用する GSS-API フレームワークのインプリメンテーションです。それはまた、Kerberos 信任状を作成し使用するための Java Authentication and Authorization Service (JAAS) ログイン・モジュールとしても機能します。さらに、それらの信任状を使用する場合、JGSS に JAAS 権限を実行させせることもできます。

IBM JGSS には、ネイティブ IBM i JGSS プロバイダー、Java JGSS プロバイダー、および Kerberos 信任状管理ツール (kinit、ktab、および klist) の Java バージョンが組み込まれています。

注: ネイティブ IBM i JGSS プロバイダーは、ネイティブ IBM i Network Authentication Services (NAS) ライブラリーを使用します。ネイティブ・プロバイダーを使用する場合、ネイティブ IBM i Kerberos ユーティリティーを使用しなければなりません。詳しくは、JGSS プロバイダー を参照してください。

Security enhancement from Sun Microsystems, Inc.

Internet Engineering Task Force (IETF) RFC 2743 Generic Security Services Application Programming Interface Version 2, Update 1

IETF RFC 2853 Generic Security Service API Version 2: Java Bindings

The X/Open Group GSS-API Extensions for DCE

JGSS の概念

JGSS の操作は、Generic Security Service Application Programming Interface (GSS-API) によって標準化されているように、4 つの異なる段階で構成されています。

この段階は次のとおりです。

1. プリンシパル用信任状の収集。
2. 対等プリンシパル通信間のセキュリティー・コンテキストの作成および設定。
3. 対等間のセキュア・メッセージの交換。
4. リソースのクリーンアップおよびリリース。

さらに、JGSS は Java Cryptographic Architecture を活用し、異なるセキュリティー・メカニズムのシームレスなプラグを可能にします。

以下のリンクから、これら重要な JGSS の概念についての高水準の説明を読むことができます。

JGSS プリンシパルおよび信任状:

アプリケーションが対等機能と JGSS セキュア通信を行うための ID をプリンシパルと呼びます。プリンシパルは、実ユーザー、または自動サービスの場合もあります。プリンシパルはそのメカニズムのもとで、ID を証明するものとしてセキュリティー・メカニズム特定信任状を獲得します。

たとえば、Kerberos メカニズムを使用する場合、プリンシパルの信任状は Kerberos 鍵配布センター (KDC) によって発行されるチケット許可チケットの形式をとります。マルチ・メカニズム環境では、GSS-API 信任状は複数の信任状エレメントを含むことができ、各エレメントは 1 つの基礎となるメカニズム信任状を表します。

GSS-API 規格では、プリンシパルが信任状を獲得する方法は規定されておらず、GSS-API インプリメンテーションは信任状の獲得手段を提供しないのが普通です。プリンシパルは GSS-API を使用する前に信任状を得ます。GSS-API はプリンシパルのために信任状を得るセキュリティー・メカニズムを単に照会するにすぎません。

IBM JGSS には、Java バージョンの Kerberos 信任状管理ツール com.ibm.security.krb5.internal.tools Class Kinit、com.ibm.security.krb5.internal.tools Class Ktab、および com.ibm.security.krb5.internal.tools Class Klist が組み込まれています。さらに、IBM JGSS は、JAAS を使用するオプションの Kerberos ログイン・インターフェースを提供することによって標準 GSS-API を強化します。純正の Java JGSS プロバイダーはオプションのログイン・インターフェースをサポートしていますが、ネイティブ IBM i プロバイダーはこれをサポートしていません。

関連概念

378 ページの『Kerberos 信任状の取得および秘密鍵の作成』
GSS-API は、信任状を取得するための方法を定義していません。 このため、IBM JGSS Kerberos メカニズムでは、ユーザーが、Kerberos 信任状を取得することが必要です。 このトピックでは、Kerberos 証明書を取得して秘密鍵を作成する方法、および Kerberos ログインおよび許可検査を実行するための JAAS の使用方法を学び、Java 仮想マシン (JVM) が必要とする JAAS 許可のリストを学習できます。

375 ページの『JGSS プロバイダー』
IBM JGSS には、ネイティブ IBM i JGSS プロバイダーおよび純粋の Java JGSS プロバイダーが含まれています。どのプロバイダーを選択して使用するかは、アプリケーションの必要に応じて異なります。

com.ibm.security.krb5.internal.tools Class Klist:

このクラスは、信任状キャッシュおよびキー・タブにある項目をリストするためのコマンド行ツールとして実行できます。

```
java.lang.Object
|
+--com.ibm.security.krb5.internal.tools.Klist
```

public class **Klist**
extends java.lang.Object

このクラスは、信任状キャッシュおよびキー・タブにある項目をリストするためのコマンド行ツールとして実行できます。

コンストラクターの要約

`Klist()`

メソッドの要約

static void	`main(java.lang.String[] args)` コマンド行で呼び出すことができるメインプログラム。

クラス java.lang.Object から継承されるメソッド

`equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

コンストラクターの詳細

Klist

```
public Klist()
```

メソッドの詳細

main

```
public static void main(java.lang.String[] args)
```

コマンド行で呼び出すことができるメインプログラム。

使用法: java com.ibm.security.krb5.tools.Klist [[-c] [-f] [-e] [-a]] [-k [-t] [-K]] [name]

信任状キャッシュに使用可能なオプション:
- **-f** は信任状フラグを示します
- **-e** は暗号化タイプを示します
- **-a** はアドレス・リストを表示します

キー・タブに使用可能なオプション:
- **-t** はキー・タブ項目のタイム・スタンプを示します
- **-K** はキー・タブ項目の DES キーを示します

com.ibm.security.krb5.internal.tools Class Kinit:

Kerberos v5 チケットを取得するための Kinit ツール。

```
java.lang.Object
|
+--com.ibm.security.krb5.internal.tools.Kinit
```

public class **Kinit**
extends java.lang.Object

Kerberos v5 チケットを取得するための Kinit ツール。

コンストラクターの要約

`Kinit(java.lang.String[] args)`
新規の Kinit オブジェクトを構成します。

メソッドの要約

static void	main(java.lang.String[] args) メイン・メソッドは、チケット要求のためのユーザー・コマンド行入力を受け入れるために使用されます。

クラス java.lang.Object から継承されるメソッド

```
equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait
```

コンストラクターの詳細

Kinit

```
public Kinit(java.lang.String[] args)
      throws java.io.IOException,
             RealmException,
             KrbException
```

新規の Kinit オブジェクトを構成します。

パラメーター:

args - チケット要求オプションの配列。使用可能なオプションは、-f、-F、-p、-P、-c、-k、principal、password です。

スロー:

java.io.IOException - 入出力エラーが発生した場合。
RealmException - レルムをインスタンス化できなかった場合。
KrbException - Kerberos 操作中にエラーが発生した場合。

メソッドの詳細

main

```
public static void main(java.lang.String[] args)
```

メイン・メソッドは、チケット要求のためのユーザー・コマンド行入力を受け入れるために使用されます。

使用法: java com.ibm.security.krb5.tools.Kinit [-f] [-F] [-p] [-P] [-k] [-c cache name] [principal] [password]

- **-f** 転送可能
- **-F** 転送不可
- **-p** プロキシー可能
- **-P** プロキシー不可
- **-c** キャッシュ名 (すなわち、FILE:d:¥temp¥mykrb5cc)
- **-k** キー・タブを使用する
- **-t** キー・タブ・ファイル名
- principal プリンシパル名 (すなわち、qwedf qwedf@IBM.COM)
- password プリンシパルの Kerberos パスワード

java com.ibm.security.krb5.tools.Kinit -help を使用して、ヘルプ・メニューを立ち上げます。

現在、ファイル・ベースの信任状キャッシュのみサポートされています。デフォルトでは、KDC から取得したチケットを保管するために、krb5cc_{user.name} という名前のキャッシュ・ファイルが {user.home} ディレクトリーに生成されます。たとえば、Windows NT では、c:\winnt\profiles\qwedf\krb5cc_qwedf になります。ここで、qwedf は {user.name} で、c:\winnt\profile\qwedf は {user.home} です。{user.home} は、Kerberos によって Java システム・プロパティー "user.home" から取得されます。時として、{user.home} がヌルである場合 (ほとんどない)、キャッシュ・ファイルはプログラムが実行している現行ディレクトリーに保管されます。 {user.name} は、オペレーティング・システムのログイン・ユーザー名です。これは、ユーザーのプリンシパル名と異なっている可能性があります。 1 人のユーザーが複数のプリンシパル名を使用できますが、信任状キャッシュの 1 次プリンシパルにできるのは 1 つだけです。これは 1 つの

キャッシュ・ファイルが 1 つの特定のユーザー・プリンシパル用のチケットしか保管できないことを意味します。ユーザーが次の Kinit でプリンシパル名を交換する場合、デフォルトで、新規のチケット用に生成されるキャッシュ・ファイルが古いキャッシュ・ファイルを上書きします。上書きされないようにするには、新規のチケットを要求するときに、別のディレクトリーまたは別のキャッシュ・ファイルを指定する必要があります。

キャッシュ・ファイルの場所

ユーザー固有のキャッシュ・ファイルの名前および場所を指定するには、いくつかの方法があります。Kerberos が検索する順序は以下に示すとおりです。

1. **-c** オプション。 java com.ibm.security.krb5.tools.Kinit -c FILE:<ユーザー固有のディレクトリーおよびファイル名>。 "FILE:" は、信任状キャッシュのタイプを識別するための接頭部です。デフォルトはファイル・ベースのタイプです。
2. 実行時に、-DKRB5CCNAME=FILE:<ユーザー固有のディレクトリーおよびファイル名> を使用して、Java システム・プロパティー "KRB5CCNAME" を設定します。
3. 実行時の前に、環境変数 "KRB5CCNAME" をコマンド・プロンプトで設定します。オペレーティング・システムが異なると、環境変数を設定する方法も異なります。たとえば、Windows では set KRB5CCNAME=FILE:<ユーザー固有のディレクトリーおよびファイル名> を使用しますが、UNIX では export KRB5CCNAME=FILE:<ユーザー固有のディレクトリーおよびファイル名> を使用します。Kerberos では、システム固有のコマンドに依存して環境変数を検索することに注意してください。UNIX で使用されるコマンドは、"/usr/bin/env" です。

KRB5CCNAME には大/小文字の区別があり、すべて大文字です。

KRB5CCNAME が上記の説明のとおりに設定されていない場合、デフォルトのキャッシュ・ファイルが使用されます。デフォルト・キャッシュは、次の順序で位置指定されます。

1. Unix プラットフォームでは /tmp/krb5cc_<uid>。ここで、<uid> は Kinit JVM を稼働しているユーザーのユーザー ID です。
2. <user.home>/krb5cc_<user.name>。ここで、<user.home> および <user.name> はそれぞれ Java user.home および user.name プロパティーです。
3. <user.home>/krb5cc (<user.name> を JVM から取得できない場合)

KDC 通信タイムアウト

Kinit は、信任状であるチケット許可チケットを獲得するために鍵配布センター (KDC) と通信します。この通信は、KDC が一定の期間内に応答しない場合にタイムアウトするように設定できます。タイムアウト期間は、Kerberos 構成ファイル内の libdefaults スタンザ (すべての KDC に適用可能) または個々の KDC スタンザで (ミリ秒単位で) 設定できます。デフォルトのタイムアウト値は 30 秒です。

com.ibm.security.krb5.internal.tools Class Ktab:

このクラスは、ユーザーがキー・テーブル内の項目を管理するのを助けるためのコマンド行ツールとして実行できます。使用可能な関数には、list/add/update/delete サービス・キーが含まれます。

```
java.lang.Object
|
+--com.ibm.security.krb5.internal.tools.Ktab
```

public class **Ktab**
extends java.lang.Object

このクラスは、ユーザーがキー・テーブル内の項目を管理するのを助けるためのコマンド行ツールとして実行できます。使用可能な関数には、list/add/update/delete サービス・キーが含まれます。

コンストラクターの要約

`Ktab()`

メソッドの要約

`static void`	`main(java.lang.String[] args)` コマンド行で呼び出すことができるメインプログラム。

クラス java.lang.Object から継承されるメソッド

`equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

コンストラクターの詳細

Ktab

`public Ktab()`

メソッドの詳細

main

`public static void main(java.lang.String[] args)`

コマンド行で呼び出すことができるメインプログラム。

使用法: java com.ibm.security.krb5.tools.Ktab <オプション>

Ktab に使用可能なオプション:

- **-l** キー・タブ名および項目をリストします
- **-a** <*principal name*>(<*password*>) 項目をキー・タブに追加します
- **-d** <*principal name*> 項目をキー・タブから削除します
- **-k** <*keytab name*> キー・タブ名およびパスに接頭部 FILE: を付けて指定します
- **-help** 指示を表示します

JGSS コンテキストの設定:

セキュリティー信任状を獲得した後、2 つの通信対等機能はその信任状を使用してセキュリティー・コンテキストを設定します。 2 つの対等機能は 1 つの結合コンテキストを設定しますが、各対等機能はそのコンテキストの自分自身のローカル・コピーを維持します。コンテキストの設定には、開始側ピアが受入側ピアに対して、自分自身を認証することが含まれます。イニシエーターは相互参照の要求を選択することができますが、その場合、アクセプターは自身をイニシエーターに認証します。

コンテキストの設定が完了する際、設定されたコンテキストは 2 つの対等機能間でのその後のセキュア・メッセージ交換を可能にする状態情報 (共有暗号鍵など) を作成します。

JGSS メッセージの保護および交換:

コンテキストが設定されると、2 つの対等機能のセキュア・メッセージ交換が可能になります。メッセージの発信元は、メッセージをエンコードするためのローカル GSS-API インプリメンテーションを呼び出します。これにより、メッセージの保全性を確実することができ、ある場合はメッセージの機密性を確実にすることができます。その後、アプリケーションは対等機能に結果トークンを送ります。

対等機能のローカル GSS-API インプリメンテーションは、設定されたコンテキストからの情報を以下の方法で使用します。

- メッセージの保全性を検証する
- メッセージを暗号解読する (メッセージが暗号化されている場合)

リソースのクリーンアップおよび解放:

リソースを解放するために、JGSS アプリケーションは不要なコンテキストを削除します。 JGSS アプリケーションは削除されたコンテキストにアクセスすることはできますが、メッセージ交換のためにそれを使用すると例外が発生します。

セキュリティー・メカニズム:

GSS-API は、1 つ以上の基礎となるセキュリティー・メカニズムの抽象フレームワークから構成されています。どのようにフレームワークが基礎となるセキュリティー・メカニズムと相互作用するかは、インプリメンテーションによって異なります。

そのようなインプリメンテーションは 2 つの一般カテゴリーに分けられます。

- 一方のモノリシックなインプリメンテーションは、フレームワークを単一のメカニズムに強力にバインドします。この種類のインプリメンテーションでは、他のメカニズムや同じメカニズムの別のインプリメンテーションでさえ使用できなくなります。
- もう一方の高度なモジュラー・インプリメンテーションは、使いやすさと柔軟性を提供します。この種類のインプリメンテーションでは、別のセキュリティー・メカニズムとそのインプリメンテーションをフレームワークにシームレスかつ容易に結合することができます。

IBM JGSS は後者のカテゴリーに属します。モジュラー・インプリメンテーションとして、IBM JGSS は Java Cryptographic Architecture (JCA) によって定義されたプロバイダー・フレームワークを活用し、すべての基本メカニズムを (JCA) プロバイダーとして取り扱います。 JGSS プロバイダーは JGSS セキュリティー・メカニズムの具体的なインプリメンテーションを提供します。アプリケーションは複数のメカニズムをインスタンス化して使用します。

プロバイダーが複数のメカニズムをサポートするのは可能であり、JGSS は異なるセキュリティー・メカニズムを簡単に使用できるようにします。しかし、GSS-API は、複数のメカニズムが利用可能な場合に、2 つの通信対等機能がメカニズムを選択する手段を提供しません。メカニズムを選択する 1 つの方法は、Simple And Protected GSS-API Negotiating Mechanism (SPNEGO) で開始することです。これは、2 つの対等機能間の実際のメカニズムを折衝する疑似メカニズムです。 IBM JGSS には SPNEGO メカニズムは含まれていません。

SPNEGO について詳しくは、Internet Engineering Task Force (IETF) RFC 2478 The Simple and Protected GSS-API Negotiation Mechanism をご覧ください。

IBM JGSS を使用するようにサーバーを構成する

JGSS を使用するようにサーバーを構成する方法は、ご使用のシステムでどのバージョンの Java 2 Platform Standard Edition (J2SE) が稼働しているかによって異なります。

JGSS を使用するように IBM i を構成する:

ご使用のサーバー上で Java 2 Software Development Kit (J2SDK) バージョン 1.4 またはそれ以上を使用する場合、JGSS はすでに構成済みです。デフォルトの構成は、純粋の Java JGSS プロバイダーを使用します。

JGSS プロバイダーの変更

純粋の Java JGSS プロバイダーの代わりにネイティブ IBM i JGSS プロバイダーを使用するように、JGSS を構成することができます。ネイティブ・プロバイダーを使用するように JGSS を構成した後は、2 つのプロバイダーを容易に切り替えることができるようになります。詳しくは、『JGSS プロバイダー』を参照してください。

セキュリティー・マネージャー

使用可能な Java セキュリティー・マネージャーで JGSS アプリケーションを実行する場合、セキュリティー・マネージャーの使用を参照してください。

JGSS プロバイダー:

IBM JGSS には、ネイティブ IBM i JGSS プロバイダーおよび純粋の Java JGSS プロバイダーが含まれています。どのプロバイダーを選択して使用するかは、アプリケーションの必要に応じて異なります。

純粋の Java JGSS プロバイダーには、以下の機能があります。
- アプリケーションに対して最高レベルのポータビリティーを保証する。
- オプショナルの JAAS Kerberos ログイン・インターフェースと連動する。
- Java Kerberos 信任状管理ツールとの互換性。

ネイティブ IBM i JGSS プロバイダーには、以下の機能があります。
- ネイティブ IBM i Kerberos ライブラリーの使用。
- Qshell Kerberos 信任状管理ツールとの互換性。
- JGSS アプリケーションの高速実行。

注: 両方の JGSS プロバイダーは GSS-API 仕様に従っているため、互いに互換性があります。言い換えれば、純粋の Java JGSS プロバイダーを使用するアプリケーションは、ネイティブ IBM i JGSS プロバイダーを使用するアプリケーションとの相互運用が可能であるということです。

JGSS プロバイダーの変更

以下の方法の 1 つを使用して、JGSS プロバイダーを容易に変更することができます。
- `${java.home}/lib/security/java.security` のセキュリティー・プロバイダー・リストを編集する

  注: ${java.home} は、サーバー上で使用している Java のバージョンの位置へのパスを示しています。たとえば、J2SE バージョン 1.5 を使用している場合、${java.home} は /QIBM/ProdData/Java400/jdk15 です。
- GSSManager.addProviderAtFront() または GSSManager.addProviderAtEnd()Specify のどちらかを使用して、JGSS アプリケーションの中でプロバイダー名を指定してください。詳しくは、『GSSManager javadoc』を参照してください。

セキュリティー・マネージャーの使用:

使用可能な Java セキュリティー・マネージャーでご使用の JGSS アプリケーションを実行する場合、アプリケーションおよび JGSS に必要なアクセス権があることを確認する必要があります。

JVM アクセス権:

JGSS が実行するアクセス制御検査に加え、Java 仮想マシン (JVM) は、ファイル、Java プロパティー、パッケージ、およびソケットを含むさまざまなリソースへのアクセス時に、許可検査を実行します。

次のリストは、JGSS の JAAS 機能を使用する場合、またはセキュリティー・マネージャーで JGSS を使用する場合に必要な許可をリストします。

- javax.security.auth.AuthPermission "modifyPrincipals"
- javax.security.auth.AuthPermission "modifyPrivateCredentials"
- javax.security.auth.AuthPermission "getSubject"
- javax.security.auth.PrivateCredentialPermission "javax.security.auth.kerberos.KerberosKey javax.security.auth.kerberos.KerberosPrincipal ¥"*¥"", "read"
- javax.security.auth.PrivateCredentialPermission "javax.security.auth.kerberos.KerberosTicket javax.security.auth.kerberos.KerberosPrincipal ¥"*¥"", "read"
- java.util.PropertyPermission "com.ibm.security.jgss.debug", "read"
- java.util.PropertyPermission "DEBUG", "read"
- java.util.PropertyPermission "java.home", "read"
- java.util.PropertyPermission "java.security.krb5.conf", "read"
- java.util.PropertyPermission "java.security.krb5.kdc", "read"
- java.util.PropertyPermission "java.security.krb5.realm", "read"
- java.util.PropertyPermission "javax.security.auth.useSubjectCredsOnly", "read"
- java.util.PropertyPermission "user.dir", "read"
- java.util.PropertyPermission "user.home", "read"
- java.lang.RuntimePermission "accessClassInPackage.sun.security.action"
- java.security.SecurityPermission "putProviderProperty.IBMJGSSProvider"

関連情報

Sun Microsystems, Inc. による「Permissions in the Java 2 SDK」

JAAS 許可検査:

IBM JGSS は、JAAS が使用可能にするプログラムが信任状を使用し、サービスにアクセスするときに、ランタイム許可検査を実行します。 Java プロパティー javax.security.auth.useSubjectCredsOnly を false に設定することによって、このオプションの JAAS 機能を使用不可にすることができます。 さらに、JGSS は、アプリケーションがセキュリティー・マネージャーを使って実行される場合のみ、許可検査を実行します。

JGSS は、現行アクセス制御コンテキストで有効な Java ポリシーに対して、許可検査を実行します。 JGSS は、次の特定の許可検査を実行します。

- javax.security.auth.kerberos.DelegationPermission
- javax.security.auth.kerberos.ServicePermission

DelegationPermission 検査

DelegationPermission により、セキュリティー・ポリシーは、Kerberos のチケット転送およびプロキシーを行う機能の使用を制御できます。これらの機能を使用して、クライアントは、サービスがクライアントの代わりとして動作することを許可できます。

DelegationPermission は、次の順序で 2 つの引数を取ります。

1. 従属プリンシパル。クライアントの代わりに、またクライアントの権限の下で動作する、サービス・プリンシパルの名前。
2. クライアントが従属プリンシパルに使用を許可するサービスの名前。

例: DelegationPermission 検査の使用

次の例では、superSecureServer が従属プリンシパル、krbtgt/REALM.IBM.COM@REALM.IBM.COM が superSecureServer にクライアントの代わりに使用を許可するサービスです。この場合、サービスはクライアントのチケット許可チケットです。つまり、superSecureServer は、クライアントの代わりにどんなサービスのチケットでも入手できるということです。

```
permission javax.security.auth.kerberos.DelegationPermission
     "¥"superSecureServer/host.ibm.com@REALM.IBM.COM¥"
     ¥"krbtgt/REALM.IBM.COM@REALM.IBM.COM¥"";
```

この例では、DelegationPermission は、superSecureServer だけが使用できる鍵配布センター (KDC) から、新しいチケット許可チケットを入手するためのクライアント許可を認可します。クライアントが新しいチケット許可チケットを superSecureServer に送信した後、 superSecureServer にはクライアントの代わりに動作する機能が付与されます。

次の例では、クライアントが、ftp サービスのみに superSecureServer がアクセスすることを許可する新しいチケットを入手できるようにします。

```
permission javax.security.auth.kerberos.DelegationPermission
     "¥"superSecureServer/host.ibm.com@REALM.IBM.COM¥"
     ¥"ftp/ftp.ibm.com@REALM.IBM.COM¥"";
```

ServicePermission 検査

ServicePermission は、コンテキストの開始および受け入れのための信任状の使用制限を検査します。コンテキスト開始側には、コンテキストを開始する許可がなければなりません。同様に、コンテキストの受入側にも、コンテキストを受け入れる許可が必要です。

例: ServicePermission 検査の使用

次の例は、クライアント・サイドが、許可をクライアントに付与することによって、ftp サービスを使ってコンテキストを開始できるようにします。

```
permission javax.security.auth.kerberos.ServicePermission
     "ftp/host.ibm.com@REALM.IBM.COM", "initiate";
```

次の例は、許可をサーバーに付与することによって、サーバー・サイドが ftp サービスの秘密鍵にアクセスし、秘密鍵を使用できるようにします。

```
permission javax.security.auth.kerberos.ServicePermission
     "ftp/host.ibm.com@REALM.IBM.COM", "accept";
```

関連情報

Sun Microsystems, Inc. の資料

IBM JGSS アプリケーションの実行

IBM Java Generic Security Service (JGSS) API 1.0 は、セキュア・アプリケーションを、さまざまな基礎となるメカニズムの複雑さおよび特殊さから保護します。 JGSS は、Java Authentication and Authorization Service (JAAS) および IBM Java Cryptography Extension (JCE) が提供する機能を使用します。

JGSS 機能には、以下のものが含まれます。

- 識別認証
- メッセージの保全性と機密性
- オプションの JAAS Kerberos ログイン・インターフェースおよび許可検査

Kerberos 信任状の取得および秘密鍵の作成:

GSS-API は、信任状を取得するための方法を定義していません。このため、IBM JGSS Kerberos メカニズムでは、ユーザーが、Kerberos 信任状を取得することが必要です。このトピックでは、Kerberos 証明書を取得して秘密鍵を作成する方法、および Kerberos ログインおよび許可検査を実行するための JAAS の使用方法を学び、Java 仮想マシン (JVM) が必要とする JAAS 許可のリストを学習できます。

信任状は、以下のいずれかの方式を使用して取得できます。

- Kinit および Ktab ツール
- オプションの JAAS Kerberos ログイン・ インターフェース

Kinit および Ktab ツール:

選択された JGSS プロバイダーは、Kerberos 信任状および秘密鍵を取得するのに使用するツールを決定します。

純正の Java JGSS プロバイダーの使用

純正の Java JGSS プロバイダーを使用している場合、IBM JGSS Kinit および Ktab ツールを使用して信任状と秘密鍵を取得します。 Kinit および Ktab ツールは、コマンド行インターフェースを使用し、他のバージョンが提供するオプションに類似したオプションを提供します。

- Kinit ツールを使用して Kerberos 信任状を取得できます。このツールは、Kerberos 配布センター (KDC) と接触し、チケット許可チケット (TGT) を取得します。TGT によって、GSS-API を使用するものも含め、Kerberos 可能サービスにアクセスできます。
- サーバーは、Ktab ツールを使用して秘密鍵を取得できます。 JGSS は、サーバー上のキー・テーブル・ファイルに秘密鍵を保管します。詳細は、Ktab Java に関する資料を参照してください。

別の方法として、アプリケーションは、JAAS ログイン・インターフェースを使って TGT と秘密鍵を取得することができます。

ネイティブ IBM i JGSS プロバイダーの使用

ネイティブ IBM i JGSS プロバイダーを使用している場合は、Qshell kinit および klist ユーティリティーを使用します。

関連概念

『JAAS Kerberos ログイン・インターフェース』
IBM JGSS は、Java Authentication and Authorizaiton Service (JAAS) Kerberos ログイン・インターフェースを備えています。 Java プロパティー javax.security.auth.useSubjectCredsOnly を false に設定することによって、この機能を使用不可にすることができます。

関連資料

370 ページの『com.ibm.security.krb5.internal.tools Class Kinit』
Kerberos v5 チケットを取得するための Kinit ツール。

372 ページの『com.ibm.security.krb5.internal.tools Class Ktab』
このクラスは、ユーザーがキー・テーブル内の項目を管理するのを助けるためのコマンド行ツールとして実行できます。 使用可能な関数には、list/add/update/delete サービス・キーが含まれます。

JAAS Kerberos ログイン・インターフェース:

IBM JGSS は、Java Authentication and Authorizaiton Service (JAAS) Kerberos ログイン・インターフェースを備えています。 Java プロパティー javax.security.auth.useSubjectCredsOnly を false に設定することによって、この機能を使用不可にすることができます。

注: 純正の Java JGSS プロバイダーではログイン・インターフェースを使用できますが、ネイティブ IBM i JGSS プロバイダーでは使用できません。

JAAS について詳しくは、Java Authentication and Authorization Service を参照してください。

JAAS および JVM 許可

セキュリティー・マネージャーを使用している場合、アプリケーションおよび JGSS に必要な JVM と JAAS 許可があるかどうか確認する必要があります。詳しくは、セキュリティー・マネージャーの使用を参照してください。

JAAS 構成ファイル・オプション

ログイン・インターフェースには、使用されるログイン・モジュールとして、com.ibm.security.auth.module.Krb5LoginModule を指定する JAAS 構成ファイルが必要です。以下の表は、Krb5LoginModule がサポートするオプションをリストしています。オプションは大文字小文字を区別しないので注意してください。

オプション名	値	デフォルト	説明
principal	<string>	なし; プロンプトに従う	Kerberos プリンシパル名
credsType	initiator	acceptor	both	initiator	JGSS 信任状タイプ
forwardable	true|false	false	転送可能チケット許可チケット (TGT) を獲得できるかどうか
proxiable	true|false	false	プロキシー可能 TGT を獲得できるかどうか
useCcache	<URL>	ccache を使用しない	指定された信任状キャッシュから TGT を検索する
useKeytab	<URL>	キー・テーブルを使用しない	指定されたキー・テーブルから秘密鍵を検索する
useDefaultCcache	true|false	デフォルト・キャッシュを使用しない	デフォルトの信任状キャッシュから TGT を検索する
useDefaultKeytab	true|false	デフォルト・キー・テーブルを使用しない	指定されたキー・テーブルから秘密鍵を検索する

Krb5LoginModule の簡単な例については、JAAS ログイン構成ファイルのサンプルを参照してください。

オプションの非互換性

プリンシパル名を除く Krb5LoginModule オプションの中には、相互に非互換であるもの、つまり一緒に指定できないものがあります。次の表は、ログイン・モジュール・オプションで、互換性のあるものと非互換であるものを示します。

表中の標識は、2 つの関連したオプションの間の関係を表します。

- X = 非互換
- N/A = 適用不可の組み合わせ
- ブランク = 互換

Krb5LoginModule オプション	credsType initiator	credsType acceptor	credsType both	forward	proxy	use Ccache	use Keytab	useDefault Ccache	useDefault Keytab
credsType=initiator		N/A	N/A				X		X
credsType=acceptor	N/A		N/A	X	X	X		X	
credsType=both	N/A	N/A							
forwardable		X				X	X	X	X
proxiable		X				X	X	X	X
useCcache		X		X	X		X	X	X
useKeytab	X			X	X	X		X	X
useDefaultCcache		X		X	X	X	X		X
useDefaultKeytab	X			X	X	X	X	X	

プリンシパル名オプション

プリンシパル名は、他のどのオプションとも組み合わせて指定できます。プリンシパル名を指定しない場合、Krb5LoginModule は、ユーザーにプリンシパル名を指定するようにというプロンプトを出します。Krb5LoginModule がユーザーにプロンプトを出すかどうかは、指定する他のオプションに依存します。

サービス・プリンシパル名のフォーマット

サービス・プリンシパル名を指定するには、次のフォーマットの 1 つを使用してください。

- <service_name> (たとえば、superSecureServer)
- <service_name>@<host> (たとえば、superSecureServer@myhost)

後者のフォーマットでは、<host> はサービスが常駐するマシンのホスト名です。完全に修飾されたホスト名を使用できます (必ずしも使用する必要はありません)。

注: JAAS は、特定の文字を区切り文字として認識します。 JAAS ストリング (プリンシパル名など) で次の文字のいずれかを使用する場合、文字を引用符で囲んでください。

_ (下線)
: (コロン)
/ (スラッシュ)
¥ (円記号)

プリンシパル名とパスワードのプロンプトを出す

JAAS 構成ファイルで指定するオプションは、Krb5LoginModule ログインが非対話式か、対話式かを決定します。

- 非対話式ログインは、どんな情報についてもプロンプトを出しません。

- 対話式ログインは、プリンシパル名、パスワード、またはその両方のプロンプトを出します。

非対話式ログイン

ログインは、信任状タイプをイニシエーター (credsType=initiator) として指定し、次のアクションのいずれかを実行する場合に非対話式に進行します。

- useCcache オプションを指定する
- useDefaultCcache オプションを true に設定する

また、信任状タイプをアクセプター、または両方 (credsType=acceptor または credsType=both) に指定し、次のアクションのいずれかを実行する場合に非対話式に進行します。

- useKeytab オプションを指定する
- useDefaultKeytab オプションを true に設定する

対話式ログイン

他の構成は、Kerberos KDC から TGT を取得できるように、プリンシパル名およびパスワードのプロンプトを出すログイン・モジュールになります。ログイン・モジュールは、プリンシパル・オプションを指定すると、パスワードのみについてプロンプトを出します。

対話式ログインでは、アプリケーションが、ログイン・コンテキストの作成時にコールバック・ハンドラーとして com.ibm.security.auth.callback.Krb5CallbackHandler を指定することが必要です。コールバック・ハンドラーには、入力を促すプロンプトを出す役割があります。

信任状タイプ・オプション

信任状タイプをイニシエーターとアクセプターの両方 (credsType=both) にする必要がある場合、Krb5LoginModule は TGT と秘密鍵の両方を取得します。ログイン・モジュールは、TGT を使ってコンテキストと秘密鍵を開始し、コンテキストを受け入れます。JAAS 構成ファイルには、ログイン・モジュールが 2 つのタイプの信任状を獲得するために十分な情報が入っているはずです。

信任状タイプ、アクセプターおよび両方については、ログイン・モジュールはサービス・プリンシパルを想定します。

構成ファイルとポリシー・ファイル:

JGSS および JAAS は、いくつかの構成ファイルおよびポリシー・ファイルに依存しています。これらのファイルを、環境およびアプリケーションに順応するように編集する必要があります。 JGSS で JAAS を使用していない場合、JAAS 構成ファイルおよびポリシー・ファイルを無視しても構いません。

注: 次の指示では、${java.home} は、サーバー上で使用している Java のバージョンの位置へのパスを示しています。たとえば、J2SE バージョン 1.5 を使用している場合、${java.home} は /QIBM/ProdData/Java400/jdk15 です。プロパティー設定の ${java.home} を、Java ホーム・ディレクトリーへの実際のパスに置換することを忘れないでください。

Kerberos 構成ファイル

IBM JGSS には、Kerberos 構成ファイルが必要です。 Kerberos 構成ファイルのデフォルト名および位置は、使用中のオペレーティング・システムに依存しています。 JGSS は、デフォルト構成ファイルを次の順序で検索します。

1. Java プロパティー java.security.krb5.conf が参照するファイル

2. ${java.home}/lib/security/krb5.conf
3. Microsoft® Windows プラットフォームの c:¥winnt¥krb5.ini
4. Solaris プラットフォームの /etc/krb5/krb5.conf
5. UNIX プラットフォームの /etc/krb5.conf

JAAS 構成ファイル

JAAS ログイン機能の使用には、JAAS 構成ファイルが必要です。以下のプロパティーの 1 つを設定することにより、JAAS 構成ファイルを指定できます。

- Java システム・プロパティー java.security.auth.login.config
- ${java.home}/lib/security/java.security ファイルのセキュリティー・プロパティー login.config.url.<integer>

詳しくは、Sun Java Authentication and Authorization Service (JAAS) Web サイトを参照してください。

JAAS ポリシー・ファイル

デフォルト・ポリシー・インプリメンテーションを使用すると、ポリシー・ファイルにアクセス権を記録することによって、JGSS は JAAS アクセス権をエンティティーに付与します。以下のプロパティーの 1 つを設定することにより、JAAS ポリシー・ファイルを指定できます。

- Java システム・プロパティー java.security.policy
- ${java.home}/lib/security/java.security ファイルのセキュリティー・プロパティー property policy.url.<integer>

J2SDK バージョン 1.4 および後続のリリースを使用している場合、JAAS への個別のポリシー・ファイルの指定はオプションです。 J2SDK バージョン 1.4 およびそれ以降のデフォルト・ポリシー・プロバイダーは、JAAS が必要とするポリシー・ファイル・エントリーをサポートします。

詳しくは、Sun Java Authentication and Authorization Service (JAAS) Web サイトを参照してください。

Java マスター・セキュリティー・プロパティー・ファイル

Java 仮想マシン (JVM) は、多数の重要なセキュリティー・プロパティーを使用します。これらのセキュリティー・プロパティーは、Java マスター・セキュリティー・プロパティー・ファイルを編集することによって設定されます。 このファイルの名前は java.security で、通常、サーバー上の ${java.home}/lib/security ディレクトリーにあります。

次のリストは、JGSS を使用するためのいくつかの関連セキュリティー・プロパティーを説明します。この説明は、java.security ファイルを編集するためのガイドとして使用してください。

注: 適用可能な場合は、説明には JGSS サンプルを実行するのに必要な適切な値が含まれます。

security.provider.<integer>: 使用する予定の JGSS プロバイダー。これは静的に暗号プロバイダー・クラスの登録も行います。 IBM JGSS は、IBM JCE プロバイダーが提供する暗号および他のセキュリティー・サービスを使用します。次の例のとおり、sun.security.provider.Sun および com.ibm.crypto.provider.IBMJCE パッケージを指定してください。

```
security.provider.1=sun.security.provider.Sun
security.provider.2=com.ibm.crypto.provider.IBMJCE
```

policy.provider: システム・ポリシー・ハンドラー・クラス。以下に例を示します。

```
policy.provider=sun.security.provider.PolicyFile
```

policy.url.<integer>: ポリシー・ファイルの URL。サンプル・ポリシー・ファイルを使用するには、次のようなエントリーを含めます。

```
policy.url.1=file:/home/user/jgss/config/java.policy
```

login.configuration.provider: JAAS ログイン構成ハンドラー・クラス。次に例を示します。

```
login.configuration.provider=com.ibm.security.auth.login.ConfigFile
```

auth.policy.provider: JAAS プリンシパルに基づくアクセス制御ポリシー・ハンドラー・クラス。次に例を示します。

```
auth.policy.provider=com.ibm.security.auth.PolicyFile
```

login.config.url.<integer>: JAAS ログイン構成ファイルの URL。サンプル構成ファイルを使用するには、次のようなエントリーを含めます。

```
login.config.url.1=file:/home/user/jgss/config/jaas.conf
```

auth.policy.url.<integer>: JAAS ポリシー・ファイルの URL。プリンシパルに基づく構成と、CodeSource に基づく構成の両方を JAAS ポリシー・ファイルに含めることができます。サンプル・ポリシー・ファイルを使用するには、次のようなエントリーを含めます。

```
auth.policy.url.1=file:/home/user/jgss/config/jaas.policy
```

信任状キャッシュおよびサーバー・キー・テーブル

ユーザー・プリンシパルは、その Kerberos 信任状を信任状キャッシュに保持します。サービス・プリンシパルは、秘密鍵をキー・テーブルに保持します。実行時に、IBM JGSS は次の方法でこれらのキャッシュを見つけます。

ユーザー信任状キャッシュ

JGSS は、次の順序でユーザー信任状キャッシュを探します。

1. Java プロパティー KRB5CCNAME が参照するファイル
2. 環境変数 KRB5CCNAME が参照するファイル
3. UNIX システムの /tmp/krb5cc_<uid>
4. ${user.home}/krb5cc_${user.name}
5. ${user.home}/krb5cc (${user.name} が取得できない場合)

サーバー・キー・テーブル

JGSS は、次の順序でサーバー・キー・テーブル・ファイルを探します。

1. Java プロパティー KRB5_KTNAME の値
2. Kerberos 構成ファイルの libdefaults スタンザにある default_keytab_name エントリー
3. ${user.home}/krb5_keytab

IBM JGSS アプリケーションの開発

セキュア・アプリケーションを開発するには、JGSS を使用します。トランスポート・トークンの生成、JGSS オブジェクトの作成、コンテキストの設定などについて学習します。

JGSS アプリケーションを開発するには、高水準 GSS-API 仕様および Java バインディング仕様を熟知している必要があります。 IBM JGSS 1.0 は、主にこれらの仕様に基づき、準拠しています。詳細は、以下のリンクを参照してください。

- RFC 2743: Generic Security Service Application Programming Interface Version 2, Update 1
- RFC 2853: Generic Security Service API Version 2: Java Bindings

IBM JGSS アプリケーション・プログラミング・ステップ:

トランスポート・トークンの使用、必要な JGSS オブジェクトの作成、コンテキストの設定と削除、およびメッセージごとのサービスの使用を含む、JGSS アプリケーションの開発に必要な複数のステップがあります。

JGSS アプリケーションの操作は、Generic Security Service Application Programming Interface (GSS-API) 操作可能モデルに従います。 JGSS 操作に重要な概念について詳しくは、JGSS の概念を参照してください。

JGSS トランスポート・トークン

重要な JGSS 操作のいくつかは、Java バイト配列の形式でトークンを生成します。 1 つの JGSS 対等機能から別の対等機能にトークンを転送するのは、アプリケーションの役割です。 JGSS は、アプリケーションがトランスポート・トークンに使用するプロトコルを、何らかの方法で抑制することはありません。アプリケーションは、JGSS トークンを他のアプリケーション (つまり JGSS でない) のデータと一緒にトランスポートすることがあります。しかし、JGSS は JGSS 特有のトークンだけを受け入れ、使用します。

JGSS アプリケーションの操作順序

JGSS 操作では、以下のリストにある順序で使用する必要がある、特定のプログラミング構成を必要とします。各ステップは、イニシエーターとアクセプターの両方に適用されます。

注: 情報には、高水準 JGSS API の使用を例示し、 アプリケーションが org.ietf.jgss パッケージをインポートすることを前提とするサンプル・ コードの断片が含まれます。多くの高水準 API が過負荷ですが、断片ではこれらのメソッドのうち最もよく使用される形式のみを示します。もちろん、ご自分の必要に最適の API メソッドを使用してください。

GSSManager の作成:

GSSManager 抽象クラスは、JGSS オブジェクトの作成のため、ファクトリーとしての役割を果たします。

GSSManager 抽象クラスは以下を作成します。

- GSSName
- GSSCredential
- GSSContext

また、GSSManager には、サポートされるセキュリティー・メカニズムおよび名前タイプを決定し、JGSS プロバイダーを指定するメソッドもあります。 GSSManager getInstance 静的メソッドを使用して、デフォルトの GSSManager のインスタンスを作成してください。

```
GSSManager manager = GSSManager.getInstance();
```

GSSName の作成:

GSSName は、GSS-API プリンシパルの ID を表します。GSSName には、サポートされる基礎となるメカニズムごとに 1 つずつ、プリンシパルのたくさんの表記が含まれることがあります。名前の表記のみを含む GSSName は、メカニズム名 (MN) と呼ばれます。

GSSManager には、ストリングまたはバイトの連続する配列から GSSName を作成するためのいくつかの過負荷メソッドがあります。メソッドは、指定された名前タイプに従ってストリングまたはバイト配列を解

釈します。通常、GSSName バイト配列メソッドを使用して、エクスポートされた名前を再構成します。エクスポートされた名前は、通常タイプ GSSName.NT_EXPORT_NAME のメカニズム・タイプです。これらのメソッドのいくつかによって、名前の作成に使用するセキュリティー・メカニズムを指定することができます。

例: GSSName の使用

次の基本コードの断片は、GSSName の使用方法を示します。

注: 注: Kerberos サービス名ストリングを、<service> または <service@host> のどちらかとして指定します。<service> はサービスの名前、<host> はサービスが実行されるマシンのホスト名です。完全に修飾されたホスト名を使用できます (必ずしも使用する必要はありません)。ストリングの @<host> 部分を省略すると、GSSName はローカル・ホスト名を使用します。

```
// Create GSSName for user foo.
GSSName fooName = manager.createName("foo", GSSName.NT_USER_NAME);

// Create a Kerberos V5 mechanism name for user foo.
Oid krb5Mech = new Oid("1.2.840.113554.1.2.2");
GSSName fooName = manager.createName("foo", GSSName.NT_USER_NAME, krb5Mech);

// Create a mechanism name from a non-mechanism name by using the GSSName
// canonicalize method.
GSSName fooName = manager.createName("foo", GSSName.NT_USER_NAME);
GSSName fooKrb5Name = fooName.canonicalize(krb5Mech);
```

***GSSCredential* の作成:**

GSSCredential には、プリンシパルの代わりにコンテキストを作成するのに必要なすべての暗号情報が含まれており、複数のメカニズム用の信任状情報も入っていることがあります。

GSSManager には、3 つの信任状作成メソッドがあります。メソッドのうち 2 つは、GSSName、信任状の存続時間、信任状入手元の 1 つ以上のメカニズム、および信任状使用タイプのパラメーターを取ります。3 番目のメソッドは、使用タイプだけを取り、他のパラメーターに関してはデフォルト値を使用します。ヌル・メカニズムの指定でも、デフォルトのメカニズムを使用します。メカニズムのヌル配列を指定すると、メソッドが、信任状をメカニズムのデフォルト設定に戻します。

注: IBM JGSS は Kerberos V5 メカニズムのみをサポートするので、これがデフォルトのメカニズムです。

アプリケーションが一度に作成できるのは、3 つの信任状タイプ (*initiate*、*accept*、または *initiate and accept*) のうち 1 つだけです。

- コンテキスト・イニシエーターは、*initiate* 信任状を作成します。
- アクセプターは *accept* 信任状を作成します。
- イニシエーターとしても振る舞うアクセプターは、*initiate and accept* 信任状を作成します。

例: 信任状の取得

次の例は、イニシエーターにデフォルト信任状を取得します。

```
GSSCredentials fooCreds = manager.createCredentials(GSSCredential.INITIATE)
```

次の例は、デフォルトの有効期間を持つイニシエーター *foo* に、Kerberos V5 信任状を取得します。

```
GSSCredential fooCreds = manager.createCredential(fooName, GSSCredential.DEFAULT_LIFETIME,
                                                  krb5Mech,GSSCredential.INITIATE);
```

次の例は、すべてデフォルトのアクセプター信任状を取得します。

```
GSSCredential serverCreds = manager.createCredential(null, GSSCredential.DEFAULT_LIFETIME,
                                                    (Oid)null, GSSCredential.ACCEPT);
```

GSSContext の作成:

IBM JGSS は、GSSManager がコンテキストの作成用に提供する 2 つのメソッドをサポートします。 これらのメソッドは、コンテキスト・イニシエーターが使用するメソッドとアクセプターが使用するメソッドです。

注: GSSManager は、前にエクスポートされたコンテキストの再作成を含む、コンテキストを作成するための 3 番目のメソッドを提供します。しかし、IBM JGSS Kerberos V5 メカニズムはエクスポートされたコンテキストの使用をサポートしていないので、IBM JGSS はこのメソッドをサポートしません。

アプリケーションは、コンテキストの受け入れ用のイニシエーター・コンテキストを使用したり、コンテキスト開始のためのアクセプター・コンテキストを使用したりすることはできません。サポートされるどちらのコンテキスト作成メソッドにも、入力としての信任状が必要です。信任状の値がヌルである場合、JGSS はデフォルト信任状を使用します。

例: GSSContext の使用

次の例では、プリンシパル (foo) がホスト (securityCentral) 上で対等機能 (superSecureServer) を使用してコンテキストを開始する際に使用するコンテキストを作成します。例では、対等機能を superSecureServer@securityCentral として指定します。作成されるコンテキストは、デフォルト期間に有効です。

```
GSSName serverName = manager.createName("superSecureServer@securityCentral",
                                        GSSName.NT_HOSTBASED_SERVICE, krb5Mech);
GSSContext fooContext = manager.createContext(serverName, krb5Mech, fooCreds,
                                              GSSCredential.DEFAULT_LIFETIME);
```

次の例は、どんな対等機能によっても開始されるコンテキストを受け入れるための superSecureServer のコンテキストを作成します。

```
GSSContext serverAcceptorContext = manager.createContext(serverCreds);
```

アプリケーションが、両方のタイプのコンテキストを作成し、同時に使用できることに注目してください。

オプションの JGSS セキュリティー・サービスの要求:

アプリケーションは、オプションのセキュリティー・サービスをどれでも要求できます。 IBM JGSS はいくつかのサービスをサポートしています。

サポートされるオプションのサービスは以下のとおりです。

- 代行
- 相互認証
- 再生検出
- 順不同検出
- 使用可能なメッセージごとの機密性
- 使用可能なメッセージごとの保全性

オプション・サービスを要求するには、アプリケーションは、コンテキスト上の適切な要求メソッドを使って明示的に要求する必要があります。これらのオプション・サービスを要求できるのはイニシエーターだけです。イニシエーターは、コンテキストの設定が始まる前に要求をする必要があります。

オプション・サービスについて詳しくは、Internet Engineering Task Force (IETF) RFC 2743 Generic Security Services Application Programming Interface Version 2, Update 1 の『Optional Service Support』を参照してください。

例: オプション・サービスの要求

次の例では、コンテキスト (fooContext) が相互認証および代行サービスを使用可能にする要求を行います。

```
fooContext.requestMutualAuth(true);
fooContext.requestCredDeleg(true);
```

JGSS コンテキストの設定:

2 つの通信する対等機能は、メッセージごとのサービスを使用する際に用いるセキュリティー・コンテキストを設定することが必要です。

イニシエーターは、そのコンテキスト上で initSecContext() を呼び出し、それによってトークンがイニシエーター・アプリケーションに戻されます。イニシエーター・アプリケーションは、コンテキスト・トークンをアクセプター・アプリケーションに移送します。アクセプターはそのコンテキスト上で acceptSecContext() を呼び出し、イニシエーターから受け取ったコンテキスト・トークンを指定します。基礎となるメカニズムおよびイニシエーターが選択したオプション・サービスによっては、acceptSecContext() は、アクセプター・アプリケーションがイニシエーター・アプリケーションに転送する必要のあるトークンを作成する場合があります。その後イニシエーター・アプリケーションは、受け取ったトークンを使用して、initSecContext() をもう一度呼び出します。

アプリケーションは、GSSContext.initSecContext() および GSSContext.acceptSecContext() に複数の呼び出しを行うことができます。また、コンテキスト設定中に、複数のトークンを対等機能と交換することもできます。したがって、コンテキスト設定の典型的なメソッドでは、アプリケーションがコンテキストを設定するまでは、ループを使って GSSContext.initSecContext() または GSSContext.acceptSecContext() を呼び出します。

例: コンテキストの設定

次の例では、コンテキスト設定のイニシエーター (foo) 側を例示します。

```
byte array[] inToken = null; // The input token is null for the first call
int inTokenLen = 0;

do {
    byte[] outToken = fooContext.initSecContext(inToken, 0, inTokenLen);

    if (outToken != null) {
        send(outToken); // transport token to acceptor
    }

    if( !fooContext.isEstablished()) {
        inToken = receive(); // receive token from acceptor
          inTokenLen = inToken.length;
    }
} while (!fooContext.isEstablished());
```

次の例では、コンテキスト設定のアクセプター側を例示します。

```
// The acceptor code for establishing context may be the following:
do {
    byte[] inToken = receive(); // receive token from initiator
    byte[] outToken =
        serverAcceptorContext.acceptSecContext(inToken, 0, inToken.length);

    if (outToken != null) {
        send(outToken); // transport token to initiator
    }
} while (!serverAcceptorContext.isEstablished());
```

JGSS のメッセージごとのサービスの使用:

セキュリティー・コンテキストの設定後、2 つの通信する対等機能が、設定されたコンテキストを介してセキュア・メッセージを交換します。

どちらの対等機能も、コンテキスト設定時にイニシエーターまたはアクセプターのどちらの役割を果たしたかに関係なく、セキュア・メッセージを発信できます。メッセージをセキュアにするため、IBM JGSS はメッセージを介して、暗号メッセージ保全コード (MIC) を計算します。オプションで、IBM JGSS は、プライバシーを確実にするために、Kerberos V5 メカニズムがメッセージを暗号化するようにできます。

メッセージの送信

IBM JGSS は、メッセージをセキュアにするための 2 つのメソッドのセット wrap() および getMIC() を提供します。

wrap() の使用

wrap メソッドは、次のアクションを実行します。

- MIC の計算
- メッセージの暗号化 (オプション)
- トークンを戻す

呼び出し側アプリケーションは、MessageProp クラスを GSSContext と組み合わせて使用し、暗号化をメッセージに適用するかどうかを指定します。

戻されたトークンには、MIC とメッセージのテキストの両方が含まれます。メッセージのテキストは、暗号文 (暗号化されたメッセージ用) か、オリジナルのプレーン・テキスト (暗号化されていないメッセージ用) のどちらかです。

getMIC() の使用

getMIC メソッドは、次のアクションを実行しますが、メッセージの暗号化はできません。

- MIC の計算
- トークンを戻す

戻されるトークンには、計算された MIC だけが入っており、オリジナル・メッセージは含まれません。したがって、MIC トークンを対等機能に移送することに加えて、MIC を検査できるように、対等機能に何らかの方法でオリジナル・メッセージが分かるようにする必要があります。

例: メッセージごとのサービスを使ってメッセージを送信する

次の例は、1 つの対等機能 (foo) が、別の対等機能 (superSecureServer) に送達するため、メッセージをラップする方法を示します。

```
byte[] message = "Ready to roll!".getBytes();
MessageProp mprop = new MessageProp(true); // foo wants the message encrypted
byte[] wrappedMessage =
    fooContext.wrap(message, 0, message.length, mprop);
send(wrappedMessage); // transfer the wrapped message to superSecureServer

// This is how superSecureServer may obtain a MIC for delivery to foo:
byte[] message = "You bet!".getBytes();
MessageProp mprop = null;  // superSecureServer is content with
                           // the default quality of protection

byte[] mic =
    serverAcceptorContext.getMIC(message, 0, message.length, mprop);
send(mic);
// send the MIC to foo. foo also needs the original message to verify the MIC
```

メッセージの受信

ラップされたメッセージの受信者は、unwrap() を使ってメッセージをデコードします。 unwrap メソッドは、次のアクションを実行します。

- メッセージに埋め込まれた暗号 MIC を検査する
- 送信側が MIC の計算に使用したオリジナル・メッセージを戻す

送信側がメッセージを暗号化した場合、unwrap() は MIC の検査前にメッセージの暗号化を解除し、その後オリジナル・プレーン・テキストを戻します。 MIC トークンの受信者は、verifyMIC() を使って指定されたメッセージを介して MIC を検査します。

対等アプリケーションは、独自のプロトコルを使って、JGSS コンテキストとメッセージ・トークンを相互に配信します。また、対等アプリケーションは、トークンが MIC または循環メッセージのどちらかを判別するため、プロトコルを定義する必要もあります。たとえば、そのようなプロトコルの一部は、Simple Authentication and Security Layer (SASL) アプリケーションが使用するプロトコルと同じほど単純 (かつ厳格) である場合があります。 SASL プロトコルは、常にコンテキスト・アクセプターが、コンテキスト設定に続くメッセージごとの (ラップされる) トークンを送信するための最初の対等機能であることを指定します。

詳細は、Simple Authentication and Security Layer (SASL) を参照してください。

例: メッセージごとのサービスを使ってメッセージを受信する

次の例は、対等機能 (superSecureServer) が、別の対等機能 (foo) から受け取ったラップ・トークンをアンラップする方法を示します。

```
MessageProp mprop = new MessageProp(false);

byte[] plaintextFromFoo =
    serverAcceptorContext.unwrap(wrappedTokenFromFoo, 0,
                                 wrappedTokenFromFoo.length, mprop);

// superSecureServer can now examine mprop to determine the message properties
// (such as whether the message was encrypted) applied by foo.

// foo verifies the MIC received from superSecureServer:

MessageProp mprop = new MessageProp(false);
fooContext.verifyMIC(micFromFoo, 0, micFromFoo.length, messageFromFoo, 0,
                     messageFromFoo.length, mprop);

// foo can now examine mprop to determine the message properties applied by
```

```
// superSecureServer. In particular, it can assert that the message was not

// encrypted since getMIC should never encrypt a message.
```

JGSS コンテキストの削除:

対等機能は、コンテキストが必要なくなると、それを削除します。 JGSS 操作では、各対等機能が、一方的にコンテキストを削除する時期を決定し、もう一方の対等機能に通知する必要はありません。

JGSS は、削除コンテキスト・トークンを定義しません。コンテキストを削除するには、対等機能は GSSContext オブジェクトの廃棄メソッドを呼び出し、コンテキストが使用するリソースがあればそれを解放します。廃棄された GSSContext オブジェクトは、アプリケーションによってヌルに設定されない限り、廃棄後もアクセス可能です。しかし、廃棄済み (ただしアクセス可能) コンテキストの使用を試みると、例外がスローされます。

JGSS アプリケーションで JAAS を使用する:

IBM JGSS には、アプリケーションが JAAS を使って信任状を取得するためのオプションの JAAS ログイン機能が含まれます。 JAAS ログイン機能がプリンシパル信任状および秘密鍵を JAAS ログイン・コンテキストの対象オブジェクトに保管した後、JGSS はその対象から信任状を検索できます。

JGSS のデフォルトの振る舞いは、その対象から信任状と秘密鍵を検索することです。 Java プロパティー javax.security.auth.useSubjectCredsOnly を false に設定することによって、この機能を使用不可にすることができます。

注: 純正の Java JGSS プロバイダーではログイン・インターフェースを使用できますが、ネイティブ IBM i JGSS プロバイダーでは使用できません。

JAAS フィーチャーについて詳しくは、 378 ページの『Kerberos 信任状の取得および秘密鍵の作成』を参照してください。

JAAS ログイン機能を使用するには、アプリケーションは次のように JAAS プログラミング・ モデルに従う必要があります。

- JAAS ログイン・コンテキストの作成
- JAAS Subject.doAs 構成の範囲内での操作

次のコードの断片は、JAAS. Subject.doAs 構造の範囲内での操作の概念を例示します。

```
static class JGSSOperations implements PrivilegedExceptionAction {
    public JGSSOperations() {}
    public Object run () throws GSSException {
        // JGSS application code goes/runs here
    }
}

public static void main(String args[]) throws Exception {
    // Create a login context that will use the Kerberos
    // callback handler
    // com.ibm.security.auth.callback.Krb5CallbackHandler

    // There must be a JAAS configuration for "JGSSClient"
    LoginContext loginContext =
        new LoginContext("JGSSClient", new Krb5CallabackHandler());
        loginContext.login();
```

```
        // Run the entire JGSS application in JAAS privileged mode
        Subject.doAsPrivileged(loginContext.getSubject(),
                               new JGSSOperations(), null);
    }
```

JGSS デバッグ

JGSS 問題を識別しようとしている場合、JGSS デバッグ機能を使用して、役立つカテゴリー化されたメッセージを生成します。

Java プロパティー com.ibm.security.jgss.debug に適切な値を設定することによって、1 つ以上のカテゴリーをオンにすることができます。複数のカテゴリーをアクティブにするには、コンマを使ってカテゴリー名を区切ります。

カテゴリーのデバッグには、次のものが関係します。

カテゴリー	説明
help	デバッグ・カテゴリーのリスト
all	すべてのカテゴリーのデバッグをオンにする
off	デバッグを完全にオフにする
app	アプリケーションのデバッグ (デフォルト)
ctx	コンテキスト操作のデバッグ
cred	信任状 (名前を含む) の操作
marsh	トークンのマーシャル
mic	MIC 操作
prov	プロバイダー操作
qop	QOP 操作
unmarsh	トークンのアンマーシャル
unwrap	アンラップ操作
wrap	ラップ操作

JGSS デバッグ・クラス

JGSS アプリケーションを方針に基づいてデバッグするには、IBM JGSS フレームワークでデバッグ・クラスを使用します。アプリケーションは、デバッグ・クラスを使用しデバッグ・カテゴリーをオン/オフにし、アクティブ・カテゴリーのデバッグ情報を表示することができます。

デフォルトのデバッグ・コンストラクターは、Java プロパティー com.ibm.security.jgss.debug を読み取り、アクティブにする (オンにする) カテゴリーを判別します。

例: アプリケーション・カテゴリーのデバッグ

次の例は、アプリケーション・カテゴリーでデバッグ情報を要求する方法を示します。

```
import com.ibm.security.jgss.debug;

Debug debug = new Debug(); // Gets categories from Java property

// Lots of work required to set up someBuffer. Test that the
// category is on before setting up for debugging.

if (debug.on(Debug.OPTS_CAT_APPLICATION)) {
```

```
    // Fill someBuffer with data.
  debug.out(Debug.OPTS_CAT_APPLICATION, someBuffer);
    // someBuffer may be a byte array or a String.
```

サンプル: IBM Java Generic Security Service (JGSS)

IBM Java Generic Security Service (JGSS) サンプル・ファイルには、クライアントおよびサーバー・プログラム、構成ファイル、ポリシー・ファイル、および Javadoc 参照情報が含まれます。サンプル・プログラムを使って JGSS セットアップをテストし、検証します。

サンプルの HTML バージョンを表示するか、またはサンプル・プログラムの Javadoc 情報およびソース・コードをダウンロードすることができます。サンプルをダウンロードすることによって、Javadoc 参照情報を表示し、コードを調査し、構成ファイルおよびポリシー・ファイルを編集し、サンプル・プログラムをコンパイルおよび実行することができます。

サンプル・プログラムの説明

JGSS サンプルには、次の 4 つのプログラムが含まれます。

- 非 JAAS サーバー
- 非 JAAS クライアント
- JAAS 使用可能サーバー
- JAAS 使用可能クライアント

JAAS 使用可能バージョンは、対応する非 JAAS バージョンと完全に相互運用しています。したがって、JAAS 使用可能クライアントを非 JAAS サーバーに対して、または非 JAAS クライアントを JAAS 使用可能サーバーに対して実行することができます。

注: 注: サンプルの実行時、構成ファイルおよびポリシー・ファイルの名前、JGSS デバッグ・オプション、およびセキュリティー・マネージャーを含む、1 つ以上のオプションの Java プロパティーを指定することができます。また、JAAS 機能をオンにしたりオフにしたりすることもできます。

サンプルは、1 サーバー構成でも 2 サーバー構成でも実行できます。 1 サーバー構成は、1 次サーバーと通信するクライアントから構成されます。 2 サーバー構成は、1 次サーバーと 2 次サーバーから構成され、1 次サーバーは 2 次サーバーに対するイニシエーター、またはクライアントとして動作します。

2 サーバー構成を使用すると、クライアントは最初にコンテキストを開始し、 ソース・メッセージを 1 次サーバーと交換します。次に、クライアントはその信任状を 1 次サーバーに委任します。次に、クライアントの代わりに、1 次サーバーがこれらの信任状を使用してコンテキストを開始し、ソース・メッセージを 2 次サーバーと交換します。また、1 次サーバーがそれ自体の代わりにクライアントとして動作する 2 サーバー構成を使用することもできます。この場合、1 次サーバーは独自の信任状を使用してコンテキストを開始し、2 次サーバーとメッセージを交換します。

1 次サーバーに対して同時に実行できるクライアントの数は決まっていません。クライアントを直接 2 次サーバーに対して実行することは可能ですが、2 次サーバーが代行信任状を使用したり、独自の信任状を使用してイニシエーターとして実行することはできません。

IBM JGSS サンプルの表示:

IBM Java Generic Security Service (JGSS) サンプル・ファイルには、クライアントおよびサーバー・プログラム、構成ファイル、ポリシー・ファイル、および Javadoc 参照情報が含まれます。次のリンクを使って、JGSS サンプルの HTML バージョンを表示してください。

関連概念

392 ページの『サンプル: IBM Java Generic Security Service (JGSS)』
IBM Java Generic Security Service (JGSS) サンプル・ファイルには、クライアントおよびサーバー・プログラム、構成ファイル、ポリシー・ファイル、および Javadoc 参照情報が含まれます。 サンプル・プログラムを使って JGSS セットアップをテストし、検証します。

関連タスク

397 ページの『サンプル: サンプル JGSS プログラムのダウンロードおよび実行』
このトピックでは、サンプル Javadoc のダウンロードおよび実行の手順を記載しています。

関連資料

501 ページの『サンプル: IBM JGSS 非 JAAS クライアント・プログラム』
この JGSS サンプル・クライアントを JGSS サンプル・サーバーとともに使用してください。

509 ページの『サンプル: IBM JGSS 非 JAAS サーバー・プログラム』
この例では、JGSS サンプル・クライアントとともに使用する JGSS サンプル・サーバーを示します。

521 ページの『サンプル: IBM JGSS JAAS 使用可能クライアント・プログラム』
このサンプル・プログラムは JAAS ログインを実行し、JAAS ログイン・コンテキスト内で操作を実行します。このプログラムでは、変数 javax.security.auth.useSubjectCredsOnly は設定せずにデフォルトの "true" のままにしてあり、Java GSS が、クライアントによって作成されたログイン・コンテキストに関連付けられている JAAS サブジェクトから信任状を獲得するようになっています。

523 ページの『サンプル: IBM JGSS JAAS 使用可能サーバー・プログラム』
このサンプル・プログラムは JAAS ログインを実行し、JAAS ログイン・コンテキスト内で操作を実行します。

サンプル: Kerberos 構成ファイル:

このトピックでは、JGSS サンプル・アプリケーションを実行するための Kerberos 構成ファイルを示します。

サンプル構成ファイルの使用に関して詳しくは、『IBM JGSS サンプルのダウンロードおよび実行』を参照してください。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
# ---------------------------------------------------------------------------------
# Kerberos configuration file for running the JGSS sample applications.
# Modify the entries to suit your environment.
#----------------------------------------------------------------------------------

[libdefaults]
 default_keytab_name   = /QIBM/UserData/OS400/NetworkAuthentication/keytab/krb5.keytab
 default_realm         = REALM.IBM.COM
 default_tkt_enctypes  = des-cbc-crc
 default_tgs_enctypes  = des-cbc-crc
 default_checksum      = rsa-md5
 kdc_timesync          = 0
 kdc_default_options   = 0x40000010
 clockskew             = 300
 check_delegate        = 1
 ccache_type           = 3
 kdc_timeout           = 60000


[realms]
 REALM.IBM.COM = {
```

```
        kdc = kdc.ibm.com:88
 }

[domain_realm]
 .ibm.com = REALM.IBM.COM
```

サンプル: JAAS ログイン構成ファイル:

このトピックでは、JGSS サンプルの JAAS ログイン構成を示します。

サンプル構成ファイルの使用に関して詳しくは、『IBM JGSS サンプルのダウンロードおよび実行』を参照してください。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
/**
 * ---------------------------------------------------------------------------------
 * JAAS Login Configuration for the JGSS samples.
 * ---------------------------------------------------------------------------------
 *
 * Code example disclaimer
 * IBM grants you a nonexclusive copyright license to use all programming code
 * examples from which you can generate similar function tailored to your own
 * specific needs.
 * All sample code is provided by IBM for illustrative purposes only.
 * These examples have not been thoroughly tested under all conditions.
 * IBM, therefore, cannot guarantee or imply reliability, serviceability, or
 * function of these programs.
 * All programs contained herein are provided to you "AS IS" without any
 * warranties of any kind.
 * The implied warranties of non-infringement, merchantability and fitness
 * for a particular purpose are expressly disclaimed.
 *
 *
 * Supported options:
 *     principal=<string>
 *     credsType=initiator|acceptor|both (default=initiator)
 *     forwardable=true|false (default=false)
 *     proxiable=true|false (default=false)
 *     useCcache=<URL_string>
 *     useKeytab=<URL_string>
 *     useDefaultCcache=true|false (default=false)
 *     useDefaultKeytab=true|false (default=false)
 *     noAddress=true|false (default=false)
 *
 * Default realm (which is obtained from the Kerberos config file) is
 * used if the principal specified does not include a realm component.
 */

JAASClient {
 com.ibm.security.auth.module.Krb5LoginModule required
  useDefaultCcache=true;
};

JAASServer {
 com.ibm.security.auth.module.Krb5LoginModule required
  credsType=acceptor useDefaultKeytab=true
  principal=gss_service/myhost.ibm.com@REALM.IBM.COM;
};
```

サンプル: JAAS ポリシー・ファイル:

このトピックでは、JGSS サンプル・アプリケーションを実行するための JAAS ポリシー・ファイルを示します。

サンプル・ポリシー・ファイルの使用に関して詳しくは、『IBM JGSS サンプルのダウンロードおよび実行』を参照してください。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
// ----------------------------------------------------------------------------
// JAAS policy file for running the JGSS sample applications.
// Modify these permissions to suit your environment.
// Not recommended for use for any purpose other than that stated above.
// In particular, do not use this policy file or its
// contents to protect resources in a production environment.
//
// Code example disclaimer
// IBM grants you a nonexclusive copyright license to use all programming code
// examples from which you can generate similar function tailored to your own
// specific needs.
// All sample code is provided by IBM for illustrative purposes only.
// These examples have not been thoroughly tested under all conditions.
// IBM, therefore, cannot guarantee or imply reliability, serviceability, or
// function of these programs.
// All programs contained herein are provided to you "AS IS" without any
// warranties of any kind.
// The implied warranties of non-infringement, merchantability and fitness
// for a particular purpose are expressly disclaimed.
//
// ----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// Permissions for client only
//-----------------------------------------------------------------------------

grant CodeBase "file:ibmjgsssample.jar",
    Principal javax.security.auth.kerberos.KerberosPrincipal
        "bob@REALM.IBM.COM"
{
    // foo needs to be able to initiate a context with the server
    permission javax.security.auth.kerberos.ServicePermission
        "gss_service/myhost.ibm.com@REALM.IBM.COM", "initiate";

    // So that foo can delegate his creds to the server
    permission javax.security.auth.kerberos.DelegationPermission
        "¥"gss_service/myhost.ibm.com@REALM.IBM.COM¥" ¥"krbtgt/REALM.IBM.COM@REALM.IBM.COM¥"";
};

//-----------------------------------------------------------------------------
// Permissions for the server only
//-----------------------------------------------------------------------------

grant CodeBase "file:ibmjgsssample.jar",
    Principal javax.security.auth.kerberos.KerberosPrincipal
        "gss_service/myhost.ibm.com@REALM.IBM.COM"
{
    // Permission for the server to accept network connections on its host
    permission java.net.SocketPermission "myhost.ibm.com", "accept";

    // Permission for the server to accept JGSS contexts
    permission javax.security.auth.kerberos.ServicePermission
        "gss_service/myhost.ibm.com@REALM.IBM.COM", "accept";

    // The server acts as a client when communicating with the secondary (backup) server
    // This permission allows the server to initiate a context with the secondary server
```

```
    permission javax.security.auth.kerberos.ServicePermission
        "gss_service2/myhost.ibm.com@REALM.IBM.COM", "initiate";
};

//-----------------------------------------------------------------------------
// Permissions for the secondary server
//-----------------------------------------------------------------------------

grant CodeBase "file:ibmjgsssample.jar",
    Principal javax.security.auth.kerberos.KerberosPrincipal
        "gss_service2/myhost.ibm.com@REALM.IBM.COM"
{
    // Permission for the secondary server to accept network connections on its host
    permission java.net.SocketPermission "myhost.ibm.com", "accept";

    // Permission for the server to accept JGSS contexts
    permission javax.security.auth.kerberos.ServicePermission
        "gss_service2/myhost.ibm.com@REALM.IBM.COM", "accept";
};
```

サンプル: Java ポリシー・ファイル:

このトピックでは、サーバー上で JGSS サンプル・アプリケーションを実行するための Java ポリシー・ファイルを示します。

サンプル・ポリシー・ファイルの使用に関して詳しくは、『IBM JGSS サンプルのダウンロードおよび実行』を参照してください。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
// -----------------------------------------------------------------
// Java policy file for running the JGSS sample applications on
// the server.
// Modify these permissions to suit your environment.
// Not recommended for use for any purpose other than that stated above.
// In particular, do not use this policy file or its
// contents to protect resources in a production environment.
//
// Code example disclaimer
// IBM grants you a nonexclusive copyright license to use all programming code
// examples from which you can generate similar function tailored to your own
// specific needs.
// All sample code is provided by IBM for illustrative purposes only.
// These examples have not been thoroughly tested under all conditions.
// IBM, therefore, cannot guarantee or imply reliability, serviceability, or
// function of these programs.
// All programs contained herein are provided to you "AS IS" without any
// warranties of any kind.
// The implied warranties of non-infringement, merchantability and fitness
// for a particular purpose are expressly disclaimed.
//
//---------------------------------------------------------------------

grant CodeBase "file:ibmjgsssample.jar" {

    // For Java 1.4
    permission javax.security.auth.AuthPermission "createLoginContext.JAASClient";
    permission javax.security.auth.AuthPermission "createLoginContext.JAASServer";

    permission javax.security.auth.AuthPermission "doAsPrivileged";

    // Permission to request a ticket from the KDC
    permission javax.security.auth.kerberos.ServicePermission
        "krbtgt/REALM.IBM.COM@REALM.IBM.COM", "initiate";
```

```
    // Permission to access sun.security.action classes
    permission java.lang.RuntimePermission "accessClassInPackage.sun.security.action";

    // A whole bunch of Java properties are accessed
    permission java.util.PropertyPermission "java.net.preferIPv4Stack", "read";
    permission java.util.PropertyPermission "java.version", "read";
    permission java.util.PropertyPermission "java.home", "read";
    permission java.util.PropertyPermission "user.home", "read";
    permission java.util.PropertyPermission "DEBUG", "read";
    permission java.util.PropertyPermission "com.ibm.security.jgss.debug", "read";
    permission java.util.PropertyPermission "java.security.krb5.kdc", "read";
    permission java.util.PropertyPermission "java.security.krb5.realm", "read";
    permission java.util.PropertyPermission "java.security.krb5.conf", "read";
    permission java.util.PropertyPermission "javax.security.auth.useSubjectCredsOnly",
    "read,write";

    // Permission to communicate with the Kerberos KDC host
    permission java.net.SocketPermission "kdc.ibm.com", "connect,accept,resolve";

    // I run the samples from my localhost
    permission java.net.SocketPermission "myhost.ibm.com", "accept,connect,resolve";
    permission java.net.SocketPermission "localhost", "listen,accept,connect,resolve";

    // Access to some possible Kerberos config locations
    // Modify the file paths as applicable to your environment
    permission java.io.FilePermission "${user.home}/krb5.ini", "read";
    permission java.io.FilePermission "${java.home}/lib/security/krb5.conf", "read";

    // Access to the Kerberos key table so we can get our server key.
    permission java.io.FilePermission
    "/QIBM/UserData/OS400/NetworkAuthentication/keytab/krb5.keytab", "read";

    // Access to the user's Kerberos credentials cache.
    permission java.io.FilePermission "${user.home}/krb5cc_${user.name}",
    "read";
};
```

サンプル: IBM JGSS サンプルの Javadoc 情報のダウンロードおよび表示:

IBM JGSS サンプル・プログラムの文書をダウンロードして表示するには、次のステップに従ってください。

1. Javadoc 情報を保管したい既存のディレクトリーを選択 (または新規のディレクトリーを作成) します。
2. そのディレクトリーに Javadoc 情報 (jgsssampledoc.zip) をダウンロードします。
3. jgsssampledoc.zip から、ディレクトリーにファイルを解凍します。
4. ブラウザーを使って index.htm ファイルにアクセスします。

サンプル: サンプル JGSS プログラムのダウンロードおよび実行:

このトピックでは、サンプル Javadoc のダウンロードおよび実行の手順を記載しています。

サンプルの変更または実行前に、『サンプル: IBM Java Generic Security Service (JGSS)』の記事にあるサンプル・プログラムの説明をお読みください。

サンプル・プログラムを実行するには、以下のタスクを実行してください。

1. サンプル・ファイルをサーバーにダウンロードする
2. サンプル・ファイルの実行を準備する
3. サンプル・プログラムを実行する

関連概念

392 ページの『サンプル: IBM Java Generic Security Service (JGSS)』
IBM Java Generic Security Service (JGSS) サンプル・ファイルには、クライアントおよびサーバー・プログラム、構成ファイル、ポリシー・ファイル、および Javadoc 参照情報が含まれます。 サンプル・プログラムを使って JGSS セットアップをテストし、検証します。

関連タスク

397 ページの『サンプル: サンプル JGSS プログラムのダウンロードおよび実行』
このトピックでは、サンプル Javadoc のダウンロードおよび実行の手順を記載しています。

サンプル: IBM JGSS サンプルのダウンロード:

このトピックでは、サーバーへのサンプル JGSS Javadoc のダウンロードの手順を記載しています。

サンプルの変更または実行前に、サンプル・プログラムの説明をお読みください。

サンプル・ファイルをダウンロードし、サーバーに保管するには、次のステップに従ってください。

1. サーバーで、サンプル・プログラム、構成ファイル、およびポリシー･ファイルを保管したい既存のディレクトリーを選択 (または新規のディレクトリーを作成) します。
2. サンプル・プログラムをダウンロード (ibmjgsssample.zip) します。
3. サーバー上のディレクトリーに、ibmjgsssample.zip からファイルを解凍します。

ibmjgsssample.jar の内容の解凍により、次のアクションが実行されます。

- サンプルの .class ファイルを含む ibmgjsssample.jar を選択されたディレクトリーに入れる
- 構成ファイルおよびポリシー・ファイルを含むサブディレクトリー (名前付き config) を作成する
- サンプルの .java ソース・ファイルを含むサブディレクトリー (名前付き src) を作成する

関連情報

関連タスクについて調べる場合、または例を見るには、以下のリンクを参照してください。

- 『サンプル: JGSS サンプル・プログラムの実行の準備』
- 399 ページの『サンプル: JGSS サンプル・プログラムの実行』
- 400 ページの『例: 非 JAAS サンプルの実行』

サンプル: JGSS サンプル・プログラムの実行の準備:

ソース・コードのダウンロード後、サンプル・プログラムを実行する前に準備を実行する必要があります。

サンプルの変更または実行前に、 392 ページの『サンプル: IBM Java Generic Security Service (JGSS)』をお読みください。

ソース・コードのダウンロード後、次のタスクを実行すると、サンプル・プログラムが実行できるようになります。

- 構成ファイルおよびポリシー・ファイルを、環境に合うように編集します。詳しくは、各構成ファイルおよびポリシー・ファイルにコメントを参照してください。
- java.security ファイルに、IBM i サーバーに合った正しい設定が含まれることを確認します。詳しくは、 381 ページの『構成ファイルとポリシー・ファイル』を参照してください。
- 変更された Kerberos 構成ファイル (krb5.conf) を、使用中の J2SDK のバージョンに適切なサーバー上のディレクトリーに入れてください。

– J2SDK のバージョン 1.4 の場合: /QIBM/ProdData/Java400/jdk14/lib/security
– J2SE のバージョン 1.5 の場合: /QIBM/ProdData/Java400/jdk15/lib/security

関連タスク

398 ページの『サンプル: IBM JGSS サンプルのダウンロード』
このトピックでは、サーバーへのサンプル JGSS Javadoc のダウンロードの手順を記載しています。

『サンプル: JGSS サンプル・プログラムの実行』
ソース・コードをダウンロードして変更した後、サンプルのうち 1 つを実行することができます。

関連資料

400 ページの『例: 非 JAAS サンプルの実行』
サンプルを実行するには、サンプル・ソース・コードをダウンロードして変更する必要があります。

サンプル: JGSS サンプル・プログラムの実行:

ソース・コードをダウンロードして変更した後、サンプルのうち 1 つを実行することができます。

サンプルの変更または実行前に、サンプル・プログラムの説明をお読みください。

サンプルを実行するには、まずサーバー・プログラムを開始する必要があります。サーバー・プログラムは、クライアント・プログラムの開始前に実行しており、接続を受け入れる準備ができていなければなりません。サーバーは、`listening on port <server_port>` と表示されると、接続を受け入れる準備ができていることになります。 <server_port > は、クライアントの開始時に指定する必要のあるポート番号なので、必ず覚えておくか、書き留めてください。

次のコマンドを使ってサンプル・プログラムを開始します。

```
java [-Dproperty1=value1 ... -DpropertyN=valueN] com.ibm.security.jgss.test.<program> [options]
```

ここで、

- [-DpropertyN=valueN] は、構成ファイルおよびポリシー・ファイルの名前、JGSS デバッグ・オプション、およびセキュリティー・マネージャーを含む、1 つ以上のオプションの Java プロパティーです。詳しくは、以下の例、および JGSS アプリケーションの実行を参照してください。
- <program> は、実行するサンプル・プログラムを指定する、必須パラメーターです (`Client`、`Server`、`JAASClient`、または `JAASServer` のいずれか)。
- [options] は、実行するサンプル・プログラムのオプション・パラメーターです。サポートされるオプションのリストを表示するには、次のコマンドを使用してください。

  ```
  java com.ibm.security.jgss.test.<program> -?
  ```

注: Java プロパティー javax.security.auth.useSubjectCredsOnly を false に設定することによって、JGSS 使用可能サンプルの JAAS 機能をオフにすることができます。もちろん、JAAS 使用可能サンプルのデフォルト値では JAAS はオンであり、プロパティー値は true です。非 JAAS クライアントおよびサーバー・プログラムは、明示的にプロパティー値を設定しない限り、プロパティーを false に設定します。

関連情報

関連タスクについて調べる場合、または例を見るには、以下のリンクを参照してください。

- 398 ページの『サンプル: JGSS サンプル・プログラムの実行の準備』
- 398 ページの『サンプル: IBM JGSS サンプルのダウンロード』
- 400 ページの『例: 非 JAAS サンプルの実行』

例: 非 JAAS サンプルの実行:

サンプルを実行するには、サンプル・ソース・コードをダウンロードして変更する必要があります。

詳細は、サンプル・プログラムのダウンロードと実行を参照してください。

1 次サーバーの始動

次のコマンドを使用して、ポート 4444 で listen する非 JAAS サーバーを始動します。サーバーはプリンシパル (superSecureServer) として稼働し、2 次サーバー (backupServer) を使用します。サーバーは、アプリケーションおよび信任状デバッグ情報も表示します。

```
java -classpath ibmjgsssample.jar
     -Dcom.ibm.security.jgss.debug="app, cred"
     com.ibm.security.jgss.test.Server -p 4444
     -n superSecureServer -s backupServer
```

この例を正常に実行すると、次のメッセージが表示されます。

```
listening on port 4444
```

2 次サーバーの始動

次のコマンドを使用して、ポート 3333 で listen し、プリンシパル backupServer として稼働する、非 JAAS 2 次サーバーを始動します。

```
java -classpath ibmjgsssample.jar
     com.ibm.security.jgss.test.Server -p 3333
     -n backupServer
```

クライアントの始動

次のコマンドを使用し (1 行に入力し)、JAAS 使用可能クライアント (myClient) を稼働します。クライアントは、ホストの 1 次サーバー (securityCentral) と通信します。クライアントは、デフォルトのセキュリティー・マネージャーを使用可能にした状態で稼働し、config ディレクトリーの JAAS 構成ファイルとポリシー・ファイル、および Java ポリシー・ファイルを使用します。 config ディレクトリーの詳細は、IBM JGSS サンプルのダウンロードを参照してください。

```
java -classpath ibmjgsssample.jar
     -Djava.security.manager
     -Djava.security.auth.login.config=config/jaas.conf
     -Djava.security.policy=config/java.policy
     -Djava.security.auth.policy=config/jaas.policy
     com.ibm.security.jgss.test.JAASClient -n myClient
     -s superSecureServer -h securityCentral:4444
```

IBM JGSS Javadoc 参照情報

IBM JGSS の Javadoc 参照情報には、org.ietf.jgss api パッケージ中のクラスおよびメソッド、およびいくつかの Kerberos 信任状管理ツールの Java バージョンが含まれます。

JGSS には、公的にアクセス可能なパッケージ (たとえば com.ibm.security.jgss および com.ibm.security.jgss.spi) が含まれますが、使用するのは標準化された org.ietf.jgss パッケージの API のみにするべきです。このパッケージだけを使用すると、アプリケーションが確実に GSS-API 仕様に準拠し、最適な相互運用性および移植性を確実にします。

- org.ietf.jgss
- 370 ページの『com.ibm.security.krb5.internal.tools Class Kinit』
- 372 ページの『com.ibm.security.krb5.internal.tools Class Ktab』

- 369 ページの『com.ibm.security.krb5.internal.tools Class Klist』

Java プログラムのパフォーマンスのチューニング

Java アプリケーションを作成する際は、Java アプリケーション・パフォーマンスのいくつかの面を考慮に入れる必要があります。

より良いパフォーマンスを得るためにできる、いくつかの処置を以下に挙げます。

- Just-In-Time コンパイラーを使用するか、共有クラス・キャッシュを使用すると Java コードのパフォーマンスが向上します。
- ガーベッジ・コレクションのパフォーマンスを最適化するための値を注意深く設定してください。
- ネイティブ・メソッドは、比較的実行時間の長いシステム機能や、Java では直接に使用できないシステム機能を開始する場合にのみ使用してください。
- アプリケーションのフローが正常でない場合は、Java 例外処理を使用します。

上記の追加情報、およびその他のパフォーマンスの考慮事項については、以下を参照してください。

- IBM Center for Java Technology Developer Kit 診断ガイド (英語)
- IBM SDK for Java トラブルシューティング (英語)

ジョブ・セッションによって、PEX が起動および終了します。通常、収集されるデータはシステム全体に渡っており、Java プログラムを含めたシステム上のすべてのジョブに関係します。場合によっては、Java アプリケーションの内部からパフォーマンス収集を開始したり停止したりしなければならないことがあります。この場合、収集時間が短くなり、呼び出し/戻りトレースによって通常生成される大量のデータを減少させることができます。 PEX は、Java スレッドの内部からは実行できません。収集を開始および停止するには、キューまたは共有メモリーを介して独立したジョブと通信するネイティブ・メソッドを作成する必要があります。この場合、2 番目のジョブが収集を適宜開始および停止します。

以下のリストは、Java のパフォーマンスに影響を与える可能性がある追加の領域を示しています。

関連概念

402 ページの『Java プロファイルのパフォーマンス測定ツール』
システム全体の中央演算処理装置 (CPU) プロファイルによって、各 Java メソッドと、Java プログラムによって使用されたすべてのシステム機能に要した、相対的な CPU 時間が計算されます。

関連情報

パフォーマンス

Java ガーベッジ・コレクション

ガーベッジ・コレクションは、プログラムが参照することのなくなったオブジェクトによって使用される記憶域を空にするプロセスです。ガーベッジ・コレクションを使用すると、プログラマーはオブジェクトを明示的に「空にする」かまたは「削除する」ために、エラーになりやすいコードを作成する必要がなくなります。このコードは、頻繁に「メモリー・リーク」プログラム・エラーという結果になります。ガーベッジ・コレクターは、ユーザー・プログラムが達することのできないオブジェクトまたはオブジェクトのグループを自動的に検出します。これを行うのは、どのプログラム構造でもそのオブジェクトを参照していないからです。オブジェクトを収集後、そのスペースを他のことに使用するように割り振ることができます。

IBM Technology for Java ガーベッジ・コレクションについて詳しくは、以下の情報を参照してください。

- Java technology, IBM style: Garbage collection policies, Part 1
- Java technology, IBM style: Garbage collection policies, Part 2
- IBM Center for Java Technology Developer Kit 診断ガイド (英語)

関連概念

403 ページの『Java パフォーマンス・データを収集する』
このトピックでは、Java パフォーマンス・データの収集および分析に関して説明します。

Java ネイティブ・メソッド呼び出しのパフォーマンスに関する考慮事項

IBM Technology for Java は、ILE と PASE for i ネイティブ・メソッドの両方の呼び出しをサポートしています。 ILE ネイティブ・メソッドの呼び出しは、PASE for i ネイティブ・メソッドの呼び出しよりも、負荷が大きくなります。

頻繁に呼び出されるネイティブ・メソッドがある場合は、そのネイティブ・メソッドは、PASE for i で実行するように作成してください。

ネイティブ・メソッドは、比較的実行時間の長いシステム機能や、Java では直接に使用できないシステム機能を開始する場合に使用してください。

Java の例外処理のパフォーマンスの考慮事項

IBM i の例外アーキテクチャーは、柔軟な割り込みおよび再試行機能を備えており、言語の混合対話も可能にします。 Java 例外を IBM i プラットフォームにスローすると、他のプラットフォームよりも負荷が大きくなることがあります。Java 例外が通常のアプリケーション・パスでルーチンに沿って使用されるのでない限り、アプリケーション・パフォーマンス全体に影響を与えないようにしなければなりません。

Java プロファイルのパフォーマンス測定ツール

システム全体の中央演算処理装置 (CPU) プロファイルによって、各 Java メソッドと、Java プログラムによって使用されたすべてのシステム機能に要した、相対的な CPU 時間が計算されます。

パフォーマンス・モニター・カウンター桁あふれ (*PMCO) の実行サイクル・イベントをトレースする Performance Explorer (PEX) 定義を使用してください。通常は、サンプルがミリ秒単位の間隔で指定されます。有効なトレース・プロファイルを収集するには、2、3 分の CPU 時間が経過するまで Java アプリケーションを実行する必要があります。これによって、100,000 以上のサンプルが生成されるはずです。PEX 報告書の印刷 (PRTPEXRPT) コマンドによって、アプリケーション全体で経過する CPU 時間のヒストグラムが作成されます。これには、すべての Java メソッドおよびシステム・レベルの活動すべてが含まれます。

注: CPU プロファイルでは、解釈される Java プログラムの相対的な CPU 使用状況は示されません。

Java Virtual Machine Tool Interface

Java Virtual Machine Tool Interface (JVMTI) は、Java 仮想マシン (JVM) を分析するためのインターフェースです。

JVM TI は、Java Virtual Machine Profiler Interface (JVMPI) と Java Virtual Machine Debugger Interface (JVMDI) にとって代わるものです。JVMTI には、JVMDI と JVMPI の両方のすべての機能、および新機能が備えられています。 JVMTI は J2SE 5.0 の一部として追加されました。 JDK 6 では、JVMDI インターフェースと JVMPI インターフェースは提供されなくなり、JVMTI が唯一の選択肢となります。

注: IBM Technology for Java は、PASE for i からの JVMTI インターフェースのみをサポートしています。その結果、ILE JVMTI エージェントを PASE for i に移植する必要があります。

関連情報

Sun Microsystems, Inc. による「Java Virtual Machine Tool Interface (JVMTI)」

Java パフォーマンス・データを収集する

このトピックでは、Java パフォーマンス・データの収集および分析に関して説明します。

「JVM ジョブの処理 (WRKJVMJOB)」CL コマンドを使用してパフォーマンス・データを収集できます。WRKJVMJOB コマンドで取得できる情報は、WRKJVMJOB コマンドを発行することによってアクセスできるほか、「ジョブの処理 (WRKJOB)」画面からでもアクセスできます。

WRKJVMJOB を使用すると、以下の情報または機能にアクセスできます。

- JVM の開始時に使用された引数とオプション。
- ILE および PASE for i の両方の環境変数。
- JVM ジョブで未解決になっている Java ロック要求。
- ガーベッジ・コレクション情報。
- Java システム・プロパティー。
- JVM に関連づけられているスレッドのリスト。
- その JVM ジョブで部分的に完了しているジョブのログ。
- JVM ジョブのスプールされた入出力ファイルを処理する機能。
- パネル・オプションから JVM ダンプ (システム・ダンプ、ヒープ・ダンプ、および Java ダンプ) を生成する機能。これらの機能は、「JVM ダンプの生成 (GENJVMDMP)」コマンドからでも使用できます。
- パネル・オプションから詳細なガーベッジ・コレクションを使用可能/使用不可にする機能。

IBM Monitoring and Diagnostic Tools for Java - Health Center (ヘルス・センター) を使用することもできます。ヘルス・センターを使用すれば、実行中の Java アプリケーションの現行の状況を評価することができます。ヘルス・センターは、パフォーマンス、メモリーの使用量と管理、最適化およびプロファイルに関する、明確でわかりやすい情報を提供します。ヘルス・センターは、プロファイル・データを解釈し、問題領域を支援する推奨事項を提供します。ヘルス・センターに関する詳細については、ヘルス・センター (英語) Web サイトを参照してください。

関連概念

14 ページの『Java システム・プロパティー』
Java システム・プロパティーにより、Java プログラムを実行する環境が決まります。 Java システム・プロパティーは、IBM i のシステム値や環境変数と似ています。

関連情報

WRKJVMJOB CL コマンドの説明

Java コマンドおよびツール

IBM i の Java を使用する際は、Qshell インタープリターか CL コマンドのいずれかで Java ツールを使用できます。

Qshell インタープリターの Java ツールは Sun Microsystems, Inc. の Java Development Kit で使用するツールと似ているため、以前に Java プログラミングの経験がある場合は、Qshell インタープリターの Java ツールの方が使いやすいかもしれません。 Qshell 環境の使用法については、Qshell トピックを参照してください。

IBM i のプログラマーは、IBM i 環境固有の Java 用 CL コマンドを使用することをお勧めします。 CL コマンドおよび System i Navigator コマンドの使用法については、この後で説明します。

関連情報

Qshell インタープリター

Java ツールおよびユーティリティー

Qshell 環境には、プログラム開発で一般的に必要とされる Java 開発ツールが組み込まれています。

上記の Java ツールは、いくつかの例外を除いて、Sun Microsystems, Inc. の公式資料に記載された構文とオプションをサポートしています。

すべての Java ツールの実行には、Qshell インタープリターを使用する必要があります。 Qshell インタープリターを起動するには、「Qshell の開始 (STRQSH または QSH)」コマンドを使用します。 Qshell インタープリターの実行中は、「QSH コマンドの入力」画面が表示されます。 Qshell のもとで実行される Java のツールやプログラムからの出力とメッセージは、すべてこの画面に表示されます。また、Java プログラムへの入力もこの画面から読み取られます。

関連概念

242 ページの『Native Abstract Windowing Toolkit』
Native Abstract Windowing Toolkit (NAWT) は、実際のツールキットではなく、Java アプリケーションおよびサーブレットで Java 2 Platform, Standard Edition (J2SE) の Abstract Windowing Toolkit (AWT) グラフィック機能を使用できるようにするためのネイティブ IBM i サポートを表すために発達した用語です。

標準 Java ツールおよびユーティリティー

Java Development Kit (JDK) の各バージョンは、Java ツールおよびユーティリティーの 1 つのバージョンを同梱しています。多くの場合、/usr/bin ディレクトリーにある Java ツールおよびユーティリティーの Qshell バージョンが、使用中の JDK のバージョンに基づいて、適切なバージョンのツールまたはユーティリティーを呼び出します。

Java ツールおよびユーティリティーの実際の位置は、JAVA_HOME 環境変数に基づいています。これは、Java アプリケーションの実行時に使用される JDK を判別します。 Java ツールおよびユーティリティーの位置は、2 つのディレクトリー、<*JAVA_HOME*>/jre/bin または <*JAVA_HOME*>/bin のいずれかにあります。ここで、<*JAVA_HOME*> は JAVA_HOME 環境変数の値です。たとえば、JAVA_HOME 環境変数が /QOpenSys/QIBM/ProdData/JavaVM/jdk60/32bit に設定され、IBM Technology for Java 6 の 32 ビット版が使用されることを示す場合、Java ツールおよびユーティリティーのディレクトリーは、次のようになります。

```
/QOpenSys/QIBM/ProdData/JavaVM/jdk60/32bit/bin
/QOpenSys/QIBM/ProdData/JavaVM/jdk60/32bit/jre/bin
```

Java ツールおよびユーティリティーを使用する際には、以下の点に注意してください。

- JAVA_HOME 環境変数が未設定であり、/usr/bin ディレクトリーにある Java ツールおよびユーティリティーのバージョンを使用している場合は、デフォルトの JDK が使用されます。 JDK の選択方法について詳しくは、 6 ページの『複数の Java Development Kit (JDK) のサポート』 を参照してください。

- <*JAVA_HOME*>/jre/bin および <*JAVA_HOME*>/bin ディレクトリーにある Java ツールおよびユーティリティーのサブセットのみが /usr/bin ディレクトリーにあります。
- IBM i でサポートされるすべての Java ツールおよびユーティリティーがすべての JDK バージョンによってサポートされているわけではありません。 Java ツールまたはユーティリティーが、使用中の JDK バージョンに対してサポートされているかどうかを判別するには、<*JAVA_HOME*>/jre/bin および <*JAVA_HOME*>/bin ディレクトリーを検索してください。
- Java ツールおよびユーティリティーに関して詳しくは、Sun JDK ツールおよびユーティリティー (英語) Web サイトを参照してください。 Sun の Web サイトに記載されていない Java ツールおよびユーティリティーに関しては、-h または -help オプションを指定して、Java ツールまたはユーティリティーを実行します。

基本的なツールおよびユーティリティー

appletviewer (Java アプレット・ビューアー)
: Web ブラウザーの外でアプレットをテストおよび実行します。

apt (アノテーション処理ツール)
: 調査対象の一連の指定されたソース・ファイルに存在するアノテーションに基づいて、アノテーション・プロセッサーを検索し、実行します。

extcheck (Extcheck ユーティリティー)
: ターゲット JAR ファイルと現在インストールされている拡張 JAR ファイルの間のバージョン競合を検出します。

jar (Java アーカイブ・ツール)
: 複数のファイルを 1 つの Java アーカイブ (JAR) ファイルに結合します。

java (Java インタープリター)
: Java クラスを実行します。 Java インタープリターは、Java プログラミング言語で記述されたプログラムを実行します。

javac (Java コンパイラー)
: Java プログラミング言語で記述されたプログラムをバイトコード (コンパイルされた Java コード) にコンパイルします。

javadoc (Java ドキュメンテーション・ジェネレーター)
: API 文書の HTML ページを Java ソース・ファイルから生成します。

javah (C ヘッダーおよびスタブ・ファイル・ジェネレーター)
: ネイティブ・メソッドを、Java プログラミング言語で記述されたコードと関連付けることができます。

javap (クラス・ファイル逆アセンブラー)
: コンパイル済みのファイルを逆アセンブルし、バイトコードの表現をプリントできます。

javaw (Java インタープリター)
: Java コマンドと同様に Java クラスを実行しますが、コンソール・ウィンドウは使用しません。

jdb (Java Debugger)
: Java プログラムのデバッグを支援します。呼び出されると、ツールがサポートされていないことを示すメッセージが表示されます。このツールの代替策については、 409 ページの『IBM i での Java プログラムのデバッグ』を参照してください。

セキュリティー・ツールおよびユーティリティー

ikeyman (iKeyman GUI ユーティリティー)

鍵、証明書、および認証要求を管理できます。詳しくは、セキュリティー・ガイド (英語) および iKeyman ユーザーズ・ガイド (英語) を参照してください。このユーティリティーのコマンド行バージョンもあります。

ikeycmd (iKeyman コマンド行ユーティリティー)

コマンド行から鍵、証明書、および認証要求を管理できます。詳しくは、セキュリティー・ガイド (英語) および iKeyman ユーザーズ・ガイド (英語) を参照してください。このユーティリティーの GUI バージョンもあります。

jarsigner (JAR 署名および検証ツール)

JAR ファイルの署名を生成し、署名付き JAR ファイルの署名を検証します。

keytool (鍵と証明書の管理ツール)

秘密鍵と、それに対応する公開鍵を認証する、関連する X.509 証明書チェーンの鍵ストア (データベース) を管理します。

kinit Kerberos チケット許可チケットを取得し、キャッシュに入れます。 `/usr/bin` にあるこのユーティリティーの Qshell バージョンは、このユーティリティーの Java バージョンを呼び出しません。どちらを使用するかを判別する場合は、 378 ページの『Kinit および Ktab ツール』を参照してください。

klist ローカル信任状キャッシュおよびキー・テーブルのエントリーを表示します。 `/usr/bin` にあるこのユーティリティーの Qshell バージョンは、このユーティリティーの Java バージョンを呼び出しません。どちらを使用するかを判別する場合は、 378 ページの『Kinit および Ktab ツール』を参照してください。

ktab ローカル・キー・テーブルに保管されたプリンシパル名およびサービス・キーを管理します。

policytool (ポリシー・ファイルの作成および管理ツール)

ご使用の Java インストールのセキュリティー・ポリシーを定義する外部ポリシー構成ファイルを作成および変更します。

国際化対応ツールおよびユーティリティー

native2ascii (ネイティブから ASCII へのコンバーター)

ネイティブ・エンコード・ファイルを Latin 1 または Unicode (あるいはその両方) でエンコードされた文字を含む ASCII ファイルに変換します。

リモート・メソッド呼び出し (RMI) ツールおよびユーティリティー

rmic (Java リモート・メソッド呼び出し (RMI) スタブ・コンバーター)

リモート・オブジェクトのスタブ、スケルトン、および結合を生成します。 RMI over Internet Inter-ORB Protocol (RMI-IIOP) サポートが含まれています。

rmid (RMI 活動化システム・デーモン)

活動化システム・デーモンを開始し、オブジェクトを Java 仮想マシン (JVM) で登録および活動化できるようにします。

rmiregistry (Java リモート・オブジェクト・レジストリー)

現行ホストの指定されたポートでリモート・オブジェクト・レジストリーを作成および開始します。

serialver (シリアル・バージョン・コマンド)
: 展開中のクラスにコピーするために適した形式で 1 つ以上のクラスの serialVersionUID を戻します。

Java IDL および RMI-IIOP ツールおよびユーティリティー

idlj (IDL から Java へのコンパイラー)
: 指定された Interface Definition Language (IDL) ファイルから Java バインディングを生成します。

orbd Common Object Request Broker Architecture (CORBA) 環境においてサーバー上の永続オブジェクトを容易に見付けて呼び出すためのサポートをクライアントに提供します。

tnameserv (CORBA Transient Naming Service)
: CORBA Transient Naming Service を開始します。

Java デプロイメント・ツールおよびユーティリティー

pack200
: Java `gzip` 圧縮プログラムを使用して、JAR ファイルを圧縮された `pack200` ファイルに変換します。

unpack200
: `pack200` によって生成された圧縮ファイルを JAR ファイルに変換します。

Java プラグイン・ツールおよびユーティリティー

HtmlConverter (Java プラグイン HTML コンバーター)
: アプレットを含む HTML ページを、Java プラグインを使用できる形式に変換します。

Java Web Start ツールおよびユーティリティー

javaws (Java Web Start)
: Java アプリケーションのデプロイメントおよび自動保守を可能にします。詳しくは、Web Start の実行 (英語) を参照してください。

Java トラブルシューティング、プロファイル作成、モニターおよび管理のツールおよびユーティリティー

jconsole (JConsole モニターおよび管理ツール)
: GUI を使用してローカルおよびリモートの JVM をモニターします。このツールは JMX に準拠しています。

jdmpview (クロスプラットフォーム・ダンプ・フォーマッター)
: ダンプを分析します。詳しくは、診断ガイド (英語) を参照してください。

jextract (ダンプ抽出)
: システム生成ダンプを `jdmpview` で使用できる共通フォーマットに変換します。詳しくは、`jdmpview` を参照してください。

Java Web サービス・ツールおよびユーティリティー

schemagen
: Java クラスで参照される名前空間ごとにスキーマ・ファイルを作成します。

wsgen JAX-WS Web サービスで使用される JAX-WS ポータブル・アーティファクトを生成します。

wsimport
: Web サービス記述言語 (WSDL) ファイルから JAX-WS ポータブル・アーティファクトを生成します。

xjc XML スキーマ・ファイルをコンパイルします。

関連概念

6 ページの『複数の Java Development Kit (JDK) のサポート』
IBM i プラットフォームでは、複数のバージョンの Java Development Kit (JDK) と Java 2 Platform, Standard Edition がサポートされています。

関連情報

IBM i5/OS における Java 仮想マシン用の IBM テクノロジー

Sun JDK ツールとユーティリティー

IBM Java ツールおよびユーティリティー

IBM は、IBM i によってサポートされている関数または機能をサポートする追加ツールを提供します。IBM Java ツールの説明については、ここに記載するトピックを参照してください。

Java hwkeytool:

`hwkeytool` アプリケーションは、Java Cryptography Extension (JCE) および Java Cryptography Architecture (JCA) でモデル 4764 暗号化コプロセッサーの暗号化機能を使用可能にします。

ハードウェアの `hwkeytool` アプリケーションは、2 つのコマンドとデフォルトの鍵ストアを除いて、`keytool` アプリケーションと同じ構文とコマンドを使用します。ハードウェア keytool では、`-genkey` および `delete` コマンドに追加のパラメーターがあります。

`-genkey` コマンドでは、以下の追加パラメーターが使用可能です。

-KeyLabel
: ハードウェア・キーの特定のラベルを設定できます。

-hardwaretype
: 鍵ペアのタイプ (公開鍵データ・セット (PKDS) または RETAINED) を判別します。

-hardwareusage
: 生成されている鍵ペアの使用法を設定します。署名のみの鍵か、署名と鍵管理の鍵のいずれかを選択します。

`delete` コマンドでは、追加パラメーター **-hardwarekey** が使用可能です。このパラメーターを使用すると、鍵ストアとハードウェアから鍵ペアを削除します。

デフォルトの鍵ストア名は `.HWkeystore` です。この鍵ストア名は、**-keystore** パラメーターを使用して変更できます。

4764 暗号化コプロセッサー

追加の Java ツールおよびユーティリティー

IBM は、IBM i Java ライセンス製品の一部ではないものの、IBM i サーバー上で使用できる追加の Java ツールおよびユーティリティーを提供します。

- **IBM Support Assistant**

| 問題判別に役立つワークベンチを提供する補足ソフトウェア。鍵となる情報の迅速な検出、反復的なステップの自動化、およびさまざまな保守性ツールの装備に焦点を当てることによって、問題の自己分析および診断と、より迅速な解決の備えができます。

- **Java 用の IBM モニターおよび診断ツール (英語)**

IBM Runtime Environments for Java を実行するアプリケーションとデプロイメントの理解、モニター、および問題診断を支援するツールおよび資料を提供します。

Java でサポートされる CL コマンド

CL 環境。この場合、Java プログラムの最適化および管理には、CL コマンドを使用します。

- 「Java 仮想マシンのジョブの表示 (DSPJVMJOB)」コマンドは、プログラム一時修正 (PTF) の適用の管理を支援するために、アクティブ JVM ジョブに関する情報を表示します。 DSPJVMJOB の詳細については、557 ページの『プログラム一時修正を適用する』を参照してください。
- 「JVM ダンプの生成 (GENJVMDMP)」コマンドは、要求があったときに Java 仮想マシン (JVM) のダンプを生成します。
- 「JVM ジョブの印刷 (PRTJVMJOB)」コマンドでは、活動ジョブで実行されている Java 仮想マシン (JVM) を印刷できます。
- JAVA コマンドと「Java の実行 (RUNJVA)」コマンドは、IBM i Java プログラムを実行します。
- 「JVM ジョブの処理 (WRKJVMJOB)」は、IBM Technology for Java Virtual Machine で実行されているジョブに関する情報を表示します。

ライセンス内部コード・オプション・パラメーター・ストリング

プログラムおよび CL コマンド API

IBM i での Java プログラムのデバッグ

ご使用のシステムで実行している Java プログラムのデバッグとトラブルシューティングには、IBM System i Debugger、システム対話式画面、Java Debug Wire Protocol 対応デバッガー、および Java の Heap Analysis Tools といった、いくつかの方法があります。

以下にいくつかのオプションを示します (ただし、この情報はすべての可能性を示すものではありません)。

システム上で実行される Java プログラムをデバッグする最も簡単な方法の 1 つは、System i Debugger を使用する方法です。System i Debugger は、サーバーのデバッグ機能をより簡単に使えるようにするためのグラフィカル・ユーザー・インターフェース (GUI) を備えています。サーバーの対話式画面を使用して Java プログラムをデバッグすることもできますが、System i Debugger を使用した方が、もっと簡単に使える GUI で同じ機能を実行することができます。

加えて、IBM i Java 仮想マシン (JVM) は、Java Platform Debugger Architecture の一部である Java Debug Wire Protocol (JDWP) をサポートしています。JDWP 対応デバッガーでは、異なるオペレーティング・システムを稼働しているクライアントからリモート・デバッグを実行することができます。 (System i Debugger でも同様の方法でリモート・デバッグを実行できますが、こちらの場合は JDWP を使用しません。) このような JDWP 対応プログラムの 1 つは、Eclipse プロジェクト・ユニバーサル・ツール・プラットフォームの Java デバッガーです。

プログラムの実行時間が長くなるとパフォーマンスが低下する場合は、誤ってメモリー・リークがコーディングされている可能性があります。プログラムのデバッグ、およびメモリー・リークの場所の探索については、 420 ページの『メモリー・リークを検出する』を参照してください。

IBM System i Debugger

419 ページの『Java Platform Debugger Architecture』
Java Platform Debugger Architecture (JPDA) は、JVM Debug Interface/JVM Tool Interface、Java Debug Wire Protocol、および Java Debug Interface で構成されます。 JPDA のこれらの部分はすべて、デバッグ操作を実行するために JDWP を使用するデバッガーのフロントエンドを使用可能にします。 デバッガー・フロントエンドは、リモート側で実行することもできますし、IBM i アプリケーションとして実行することもできます。

Java development tool debug

Eclipse project Web site

System i Debugger を使用して Java プログラムをデバッグする

システム上で実行される Java プログラムをデバッグする最も簡単な方法は、IBM System i Debugger を使用することです。System i Debugger は、システムのデバッグ機能をより使いやすくするグラフィカル・ユーザー・インターフェースを提供します。

System i Debugger を使用した、システム上で実行される Java プログラムのデバッグとテストについて詳しくは、『IBMSystem iDebugger』を参照してください。

IBM Technology for Java のシステム・デバッグ

ここでは、IBM Technology for Java JVM のデバッグのためのいくつかのオプションについて説明します。

CL コマンド行からの対話式デバッグ

System Debugger を開始する最も簡単な方法は、JAVA CL コマンドの OPTION(*DEBUG) パラメーターを使用する方法です。以下に例を示します。

```
> JAVA CLASS(Hello) OPTION(*DEBUG)
```

IBM Technology for Java JVM のデバッグを使用可能にする

IBM Technology for Java JVM ジョブを他のジョブからデバッグするためには、JVM の開始時にデバッグを使用可能にしておく必要があります。Java のデバッグは、デバッグ・エージェントによって管理されます。 JVM の開始時にこのエージェントを開始することが、首尾良く Java コードをデバッグするためのかぎです。デバッグ・エージェントと一緒に JVM が正常に開始されれば、JVM は、「サービス・ジョブ開始 (STRSRVJOB)」コマンドや「デバッグ開始 (STRDBG)」コマンドを使用する方法でも、System i Debugger のグラフィカル・ユーザー・インターフェースを使用する方法でもデバッグできます。この後のセクションでは、デバッグ・エージェントを開始するさまざまな方法について説明します。いずれの場合も、パラメーターや環境変数は、Java デバッグ・エージェントを開始する必要を示すことを目的としています。これらの説明では、最も単純な状況から始めて、徐々により複雑な状況について説明していきます。

注:

- デバッグ・エージェントは、main メソッドに入る前に JVM を中断しません。短期のプログラムの場合や main メソッドをデバッグする場合は、JVM を停止させるために追加の Java コードが必要になる場合があります。これを行う 1 つの方法は、時間待ちループを使用する方法です。別の方法として、標準入力から読み取る方法もあります。
- デバッグ・エージェントが使用可能になっていない JVM でデバッグが試行されると、JVM ジョブと保守ジョブの両方のジョブ・ログに JVAB307 診断メッセージが送信されます。メッセージ・テキストには、デバッグが使用可能になっていない JVM ジョブが示されます。このメッセージは、正常にデバッグを行うためには JVM の再始動が必要であることを示します。JVM の開始後にデバッグを使用可能にすることはできません。

CL から Java のデバッグを使用可能にする

CL から Java のデバッグを使用可能にするには、JAVA CL コマンドに `AGTPGM(D9TI)` パラメーターを追加します。以下に例を示します。

```
> JAVA CLASS(Hello) AGTPGM(D9TI)
```

Qshell または PASE 端末から Java のデバッグを使用可能にする

Qshell (QSH) または PASE 端末 (QP2TERM) から Java のデバッグを使用可能にするには、`java` の起動に `-debug` パラメーターを追加します。以下に例を示します。

```
> java -debug Hello
```

`-debug` パラメーターを使用するのが、デバッグ・エージェントを開始する一番簡単な方法です。これは、`-agentlib:d9ti` パラメーターを追加するのと同じことです。以下のようにしても、デバッグ・エージェントは開始されます。

```
> java -agentlib:d9ti Hello
```

バッチ・ジョブ JVM で Java のデバッグを使用可能にする

「ジョブの投入 (SBMJOB)」CL コマンドでバッチ・ジョブ JVM が開始される場合は、JAVA CL コマンドに `AGTPGM(D9TI)` パラメーターを追加することができます。例えば、次のようにすると、バッチ・ジョブ JVM と一緒にデバッグ・エージェントが開始されます。

```
> SBMJOB CMD(JAVA CLASS(HELLO) AGTPGM(D9TI))
       CPYENVVAR(*YES) ALWMLTTHD(*YES)
```

バッチ・ジョブが別の方法で開始される場合は、JAVA_TOOL_OPTIONS 環境変数が、デバッグ・エージェントの開始に使用されます。JAVA_TOOL_OPTIONS 環境変数は、始動時に JVM によって自動的に照会されます。この環境変数が *-debug* または *-agentlib:d9ti* に設定されていると、その JVM ではデバッグ・エージェントが開始されます。例えば、次のいずれかの方法で環境変数を設定できます。

```
> ADDENVVAR ENVVAR(JAVA_TOOL_OPTIONS) VALUE('-debug')
> ADDENVVAR ENVVAR(JAVA_TOOL_OPTIONS) VALUE('-agentlib:d9ti')
```

バッチ・ジョブがすべての環境変数を自動的に継承しない場合は、JAVA_TOOL_OPTIONS 環境変数をシステム全体に設定する必要があります。以下に例を示します。

```
> ADDENVVAR ENVVAR(JAVA_TOOL_OPTIONS) VALUE('-debug') LEVEL(*SYS)
```

注: JAVA_TOOL_OPTIONS 環境変数をシステム全体に設定すると、システムで開始されるすべての IBM Technology for Java JVM が、デバッグが使用可能にされた状態で開始されます。**この操作は、大幅なパフォーマンスの低下を招く恐れがあります。**

Java 呼び出し API を使用して作成された JVM で Java のデバッグを使用可能にする

JVM の作成に C/C++ JNI_CreateJavaVM API を使用している場合は、以下のいずれかの方法でデバッグを使用可能にできます。

- JAVA_TOOL_OPTIONS 環境変数を *-debug* に設定します。
- JAVA_TOOL_OPTIONS 環境変数を *-agentlib:d9ti* に設定します。
- JNI_CreateJavaVM C/C++ API に渡されるオプション・パラメーター・リストに `-debug` パラメーターを追加します。
- JNI_CreateJavaVM C/C++ API に渡されるオプション・パラメーター・リストに `-agentlib:d9ti` パラメーターを追加します。

加えて、デバッグするクラスの Java ソース・コードを表示するためには、DEBUGSOURCEPATH 環境変数を Java ソース・コードの基本ディレクトリーのロケーションに設定することが必要な場合があります。

System i Debugger のグラフィカル・ユーザー・インターフェースから IBM Technology for Java JVM を開始する

System i Debugger グラフィカル・ユーザー・インターフェースから IBM Technology for Java JVM を開始するためには、JVM ジョブの開始時に JAVA_HOME 環境変数を設定する必要があります。この環境変数は、JVM 開始時の「**初期化コマンド**」画面で設定できます。この画面は、System i Debugger インターフェースの「デバッグの開始」ウィンドウにあります。

例えば、32 ビット JDK 5.0 JVM を開始するためには、「**初期化コマンド**」画面に以下を追加します。

```
ADDENVVAR ENVVAR(JAVA_HOME) VALUE('/QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit')
```

注: IBM Technology for Java JVM では、ローカル変数の監視点はサポートされていません。IBM Technology for Java JVM に対する System Debug のインプリメンテーションでは、ローカル変数の監視点機能を備えていない JVMTI が使用されます。

関連情報

System i Debugger

デバッグ操作

サーバーの対話式表示を使用することができます。プログラムを実行する前にソース・コードを表示するには `*DEBUG` オプションを使用します。これにより、停止点を設定したり、プログラムを 1 ステップずつ実行して、プログラムの実行中にエラーを分析したりできます。

*DEBUG オプションを使用して Java プログラムをデバッグする

*DEBUG オプションを使用して Java プログラムをデバッグするには、以下のステップを実行します。

1. `javac` ツールの -g オプションである DEBUG オプションを使って、Java プログラムをコンパイルする。
2. クラス・ファイル (.class) とソース・ファイル (.java) をサーバー上の同じディレクトリーに挿入する。
3. IBM i コマンド行で「Java プログラムの実行 (RUNJVA)」を使って、Java プログラムを実行する。「Java プログラムの実行 (RUNJVA)」コマンドで OPTION(*DEBUG) を指定する。例: `RUNJVA CLASS(classname) OPTION(*DEBUG)`

   クラスだけがデバッグされます。 CLASS キーワードで JAR ファイル名を入力した場合は、OPTION(*DEBUG) はサポートされません。
4. Java プログラムのソースが表示される。

5. F6 (停止点の追加/消去) を押して停止点を設定するか、または F10 (ステップ) を押してプログラム内に入る。

注:

- 停止点とステップを使用するときには、Java プログラムの論理フローをチェックしてから、必要に応じて変数を表示および変更してください。
- RUNJVA コマンド上で OPTION(*DEBUG) を使用すると、Just-In-Time (JIT) コンパイラーが使用できなくなります。
- 「サービス・ジョブ開始 (STRSRVJOB)」コマンドの使用が許可されていなければ、OPTION(*DEBUG) は無視されます。

別の画面から Java プログラムをデバッグする

サーバーの対話式画面を使用して Java プログラムをデバッグしているときは、停止点に達すると必ずプログラム・ソースが表示されます。これにより、Java プログラムの表示出力が妨げられることがあります。これを避けるには、別の画面から Java プログラムをデバッグします。 Java プログラムの出力は Java コマンドが実行されている画面に表示され、プログラム・ソースは別の画面に表示されます。

Java プログラムがデバッグを使用可能にして開始されていれば、この方法で既に稼働しているプログラムをデバッグすることも可能です。

注: JVM で System i Debugger を使用するために、AGTPGM(D9TI) オプションを JAVA/RUNJVA コマンドに追加することによって、Java デバッグを使用可能にすることができます。 OPTION(*DEBUG) を使用している場合は、AGTPGM(D9TI) は必要ありません。

別の画面から Java をデバッグするには、次のようにします。

1. デバッグの設定を開始するときには、Java プログラムを必ず保留にする。

   プログラムで次のことを行うと、Java プログラムを保留できます。
   - キーボードからの入力を待機する。
   - 一定時間待機する。
   - 変数をテストするためにループする。それには、Java プログラムのループを最終的に終了させるように値を設定しておく必要があります。
2. Java プログラムが保留されたら、別の画面に移って次のステップを実行する。
   a. コマンド行に「活動ジョブの処理 (WRKACTJOB)」コマンドを入力する。
   b. Java プログラムが実行されているバッチ即時 (BCI) ジョブを見つける。 QJVACMDSRV の「サブシステム/ジョブ」リストを調べる。使用している「ユーザー ID」の「ユーザー」リストを調べる。「タイプ」を調べて、BCI を探す。
   c. オプション 5 を入力してそのジョブを処理する。
   d. 「ジョブの処理」画面の上部に、「番号 (Number)」、「ユーザー (User)」、および「ジョブ (Job)」が表示される。 `STRSRVJOB Number/User/Job` と入力する。
   e. `STRDBG CLASS(classname)` と入力する。 classname はデバッグしたい Java クラスの名前です。この名前は、Java コマンドで指定したクラス名でも、別のクラス名でもかまいません。
   f. そのクラスのソースが「モジュール・ソースの表示」画面に表示される。
   g. その Java クラス内でストップしたい位置で、F6 (停止点の追加/消去) を押して、停止点を設定する。デバッグする他のクラス、プログラム、サービス・プログラムを追加するには、F14 を押す。
   h. F12 (再開) を押してプログラムの実行を続ける。

3. 元の Java プログラムの保留を停止する。停止点に達すると、「モジュール・ソースの表示」画面が、「サービス・ジョブ開始 (STRSRVJOB)」コマンドと「デバッグ開始 (STRDBG)」コマンドが入力された画面に表示されます。 Java プログラムが終了すると、Job being serviced ended (サービス対象のジョブが終了しました) というメッセージが表示されます。
4. 「デバッグ・モード終了 (ENDDBG)」コマンドを入力する。
5. 「サービス・ジョブ終了 (ENDSRVJOB)」コマンドを入力する。

Java プログラムをデバッグするときに、Java プログラムは実際にはバッチ即時 (BCI) ジョブの Java 仮想マシンで実行されます。ソース・コードが対話式画面に表示されますが、Java プログラムはそこでは実行されません。これは、別のジョブ (サービス・ジョブ) で実行されます。 Java 仮想マシンを呼び出すまで BCI ジョブが待機するかどうかを制御するこの変数の詳細については、QIBM_CHILD_JOB_SNDINQMSG 環境変数を参照してください。

関連情報

System i Debugger

Java プログラムの初期デバッグ画面:

Java プログラムをデバッグする際には、そのプログラムについて、以下のサンプル画面に従ってください。以下の画面では、Hellod という名前のサンプル・プログラムが示されています。

- ADDENVVAR ENVVAR(CLASSPATH) VALUE ('/MYDIR') と入力します。
- コマンド RUNJVA CLASS(HELLOD) OPTION(*DEBUG) を入力します。 HELLOD の箇所には、実際の Java プログラムの名前を入れてください。
- 「モジュール・ソースの表示」画面が表示されるのを待機します。これは、HELLOD Java プログラムのソースです。

```
+--------------------------------------------------------------------------------+
|                             モジュール・ソースの表示                           |
|                                                                                |
| クラス・ファイル名:   HELLOD                                                    |
|      1  import java.lang.*;                                                    |
|      2                                                                         |
|      3  public class Hellod extends Object                                     |
|      4  {                                                                      |
|      5  int k;                                                                 |
|      6  int l;                                                                 |
|      7  int m;                                                                 |
|      8  int n;                                                                 |
|      9  int o;                                                                 |
|     10  int p;                                                                 |
|     11  String myString;                                                       |
|     12  Hellod myHellod;                                                       |
|     13  int myArray[];                                                         |
|     14                                                                         |
|     15  public Hellod()                                                        |
|                                                                   続く...      |
| デバッグ . . .                                                                 |
|                                                                                |
| F3=プログラム終了   F6=停止点の追加/消去        F10=ステップ                     |
| F11=変数の表示      F12=再開     F17=変数監視    F24=キーの続き                   |
|                                                                                |
+--------------------------------------------------------------------------------+
```

- F14 (モジュール・リストの処理) を押します。
- 「モジュール・リストの処理」画面が表示されます。オプション 1 (プログラムの追加) を入力すると、デバッグする他のクラスおよびプログラムを追加することができます。それらのソースを表示するには、オプション 5 (モジュール・ソースの表示) を使用します。

```
+--------------------------------------------------------------------------------+
|                        モジュール・リストの処理                                |
|                                                   システム:   AS400           |
| オプションを入力して、実行キーを押してください。                               |
|   1=プログラム追加     4=プログラム除去    5=モジュール・ソースの表示          |
|   8=モジュール停止点の処理                                                     |
|                                                                                |
| OPT プログラム/モジュール  ライブラリー       タイプ                           |
|                            *LIBL             *SRVPGM                          |
|         HELLOD                               *CLASS      選択済み             |
|                                                                                |
|                                                                                |
|                                                                                |
|                                                                                |
|                                                                                |
|                                                                                |
|                                                                                |
|                                                                                |
|                                                                    終わり     |
| コマンド                                                                       |
| ===>                                                                           |
| F3=終了   F4=プロンプト   F5=最新表示   F9=コマンドの複写   F12=取り消し       |
| F22=クラス・ファイル名の表示                                                   |
|                                                                                |
+--------------------------------------------------------------------------------+
```

- デバッグするクラスを追加するときには、「PROGRAM/MODULE」入力フィールドよりも長いパッケージ修飾クラス名を入力することが必要になる可能性があります。より長い名前を入力するには、以下のステップに従ってください。
  1. オプション 1 (プログラムの追加) を入力します。
  2. 「PROGRAM/MODULE」フィールドをブランクのままにします。
  3. 「LIBRARY」フィールドを *LIBL のままにします。
  4. 「TYPE」として *CLASS を入力します。
  5. Enter を押します。
  6. パッケージ修飾クラス・ファイル名を入力するためのスペースがあるポップアップ・ウィンドウが表示されます。例: `pkgname1.pkgname2.classname`

停止点の設定:

停止点を使用して、プログラムの実行を制御できます。停止点は、特定のステートメントでプログラムの実行を停止させます。

停止点を設定するには、以下のステップを実行してください。

1. 停止点を設定したいコードの行にカーソルを置く。
2. F6 (停止点の追加/消去) を押して、停止点を設定する。
3. F12 (再開) を押してプログラムを実行する。

注: 停止点を設定したコード行が実行される直前に、プログラム・ソースが表示され、停止点に達したことが示されます。

```
+--------------------------------------------------------------------------------+
|                          モジュール・ソースの表示                              |
|                                                                                |
|現行スレッド:   00000019     停止スレッド:   00000019                           |
|クラス・ファイル名:   Hellod                                                    |
|35  public static void main(String[] args)                                     |
|36      {                                                                       |
```

```
|37    int i,j,h,B[],D[][];                                                      |
|38      Hellod A=new Hellod();                                                  |
|39      A.myHellod = A;                                                         |
|40      Hellod C[];                                                             |
|41     C = new Hellod[5];                                                       |
|42      for (int counter=0; counter<2; counter++) {                             |
|43        C[counter] = new Hellod();                                            |
|44        C[counter].myHellod = C[counter];                                     |
|45      }                                                                       |
|46      C[2] = A;                                                               |
|47      C[0].myString = null;                                                   |
|48      C[0].myHellod = null;                                                   |
|                                                                                |
|49      A.method1();                                                            |
|デバッグ . . .                                                                  |
|                                                                                |
|F3=プログラム終了   F6=停止点の追加/消去      F10=ステップ                      |
|F11=変数の表示      F12=再開   F17=変数監視   F24=キーの続き                    |
|停止点が行 41 に追加されました。                                                |
+--------------------------------------------------------------------------------+
```

停止点に達するか、ステップが完了したなら、TBREAK コマンドを使用して、停止点が現行スレッドにのみ適用されるように設定できます。

Java プログラムのステップ実行:

デバッグしながらプログラムを 1 ステップずつ実行することができます。他の関数をステップオーバーしたり、ステップイントゥすることができます。 Java プログラムおよびネイティブ・メソッドは、ステップ関数を使用することができます。

最初にプログラム・ソースが表示されると、ステップ実行を開始することができます。プログラムは、最初のステートメントを実行する前に停止します。 F10 (ステップ) を押してください。プログラムを 1 ステップずつ実行するには、F10 (ステップ) を押し続けてください。プログラムが呼び出す関数をステップイントゥするには、F22 (ステップイン) を押してください。また、停止点に達した場合に、いつでもステップ実行を開始することができます。停止点を設定することについては、『停止点の設定』のトピックを参照してください。

```
+--------------------------------------------------------------------------------+
|                          モジュール・ソースの表示                              |
|                                                                                |
|現行スレッド:   00000019     停止スレッド:   00000019                           |
|クラス・ファイル名:   Hellod                                                    |
|35  public static void main(String[] args)                                      |
|36      {                                                                       |
|37    int i,j,h,B[],D[][];                                                      |
|38      Hellod A=new Hellod();                                                  |
|39      A.myHellod = A;                                                         |
|40      Hellod C[];                                                             |
|41     C = new Hellod[5];                                                       |
|42      for (int counter=0; counter<2; counter++) {                             |
|43        C[counter] = new Hellod();                                            |
|44        C[counter].myHellod = C[counter];                                     |
|45      }                                                                       |
|46      C[2] = A;                                                               |
|47      C[0].myString = null;                                                   |
|48      C[0].myHellod = null;                                                   |
|49      A.method1();                                                            |
|デバッグ . . .                                                                  |
|                                                                                |
|F3=プログラム終了   F6=停止点の追加/消去      F10=ステップ                      |
|F11=変数の表示      F12=再開   F17=変数監視   F24=キーの続き                    |
|スレッド 00000019 の行 42 でステップが完了した。                                |
+--------------------------------------------------------------------------------+
```

プログラムの実行を続けるには、F12 (再開) を押してください。

Java プログラム中の変数を評価する:

停止点またはステップでプログラムの実行が停止したときに変数を評価するには、2 つの方法があります。

- オプション 1: デバッグ・コマンド行で、`EVAL VariableName` を入力する。
- オプション 2: 表示されたソース・コード中の変数名にカーソルを移動させて、F11 (変数の表示) を押す。

注: また、EVAL コマンドを使用して、変数の内容を変更することができます。 EVAL コマンドのバリエーションについて詳しくは、「WebSphere Development Studio: ILE C/C++ Programmer's Guide」(SC09-2712) およびオンライン・ヘルプの情報を参照してください。

Java プログラム中の変数を見るときは、次のことに注意してください。

- Java クラスのインスタンスを表す変数を評価する場合は、画面の最初の行には変数がどんな種類のオブジェクトであるかが表示される。また、オブジェクトの ID も表示されます。最初の表示行のあとに、オブジェクトの各フィールドのコンテンツが表示されます。変数がヌルである場合は、画面の最初の行には変数がヌルであることが表示されます。アスタリスクは、各フィールド (ヌル・オブジェクト) のコンテンツを表します。
- Java ストリング・オブジェクトを表す変数を評価する場合は、ストリングのコンテンツが表示される。ストリングがヌルである場合は、ヌルが表示されます。
- ストリングやオブジェクトを表す変数は変更できない。
- 配列を表す変数を評価する場合は、"ARR" に続いてその配列の ID が表示される。変数名の添え字を使用して、配列の要素を評価することができます。配列がヌルである場合は、ヌルが表示されます。
- 配列を表す変数は変更できない。配列がストリングでもオブジェクトでない場合は、配列の要素を変更することができます。
- 配列を表す変数では、配列中に要素がいくつあるかを調べるために `arrayname.length` を指定することができる。
- クラスのフィールドを表す変数のコンテンツを調べたい場合は、`classvariable.fieldname` を指定できる。
- 初期化される前の変数を評価しようとすると、次のいずれかが生じる。 「変数を表示することができません。(`Variable not available to display`)」 というメッセージが表示されるか、または変数の初期化されていない内容が表示されます (予期せぬ値になります)。

Java およびネイティブ・メソッド・プログラムをデバッグする:

Java プログラムとネイティブ・メソッド・プログラムを同時にデバッグできます。対話式画面でソースをデバッグする一方で、サービス・プログラム (*SRVPGM) 内にある、C でプログラミングされたネイティブ・メソッドをデバッグできます。 *SRVPGM はデバッグ・データ付きでコンパイルおよび作成されている必要があります。

サーバーの対話式画面を使用して、Java プログラムとネイティブ・メソッド・プログラムを同時にデバッグするには、以下の手順のようにします。

1. Java プログラム・ソースが表示されるときに F14 (モジュール・リストの処理) を押して、「モジュール・リストの処理 (WRKMODLST)」画面を表示する。
2. オプション 1 (プログラムの追加) を選択して、サービス・プログラムを追加する。

3. オプション 5 (モジュール・ソースの表示) を選択して、デバッグしたい *MODULE とソースを表示する。
4. F6 (停止点の追加/消去) を押して、サービス・プログラムに停止点を設定する。停止点の設定について詳しくは、 415 ページの『停止点の設定』を参照してください。
5. F12 (再開) を押してプログラムを実行する。

注: サービス・プログラム内の停止点に達すると、プログラムの実行が停止し、サービス・プログラムのソースが表示されます。

QIBM_CHILD_JOB_SNDINQMSG 環境変数をデバッグに使用する

QIBM_CHILD_JOB_SNDINQMSG 環境変数は、Java 仮想マシンが実行されるバッチ即時 (BCI) ジョブが、Java 仮想マシンが起動されるまで待機するかどうかを制御する変数です。

「Java プログラムの実行 (RUNJVA)」コマンドの実行時に環境変数を 1 に設定すると、メッセージがユーザーのメッセージ待ち行列に送られます。このメッセージは、BCI ジョブ内で Java 仮想マシンが開始される前に送られます。このメッセージは次のような形式です。

```
Spawned (child) process 023173/JOB/QJVACMDSRV is stopped (G C)
```

このメッセージを表示するには、SYSREQ と入力して、オプション 4 を選択します。

このメッセージに対する応答が入力されるまで BCI ジョブが待機します。 (G) という応答で Java 仮想マシンが起動します。

メッセージに応答する前に、BCI ジョブが呼び出す *SRVPGM または *PGM で停止点を設定できます。

注: この時点では Java 仮想マシンが起動していないため、Java クラスに停止点を設定することはできません。

カスタム・クラス・ローダーを通してロードされた Java クラスをデバッグする

サーバーの対話式画面を使用して、カスタム・クラス・ローダーを通してロードされたクラスをデバッグするには、以下の手順のようにします。

1. ソース・コードが含まれているディレクトリー、またはパッケージ修飾クラスの場合はそのパッケージ名の開始ディレクトリーを DEBUGSOURCEPATH 環境変数に設定する。

   たとえば、カスタム・クラス・ローダーが /MYDIR の下にあるクラスをロードする場合は、次のようにします。

   ```
   ADDENVVAR ENVVAR(DEBUGSOURCEPATH) VALUE('/MYDIR')
   ```
2. 「モジュール・ソースの表示」画面からデバッグ・ビューにそのクラスを追加する。

   そのクラスが既に Java 仮想マシン (JVM) にロードされている場合は、通常のように *CLASS を追加し、デバッグするソース・コードを表示します。

   たとえば、pkg1/test14/class のソースを表示するには、次のように入力します。

   ```
   Opt     Program/module     Library        Type
   1       pkg1.test14_       *LIBL          *CLASS
   ```

   クラスが JVM にロードされていない場合は、前述と同様の手順で *CLASS を追加してください。その結果、「**Java クラス・ファイルが使用できません。(Java class file not available)**」というメッセー

ジが表示されます。ここで、プログラム処理を再開します。指定された名前と合致するクラスの任意のメソッドが入力されると、JVM は自動的に停止します。そのクラスのソース・コードが表示され、デバッグが可能になります。

サーブレットをデバッグする

サーブレットのデバッグは、カスタム・クラス・ローダーを通してロードされるクラスをデバッグする、特殊なケースです。サーブレットは、IBM WebSphere Application Server for IBM i または IBM Integrated Web Application Server for i などのようなアプリケーション・サーバーの Java ランタイムで実行されます。サーブレットのデバッグにはいくつかのオプションがあります。

サーブレットのデバッグは、『カスタム・クラス・ローダーを通してロードされた Java クラスをデバッグする』にある指示に従って行うことができます。

また、以下のステップを実行することにより、サーバーの対話式画面を使用してサーブレットをデバッグすることもできます。

1. Qshell インタープリターで `javac -g` コマンドを使用して、サーブレットをコンパイルします。
2. ソース・コード (.java ファイル) とコンパイル済みコード (.class ファイル) をクラスパス内のディレクトリーにコピーします。
3. サーバーを開始します。
4. サーブレットを実行するジョブで「サービス・ジョブ開始 (STRSRVJOB)」コマンドを実行します。
5. STRDBG CLASS(myServlet) を実行します。myServlet はサーブレットの名前です。ソースが表示されるはずです。
6. サーブレットに停止点を設定して F12 を押します。
7. サーブレットを実行します。サーブレットが停止点に達しても、デバッグを継続できます。

システム上で実行される Java プログラムやサーブレットをデバッグする別の方法は、IBM System i Debugger を使用する方法です。 System i Debugger は、システムのデバッグ機能をより使いやすくするグラフィカル・ユーザー・インターフェースを提供します。

関連情報

System i Debugger

Java Platform Debugger Architecture

Java Platform Debugger Architecture (JPDA) は、JVM Debug Interface/JVM Tool Interface、Java Debug Wire Protocol、および Java Debug Interface で構成されます。 JPDA のこれらの部分はすべて、デバッグ操作を実行するために JDWP を使用するデバッガーのフロントエンドを使用可能にします。デバッガー・フロントエンドは、リモート側で実行することもできますし、IBM i アプリケーションとして実行することもできます。

Java Virtual Machine Tool Interface (JVMTI)

JVMTI は、Java Virtual Machine Debug Interface (JVMDI) と Java Virtual Machine Profiler Interface (JVMPI) にとって代わるものです。JVMTI には、JVMDI と JVMPI の両方のすべての機能、および新機能が備えられています。 JVMTI は J2SE 5.0 の一部として追加されました。 JDK 6 では、JVMDI インターフェースと JVMPI インターフェースは提供されなくなり、JVMTI が唯一の選択肢となります。

JVMTI の使用について詳しくは、Sun Microsystems, Inc. の Web サイトにあるJVMTI のリファレンス・ページ (英語) を参照してください。

Java Virtual Machine Debug Interface (JDK 1.4 のみ)

Java 2 SDK (J2SDK), Standard Edition では、JVMDI は Sun Microsystems, Inc. のプラットフォーム・アプリケーション・プログラミング・インターフェース (API) の一部です。JVMDI を使用すると、誰でも C コードで IBM i 用の Java デバッガーを作成することができます。デバッガーでは、JVMDI インターフェースを使用するため、Java 仮想マシンの内部構造を認識する必要はありません。 JVMDI は、Java 仮想マシンに最も近い、JPDA の最低レベルのインターフェースです。

Java Debug Wire Protocol

Java Debug Wire Protocol (JDWP) は、デバッガー・プロセスと JVMDI/JVMTI の間の定義済み通信プロトコルです。 JDWP はリモート・システムから使用することもできますし、ローカル・ソケットを介して使用することもできます。これは、JVMDI/JVMTI から取り外された 1 つの階層です。

JDWP を QShell で開始する

JDWP を開始して Java クラス SomeClass を実行するには、QShell で次のコマンドを入力してください。

```
java -interpret -agentlib:jdwp=transport=dt_socket,
address=8000,server=y,suspend=n SomeClass
```

この例では、JDWP は TCP/IP ポート 8000 上でリモート・デバッガーからの接続を listen しますが、任意のポート番号を使用できます。 dt_socket は JDWP トランスポートを処理する SRVPGM の名前で変わることはできません。

-agentlib で使用できる追加のオプションについては、Sun Microsystems, Inc. の 『 Sun VM Invocation Options』を参照してください。これらのオプションは、IBM i 上の JDK 1.4 および 1.5 の両方で選択可能です。

JDWP を CL コマンド行から開始する

JDWP を開始して Java クラス SomeClass を実行するには、次のコマンドを入力してください。

```
| JAVA CLASS(SomeClass) INTERPRET(*YES)
|      PROP((os400.xrun.option 'jdwp:transport=dt_socket,address=8000,server=y,suspend=n'))
```

Java Debug Interface

Java Debug Interface (JDI) は、ツール開発用に用意されているハイレベル Java 言語インターフェースです。 JDI では、Java クラス定義を使用することで、JVMDI/JVMTI および JDWP の複雑さが隠されています。 JDI は rt.jar ファイルに入っているので、デバッガーのフロントエンドは、 Java がインストールされているすべてのプラットフォーム上に存在することになります。

Java 用のデバッガーを作成する場合、JDI は最も単純なインターフェースであり、 コードはプラットフォームに依存していないので、JDI を使用してください。

JDPA の詳細については、Sun Microsystems, Inc. の Java Platform Debugger Architecture を参照してください。

メモリー・リークを検出する

プログラムの実行時間が長くなるとパフォーマンスが低下する場合は、誤ってメモリー・リークがコーディングされている可能性があります。

OutOfMemory 例外が一度もスローされなくても、メモリー・リークによって、OutOfMemoryError 例外が発生するか、パフォーマンスが徐々に低下します。メモリー・リークの検出について詳しくは、 IBM SDK for Java インフォメーション・センターの Java トラブルシューティング・ドキュメントを参照してください。

「JVM ダンプの生成」コマンドの使用

「JVM ダンプの生成」(GENJVMDMP) CL コマンドを使用することによって、Java、システム、およびヒープのダンプを生成できます。

GENJVMDMP コマンドは、要求があったときに Java 仮想マシン (JVM) のダンプを生成します。以下のタイプのダンプを生成できます。

***JAVA**
: JVM および JVM 内で稼働している Java アプリケーションについての診断情報を含む複数のファイルを生成します。

***SYSTEM**
: ダンプが開始されたときに実行されていたジョブのバイナリー形式のロー・メモリー・イメージを生成します。

***HEAP**
: 解放されていないすべてのヒープ・スペース割り振りのダンプを生成します。

関連情報

「JVM ダンプの生成 (GENJVMDMP)」CL コマンド

Java コード例

IBM i の Java のコード例の一覧を示します。

国際化対応

- 424 ページの『例: java.util.DateFormat クラスを使用して日付を国際化する』
- 424 ページの『例: java.util.NumberFormat クラスを使用して数値表示を国際化する』
- 425 ページの『例: java.util.ResourceBundle クラスを使用してロケール固有データを国際化する』

JDBC

- 426 ページの『例: Access プロパティー』
- 140 ページの『例: BLOB』
- 430 ページの『例: IBM Developer Kit for Java の CallableStatement インターフェース』
- 120 ページの『例: 他のステートメントのカーソルを介してテーブルから値を除去する』
- 144 ページの『例: CLOB』
- 51 ページの『例: UDBDataSource を作成して JNDI でバインドする』
- 53 ページの『例: UDBDataSource の作成およびユーザー ID とパスワードの取得』
- 52 ページの『例: UDBDataSourceBind を作成して DataSource プロパティーを設定する』
- 63 ページの『例: DatabaseMetaData インターフェースを使用して表のリストを戻す』
- 51 ページの『例: UDBDataSource を作成して JNDI でバインドする』
- 147 ページの『例: Datalink』

- 148 ページの『例: 特殊タイプ』
- 183 ページの『例: SQL ステートメントを Java アプリケーションに組み込む』
- 87 ページの『例: トランザクションを終了する』
- 40 ページの『例: 無効なユーザー ID とパスワード』
- 33 ページの『例: JDBC』
- 81 ページの『例: 単一トランザクション上で動作する複数の接続』
- 53 ページの『例: UDBDataSource をバインドする前に初期コンテキストを取得する』
- 97 ページの『例: ParameterMetaData』
- 122 ページの『例: 他のステートメントのカーソルを介してステートメントで値を変更する』
- 125 ページの『例: ResultSet インターフェース』
- 113 ページの『例: ResultSet の感度』
- 111 ページの『例: 感知および非感知の ResultSet』
- 128 ページの『例: UDBDataSource および UDBConnectionPoolDataSource で接続プーリングをセットアップする』
- 67 ページの『例: SQLException』
- 89 ページの『例: トランザクションを中断して再開する』
- 84 ページの『例: 中断状態の ResultSets』
- 128 ページの『例: 接続プーリングのパフォーマンスをテストする』
- 131 ページの『例: 2 つの DataSource のパフォーマンスをテストする』
- 141 ページの『例: BLOB の更新』
- 145 ページの『例: CLOB の更新』
- 83 ページの『例: 複数のトランザクションで単一の接続を使用する』
- 142 ページの『例: BLOB の使用』
- 146 ページの『例: CLOB の使用』
- 152 ページの『DB2CachedRowSet の作成とデータ取り込み』
- 152 ページの『DB2CachedRowSet の作成とデータ取り込み』
- 79 ページの『例: トランザクションを処理するために JTA を使用する』
- 64 ページの『例: 複数の列を持ったメタデータ ResultSet を使用する』
- 477 ページの『例: ネイティブ JDBC と IBM Toolbox for Java JDBC を同時に使用する』
- 98 ページの『例: ResultSet を取得するために PreparedStatement を使用する』
- 152 ページの『DB2CachedRowSet の作成とデータ取り込み』
- 152 ページの『DB2CachedRowSet の作成とデータ取り込み』
- 152 ページの『DB2CachedRowSet の作成とデータ取り込み』
- 152 ページの『DB2CachedRowSet の作成とデータ取り込み』
- 93 ページの『例: Statement オブジェクトの executeUpdate メソッドを使用する』

Java Authentication and Authorization Service

- 482 ページの『例: JAAS HelloWorld』
- 492 ページの『例: JAAS SampleThreadSubjectLogin』

Java Generic Security Service

- 501 ページの『サンプル: IBM JGSS 非 JAAS クライアント・プログラム』
- 509 ページの『サンプル: IBM JGSS 非 JAAS サーバー・プログラム』
- 521 ページの『サンプル: IBM JGSS JAAS 使用可能クライアント・プログラム』
- 523 ページの『サンプル: IBM JGSS JAAS 使用可能サーバー・プログラム』

Java セキュア・ソケット拡張機能

- 290 ページの『例: IBM Java Secure Sockets Extension 1.4』

Java と他のプログラム言語

- 227 ページの『例: java.lang.Runtime.exec() を使用して CL プログラムを呼び出す』
- 227 ページの『例: java.lang.Runtime.exec() を使用して CL コマンドを呼び出す』
- 226 ページの『例: java.lang.Runtime.exec() を使用して別の Java プログラムを呼び出す』
- 233 ページの『例: ILE C から Java を呼び出す』
- 233 ページの『例: RPG から Java を呼び出す』
- 232 ページの『例: プロセス間通信に入出力ストリームを使用する』
- 222 ページの『例: Java 呼び出し API』
- 216 ページの『例: Java 用の IBM PASE for i ネイティブ・メソッド』
- ソケット
- 214 ページの『例: Java 用の ILE ネイティブ・メソッド』

SQLJ

- 183 ページの『例: SQL ステートメントを Java アプリケーションに組み込む』

Secure Sockets Layer

- 270 ページの『例: クライアントのソケット・ファクトリーを使用するように Java コードを変更する』
- 268 ページの『例: サーバーのソケット・ファクトリーを使用するように Java コードを変更する』
- 274 ページの『例: Secure Sockets Layer を使用するように Java クライアントを変更する』
- 272 ページの『例: Secure Sockets Layer を使用するように Java サーバーを変更する』

IBM は、お客様に、すべてのプログラム・コードのサンプルを使用することができる非独占的な著作使用権を許諾します。お客様は、このサンプル・コードから、お客様独自の特別のニーズに合わせた類似のプログラムを作成することができます。

強行法規で除外を禁止されている場合を除き、IBM、そのプログラム開発者、および供給者は「プログラム」および「プログラム」に対する技術的サポートがある場合にはその技術的サポートについて、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任を負わないものとします。

IBM、そのプログラム開発者、または供給者は、いかなる場合においてもその予見の有無を問わず、以下に対する責任を負いません。

1. データの喪失、または損傷。
2. 直接損害、特別損害、付随的損害、間接損害、または経済上の結果的損害
3. 逸失した利益、ビジネス上の収益、あるいは節約すべかりし費用

国または地域によっては、法律の強行規定により、上記の責任の制限が適用されない場合があります。

例: java.util.DateFormat クラスを使用して日付を国際化する

この例では、ロケールを使用して日付を形式化する方法を示します。

例 1: 日付を国際化するための java.util.DateFormat クラスの使用

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
//************************
// File: DateExample.java
//************************

import java.text.*;
import java.util.*;
import java.util.Date;

public class DateExample {

   public static void main(String args[]) {

     // Get the Date
     Date now = new Date();

     // Get date formatters for default, German, and French locales
     DateFormat theDate = DateFormat.getDateInstance(DateFormat.LONG);
     DateFormat germanDate = DateFormat.getDateInstance(DateFormat.LONG, Locale.GERMANY);
     DateFormat frenchDate = DateFormat.getDateInstance(DateFormat.LONG, Locale.FRANCE);

     // Format and print the dates
     System.out.println("Date in the default locale: " + theDate.format(now));
     System.out.println("Date in the German locale : " + germanDate.format(now));
     System.out.println("Date in the French locale : " + frenchDate.format(now));
   }
}
```

27 ページの『例: 国際化 Java プログラムを作成する』
特定の地域に Java プログラムをカスタマイズする必要がある場合は、Java ロケールを使用して、国際化 Java プログラムを作成できます。

例: java.util.NumberFormat クラスを使用して数値表示を国際化する

この例では、ロケールを使用して数値を形式化する方法を示します。

例 1: 数値出力を国際化するための java.util.NumberFormat クラスの使用

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
//**************************
// File: NumberExample.java
//**************************

 import java.lang.*;
 import java.text.*;
 import java.util.*;

 public class NumberExample {

     public static void main(String args[]) throws NumberFormatException {

       // The number to format
       double number = 12345.678;
```

```
        // Get formatters for default, Spanish, and Japanese locales
        NumberFormat defaultFormat = NumberFormat.getInstance();
        NumberFormat spanishFormat = NumberFormat.getInstance(new
 Locale("es", "ES"));
        NumberFormat japaneseFormat = NumberFormat.getInstance(Locale.JAPAN);

        // Print out number in the default, Spanish, and Japanese formats
        // (Note: NumberFormat is not necessary for the default format)
        System.out.println("The number formatted for the default locale; " +
                           defaultFormat.format(number));
        System.out.println("The number formatted for the Spanish locale; " +
                           spanishFormat.format(number));
        System.out.println("The number formatted for the Japanese locale; " +
                           japaneseFormat.format(number));
      }
 }
```

27 ページの『例: 国際化 Java プログラムを作成する』
特定の地域に Java プログラムをカスタマイズする必要がある場合は、Java ロケールを使用して、国際化 Java プログラムを作成できます。

例: java.util.ResourceBundle クラスを使用してロケール固有データを国際化する

この例では、リソース・バンドルとともにロケールを使用して、プログラム・ストリングを国際化する方法を示します。

ResourceBundleExample プログラムが意図されたとおりに機能するためには、以下のプロパティー・ファイルが必要です。

RBExample.properties の内容
Hello.text=Hello

RBExample_de.properties の内容
Hello.text=Guten Tag

RBExample_fr_FR.properties の内容
Hello.text=Bonjour

例 1: ロケール固有データを国際化するための java.util.ResourceBundle クラスの使用

注: サンプル・コードをご使用の場合は、560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
//*********************************
// File: ResourceBundleExample.java
//*********************************

     import java.util.*;

public class ResourceBundleExample {
      public static void main(String args[]) throws MissingResourceException {

          String resourceName = "RBExample";
          ResourceBundle rb;

          // Default locale
          rb = ResourceBundle.getBundle(resourceName);
          System.out.println("Default : " + rb.getString("Hello" + ".text"));

          // Request a resource bundle with explicitly specified locale
          rb = ResourceBundle.getBundle(resourceName, Locale.GERMANY);
```

```
            System.out.println("German : " + rb.getString("Hello" + ".text"));

            // No property file for China in this example... use default
            rb = ResourceBundle.getBundle(resourceName, Locale.CHINA);
            System.out.println("Chinese : " + rb.getString("Hello" + ".text"));

            // Here is another way to do it...
            Locale.setDefault(Locale.FRANCE);
            rb = ResourceBundle.getBundle(resourceName);
            System.out.println("French : " + rb.getString("Hello" + ".text"));

            // No property file for China in this example... use default, which is now fr_FR.
            rb = ResourceBundle.getBundle(resourceName, Locale.CHINA);
            System.out.println("Chinese : " + rb.getString("Hello" + ".text"));
    }
}
```

27 ページの『例: 国際化 Java プログラムを作成する』
特定の地域に Java プログラムをカスタマイズする必要がある場合は、Java ロケールを使用して、国際化 Java プログラムを作成できます。

例: Access プロパティー

ここでは、Java Access プロパティーの使用方法の例を示します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
// This program assumes directory cujosql exists.
import java.sql.*;
import javax.sql.*;
import javax.naming.*;

public class AccessPropertyTest {
    public String url = "jdbc:db2:*local";
    public Connection connection = null;

    public static void main(java.lang.String[] args)
    throws Exception
    {
        AccessPropertyTest test = new AccessPropertyTest();

        test.setup();

        test.run();
        test.cleanup();
    }


/**
Set up the DataSource used in the testing.
**/
    public void setup()
    throws Exception
    {
        Class.forName("com.ibm.db2.jdbc.app.DB2Driver");

        connection = DriverManager.getConnection(url);
        Statement s = connection.createStatement();
        try {
            s.executeUpdate("DROP TABLE CUJOSQL.TEMP");
        } catch (SQLException e) { // Ignore it - it doesn't exist
        }

        try {
            String sql = "CREATE PROCEDURE CUJOSQL.TEMP "
```

```
            + " LANGUAGE SQL SPECIFIC CUJOSQL.TEMP "
            + " MYPROC: BEGIN"
            + "   RETURN 11;"
            + " END MYPROC";
        s.executeUpdate(sql);
    } catch (SQLException e) {
        // Ignore it - it exists.
    }
    s.executeUpdate("create table cujosql.temp (col1 char(10))");
    s.executeUpdate("insert into cujosql.temp values ('compare')");
    s.close();
}


public void resetConnection(String property)
throws SQLException
{
    if (connection != null)
        connection.close();

    connection = DriverManager.getConnection(url + ";access=" + property);
}


public boolean canQuery() {
    Statement s = null;
    try {
        s = connection.createStatement();
        ResultSet rs = s.executeQuery("SELECT * FROM cujosql.temp");
        if (rs == null)
            return false;

        rs.next();

        if (rs.getString(1).equals("compare   "))
            return true;

        return false;

    } catch (SQLException e) {
        // System.out.println("Exception: SQLState(" +
        //                    e.getSQLState() + ") " + e + " (" + e.getErrorCode() + ")");
        return false;
    } finally {
        if (s != null) {
            try {
                s.close();
            } catch (Exception e) {
                // Ignore it.
            }
        }
    }
}


public boolean canUpdate() {
    Statement s = null;
    try {
        s = connection.createStatement();
        int count = s.executeUpdate("INSERT INTO CUJOSQL.TEMP VALUES('x')");
        if (count != 1)
            return false;

        return true;

    } catch (SQLException e) {
        //System.out.println("Exception: SQLState(" +
```

```
        //                   e.getSQLState() + ") " + e + " (" + e.getErrorCode() + ")");
        return false;
    } finally {
        if (s != null) {
            try {
                s.close();
            } catch (Exception e) {
                // Ignore it.
            }
        }
    }
}


public boolean canCall() {
    CallableStatement s = null;
    try {
        s = connection.prepareCall("? = CALL CUJOSQL.TEMP()");
        s.registerOutParameter(1, Types.INTEGER);
        s.execute();
        if (s.getInt(1) != 11)
            return false;

        return true;

    } catch (SQLException e) {
        //System.out.println("Exception: SQLState(" +
        //                   e.getSQLState() + ") " + e + " (" + e.getErrorCode() + ")");
        return false;
    } finally {
        if (s != null) {
            try {
                s.close();
            } catch (Exception e) {
                // Ignore it.
            }
        }
    }
}


public void run()
throws SQLException
{
    System.out.println("Set the connection access property to read only");
    resetConnection("read only");

    System.out.println("Can run queries  -->" + canQuery());
    System.out.println("Can run updates  -->" + canUpdate());
    System.out.println("Can run sp calls -->" + canCall());

    System.out.println("Set the connection access property to read call");
    resetConnection("read call");

    System.out.println("Can run queries  -->" + canQuery());
    System.out.println("Can run updates  -->" + canUpdate());
    System.out.println("Can run sp calls -->" + canCall());

    System.out.println("Set the connection access property to all");
    resetConnection("all");

    System.out.println("Can run queries  -->" + canQuery());
    System.out.println("Can run updates  -->" + canUpdate());
    System.out.println("Can run sp calls -->" + canCall());

}
```

```
    public void cleanup() {
        try {
            connection.close();
        } catch (Exception e) {
            // Ignore it.
        }
    }
}
```

例: BLOB

以下は、 BLOB をデータベースに書き込んだり、データベースから検索したりする方法の例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
/////////////////////////////////////////
// PutGetBlobs is an example application
// that shows how to work with the JDBC
// API to obtain and put BLOBs to and from
// database columns.
//
// The results of running this program
// are that there are two BLOB values
// in a new table. Both are identical
// and contain 500k of random byte
// data.
/////////////////////////////////////////
import java.sql.*;
import java.util.Random;

public class PutGetBlobs {
   public static void main(String[] args)
   throws SQLException
   {
       // Register the native JDBC driver.
       try {
          Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
      } catch (Exception e) {
          System.exit(1);  // Setup error.
      }

      // Establish a Connection and Statement with which to work.
      Connection c = DriverManager.getConnection("jdbc:db2:*local");
      Statement s = c.createStatement();

      // Clean up any previous run of this application.
      try {
          s.executeUpdate("DROP TABLE CUJOSQL.BLOBTABLE");
      } catch (SQLException e) {
          // Ignore it - assume the table did not exist.
      }

      // Create a table with a BLOB column. The default BLOB column
      // size is 1 MB.
      s.executeUpdate("CREATE TABLE CUJOSQL.BLOBTABLE (COL1 BLOB)");

      // Create a PreparedStatement object that allows you to put
      // a new Blob object into the database.
      PreparedStatement ps = c.prepareStatement("INSERT INTO CUJOSQL.BLOBTABLE VALUES(?)");

      // Create a big BLOB value...
      Random random = new Random ();
      byte [] inByteArray = new byte[500000];
      random.nextBytes (inByteArray);
```

```
      // Set the PreparedStatement parameter. Note: This is not
      // portable to all JDBC drivers. JDBC drivers do not have
      // support when using setBytes for BLOB columns. This is used to
      // allow you to generate new BLOBs. It also allows JDBC 1.0
      // drivers to work with columns containing BLOB data.
      ps.setBytes(1, inByteArray);

      // Process the statement, inserting the BLOB into the database.
      ps.executeUpdate();

      // Process a query and obtain the BLOB that was just inserted out
      // of the database as a Blob object.
      ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.BLOBTABLE");
      rs.next();
      Blob blob = rs.getBlob(1);


      // Put that Blob back into the database through
      // the PreparedStatement.
      ps.setBlob(1, blob);
      ps.execute();

      c.close(); // Connection close also closes stmt and rs.
   }
}
```

例: IBM Developer Kit for Java の CallableStatement インターフェース

次に、CallableStatement インターフェースの使用法の例を示します。

例: CallableStatement インターフェース

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
// Connect to the server.
Connection c = DriverManager.getConnection("jdbc:db2://mySystem");

// Create the CallableStatement object.
// It precompiles the specified call to a stored procedure.
// The question marks indicate where input parameters must be set and
// where output parameters can be retrieved.
// The first two parameters are input parameters, and the third parameter is an output parameter.
CallableStatement cs = c.prepareCall("CALL MYLIBRARY.ADD (?, ?, ?)");

// Set input parameters.
cs.setInt (1, 123);
cs.setInt (2, 234);

// Register the type of the output parameter.
cs.registerOutParameter (3, Types.INTEGER);

// Run the stored procedure.
cs.execute ();

// Get the value of the output parameter.
int sum = cs.getInt (3);

// Close the CallableStatement and the Connection.
cs.close();
c.close();
```

100 ページの『CallableStatement』

JDBC CallableStatement インターフェースは PreparedStatement を拡張し、パラメーターの出力および入出力のサポートを提供します。 CallableStatement インターフェースは、PreparedStatement インターフェースによって提供される入力パラメーターもサポートします。

例: 他のステートメントのカーソルを介してテーブルから値を除去する

この Java の例では、他のステートメントのカーソルを介してテーブルから値を除去する方法を示します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;

public class UsingPositionedDelete {
    public Connection connection = null;
    public static void main(java.lang.String[] args) {
        UsingPositionedDelete test = new UsingPositionedDelete();

        test.setup();
        test.displayTable();

        test.run();
        test.displayTable();

        test.cleanup();
    }



/**
Handle all the required setup work.
**/
    public void setup() {
        try {
            // Register the JDBC driver.
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");

            connection = DriverManager.getConnection("jdbc:db2:*local");

            Statement s = connection.createStatement();
            try {
                s.executeUpdate("DROP TABLE CUJOSQL.WHERECUREX");
            } catch (SQLException e) {
                // Ignore problems here.
            }

            s.executeUpdate("CREATE TABLE CUJOSQL.WHERECUREX ( " +
                            "COL_IND INT, COL_VALUE CHAR(20)) ");

            for (int i = 1; i <= 10; i++) {
                s.executeUpdate("INSERT INTO CUJOSQL.WHERECUREX VALUES(" + i + ", 'FIRST')");
            }

            s.close();

        } catch (Exception e) {
            System.out.println("Caught exception: " + e.getMessage());
            e.printStackTrace();
        }
    }


/**
In this section, all the code to perform the testing should
```

```
be added. If only one connection to the database is needed,
the global variable 'connection' can be used.
**/
    public void run() {
        try {
            Statement stmt1 = connection.createStatement();

            // Update each value using next().
            stmt1.setCursorName("CUJO");
            ResultSet rs = stmt1.executeQuery ("SELECT * FROM CUJOSQL.WHERECUREX " +
                                               "FOR UPDATE OF COL_VALUE");

            System.out.println("Cursor name is " + rs.getCursorName());

            PreparedStatement stmt2 = connection.prepareStatement
                                      ("DELETE FROM " + " CUJOSQL.WHERECUREX WHERE CURRENT OF " +
                                      rs.getCursorName ());


            // Loop through the ResultSet and update every other entry.
            while (rs.next ()) {
                if (rs.next())
                    stmt2.execute ();
            }


            // Clean up the resources after they have been used.
            rs.close ();
            stmt2.close ();


        } catch (Exception e) {
            System.out.println("Caught exception: ");
            e.printStackTrace();
        }
    }



/**
In this section, put all clean-up work for testing.
**/
    public void cleanup() {
        try {
            // Close the global connection opened in setup().
            connection.close();

        } catch (Exception e) {
            System.out.println("Caught exception: ");
            e.printStackTrace();
        }
    }



/**
Display the contents of the table.
**/
    public void displayTable()
    {
        try {
            Statement s = connection.createStatement();
            ResultSet rs = s.executeQuery ("SELECT * FROM CUJOSQL.WHERECUREX");

            while (rs.next ()) {
                System.out.println("Index " + rs.getInt(1) + " value " + rs.getString(2));
            }
```

```
            rs.close ();
            s.close();
            System.out.println("-----------------------------------------");
        } catch (Exception e) {
            System.out.println("Caught exception: ");
            e.printStackTrace();
        }
    }
}
```

例: CLOB

以下は、 CLOB をデータベースに書き込んだり、データベースから検索したりする方法の例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
/////////////////////////////////////////
// PutGetClobs is an example application
// that shows how to work with the JDBC
// API to obtain and put CLOBs to and from
// database columns.
//
// The results of running this program
// are that there are two CLOB values
// in a new table. Both are identical
// and contain about 500k of repeating
// text data.
/////////////////////////////////////////
import java.sql.*;

public class PutGetClobs {
   public static void main(String[] args)
   throws SQLException
   {
       // Register the native JDBC driver.
       try {
          Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
      } catch (Exception e) {
          System.exit(1);  // Setup error.
      }

      // Establish a Connection and Statement with which to work.
      Connection c = DriverManager.getConnection("jdbc:db2:*local");
      Statement s = c.createStatement();

      // Clean up any previous run of this application.
      try {
          s.executeUpdate("DROP TABLE CUJOSQL.CLOBTABLE");
      } catch (SQLException e) {
          // Ignore it - assume the table did not exist.
      }

      // Create a table with a CLOB column. The default CLOB column
      // size is 1 MB.
      s.executeUpdate("CREATE TABLE CUJOSQL.CLOBTABLE (COL1 CLOB)");

      // Create a PreparedStatement object that allow you to put
      // a new Clob object into the database.
      PreparedStatement ps = c.prepareStatement("INSERT INTO CUJOSQL.CLOBTABLE VALUES(?)");

      // Create a big CLOB value...
      StringBuffer buffer = new StringBuffer(500000);
      while (buffer.length() < 500000) {
          buffer.append("All work and no play makes Cujo a dull boy.");
```

```
        }
        String clobValue = buffer.toString();

        // Set the PreparedStatement parameter. This is not
        // portable to all JDBC drivers. JDBC drivers do not have
        // to support setBytes for CLOB columns. This is done to
        // allow you to generate new CLOBs. It also
        // allows JDBC 1.0 drivers a way to work with columns containing
        // Clob data.
        ps.setString(1, clobValue);

        // Process the statement, inserting the clob into the database.
        ps.executeUpdate();

        // Process a query and get the CLOB that was just inserted out of the
        // database as a Clob object.
        ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.CLOBTABLE");
        rs.next();
        Clob clob = rs.getClob(1);


        // Put that Clob back into the database through
        // the PreparedStatement.
        ps.setClob(1, clob);
        ps.execute();

        c.close(); // Connection close also closes stmt and rs.
    }
}
```

例: UDBDataSource を作成して JNDI でバインドする

次に、UDBDataSource を作成し、それを JNDI でバインドする方法の例を示します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
// Import the required packages. At deployment time,
// the JDBC driver-specific class that implements
// DataSource must be imported.
import java.sql.*;
import javax.naming.*;
import com.ibm.db2.jdbc.app.UDBDataSource;

public class UDBDataSourceBind
{
    public static void main(java.lang.String[] args)
    throws Exception
    {
        // Create a new UDBDataSource object and give it
        // a description.
        UDBDataSource ds = new UDBDataSource();
        ds.setDescription("A simple UDBDataSource");

        // Retrieve a JNDI context. The context serves
        // as the root for where objects are bound or
        // found in JNDI.
        Context ctx = new InitialContext();

        // Bind the newly created UDBDataSource object
        // to the JNDI directory service, giving it a name
        // that can be used to look up this object again
        // at a later time.
        ctx.rebind("SimpleDS", ds);
    }
}
```

例: UDBDataSource の作成およびユーザー ID とパスワードの取得

以下は、UDBDataSource を作成し、実行時に getConnection メソッドを使用してユーザー ID とパスワードを取得する方法の例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
/// Import the required packages. There is
// no driver-specific code needed in runtime
// applications.
import java.sql.*;
import javax.sql.*;
import javax.naming.*;

public class UDBDataSourceUse2
{
    public static void main(java.lang.String[] args)
    throws Exception
    {
        // Retrieve a JNDI context. The context serves
        // as the root for where objects are bound or
        // found in JNDI.
        Context ctx = new InitialContext();

        // Retrieve the bound UDBDataSource object using the
        // name with which it was previously bound. At runtime,
        // only the DataSource interface is used, so there
        // is no need to convert the object to the UDBDataSource
        // implementation class. (There is no need to know
        // what the implementation class is. The logical JNDI name
        // is only required).
        DataSource ds = (DataSource) ctx.lookup("SimpleDS");


        // Once the DataSource is obtained, it can be used to establish
        // a connection. The user profile cujo and password newtiger
        // used to create the connection instead of any default user
        // ID and password for the DataSource.
        Connection connection = ds.getConnection("cujo", "newtiger");

        // The connection can be used to create Statement objects and
        // update the database or process queries as follows.
        Statement statement = connection.createStatement();
        ResultSet rs = statement.executeQuery("select * from qsys2.sysprocs");
        while (rs.next()) {
            System.out.println(rs.getString(1) + "." + rs.getString(2));
        }

        // The connection is closed before the application ends.
        connection.close();
    }
}
```

例: UDBDataSourceBind を作成して DataSource プロパティーを設定する

次に、UDBDataSource を作成し、DataSource のプロパティーとしてユーザー ID とパスワードを設定する方法の例を示します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
// Import the required packages. At deployment time,
// the JDBC driver-specific class that implements
// DataSource must be imported.
import java.sql.*;
import javax.naming.*;
import com.ibm.db2.jdbc.app.UDBDataSource;

public class UDBDataSourceBind2
{
    public static void main(java.lang.String[] args)
    throws Exception
    {
        // Create a new UDBDataSource object and give it
        // a description.
        UDBDataSource ds = new UDBDataSource();
        ds.setDescription("A simple UDBDataSource "  +
                          "with cujo as the default " +
                          "profile to connect with.");

        // Provide a user ID and password to be used for
        // connection requests.
        ds.setUser("cujo");
        ds.setPassword("newtiger");

        // Retrieve a JNDI context. The context serves
        // as the root for where objects are bound or
        // found in JNDI.
        Context ctx = new InitialContext();

        // Bind the newly created UDBDataSource object
        // to the JNDI directory service, giving it a name
        // that can be used to look up this object again
        // at a later time.
        ctx.rebind("SimpleDS2", ds);
    }
}
```

例: DatabaseMetaData インターフェースを使用して表のリストを戻す

次の例は、テーブルのリストを戻す方法を示しています。

注: サンプル・コードをご使用の場合は、560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
// Connect to the server.
Connection c = DriverManager.getConnection("jdbc:db2:mySystem");

// Get the database meta data from the connection.
DatabaseMetaData dbMeta = c.getMetaData();

// Get a list of tables matching this criteria.
String catalog = "myCatalog";
String schema = "mySchema";
String table  = "myTable%"; // % indicates search pattern
String types[] = {"TABLE", "VIEW", "SYSTEM TABLE"}:
ResultSet rs = dbMeta.getTables(catalog, schema, table, types);

// ... iterate through the ResultSet to get the values.

// Close the connection.
c.close():
```

例: Datalink

このサンプル・アプリケーションでは、JDBC API を使用して Datalink データベース列をハンドルする方法を示します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
/////////////////////////////////////////
// PutGetDatalinks is an example application
// that shows how to use the JDBC
// API to handle datalink database columns.
/////////////////////////////////////////
import java.sql.*;
import java.net.URL;
import java.net.MalformedURLException;

public class PutGetDatalinks {
    public static void main(String[] args)
    throws SQLException
    {
        // Register the native JDBC driver.
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
        } catch (Exception e) {
            System.exit(1);  // Setup error.
        }

        // Establish a Connection and Statement with which to work.
        Connection c = DriverManager.getConnection("jdbc:db2:*local");
        Statement s = c.createStatement();

        // Clean up any previous run of this application.
        try {
            s.executeUpdate("DROP TABLE CUJOSQL.DLTABLE");
        } catch (SQLException e) {
            // Ignore it - assume the table did not exist.
        }

        // Create a table with a datalink column.
        s.executeUpdate("CREATE TABLE CUJOSQL.DLTABLE (COL1 DATALINK)");

        // Create a PreparedStatement object that allows you to add
        // a new datalink into the database. Since conversing
        // to a datalink cannot be accomplished directly in the database, you
        // can code the SQL statement to perform the explicit conversion.
        PreparedStatement ps = c.prepareStatement("INSERT INTO CUJOSQL.DLTABLE
                                           VALUES(DLVALUE( CAST(? AS VARCHAR(100))))");

        // Set the datalink. This URL points you to a topic about
        // the new features of JDBC 3.0.
        ps.setString (1, "http://www.ibm.com/developerworks/java/library/j-jdbcnew/index.html");
        // Process the statement, inserting the CLOB into the database.
        ps.executeUpdate();

        // Process a query and obtain the CLOB that was just inserted out of the
        // database as a Clob object.
        ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.DLTABLE");
        rs.next();
        String datalink = rs.getString(1);


        // Put that datalink value into the database through
        // the PreparedStatement. Note: This function requires JDBC 3.0
        // support.
        /*
```

```
        try {
            URL url = new URL(datalink);
            ps.setURL(1, url);
            ps.execute();
        } catch (MalformedURLException mue) {
            // Handle this issue here.
        }

        rs = s.executeQuery("SELECT * FROM CUJOSQL.DLTABLE");
        rs.next();
        URL url = rs.getURL(1);
        System.out.println("URL value is " + url);
        */

        c.close(); // Connection close also closes stmt and rs.
    }
}
```

例: 特殊タイプ

以下に、特殊タイプの使用法の例を示します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
/////////////////////////////////////////
// This example program shows examples of
// various common tasks that can be done
// with distinct types.
/////////////////////////////////////////
import java.sql.*;

public class Distinct {
    public static void main(String[] args)
    throws SQLException
    {
        // Register the native JDBC driver.
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
        } catch (Exception e) {
            System.exit(1);  // Setup error.
        }

        Connection c = DriverManager.getConnection("jdbc:db2:*local");
        Statement s = c.createStatement();

        // Clean up any old runs.
        try {
            s.executeUpdate("DROP TABLE CUJOSQL.SERIALNOS");
        } catch (SQLException e) {
            // Ignore it and assume the table did not exist.
        }

        try {
            s.executeUpdate("DROP DISTINCT TYPE CUJOSQL.SSN");
        } catch (SQLException e) {
            // Ignore it and assume the table did not exist.
        }

        // Create the type, create the table, and insert a value.
        s.executeUpdate("CREATE DISTINCT TYPE CUJOSQL.SSN AS CHAR(9)");
        s.executeUpdate("CREATE TABLE CUJOSQL.SERIALNOS (COL1 CUJOSQL.SSN)");

        PreparedStatement ps = c.prepareStatement("INSERT INTO CUJOSQL.SERIALNOS VALUES(?)");
        ps.setString(1, "399924563");
        ps.executeUpdate();
```

```
        ps.close();

        // You can obtain details about the types available with new metadata in
        // JDBC 2.0
        DatabaseMetaData dmd = c.getMetaData();

        int types[] = new int[1];
        types[0] = java.sql.Types.DISTINCT;

        ResultSet rs = dmd.getUDTs(null, "CUJOSQL", "SSN", types);
        rs.next();
        System.out.println("Type name " + rs.getString(3) +
                           " has type " + rs.getString(4));


        // Access the data you have inserted.
        rs = s.executeQuery("SELECT COL1 FROM CUJOSQL.SERIALNOS");
        rs.next();
        System.out.println("The SSN is " + rs.getString(1));

        c.close(); // Connection close also closes stmt and rs.
    }
}
```

例: SQL ステートメントを Java アプリケーションに組み込む

以下の SQLJ アプリケーション例、App.sqlj は、静的 SQL を使用して更新データを DB2 サンプル・データベースの EMPLOYEE テーブルから検索します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

#sql iterator App_Cursor1 (String empno, String firstnme) ; // 1
#sql iterator App_Cursor2 (String) ;

class App
{

  /**********************
   **  Register Driver **
   **********************/

 static
 {
   try
   {
     Class.forName("com.ibm.db2.jdbc.app.DB2Driver").newInstance();
   }
   catch (Exception e)
   {
     e.printStackTrace();
   }
 }

  /********************
   **      Main      **
   ********************/

 public static void main(String argv[])
 {
   try
   {
```

```
App_Cursor1 cursor1;
App_Cursor2 cursor2;

String str1 = null;
String str2 = null;
long   count1;

// URL is jdbc:db2:dbname
String url = "jdbc:db2:sample";

DefaultContext ctx = DefaultContext.getDefaultContext();
if (ctx == null)
{
  try
  {
    // connect with default id/password
    Connection con = DriverManager.getConnection(url);
    con.setAutoCommit(false);
    ctx = new DefaultContext(con);
  }
  catch (SQLException e)
  {
    System.out.println("Error: could not get a default context");
    System.err.println(e) ;
    System.exit(1);
  }
  DefaultContext.setDefaultContext(ctx);
}

// retrieve data from the database
System.out.println("Retrieve some data from the database.");
#sql cursor1 = {SELECT empno, firstnme FROM employee}; // 2

// display the result set
// cursor1.next() returns false when there are no more rows
System.out.println("Received results:");
while (cursor1.next()) // 3
{
  str1 = cursor1.empno(); // 4
  str2 = cursor1.firstnme();

  System.out.print (" empno= " + str1);
  System.out.print (" firstname= " + str2);
  System.out.println("");
}
cursor1.close(); // 9

// retrieve number of employee from the database
#sql { SELECT count(*) into :count1 FROM employee }; // 5
if (1 == count1)
  System.out.println ("There is 1 row in employee table");
else
  System.out.println ("There are " + count1
                      + " rows in employee table");

// update the database
System.out.println("Update the database.");
#sql { UPDATE employee SET firstnme = 'SHILI' WHERE empno = '000010' };

// retrieve the updated data from the database
System.out.println("Retrieve the updated data from the database.");
str1 = "000010";
#sql cursor2 = {SELECT firstnme FROM employee WHERE empno = :str1}; // 6

// display the result set
// cursor2.next() returns false when there are no more rows
System.out.println("Received results:");
```

```
      while (true)
      {
        #sql { FETCH :cursor2 INTO :str2 }; // 7
        if (cursor2.endFetch()) break; // 8

        System.out.print (" empno= " + str1);
        System.out.print (" firstname= " + str2);
        System.out.println("");
      }
      cursor2.close(); // 9

      // rollback the update
      System.out.println("Rollback the update.");
      #sql { ROLLBACK work };
      System.out.println("Rollback done.");
    }
    catch( Exception e )
    {
      e.printStackTrace();
    }
  }
}
```

[1] 反復子を宣言する。このセクションでは、次の 2 種類の反復子を宣言します。

- App_Cursor1: 列データのタイプおよび名前を宣言して、列名 (列に結び付けられた名前) に応じた列の値を戻します。
- App_Cursor2: 列データのタイプを宣言して、列位置 (列に結び付けられた定位置) に応じた列の値を戻します。

[2] 反復子を初期設定する。反復子オブジェクト cursor1 が照会の結果を使用して初期設定されます。照会は結果を cursor1 に格納します。

[3] 反復子を次の行に進める。 cursor1.next() メソッドは、検索する行がなくなった場合にブール値の偽を戻します。

[4] データを移動する。名前付きアクセス機構メソッド empno() は、現在の行にある empno という名前の列の値を戻します。名前付きアクセス機構メソッド firstnme() は、現在の行にある firstnme() という名前の列の値を戻します。

[5] データをホスト変数に SELECT する。 SELECT ステートメントは、テーブル内の行数をホスト変数 count1 に渡します。

[6] 反復子を初期設定する。反復子オブジェクト cursor2 が照会の結果を使用して初期設定されます。照会は結果を cursor2 に格納します。

[7] データを検索する。 FETCH ステートメントは、結果テーブルから ByPos カーソル内で宣言された最初の列の現行値を、ホスト変数 str2 に戻します。

[8] FETCH.INTO ステートメントが成功したかを検査する。 endFetch() メソッドは、反復子が行に位置していない場合、つまり行を取り出す前回の試行が失敗した場合に、ブール値の真を戻します。 endFetch() メソッドは、行を取り出す前回の試行が成功した場合に、偽を戻します。 DB2 は next() メソッドが呼び出されたときに行の取り出しを試行します。 FETCH...INTO ステートメントは、暗黙的に next() メソッドを呼び出します。

[9] 反復子をクローズする。 close() メソッドは、反復子が保持しているリソースを解放します。反復子を明示的にクローズして、システム・リソースが適時に解放されるようにしてください。

例: トランザクションを終了する

以下は、アプリケーション内でトランザクションを終了する方法の例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.transaction.*;
import javax.transaction.xa.*;
import com.ibm.db2.jdbc.app.*;

public class JTATxEnd {

    public static void main(java.lang.String[] args) {
        JTATxEnd test = new JTATxEnd();

        test.setup();
        test.run();
    }


    /**
     * Handle the previous cleanup run so that this test can recommence.
     */
    public void setup() {

        Connection c = null;
        Statement s = null;
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            c = DriverManager.getConnection("jdbc:db2:*local");
            s = c.createStatement();

            try {
                s.executeUpdate("DROP TABLE CUJOSQL.JTATABLE");
            } catch (SQLException e) {
                // Ignore... does not exist
            }

            s.executeUpdate("CREATE TABLE CUJOSQL.JTATABLE (COL1 CHAR (50))");
            s.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('Fun with JTA')");
            s.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('JTA is fun.)");

            s.close();
        } finally {
            if (c != null) {
                c.close();
            }
        }
    }


    /**
     * This test use JTA support to handle transactions.
     */
    public void run() {
        Connection c = null;

        try {
            Context ctx = new InitialContext();

            // Assume the data source is backed by a UDBXADataSource.
            UDBXADataSource ds = (UDBXADataSource) ctx.lookup("XADataSource");
```

```
            // From the DataSource, obtain an XAConnection object that
            // contains an XAResource and a Connection object.
            XAConnection  xaConn = ds.getXAConnection();
            XAResource    xaRes  = xaConn.getXAResource();
            Connection    c      = xaConn.getConnection();

            // For XA transactions, transaction identifier is required.
            // An implementation of the XID interface is not included
            // with the JDBC driver. See Transactions with JTA for a
            // description of this interface to build a class for it.
            Xid xid = new XidImpl();

            // The connection from the XAResource can be used as any other
            // JDBC connection.
            Statement stmt = c.createStatement();

            // The XA resource must be notified before starting any
            // transactional work.
            xaRes.start(xid, XAResource.TMNOFLAGS);

            // Create a ResultSet during JDBC processing and fetch a row.
            ResultSet rs = stmt.executeUpdate("SELECT * FROM CUJOSQL.JTATABLE");
            rs.next();

            // When the end method is called, all ResultSet cursors close.
            // Accessing the ResultSet after this point results in an
            // exception being thrown.
            xaRes.end(xid, XAResource.TMNOFLAGS);

            try {
                String value = rs.getString(1);
                System.out.println("Something failed if you receive this message.");
            } catch (SQLException e) {
                System.out.println("The expected exception was thrown.");
            }

            // Commit the transaction to ensure that all locks are
            // released.
            int rc = xaRes.prepare(xid);
            xaRes.commit(xid, false);


        } catch (Exception e) {
            System.out.println("Something has gone wrong.");
            e.printStackTrace();
        } finally {
            try {
                if (c != null)
                    c.close();
            } catch (SQLException e) {
                System.out.println("Note:  Cleaup exception.");
                e.printStackTrace();
            }
        }
    }
}
```

75 ページの『JDBC 分散トランザクション』
通常、Java Database Connectivity (JDBC) のトランザクションはローカルです。 これは、単一の接続がトランザクションのすべての作業を行い、その接続では一度に 1 つのトランザクションだけが動作できることを意味します。

例: 無効なユーザー ID とパスワード

以下は、SQL 命名モードでの Connection プロパティーの使用法の例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
///////////////////////////////////////////////////////////////////////////////////
//
// InvalidConnect example.
//
// This program uses the Connection property in SQL naming mode.
//
///////////////////////////////////////////////////////////////////////////////////
//
// This source is an example of the IBM Developer for Java JDBC driver.
// IBM grants you a nonexclusive license to use this as an example
// from which you can generate similar function tailored to
// your own specific needs.
//
// This sample code is provided by IBM for illustrative purposes
// only. These examples have not been thoroughly tested under all
// conditions. IBM, therefore, cannot guarantee or imply
// reliability, serviceability, or function of these programs.
//
// All programs contained herein are provided to you "AS IS"
// without any warranties of any kind.  The implied warranties of
// merchantability and fitness for a particular purpose are
// expressly disclaimed.
//
// IBM Developer Kit for Java
// (C) Copyright IBM Corp. 2001
// All rights reserved.
// US Government Users Restricted Rights -
// Use, duplication, or disclosure restricted
// by GSA ADP Schedule Contract with IBM Corp.
//
///////////////////////////////////////////////////////////////////////////////////
import java.sql.*;
import java.util.*;

public class InvalidConnect {

    public static void main(java.lang.String[] args)
    {
        // Register the driver.
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
        } catch (ClassNotFoundException cnf) {
            System.out.println("ERROR: JDBC driver did not load.");
            System.exit(0);
        }

        // Attempt to obtain a connection without specifying any user or
        // password. The attempt works and the connection uses the
        // same user profile under which the job is running.
        try {
            Connection c1 = DriverManager.getConnection("jdbc:db2:*local");
            c1.close();
        } catch (SQLException e) {
            System.out.println("This test should not get into this exception path.");
            e.printStackTrace();
            System.exit(1);
        }

        try {
            Connection c2 = DriverManager.getConnection("jdbc:db2:*local",
                                                        "notvalid", "notvalid");
        } catch (SQLException e) {
            System.out.println("This is an expected error.");
            System.out.println("Message is " + e.getMessage());
```

```
            System.out.println("SQLSTATE is " + e.getSQLState());
        }

    }
}
```

例: JDBC

以下に、BasicJDBC プログラムの使用法の例を示します。このプログラムでは、IBM Developer Kit for Java のネイティブ JDBC ドライバーを使用して簡単な表を作成し、その表のデータを表示する照会を処理します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
//////////////////////////////////////////////////////////////////////////////////
//
// BasicJDBC example.  This program uses the native JDBC driver for the
// Developer Kit for Java to build a simple table and process a query
// that displays the data in that table.
//
// Command syntax:
//    BasicJDBC
//
//////////////////////////////////////////////////////////////////////////////////
//
// This source is an example of the native JDBC driver.
// IBM grants you a nonexclusive license to use this as an example
// from which you can generate similar function tailored to
// your own specific needs.
//
// This sample code is provided by IBM for illustrative purposes
// only. These examples have not been thoroughly tested under all
// conditions. IBM, therefore, cannot guarantee or imply
// reliability, serviceability, or function of these programs.
//
// All programs contained herein are provided to you "AS IS"
// without any warranties of any kind.  The implied warranties of
// merchantability and fitness for a particular purpose are
// expressly disclaimed.
//
// IBM Developer Kit for Java
// (C) Copyright IBM Corp. 2001
// All rights reserved.
// US Government Users Restricted Rights -
// Use, duplication, or disclosure restricted
// by GSA ADP Schedule Contract with IBM Corp.
//
//////////////////////////////////////////////////////////////////////////////////

// Include any Java classes that are to be used. In this application,
// many classes from the java.sql package are used and the
// java.util.Properties class is also used as part of obtaining
// a connection to the database.
import java.sql.*;
import java.util.Properties;

// Create a public class to encapsulate the program.
public class BasicJDBC {

    // The connection is a private variable of the object.
    private Connection connection = null;

    // Any class that is to be an 'entry point' for running
    // a program must have a main method. The main method
```

```
// is where processing begins when the program is called.
public static void main(java.lang.String[] args) {

    // Create an object of type BasicJDBC. This
    // is fundamental to object-oriented programming. Once
    // an object is created, call various methods on
    // that object to accomplish work.
    // In this case, calling the constructor for the object
    // creates a database connection that the other
    // methods use to do work against the database.
    BasicJDBC test = new BasicJDBC();

    // Call the rebuildTable method. This method ensures that
    // the table used in this program exists and looks
    // correct. The return value is a boolean for
    // whether or not rebuilding the table completed
    // successfully. If it did no, display a message
    // and exit the program.
    if (!test.rebuildTable()) {
        System.out.println("Failure occurred while setting up " +
                           " for running the test.");
        System.out.println("Test will not continue.");
        System.exit(0);
    }

    // The run query method is called next. This method
    // processes an SQL select statement against the table that
    // was created in the rebuildTable method. The output of
    // that query is output to standard out for you to view.
    test.runQuery();

    // Finally, the cleanup method is called. This method
    // ensures that the database connection that the object has
    // been hanging on to is closed.
    test.cleanup();
}


/**
This is the constructor for the basic JDBC test. It creates a database
connection that is stored in an instance variable to be used in later
method calls.
**/
public BasicJDBC() {

    // One way to create a database connection is to pass a URL
    // and a java Properties object to the DriverManager. The following
    // code constructs a Properties object that has your user ID and
    // password. These pieces of information are used for connecting
    // to the database.
    Properties properties = new Properties ();
    properties.put("user", "cujo");
    properties.put("password", "newtiger");

    // Use a try/catch block to catch all exceptions that can come out of the
    // following code.
    try {
        // The DriverManager must be aware that there is a JDBC driver available
        // to handle a user connection request. The following line causes the
        // native JDBC driver to be loaded and registered with the DriverManager.
        Class.forName("com.ibm.db2.jdbc.app.DB2Driver");

        // Create the database Connection object that this program uses in all
        // the other method calls that are made. The following code specifies
        // that a connection is to be established to the local database and that
        // that connection should conform to the properties that were set up
        // previously (that is, it should use the user ID and password specified).
```

```
        connection = DriverManager.getConnection("jdbc:db2:*local", properties);

    } catch (Exception e) {
        // If any of the lines in the try/catch block fail, control transfers to
        // the following line of code. A robust application tries to handle the
        // problem or provide more details to you. In this program, the error
        // message from the exception is displayed and the application allows
        // the program to return.
        System.out.println("Caught exception: " + e.getMessage());
    }
}


/**
Ensures that the qgpl.basicjdbc table looks you want it to at the start of
the test.

@returns boolean    Returns true if the table was rebuild successfully;
                    returns false if any failure occurred.
**/
public boolean rebuildTable() {
    // Wrap all the functionality in a try/catch block so an attempt is
    // made to handle any errors that may happen within this method.
    try {

        // Statement objects are used to process SQL statements against the
        // database. The Connection object is used to create a Statement
        // object.
        Statement s = connection.createStatement();

        try {
            // Build the test table from scratch. Process an update statement
            // that attempts to delete the table if it currently exists.
            s.executeUpdate("drop table qgpl.basicjdbc");
        } catch (SQLException e) {
            // Do not perform anything if an exception occurred. Assume
            // that the problem is that the table that was dropped does not
            // exist and that it can be created next.
        }

        // Use the statement object to create our table.
        s.executeUpdate("create table qgpl.basicjdbc(id int, name char(15))");

        // Use the statement object to populate our table with some data.
        s.executeUpdate("insert into qgpl.basicjdbc values(1, 'Frank Johnson')");
        s.executeUpdate("insert into qgpl.basicjdbc values(2, 'Neil Schwartz')");
        s.executeUpdate("insert into qgpl.basicjdbc values(3, 'Ben Rodman')");
        s.executeUpdate("insert into qgpl.basicjdbc values(4, 'Dan Gloore')");

        // Close the SQL statement to tell the database that it is no longer
        // needed.
        s.close();

        // If the entire method processed successfully, return true. At this point,
        // the table has been created or refreshed correctly.
        return true;

    } catch (SQLException sqle) {
        // If any of our SQL statements failed (other than the drop of the table
        // that was handled in the inner try/catch block), the error message is
        // displayed and false is returned to the caller, indicating that the table
        // may not be complete.
        System.out.println("Error in rebuildTable: " + sqle.getMessage());
        return false;
    }
}
```

```
/**
Runs a query against the demonstration table and the results are displayed to
standard out.
**/
public void runQuery() {
    // Wrap all the functionality in a try/catch block so an attempts is
    // made to handle any errors that might happen within this
    // method.
    try {
        // Create a Statement object.
        Statement s = connection.createStatement();

        // Use the statement object to run an SQL query. Queries return
        // ResultSet objects that are used to look at the data the query
        // provides.
        ResultSet rs = s.executeQuery("select * from qgpl.basicjdbc");

        // Display the top of our 'table' and initialize the counter for the
        // number of rows returned.
        System.out.println("--------------------");
        int i = 0;

        // The ResultSet next method is used to process the rows of a
        // ResultSet. The next method must be called once before the
        // first data is available for viewing. As long as next returns
        // true, there is another row of data that can be used.
        while (rs.next()) {

            // Obtain both columns in the table for each row and write a row to
            // our on-screen table with the data. Then, increment the count
            // of rows that have been processed.
            System.out.println("| " + rs.getInt(1) + " | " + rs.getString(2) + "|");
            i++;
        }

        // Place a border at the bottom on the table and display the number of rows
        // as output.
        System.out.println("--------------------");
        System.out.println("There were " + i + " rows returned.");
        System.out.println("Output is complete.");

    } catch (SQLException e) {
        // Display more information about any SQL exceptions that are
        // generated as output.
        System.out.println("SQLException exception: ");
        System.out.println("Message:....." + e.getMessage());
        System.out.println("SQLState:...." + e.getSQLState());
        System.out.println("Vendor Code:." + e.getErrorCode());
        e.printStackTrace();
    }
}


/**
The following method ensures that any JDBC resources that are still
allocated are freed.
**/
public void cleanup() {
    try {
        if (connection != null)
            connection.close();
    } catch (Exception e) {
        System.out.println("Caught exception: ");
```

```
            e.printStackTrace();
        }
    }
}
```

例: 単一トランザクション上で動作する複数の接続

以下は、単一トランザクション上で動作する複数の接続の使用法の例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.transaction.*;
import javax.transaction.xa.*;
import com.ibm.db2.jdbc.app.*;
public class JTAMultiConn {
   public static void main(java.lang.String[] args) {
      JTAMultiConn test = new JTAMultiConn();
      test.setup();
      test.run();
   }
/**
* Handle the previous cleanup run so that this test can recommence.
*/
   public void setup() {
      Connection c = null;
      Statement s = null;
      try {
         Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
         c = DriverManager.getConnection("jdbc:db2:*local");
         s = c.createStatement();
         try {
            s.executeUpdate("DROP TABLE CUJOSQL.JTATABLE");
         }
         catch (SQLException e) {
         // Ignore... does not exist
         }
         s.executeUpdate("CREATE TABLE CUJOSQL.JTATABLE (COL1 CHAR
                         (50))");
               s.close();
      }
      finally {
         if (c != null) {
            c.close();
         }
      }
   }
/**
* This test uses JTA support to handle transactions.
*/
   public void run() {
      Connection c1 = null;
      Connection c2 = null;
      Connection c3 = null;
      try {
         Context ctx = new InitialContext();
         // Assume the data source is backed by a UDBXADataSource.
         UDBXADataSource ds = (UDBXADataSource)
                              ctx.lookup("XADataSource");
         // From the DataSource, obtain some XAConnection objects that
         // contain an XAResource and a Connection object.
         XAConnection xaConn1 = ds.getXAConnection();
         XAConnection xaConn2 = ds.getXAConnection();
```

```
   XAConnection xaConn3 = ds.getXAConnection();
   XAResource xaRes1 = xaConn1.getXAResource();
   XAResource xaRes2 = xaConn2.getXAResource();
   XAResource xaRes3 = xaConn3.getXAResource();
   c1 = xaConn1.getConnection();
   c2 = xaConn2.getConnection();
   c3 = xaConn3.getConnection();
   Statement stmt1 = c1.createStatement();
   Statement stmt2 = c2.createStatement();
   Statement stmt3 = c3.createStatement();
   // For XA transactions, a transaction identifier is required.
   // Support for creating XIDs is again left to the application
   // program.
   Xid xid = JDXATest.xidFactory();
   // Perform some transactional work under each of the three
   // connections that have been created.
   xaRes1.start(xid, XAResource.TMNOFLAGS);
   int count1 = stmt1.executeUpdate("INSERT INTO " + tableName + "VALUES('Value 1-A')");
   xaRes1.end(xid, XAResource.TMNOFLAGS);

   xaRes2.start(xid, XAResource.TMJOIN);
   int count2 = stmt2.executeUpdate("INSERT INTO " + tableName + "VALUES('Value 1-B')");
   xaRes2.end(xid, XAResource.TMNOFLAGS);

   xaRes3.start(xid, XAResource.TMJOIN);
   int count3 = stmt3.executeUpdate("INSERT INTO " + tableName + "VALUES('Value 1-C')");
   xaRes3.end(xid, XAResource.TMSUCCESS);
   // When completed, commit the transaction as a single unit.
   // A prepare() and commit() or 1 phase commit() is required for
   // each separate database (XAResource) that participated in the
   // transaction. Since the resources accessed (xaRes1, xaRes2, and xaRes3)
   // all refer to the same database, only one prepare or commit is required.
   int rc = xaRes.prepare(xid);
   xaRes.commit(xid, false);
}
catch (Exception e) {
   System.out.println("Something has gone wrong.");
   e.printStackTrace();
}
finally {
   try {
      if (c1 != null) {
         c1.close();
      }
   }
   catch (SQLException e) {
      System.out.println("Note: Cleaup exception " +
                         e.getMessage());
   }
   try {
      if (c2 != null) {
         c2.close();
      }
   }
   catch (SQLException e) {
      System.out.println("Note: Cleaup exception " +
                         e.getMessage());
   }
   try {
      if (c3 != null) {
         c3.close();
      }
   }
   catch (SQLException e) {
      System.out.println("Note: Cleaup exception " +
                         e.getMessage());
```

```
            }
        }
    }
}
```

例: UDBDataSource をバインドする前に初期コンテキストを取得する

次の例では、UDBDataSource をバインドするにあたって、その前に初期コンテキストを取得します。その後、そのコンテキスト上で lookup メソッドを使用して、アプリケーションが使用する DataSource タイプのオブジェクトを戻します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
// Import the required packages. There is no
// driver-specific code needed in runtime
// applications.
import java.sql.*;
import javax.sql.*;
import javax.naming.*;

public class UDBDataSourceUse
{
    public static void main(java.lang.String[] args)
    throws Exception
    {
        // Retrieve a JNDI context. The context serves
        // as the root for where objects are bound or
        // found in JNDI.
        Context ctx = new InitialContext();

        // Retrieve the bound UDBDataSource object using the
        // name with which it was previously bound. At runtime,
        // only the DataSource interface is used, so there
        // is no need to convert the object to the UDBDataSource
        // implementation class. (There is no need to know what
        // the implementation class is. The logical JNDI name is
        // only required).
        DataSource ds = (DataSource) ctx.lookup("SimpleDS");


        // Once the DataSource is obtained, it can be used to establish
        // a connection. This Connection object is the same type
        // of object that is returned if the DriverManager approach
        // to establishing connection is used. Thus, so everything from
        // this point forward is exactly like any other JDBC
        // application.
        Connection connection = ds.getConnection();

        // The connection can be used to create Statement objects and
        // update the database or process queries as follows.
        Statement statement = connection.createStatement();
        ResultSet rs = statement.executeQuery("select * from qsys2.sysprocs");
        while (rs.next()) {
            System.out.println(rs.getString(1) + "." + rs.getString(2));
        }

        // The connection is closed before the application ends.
        connection.close();
    }
}
```

例: ParameterMetaData

これは、ParameterMetaData インターフェースを使用して、パラメーターについての情報を検索するときの一例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
//////////////////////////////////////////////////////////////////////////////////
//
// ParameterMetaData example. This program demonstrates
// the new support of JDBC 3.0 for learning information
// about parameters to a PreparedStatement.
//
// Command syntax:
//    java PMD
//
//////////////////////////////////////////////////////////////////////////////////
//
// This source is an example of the IBM Developer for Java JDBC driver.
// IBM grants you a nonexclusive license to use this as an example
// from which you can generate similar function tailored to
// your own specific needs.
//
// This sample code is provided by IBM for illustrative purposes
// only. These examples have not been thoroughly tested under all
// conditions. IBM, therefore, cannot guarantee or imply
// reliability, serviceability, or function of these programs.
//
// All programs contained herein are provided to you "AS IS"
// without any warranties of any kind.  The implied warranties of
// merchantability and fitness for a particular purpose are
// expressly disclaimed.
//
// IBM Developer Kit for Java
// (C) Copyright IBM Corp. 2001
// All rights reserved.
// US Government Users Restricted Rights -
// Use, duplication, or disclosure restricted
// by GSA ADP Schedule Contract with IBM Corp.
//
//////////////////////////////////////////////////////////////////////////////////

import java.sql.*;

public class PMD {

    // Program entry point.
    public static void main(java.lang.String[] args)
    throws Exception
    {
        // Obtain setup.
        Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
        Connection c = DriverManager.getConnection("jdbc:db2:*local");
        PreparedStatement ps = c.prepareStatement("INSERT INTO CUJOSQL.MYTABLE VALUES(?, ?, ?)");
        ParameterMetaData pmd = ps.getParameterMetaData();

        for (int i = 1; i < pmd.getParameterCount(); i++) {
            System.out.println("Parameter number " + i);
            System.out.println("  Class name is " + pmd.getParameterClassName(i));
            // Note: Mode relates to input, output or inout
            System.out.println("  Mode is " + pmd.getParameterClassName(i));
            System.out.println("  Type is " + pmd.getParameterType(i));
            System.out.println("  Type name is " + pmd.getParameterTypeName(i));
            System.out.println("  Precision is " + pmd.getPrecision(i));
            System.out.println("  Scale is " + pmd.getScale(i));
```

```
            System.out.println("  Nullable? is " + pmd.isNullable(i));
            System.out.println("  Signed? is " + pmd.isSigned(i));
        }
    }
}
```

例: 他のステートメントのカーソルを介してステートメントで値を変更する

この Java の例では、他のステートメントのカーソルを介したステートメントによる値の変更方法を示します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;

public class UsingPositionedUpdate {
    public Connection connection = null;
    public static void main(java.lang.String[] args) {

        UsingPositionedUpdate test = new UsingPositionedUpdate();

        test.setup();
        test.displayTable();

        test.run();
        test.displayTable();

        test.cleanup();
    }



/**
Handle all the required setup work.
**/
    public void setup() {
        try {
            // Register the JDBC driver.
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");

            connection = DriverManager.getConnection("jdbc:db2:*local");

            Statement s = connection.createStatement();
            try {
                s.executeUpdate("DROP TABLE CUJOSQL.WHERECUREX");
            } catch (SQLException e) {
                // Ignore problems here.
            }

            s.executeUpdate("CREATE TABLE CUJOSQL.WHERECUREX ( " +
                            "COL_IND INT, COL_VALUE CHAR(20)) ");

            for (int i = 1; i <= 10; i++) {
                s.executeUpdate("INSERT INTO CUJOSQL.WHERECUREX VALUES(" + i + ", 'FIRST')");
            }

            s.close();

        } catch (Exception e) {
            System.out.println("Caught exception: " + e.getMessage());
            e.printStackTrace();
        }
    }
```

```
/**
In this section, all the code to perform the testing should
be added. If only one connection to the database is required,
the global variable 'connection' can be used.
**/
    public void run() {
        try {
            Statement stmt1 = connection.createStatement();

            // Update each value using next().
            stmt1.setCursorName("CUJO");
            ResultSet rs = stmt1.executeQuery ("SELECT * FROM CUJOSQL.WHERECUREX " +
                                               "FOR UPDATE OF COL_VALUE");

            System.out.println("Cursor name is " + rs.getCursorName());

            PreparedStatement stmt2 = connection.prepareStatement ("UPDATE "
                                                                   + " CUJOSQL.WHERECUREX
                                                                       SET COL_VALUE = 'CHANGED'
                                                                       WHERE CURRENT OF "
                                                                   + rs.getCursorName ());


            // Loop through the ResultSet and update every other entry.
            while (rs.next ()) {
                if (rs.next())
                    stmt2.execute ();
            }


            // Clean up the resources after they have been used.
            rs.close ();
            stmt2.close ();


        } catch (Exception e) {
            System.out.println("Caught exception: ");
            e.printStackTrace();
        }
    }



/**
In this section, put all clean-up work for testing.
**/
    public void cleanup() {
        try {
            // Close the global connection opened in setup().
            connection.close();

        } catch (Exception e) {
            System.out.println("Caught exception: ");
            e.printStackTrace();
        }
    }



/**
Display the contents of the table.
**/
    public void displayTable()
    {
        try {
            Statement s = connection.createStatement();
            ResultSet rs = s.executeQuery ("SELECT * FROM CUJOSQL.WHERECUREX");
```

```
            while (rs.next ()) {
                System.out.println("Index " + rs.getInt(1) + " value " + rs.getString(2));
            }

            rs.close ();
            s.close();
            System.out.println("------------------------------------------");
        } catch (Exception e) {
            System.out.println("Caught exception: ");
            e.printStackTrace();
        }
    }
}
```

例: ResultSet インターフェース

以下は、ResultSet インターフェースの使用法の例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;

/**
ResultSetExample.java

This program demonstrates using a ResultSetMetaData and
a ResultSet to display all the data in a table even though
the program that gets the data does not know what the table
is going to look like (the user passes in the values for the
table and library).
**/
public class ResultSetExample {

    public static void main(java.lang.String[] args)
    {
        if (args.length != 2) {
            System.out.println("Usage:  java ResultSetExample <library> <table>");
            System.out.println(" where <library> is the library that contains <table>");
            System.exit(0);
        }

        Connection con = null;
        Statement s = null;
        ResultSet rs = null;
        ResultSetMetaData rsmd = null;

        try {
            // Get a database connection and prepare a statement.
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            con = DriverManager.getConnection("jdbc:db2:*local");

            s = con.createStatement();

            rs = s.executeQuery("SELECT * FROM " + args[0] + "." + args[1]);
            rsmd = rs.getMetaData();

            int colCount = rsmd.getColumnCount();
            int rowCount = 0;
            while (rs.next()) {
                rowCount++;
                System.out.println("Data for row " + rowCount);
                for (int i = 1; i <= colCount; i++)
                    System.out.println("   Row " + i + ": " + rs.getString(i));
            }
```

```
        } catch (Exception e) {
            // Handle any errors.
            System.out.println("Oops... we have an error... ");
            e.printStackTrace();
        } finally {
            // Ensure we always clean up.  If the connection gets closed, the
            // statement under it closes as well.
            if (con != null) {
                try {
                    con.close();
                } catch (SQLException e) {
                    System.out.println("Critical error - cannot close connection object");
                }
            }
        }
    }
}
```

例: ResultSet の感度

以下の例は、ResultSet の感度に基づいた、変更が SQL ステートメントの where 文節に与える影響を示しています。

この例にはフォーマット設定の正しくない箇所があるかもしれません。これは、この例を印刷ページに収めるためです。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;

public class Sensitive2 {

    public Connection connection = null;

    public static void main(java.lang.String[] args) {
        Sensitive2 test = new Sensitive2();

        test.setup();
        test.run("sensitive");
        test.cleanup();

        test.setup();
        test.run("insensitive");
        test.cleanup();
    }


    public void setup() {

        try {
            System.out.println("Native JDBC used");
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            connection = DriverManager.getConnection("jdbc:db2:*local");

            Statement s = connection.createStatement();
            try {
                s.executeUpdate("drop table cujosql.sensitive");
            } catch (SQLException e) {
                // Ignored.
            }
```

```
        s.executeUpdate("create table cujosql.sensitive(col1 int)");
        s.executeUpdate("insert into cujosql.sensitive values(1)");
        s.executeUpdate("insert into cujosql.sensitive values(2)");
        s.executeUpdate("insert into cujosql.sensitive values(3)");
        s.executeUpdate("insert into cujosql.sensitive values(4)");
        s.executeUpdate("insert into cujosql.sensitive values(5)");


        try {
            s.executeUpdate("drop table cujosql.sensitive2");
        } catch (SQLException e) {
            // Ignored.
        }

        s.executeUpdate("create table cujosql.sensitive2(col2 int)");
        s.executeUpdate("insert into cujosql.sensitive2 values(1)");
        s.executeUpdate("insert into cujosql.sensitive2 values(2)");
        s.executeUpdate("insert into cujosql.sensitive2 values(3)");
        s.executeUpdate("insert into cujosql.sensitive2 values(4)");
        s.executeUpdate("insert into cujosql.sensitive2 values(5)");

        s.close();

    } catch (Exception e) {
        System.out.println("Caught exception: " + e.getMessage());
        if (e instanceof SQLException) {
            SQLException another = ((SQLException) e).getNextException();
            System.out.println("Another:  " + another.getMessage());
        }
    }
}


public void run(String sensitivity) {
    try {

        Statement s = null;
        if (sensitivity.equalsIgnoreCase("insensitive")) {
            System.out.println("creating a TYPE_SCROLL_INSENSITIVE cursor");
            s = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                ResultSet.CONCUR_READ_ONLY);
        } else {
            System.out.println("creating a TYPE_SCROLL_SENSITIVE cursor");
            s = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                ResultSet.CONCUR_READ_ONLY);
        }

        ResultSet rs = s.executeQuery("select col1, col2 From cujosql.sensitive,
                               cujosql.sensitive2 where col1 = col2");

        rs.next();
        System.out.println("value is " + rs.getInt(1));
        rs.next();
        System.out.println("value is " + rs.getInt(1));
        rs.next();
        System.out.println("value is " + rs.getInt(1));
        rs.next();
        System.out.println("value is " + rs.getInt(1));

        System.out.println("fetched the four rows...");

        // Another statement creates a value that does not fit the where clause.
        Statement s2 =
```

```
            connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
            ResultSet.CONCUR_UPDATEABLE);
            ResultSet rs2 = s2.executeQuery("select *
            from cujosql.sensitive where col1 = 5 FOR UPDATE");
            rs2.next();
            rs2.updateInt(1, -1);
            rs2.updateRow();
            s2.close();

            if (rs.next()) {
                System.out.println("There is still a row: " + rs.getInt(1));
            } else {
                System.out.println("No more rows.");
            }

        } catch (SQLException e) {
            System.out.println("SQLException exception: ");
            System.out.println("Message:....." + e.getMessage());
            System.out.println("SQLState:...." + e.getSQLState());
            System.out.println("Vendor Code:." + e.getErrorCode());
            System.out.println("----------------------------");
            e.printStackTrace();
        }
        catch (Exception ex) {
            System.out.println("An exception other
            than an SQLException was thrown: ");
            ex.printStackTrace();
        }
    }



    public void cleanup() {
        try {
            connection.close();
        } catch (Exception e) {
            System.out.println("Caught exception: ");
            e.printStackTrace();
        }
    }
}
```

例: 感知および非感知の ResultSet

以下の例は、テーブルに行が挿入される際の、感知 ResultSet と非感知 ResultSet との違いを示しています。

注: サンプル・コードをご使用の場合は、560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;

public class Sensitive {

    public Connection connection = null;

    public static void main(java.lang.String[] args) {
        Sensitive test = new Sensitive();

        test.setup();
        test.run("sensitive");
        test.cleanup();
```

```
        test.setup();
        test.run("insensitive");
        test.cleanup();
    }


    public void setup() {

        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            connection = DriverManager.getConnection("jdbc:db2:*local");

            Statement s = connection.createStatement();
            try {
                s.executeUpdate("drop table cujosql.sensitive");
            } catch (SQLException e) {
                // Ignored.
            }

            s.executeUpdate("create table cujosql.sensitive(col1 int)");
            s.executeUpdate("insert into cujosql.sensitive values(1)");
            s.executeUpdate("insert into cujosql.sensitive values(2)");
            s.executeUpdate("insert into cujosql.sensitive values(3)");
            s.executeUpdate("insert into cujosql.sensitive values(4)");
            s.executeUpdate("insert into cujosql.sensitive values(5)");
            s.close();

        } catch (Exception e) {
            System.out.println("Caught exception: " + e.getMessage());
            if (e instanceof SQLException) {
                SQLException another = ((SQLException) e).getNextException();
                System.out.println("Another:  " + another.getMessage());
            }
        }
    }


    public void run(String sensitivity) {
        try {
            Statement s = null;
            if (sensitivity.equalsIgnoreCase("insensitive")) {
                System.out.println("creating a TYPE_SCROLL_INSENSITIVE cursor");
                s = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                    ResultSet.CONCUR_READ_ONLY);
            } else {
                System.out.println("creating a TYPE_SCROLL_SENSITIVE cursor");
                s = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                    ResultSet.CONCUR_READ_ONLY);
            }

            ResultSet rs = s.executeQuery("select * From cujosql.sensitive");

            // Fetch the five values that are there.
            rs.next();
            System.out.println("value is " + rs.getInt(1));
            rs.next();
            System.out.println("value is " + rs.getInt(1));
            rs.next();
            System.out.println("value is " + rs.getInt(1));
            rs.next();
            System.out.println("value is " + rs.getInt(1));
            rs.next();
```

```
            System.out.println("value is " + rs.getInt(1));
            System.out.println("fetched the five rows...");

            // Note:  If you fetch the last row, the ResultSet looks
            //        closed and subsequent new rows that are added
            //        are not be recognized.

            // Allow another statement to insert a new value.
            Statement s2 = connection.createStatement();
            s2.executeUpdate("insert into cujosql.sensitive values(6)");
            s2.close();

            // Whether a row is recognized is based on the sensitivity setting.
            if (rs.next()) {
                System.out.println("There is a row now: " + rs.getInt(1));
            } else {
                System.out.println("No more rows.");
            }


        } catch (SQLException e) {
            System.out.println("SQLException exception: ");
            System.out.println("Message:....." + e.getMessage());
            System.out.println("SQLState:...." + e.getSQLState());
            System.out.println("Vendor Code:." + e.getErrorCode());
            System.out.println("-------------------------------------");
            e.printStackTrace();
        }
        catch (Exception ex) {
            System.out.println("An exception other than an SQLException was thrown: ");
            ex.printStackTrace();
        }
    }



    public void cleanup() {
        try {
            connection.close();
        } catch (Exception e) {
            System.out.println("Caught exception: ");
            e.printStackTrace();
        }
    }
}
```

例: UDBDataSource および UDBConnectionPoolDataSource で接続プーリングをセットアップする

以下に、UDBDataSource および UDBConnectionPoolDataSource で接続プーリングを使用する方法の例を示します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;
import javax.naming.*;
import com.ibm.db2.jdbc.app.UDBDataSource;
import com.ibm.db2.jdbc.app.UDBConnectionPoolDataSource;

public class ConnectionPoolingSetup
{
```

```
    public static void main(java.lang.String[] args)
    throws Exception
    {
        // Create a ConnectionPoolDataSource implementation
        UDBConnectionPoolDataSource cpds = new UDBConnectionPoolDataSource();
        cpds.setDescription("Connection Pooling DataSource object");

        // Establish a JNDI context and bind the connection pool data source
        Context ctx = new InitialContext();
        ctx.rebind("ConnectionSupport", cpds);

        // Create a standard data source that references it.
        UDBDataSource ds = new UDBDataSource();
        ds.setDescription("DataSource supporting pooling");
        ds.setDataSourceName("ConnectionSupport");
        ctx.rebind("PoolingDataSource", ds);
  }
}
```

例: SQLException

以下に、SQLException をキャッチし、提供されたすべての情報をダンプする例を示します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;

public class ExceptionExample {

    public static Connection connection = null;

    public static void main(java.lang.String[] args) {

        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            connection = DriverManager.getConnection("jdbc:db2:*local");

            Statement s = connection.createStatement();
            int count = s.executeUpdate("insert into cujofake.cujofake values(1, 2,3)");

            System.out.println("Did not expect that table to exist.");


        } catch (SQLException e) {
            System.out.println("SQLException exception: ");
            System.out.println("Message:....." + e.getMessage());
            System.out.println("SQLState:...." + e.getSQLState());
            System.out.println("Vendor Code:." + e.getErrorCode());
            System.out.println("-----------------------------------------------------");
            e.printStackTrace();
        } catch (Exception ex) {
            System.out.println("An exception other than an SQLException was thrown: ");
            ex.printStackTrace();
        } finally {
            try {
                if (connection != null) {
                    connection.close();
                }
            } catch (SQLException e) {
                System.out.println("Exception caught attempting to shutdown...");
            }
        }
    }
}
```

例: トランザクションを中断して再開する

以下は、中断され、その後再開されるトランザクションの例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.transaction.*;
import javax.transaction.xa.*;
import com.ibm.db2.jdbc.app.*;
import javax.naming.InitialContext;
import javax.naming.Context;

public class JTATxSuspend {

    public static void main(java.lang.String[] args) {
        JTATxSuspend test = new JTATxSuspend();

        test.setup();
        test.run();
    }


    /**
     * Handle the previous cleanup run so that this test can recommence.
     */
    public void setup() {

        Connection c = null;
        Statement s = null;
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            c = DriverManager.getConnection("jdbc:db2:*local");
            s = c.createStatement();

            try {
                s.executeUpdate("DROP TABLE CUJOSQL.JTATABLE");
            } catch (SQLException e) {
                // Ignore... doesn't exist
            }

            s.executeUpdate("CREATE TABLE CUJOSQL.JTATABLE (COL1 CHAR (50))");
            s.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('Fun with JTA')");
            s.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('JTA is fun.)");

            s.close();
        } finally {
            if (c != null) {
                c.close();
            }
        }
    }


    /**
     * This test uses JTA support to handle transactions.
     */
    public void run() {
        Connection c = null;

        try {
            Context ctx = new InitialContext();

            // Assume the data source is backed by a UDBXADataSource.
```

```
UDBXADataSource ds = (UDBXADataSource) ctx.lookup("XADataSource");

// From the DataSource, obtain an XAConnection object that
// contains an XAResource and a Connection object.
XAConnection  xaConn = ds.getXAConnection();
XAResource    xaRes  = xaConn.getXAResource();
c      = xaConn.getConnection();

// For XA transactions, a transaction identifier is required.
// An implementation of the XID interface is not included with
// the JDBC driver. See topic "Transactions with JTA" for a
// description of this interface to build a class for it.
Xid xid = new XidImpl();

// The connection from the XAResource can be used as any other
// JDBC connection.
Statement stmt = c.createStatement();

// The XA resource must be notified before starting any
// transactional work.
xaRes.start(xid, XAResource.TMNOFLAGS);

// Create a ResultSet during JDBC processing and fetch a row.
ResultSet rs = stmt.executeQuery("SELECT * FROM CUJOSQL.JTATABLE");
rs.next();

// The end method is called with the suspend option.
// ResultSets associated with the current transaction are 'on hold'.
// They are neither gone nor accessible in this state.
xaRes.end(xid, XAResource.TMSUSPEND);


// Other work can be performed with the transaction.
// As an example, you can create a statement and process a query.
// This work and any other transactional work that the transaction may
// perform is separate from the work done previously under the XID.
Statement nonXAStmt = c.createStatement();
ResultSet nonXARS = nonXAStmt.executeQuery("SELECT * FROM CUJOSQL.JTATABLE");
while (nonXARS.next()) {
    // Process here...
}
nonXARS.close();
nonXAStmt.close();


// If an attempt is made to use any suspended transactions
// resources, an exception results.
try {
    rs.getString(1);
    System.out.println("Value of the first row is " + rs.getString(1));
} catch (SQLException e) {
    System.out.println("This was an expected exception - " +
                       "suspended ResultSet was used.");
}


// Resume the suspended transaction and complete the work on it.
// The ResultSet is exactly as it was before the suspension.
xaRes.start(newXid, XAResource.TMRESUME);
rs.next();
System.out.println("Value of the second row is " + rs.getString(1));


// When the transaction has completed, end it
// and commit any work under it.
xaRes.end(xid, XAResource.TMNOFLAGS);
int rc = xaRes.prepare(xid);
```

```
            xaRes.commit(xid, false);


        } catch (Exception e) {
            System.out.println("Something has gone wrong.");
            e.printStackTrace();
        } finally {
            try {
                if (c != null)
                    c.close();
            } catch (SQLException e) {
                System.out.println("Note:  Cleaup exception.");
                e.printStackTrace();
            }
        }
    }
}
```

例: 中断状態の ResultSets

以下は、作業を実行するために Statement オブジェクトを別のトランザクションで再処理する方法の例です。

注: サンプル・コードをご使用の場合は、560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.transaction.*;
import javax.transaction.xa.*;
import com.ibm.db2.jdbc.app.*;

public class JTATxEffect {

    public static void main(java.lang.String[] args) {
        JTATxEffect test = new JTATxEffect();

        test.setup();
        test.run();
    }


    /**
     * Handle the previous cleanup run so that this test can recommence.
     */
    public void setup() {

        Connection c = null;
        Statement s = null;
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            c = DriverManager.getConnection("jdbc:db2:*local");
            s = c.createStatement();

            try {
                s.executeUpdate("DROP TABLE CUJOSQL.JTATABLE");
            } catch (SQLException e) {
                // Ignore... does not exist
            }

            s.executeUpdate("CREATE TABLE CUJOSQL.JTATABLE (COL1 CHAR (50))");
            s.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('Fun with JTA')");
            s.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('JTA is fun.)");

            s.close();
```

```
    } finally {
        if (c != null) {
            c.close();
        }
    }
}


/**
 * This test uses JTA support to handle transactions.
 */
public void run() {
    Connection c = null;

    try {
        Context ctx = new InitialContext();

        // Assume the data source is backed by a UDBXADataSource.
        UDBXADataSource ds = (UDBXADataSource) ctx.lookup("XADataSource");

        // From the DataSource, obtain an XAConnection object that
        // contains an XAResource and a Connection object.
        XAConnection  xaConn = ds.getXAConnection();
        XAResource    xaRes  = xaConn.getXAResource();
        Connection    c      = xaConn.getConnection();

        // For XA transactions, a transaction identifier is required.
        // An implementation of the XID interface is not included with
        // the JDBC driver. See Transactions with JTA
        // for a description of this interface to build a
        // class for it.
        Xid xid = new XidImpl();

        // The connection from the XAResource can be used as any other
        // JDBC connection.
        Statement stmt = c.createStatement();

        // The XA resource must be notified before starting any
        // transactional work.
        xaRes.start(xid, XAResource.TMNOFLAGS);

        // Create a ResultSet during JDBC processing and fetch a row.
        ResultSet rs = stmt.executeUpdate("SELECT * FROM CUJOSQL.JTATABLE");
        rs.next();

        // The end method is called with the suspend option.
        // ResultSets associated with the current transaction are 'on hold'.
        // They are neither gone nor accessible in this state.
        xaRes.end(xid, XAResource.TMSUSPEND);


        // In the meantime, other work can be done outside the transaction.
        // The ResultSets under the transaction can be closed if the
        // Statement object used to create them is reused.
        ResultSet nonXARS = stmt.executeQuery("SELECT * FROM CUJOSQL.JTATABLE");
        while (nonXARS.next()) {
            // Process here...
        }


        // Attempt to go back to the suspended transaction. The suspended
        // transaction's ResultSet has disappeared because the statement
        // has been processed again.
        xaRes.start(newXid, XAResource.TMRESUME);
        try {
            rs.next();
        } catch (SQLException ex) {
```

```
                System.out.println("This exception is expected. " +
                                   "The ResultSet closed due to another process.");
            }


            // When the transaction had completed, end it
            // and commit any work under it.
            xaRes.end(xid, XAResource.TMNOFLAGS);
            int rc = xaRes.prepare(xid);
            xaRes.commit(xid, false);


        } catch (Exception e) {
            System.out.println("Something has gone wrong.");
            e.printStackTrace();
        } finally {
            try {
                if (c != null)
                    c.close();
            } catch (SQLException e) {
                System.out.println("Note:  Cleaup exception.");
                e.printStackTrace();
            }
        }
    }
}
```

例: 接続プーリングのパフォーマンスをテストする

以下に、プーリングされたときのパフォーマンスとプーリングされていないときのパフォーマンスを対比してテストする方法を示します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;
import javax.naming.*;
import java.util.*;
import javax.sql.*;

public class ConnectionPoolingTest
{
    public static void main(java.lang.String[] args)
    throws Exception
    {
        Context ctx = new InitialContext();
        // Do the work without a pool:
        DataSource ds = (DataSource) ctx.lookup("BaseDataSource");
        System.out.println("¥nStart timing the non-pooling DataSource version...");

        long startTime = System.currentTimeMillis();
        for (int i = 0; i < 100; i++) {
            Connection c1 = ds.getConnection();
            c1.close();
        }
        long endTime = System.currentTimeMillis();
        System.out.println("Time spent: " + (endTime - startTime));

        // Do the work with pooling:
        ds = (DataSource) ctx.lookup("PoolingDataSource");
        System.out.println("¥nStart timing the pooling version...");

        startTime = System.currentTimeMillis();
        for (int i = 0; i < 100; i++) {
            Connection c1 = ds.getConnection();
            c1.close();
```

```
        }
        endTime = System.currentTimeMillis();
        System.out.println("Time spent: " + (endTime - startTime));
    }
}
```

例: 2 つの DataSource のパフォーマンスをテストする

次に、接続プーリングだけを使用する 1 つの DataSource と、ステートメントと接続プーリングを使用する別の DataSource をテストする例を示します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;
import javax.naming.*;
import java.util.*;
import javax.sql.*;
import com.ibm.db2.jdbc.app.UDBDataSource;
import com.ibm.db2.jdbc.app.UDBConnectionPoolDataSource;

public class StatementPoolingTest
{

    public static void main(java.lang.String[] args)
    throws Exception
    {
        Context ctx = new InitialContext();

        System.out.println("deploying statement pooling data source");
        deployStatementPoolDataSource();

        // Do the work with connection pooling only.
        DataSource ds = (DataSource) ctx.lookup("PoolingDataSource");
        System.out.println("¥nStart timing the connection pooling only version...");

        long startTime = System.currentTimeMillis();
        for (int i = 0; i < 100; i++) {
            Connection c1 = ds.getConnection();
            PreparedStatement ps = c1.prepareStatement("select * from qsys2.sysprocs");
            ResultSet rs = ps.executeQuery();
            c1.close();
        }
        long endTime = System.currentTimeMillis();
        System.out.println("Time spent: " + (endTime - startTime));


        // Do the work with statement pooling added.
        ds = (DataSource) ctx.lookup("StatementPoolingDataSource");
        System.out.println("¥nStart timing the statement pooling version...");

        startTime = System.currentTimeMillis();
        for (int i = 0; i < 100; i++) {
            Connection c1 = ds.getConnection();
            PreparedStatement ps = c1.prepareStatement("select * from qsys2.sysprocs");
            ResultSet rs = ps.executeQuery();
            c1.close();
        }
        endTime = System.currentTimeMillis();
        System.out.println("Time spent: " + (endTime - startTime));
    }


    private static void deployStatementPoolDataSource()
    throws Exception
```

```
    {
        // Create a ConnectionPoolDataSource implementation
        UDBConnectionPoolDataSource cpds = new UDBConnectionPoolDataSource();
        cpds.setDescription("Connection Pooling DataSource object with Statement pooling");
        cpds.setMaxStatements(10);

        // Establish a JNDI context and bind the connection pool data source
        Context ctx = new InitialContext();
        ctx.rebind("StatementSupport", cpds);


        // Create a standard datasource that references it.
        UDBDataSource ds = new UDBDataSource();
        ds.setDescription("DataSource supporting statement pooling");
        ds.setDataSourceName("StatementSupport");
        ctx.rebind("StatementPoolingDataSource", ds);

    }
}
```

例: BLOB の更新

以下は、Java アプリケーション中で BLOB を更新する方法の例です。

注: サンプル・コードをご使用の場合は、560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
/////////////////////////////////////////
// UpdateBlobs is an example application
// that shows some of the APIs providing
// support for changing Blob objects
// and reflecting those changes to the
// database.
//
// This program must be run after
// the PutGetBlobs program has completed.
/////////////////////////////////////////
import java.sql.*;

public class UpdateBlobs {
   public static void main(String[] args)
   throws SQLException
   {
       // Register the native JDBC driver.
       try {
          Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
      } catch (Exception e) {
          System.exit(1);  // Setup error.
      }

      Connection c = DriverManager.getConnection("jdbc:db2:*local");
      Statement s = c.createStatement();

      ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.BLOBTABLE");

      rs.next();
      Blob blob1 = rs.getBlob(1);
      rs.next();
      Blob blob2 = rs.getBlob(1);


      // Truncate a BLOB.
      blob1.truncate((long) 150000);
      System.out.println("Blob1's new length is " + blob1.length());

      // Update part of the BLOB with a new byte array.
```

```
        // The following code obtains the bytes that are at
        // positions 4000-4500 and set them to positions 500-1000.

        // Obtain part of the BLOB as a byte array.
        byte[] bytes = blob1.getBytes(4000L, 4500);

        int bytesWritten = blob2.setBytes(500L, bytes);

        System.out.println("Bytes written is " + bytesWritten);

        // The bytes are now found at position 500 in blob2
        long startInBlob2 = blob2.position(bytes, 1);

        System.out.println("pattern found starting at position " + startInBlob2);

        c.close(); // Connection close also closes stmt and rs.
    }
}
```

例: CLOB の更新

以下は、Java アプリケーション中で CLOB を更新する方法の例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
/////////////////////////////////////////
// UpdateClobs is an example application
// that shows some of the APIs providing
// support for changing Clob objects
// and reflecting those changes to the
// database.
//
// This program must be run after
// the PutGetClobs program has completed.
/////////////////////////////////////////
import java.sql.*;

public class UpdateClobs {
   public static void main(String[] args)
   throws SQLException
   {
       // Register the native JDBC driver.
       try {
          Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
      } catch (Exception e) {
          System.exit(1);  // Setup error.
      }

      Connection c = DriverManager.getConnection("jdbc:db2:*local");
      Statement s = c.createStatement();

      ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.CLOBTABLE");

      rs.next();
      Clob clob1 = rs.getClob(1);
      rs.next();
      Clob clob2 = rs.getClob(1);


      // Truncate a CLOB.
      clob1.truncate((long) 150000);
      System.out.println("Clob1's new length is " + clob1.length());

      // Update a portion of the CLOB with a new String value.
      String value = "Some new data for once";
```

```
        int charsWritten = clob2.setString(500L, value);

        System.out.println("Characters written is " + charsWritten);

        // The bytes can be found at position 500 in clob2
        long startInClob2 = clob2.position(value, 1);

        System.out.println("pattern found starting at position " + startInClob2);

        c.close(); // Connection close also closes stmt and rs.
    }
}
```

例: 複数のトランザクションで単一の接続を使用する

以下は、複数のトランザクションでの単一接続の使用法の例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.transaction.*;
import javax.transaction.xa.*;
import com.ibm.db2.jdbc.app.*;

public class JTAMultiTx {

    public static void main(java.lang.String[] args) {
        JTAMultiTx test = new JTAMultiTx();

        test.setup();
        test.run();
    }


    /**
     * Handle the previous cleanup run so that this test can recommence.
     */
    public void setup() {

        Connection c = null;
        Statement s = null;
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            c = DriverManager.getConnection("jdbc:db2:*local");
            s = c.createStatement();

            try {
                s.executeUpdate("DROP TABLE CUJOSQL.JTATABLE");
            } catch (SQLException e) {
                // Ignore... does not exist
            }

            s.executeUpdate("CREATE TABLE CUJOSQL.JTATABLE (COL1 CHAR (50))");

            s.close();
        } finally {
            if (c != null) {
                c.close();
            }
        }
    }
```

```
/**
 * This test uses JTA support to handle transactions.
 */
public void run() {
    Connection c = null;

    try {
        Context ctx = new InitialContext();

        // Assume the data source is backed by a UDBXADataSource.
        UDBXADataSource ds = (UDBXADataSource) ctx.lookup("XADataSource");

        // From the DataSource, obtain an XAConnection object that
        // contains an XAResource and a Connection object.
        XAConnection  xaConn = ds.getXAConnection();
        XAResource    xaRes  = xaConn.getXAResource();
        Connection    c      = xaConn.getConnection();
        Statement     stmt   = c.createStatement();

        // For XA transactions, a transaction identifier is required.
        // This is not meant to imply that all the XIDs are the same.
        // Each XID must be unique to distinguish the various transactions
        // that occur.
        // Support for creating XIDs is again left to the application
        // program.
        Xid xid1 = JDXATest.xidFactory();
        Xid xid2 = JDXATest.xidFactory();
        Xid xid3 = JDXATest.xidFactory();

        // Do work under three transactions for this connection.
        xaRes.start(xid1, XAResource.TMNOFLAGS);
        int count1 = stmt.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('Value 1-A')");
        xaRes.end(xid1, XAResource.TMNOFLAGS);

        xaRes.start(xid2, XAResource.TMNOFLAGS);
        int count2 = stmt.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('Value 1-B')");
        xaRes.end(xid2, XAResource.TMNOFLAGS);

        xaRes.start(xid3, XAResource.TMNOFLAGS);
        int count3 = stmt.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('Value 1-C')");
        xaRes.end(xid3, XAResource.TMNOFLAGS);

        // Prepare all the transactions
        int rc1 = xaRes.prepare(xid1);
        int rc2 = xaRes.prepare(xid2);
        int rc3 = xaRes.prepare(xid3);

        // Two of the transactions commit and one rolls back.
        // The attempt to insert the second value into the table is
        // not committed.
        xaRes.commit(xid1, false);
        xaRes.rollback(xid2);
        xaRes.commit(xid3, false);


    } catch (Exception e) {
        System.out.println("Something has gone wrong.");
        e.printStackTrace();
    } finally {
        try {
            if (c != null)
                c.close();
        } catch (SQLException e) {
            System.out.println("Note:  Cleaup exception.");
            e.printStackTrace();
```

```
            }
        }
    }
}
```

例: BLOB の使用

以下は、Java アプリケーション中で BLOB を使用する方法の例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
/////////////////////////////////////////
// UseBlobs is an example application
// that shows some of the APIs associated
// with Blob objects.
//
// This program must be run after
// the PutGetBlobs program has completed.
/////////////////////////////////////////
import java.sql.*;

public class UseBlobs {
   public static void main(String[] args)
   throws SQLException
   {
       // Register the native JDBC driver.
       try {
          Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
      } catch (Exception e) {
          System.exit(1);  // Setup error.
      }

      Connection c = DriverManager.getConnection("jdbc:db2:*local");
      Statement s = c.createStatement();

      ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.BLOBTABLE");

      rs.next();
      Blob blob1 = rs.getBlob(1);
      rs.next();
      Blob blob2 = rs.getBlob(1);


      // Determine the length of a LOB.
      long end = blob1.length();
      System.out.println("Blob1 length is " + blob1.length());

      // When working with LOBs, all indexing that is related to them
      // is 1-based, and is not 0-based like strings and arrays.
      long startingPoint = 450;
      long endingPoint = 500;

      // Obtain part of the BLOB as a byte array.
      byte[] outByteArray = blob1.getBytes(startingPoint, (int)endingPoint);

      // Find where a sub-BLOB or byte array is first found within a
      // BLOB. The setup for this program placed two identical copies of
      // a random BLOB into the database. Thus, the start position of the
      // byte array extracted from blob1 can be found in the starting
      // position in blob2. The exception would be if there were 50
      // identical random bytes in the LOBs previously.
      long startInBlob2 = blob2.position(outByteArray, 1);

      System.out.println("pattern found starting at position " + startInBlob2);
```

```
      c.close(); // Connection close closes stmt and rs too.
   }
}
```

例: CLOB の使用

以下は、Java アプリケーション中で CLOB を使用する方法の例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
/////////////////////////////////////////
// UpdateClobs is an example application
// that shows some of the APIs providing
// support for changing Clob objects
// and reflecting those changes to the
// database.
//
// This program must be run after
// the PutGetClobs program has completed.
/////////////////////////////////////////
import java.sql.*;

public class UseClobs {
   public static void main(String[] args)
   throws SQLException
   {
       // Register the native JDBC driver.
       try {
          Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
      } catch (Exception e) {
          System.exit(1);  // Setup error.
      }

      Connection c = DriverManager.getConnection("jdbc:db2:*local");
      Statement s = c.createStatement();

      ResultSet rs = s.executeQuery("SELECT * FROM CUJOSQL.CLOBTABLE");

      rs.next();
      Clob clob1 = rs.getClob(1);
      rs.next();
      Clob clob2 = rs.getClob(1);


      // Determine the length of a LOB.
      long end = clob1.length();
      System.out.println("Clob1 length is " + clob1.length());

      // When working with LOBs, all indexing that is related to them
      // is 1-based, and not 0-based like strings and arrays.
      long startingPoint = 450;
      long endingPoint = 50;

      // Obtain part of the CLOB as a byte array.
      String outString = clob1.getSubString(startingPoint, (int)endingPoint);
      System.out.println("Clob substring is " + outString);

      // Find where a sub-CLOB or string is first found within a
      // CLOB. The setup for this program placed two identical copies of
      // a repeating CLOB into the database. Thus, the start position of the
      // string extracted from clob1 can be found in the starting
      // position in clob2 if the search begins close to the position where
   // the string starts.
      long startInClob2 = clob2.position(outString, 440);
```

```
        System.out.println("pattern found starting at position " + startInClob2);

        c.close(); // Connection close also closes stmt and rs.
    }
}
```

例: トランザクションを処理するために JTA を使用する

以下は、アプリケーション内でトランザクションを処理するための Java Transaction API (JTA) の使用法の例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.transaction.*;
import javax.transaction.xa.*;
import com.ibm.db2.jdbc.app.*;

public class JTACommit {

    public static void main(java.lang.String[] args) {
        JTACommit test = new JTACommit();

        test.setup();
        test.run();
    }


    /**
     * Handle the previous cleanup run so that this test can recommence.
     */
    public void setup() {

        Connection c = null;
        Statement s = null;
        try {
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
            c = DriverManager.getConnection("jdbc:db2:*local");
            s = c.createStatement();

            try {
                s.executeUpdate("DROP TABLE CUJOSQL.JTATABLE");
            } catch (SQLException e) {
                // Ignore... does not exist
            }

            s.executeUpdate("CREATE TABLE CUJOSQL.JTATABLE (COL1 CHAR (50))");
            s.close();
        } finally {
            if (c != null) {
                c.close();
            }
        }
    }


    /**
     * This test uses JTA support to handle transactions.
     */
    public void run() {
        Connection c = null;
```

```
        try {
            Context ctx = new InitialContext();

            // Assume the data source is backed by a UDBXADataSource.
            UDBXADataSource ds = (UDBXADataSource) ctx.lookup("XADataSource");

            // From the DataSource, obtain an XAConnection object that
            // contains an XAResource and a Connection object.
            XAConnection  xaConn = ds.getXAConnection();
            XAResource    xaRes  = xaConn.getXAResource();
            Connection    c      = xaConn.getConnection();

            // For XA transactions, a transaction identifier is required.
            // An implementation of the XID interface is not included with the
            // JDBC driver. See Transactions with JTA for a description of
            // this interface to build a class for it.
            Xid xid = new XidImpl();

            // The connection from the XAResource can be used as any other
            // JDBC connection.
            Statement stmt = c.createStatement();

            // The XA resource must be notified before starting any
            // transactional work.
            xaRes.start(xid, XAResource.TMNOFLAGS);

            // Standard JDBC work is performed.
            int count =
              stmt.executeUpdate("INSERT INTO CUJOSQL.JTATABLE VALUES('JTA is pretty fun.')");

            // When the transaction work has completed, the XA resource must
            // again be notified.
            xaRes.end(xid, XAResource.TMSUCCESS);

            // The transaction represented by the transaction ID is prepared
            // to be committed.
            int rc = xaRes.prepare(xid);

            // The transaction is committed through the XAResource.
            // The JDBC Connection object is not used to commit
            // the transaction when using JTA.
            xaRes.commit(xid, false);


        } catch (Exception e) {
            System.out.println("Something has gone wrong.");
            e.printStackTrace();
        } finally {
            try {
                if (c != null)
                    c.close();
            } catch (SQLException e) {
                System.out.println("Note:  Cleaup exception.");
                e.printStackTrace();
            }
        }
    }
}
```

例: 複数の列を持ったメタデータ ResultSet を使用する

以下は、複数の列があるメタデータ ResultSet を使用する方法の例です。

注: サンプル・コードをご使用の場合は、560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
///////////////////////////////////////////////////////////////////////////////////
//
// SafeGetUDTs example. This program demonstrates one way to deal with
// metadata ResultSets that have more columns in JDK 1.4 than they
// had in previous releases.
//
// Command syntax:
//    java SafeGetUDTs
//
///////////////////////////////////////////////////////////////////////////////////
//
// This source is an example of the IBM Developer for Java JDBC driver.
// IBM grants you a nonexclusive license to use this as an example
// from which you can generate similar function tailored to
// your own specific needs.
//
// This sample code is provided by IBM for illustrative purposes
// only. These examples have not been thoroughly tested under all
// conditions. IBM, therefore, cannot guarantee or imply
// reliability, serviceability, or function of these programs.
//
// All programs contained herein are provided to you "AS IS"
// without any warranties of any kind.  The implied warranties of
// merchantability and fitness for a particular purpose are
// expressly disclaimed.
//
// IBM Developer Kit for Java
// (C) Copyright IBM Corp. 2001
// All rights reserved.
// US Government Users Restricted Rights -
// Use, duplication, or disclosure restricted
// by GSA ADP Schedule Contract with IBM Corp.
//
///////////////////////////////////////////////////////////////////////////////////

import java.sql.*;

public class SafeGetUDTs {

    public static int jdbcLevel;

    // Note: Static block runs before main begins.
    // Therefore, there is access to jdbcLevel in
    // main.
    {
        try {
            Class.forName("java.sql.Blob");

            try {
                Class.forName("java.sql.ParameterMetaData");
                // Found a JDBC 3.0 interface.  Must support JDBC 3.0.
                jdbcLevel = 3;
            } catch (ClassNotFoundException ez) {
                // Could not find the JDBC 3.0 ParameterMetaData class.
                // Must be running under a JVM with only JDBC 2.0
                // support.
                jdbcLevel = 2;
            }

        } catch (ClassNotFoundException ex) {
            // Could not find the JDBC 2.0 Blob class.  Must be
            // running under a JVM with only JDBC 1.0 support.
            jdbcLevel = 1;
        }
    }

    // Program entry point.
```

```
    public static void main(java.lang.String[] args)
    {
        Connection c = null;

        try {
            // Get the driver registered.
            Class.forName("com.ibm.db2.jdbc.app.DB2Driver");

            c = DriverManager.getConnection("jdbc:db2:*local");
            DatabaseMetaData dmd = c.getMetaData();

            if (jdbcLevel == 1) {
                System.out.println("No support is provided for getUDTs.  Just return.");
                System.exit(1);
            }

            ResultSet rs = dmd.getUDTs(null, "CUJOSQL", "SSN%", null);
            while (rs.next()) {

                // Fetch all the columns that have been available since the
                // JDBC 2.0 release.
                System.out.println("TYPE_CAT is " + rs.getString("TYPE_CAT"));
                System.out.println("TYPE_SCHEM is " + rs.getString("TYPE_SCHEM"));
                System.out.println("TYPE_NAME is " + rs.getString("TYPE_NAME"));
                System.out.println("CLASS_NAME is " + rs.getString("CLASS_NAME"));
                System.out.println("DATA_TYPE is " + rs.getString("DATA_TYPE"));
                System.out.println("REMARKS is " + rs.getString("REMARKS"));

                // Fetch all the columns that were added in JDBC 3.0.
                if (jdbcLevel > 2) {
                    System.out.println("BASE_TYPE is " + rs.getString("BASE_TYPE"));
                }
            }
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        } finally {
            if (c != null) {
                try {
                    c.close();
                } catch (SQLException e) {
                    // Ignoring shutdown exception.
                }
            }
        }
    }
}
```

例: ネイティブ JDBC と IBM Toolbox for Java JDBC を同時に使用する

以下に、プログラム中でネイティブ JDBC 接続と IBM Toolbox for Java JDBC 接続を使用する方法を示します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
//////////////////////////////////////////////////////////////////////////////////
//
// GetConnections example.
//
// This program demonstrates being able to use both JDBC drivers at
// once in a program. Two Connection objects are created in this
// program. One is a native JDBC connection and one is a IBM Toolbox for Java
// JDBC connection.
//
```

```
// This technique is convenient because it allows you to use different
// JDBC drivers for different tasks concurrently. For example, the
// IBM Toolbox for Java JDBC driver is ideal for connecting to a remote IBM i
// server and the native JDBC driver is faster for local connections.
// You can use the strengths of each driver concurrently in your
// application by writing code similar to this example.
//
//////////////////////////////////////////////////////////////////////////////////
//
// This source is an example of the IBM Developer for Java JDBC driver.
// IBM grants you a nonexclusive license to use this as an example
// from which you can generate similar function tailored to
// your own specific needs.
//
// This sample code is provided by IBM for illustrative purposes
// only. These examples have not been thoroughly tested under all
// conditions. IBM, therefore, cannot guarantee or imply
// reliability, serviceability, or function of these programs.
//
// All programs contained herein are provided to you "AS IS"
// without any warranties of any kind.  The implied warranties of
// merchantability and fitness for a particular purpose are
// expressly disclaimed.
//
// IBM Developer Kit for Java
// (C) Copyright IBM Corp. 2001
// All rights reserved.
// US Government Users Restricted Rights -
// Use, duplication, or disclosure restricted
// by GSA ADP Schedule Contract with IBM Corp.
//
//////////////////////////////////////////////////////////////////////////////////
import java.sql.*;
import java.util.*;

public class GetConnections {

   public static void main(java.lang.String[] args)
   {
       // Verify input.
       if (args.length != 2) {
          System.out.println("Usage (CL command line):  java GetConnections PARM(<user> <password>)");
          System.out.println(" where <user> is a valid IBM i user ID");
          System.out.println("   and <password> is the password for that user ID");
          System.exit(0);
       }

       // Register both drivers.
      try {
         Class.forName("com.ibm.db2.jdbc.app.DB2Driver");
         Class.forName("com.ibm.as400.access.AS400JDBCDriver");
      } catch (ClassNotFoundException cnf) {
         System.out.println("ERROR: One of the JDBC drivers did not load.");
         System.exit(0);
      }

      try {
         // Obtain a connection with each driver.
         Connection conn1 = DriverManager.getConnection("jdbc:db2://localhost", args[0], args[1]);
         Connection conn2 = DriverManager.getConnection("jdbc:as400://localhost", args[0], args[1]);

         // Verify that they are different.
         if (conn1 instanceof com.ibm.db2.jdbc.app.DB2Connection)
            System.out.println("conn1 is running under the native JDBC driver.");
         else
            System.out.println("There is something wrong with conn1.");
```

```
         if (conn2 instanceof com.ibm.as400.access.AS400JDBCConnection)
            System.out.println("conn2 is running under the IBM Toolbox for Java JDBC driver.");
         else
            System.out.println("There is something wrong with conn2.");

         conn1.close();
         conn2.close();
      } catch (SQLException e) {
         System.out.println("ERROR: " + e.getMessage());
      }
   }
}
```

例: ResultSet を取得するために PreparedStatement を使用する

これは、PreparedStatement オブジェクトの executeQuery メソッドを使用して、ResultSet を入手するときの一例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;
import java.util.Properties;

public class PreparedStatementExample {

    public static void main(java.lang.String[] args)
    {
        // Load the following from a properties object.
        String DRIVER = "com.ibm.db2.jdbc.app.DB2Driver";
        String URL    = "jdbc:db2://*local";

        // Register the native JDBC driver. If the driver cannot
        // be registered, the test cannot continue.
        try {
            Class.forName(DRIVER);
        } catch (Exception e) {
            System.out.println("Driver failed to register.");
            System.out.println(e.getMessage());
            System.exit(1);
        }

        Connection c = null;
        Statement s = null;

        //    This program creates a table that is
        //    used by prepared statements later.
        try {
            // Create the connection properties.
            Properties properties = new Properties ();
            properties.put ("user", "userid");
            properties.put ("password", "password");

            // Connect to the local database.
            c = DriverManager.getConnection(URL, properties);

            // Create a Statement object.
            s = c.createStatement();
            // Delete the test table if it exists.  Note that
            // this example assumes throughout that the collection
            // MYLIBRARY exists on the system.
            try {
                s.executeUpdate("DROP TABLE MYLIBRARY.MYTABLE");
            } catch (SQLException e) {
                // Just continue... the table probably did not exist.
            }
```

```
    // Run an SQL statement that creates a table in the database.
    s.executeUpdate("CREATE TABLE MYLIBRARY.MYTABLE (NAME VARCHAR(20), ID INTEGER)");

} catch (SQLException sqle) {
    System.out.println("Database processing has failed.");
    System.out.println("Reason: " + sqle.getMessage());
} finally {
    // Close database resources
    try {
        if (s != null) {
            s.close();
        }
    } catch (SQLException e) {
        System.out.println("Cleanup failed to close Statement.");
    }
}


//    This program then uses a prepared statement to insert many
//    rows into the database.
PreparedStatement ps = null;
String[] nameArray = {"Rich", "Fred", "Mark", "Scott", "Jason",
    "John", "Jessica", "Blair", "Erica", "Barb"};
try {
    // Create a PreparedStatement object that is used to insert data into the
    // table.
    ps = c.prepareStatement("INSERT INTO MYLIBRARY.MYTABLE (NAME, ID) VALUES (?, ?)");

    for (int i = 0; i < nameArray.length; i++) {
        ps.setString(1, nameArray[i]);      // Set the Name from our array.
        ps.setInt(2, i+1);                  // Set the ID.
        ps.executeUpdate();
    }

} catch (SQLException sqle) {
    System.out.println("Database processing has failed.");
    System.out.println("Reason: " + sqle.getMessage());
} finally {
    // Close database resources
    try {
        if (ps != null) {
            ps.close();
        }
    } catch (SQLException e) {
        System.out.println("Cleanup failed to close Statement.");
    }
}


//    Use a prepared statement to query the database
//    table that has been created and return data from it. In
//    this example, the parameter used is arbitrarily set to
//    5, meaning return all rows where the ID field is less than
//    or equal to 5.
try {
    ps = c.prepareStatement("SELECT * FROM MYLIBRARY.MYTABLE " +
                            "WHERE ID <= ?");

    ps.setInt(1, 5);

    // Run an SQL query on the table.
    ResultSet rs = ps.executeQuery();
    // Display all the data in the table.
    while (rs.next()) {
        System.out.println("Employee " + rs.getString(1) + " has ID " + rs.getInt(2));
    }
```

```
        } catch (SQLException sqle) {
            System.out.println("Database processing has failed.");
            System.out.println("Reason: " + sqle.getMessage());
        } finally {
            // Close database resources
            try {
                if (ps != null) {
                    ps.close();
                }
            } catch (SQLException e) {
                System.out.println("Cleanup failed to close Statement.");
            }

            try {
                if (c != null) {
                    c.close();
                }
            } catch (SQLException e) {
                System.out.println("Cleanup failed to close Connection.");
            }

        }
    }
}
```

例: Statement オブジェクトの executeUpdate メソッドを使用する

次に、Statement オブジェクトの executeUpdate メソッドを使用する方法の例を示します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;
import java.util.Properties;

public class StatementExample {

    public static void main(java.lang.String[] args)
    {

        // Suggestion:  Load these from a properties object.
        String DRIVER = "com.ibm.db2.jdbc.app.DB2Driver";
        String URL    = "jdbc:db2://*local";

        // Register the native JDBC driver. If the driver cannot be
        // registered, the test cannot continue.
        try {
            Class.forName(DRIVER);
        } catch (Exception e) {
            System.out.println("Driver failed to register.");
            System.out.println(e.getMessage());
            System.exit(1);
        }

        Connection c = null;
        Statement s = null;

        try {
            // Create the connection properties.
            Properties properties = new Properties ();
            properties.put ("user", "userid");
            properties.put ("password", "password");

            // Connect to the local database.
            c = DriverManager.getConnection(URL, properties);
```

```
            // Create a Statement object.
            s = c.createStatement();
            // Delete the test table if it exists. Note: This
            // example assumes that the collection MYLIBRARY
            // exists on the system.
            try {
                s.executeUpdate("DROP TABLE MYLIBRARY.MYTABLE");
            } catch (SQLException e) {
                // Just continue... the table probably does not exist.
            }

            // Run an SQL statement that creates a table in the database.
            s.executeUpdate("CREATE TABLE MYLIBRARY.MYTABLE (NAME VARCHAR(20), ID INTEGER)");

            // Run some SQL statements that insert records into the table.
            s.executeUpdate("INSERT INTO MYLIBRARY.MYTABLE (NAME, ID) VALUES ('RICH', 123)");
            s.executeUpdate("INSERT INTO MYLIBRARY.MYTABLE (NAME, ID) VALUES ('FRED', 456)");
            s.executeUpdate("INSERT INTO MYLIBRARY.MYTABLE (NAME, ID) VALUES ('MARK', 789)");

            // Run an SQL query on the table.
            ResultSet rs = s.executeQuery("SELECT * FROM MYLIBRARY.MYTABLE");

            // Display all the data in the table.
            while (rs.next()) {
                System.out.println("Employee " + rs.getString(1) + " has ID " + rs.getInt(2));
            }

        } catch (SQLException sqle) {
            System.out.println("Database processing has failed.");
            System.out.println("Reason: " + sqle.getMessage());
        } finally {
            // Close database resources
            try {
                if (s != null) {
                    s.close();
                }
            } catch (SQLException e) {
                System.out.println("Cleanup failed to close Statement.");
            }
             }

            try {
                if (c != null) {
                    c.close();
                }
            } catch (SQLException e) {
                System.out.println("Cleanup failed to close Connection.");
            }
        }
    }
}
```

例: JAAS HelloWorld

ここで示されている例は、JAAS の HelloWorld をコンパイルして実行するために必要な 3 つのファイルを示しています。

HelloWorld.java

以下は、ファイル HelloWorld.java のソースです。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
/*
 * ===========================================================================
 * Licensed Materials - Property of IBM
 *
 * (C) Copyright IBM Corp. 2000 All Rights Reserved.
  *
 *  US Government Users Restricted Rights - Use, duplication or
 *  disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
 * ===========================================================================
 *
 * File: HelloWorld.java
 */

import java.io.*;
import java.util.*;
import java.security.Principal;
import java.security.PrivilegedAction;
import javax.security.auth.*;
import javax.security.auth.callback.*;
import javax.security.auth.login.*;
import javax.security.auth.spi.*;

/**
 * This SampleLogin application attempts to authenticate a user.
 *
 * If the user successfully authenticates itself,
 * the user name and number of Credentials is displayed.
 *
 * @version 1.1, 09/14/99
 */
public class HelloWorld {

    /**
     * Attempt to authenticate the user.
     */
    public static void main(String[] args) {
        // use the configured LoginModules for the "helloWorld" entry
        LoginContext lc = null;
        try {
            lc = new LoginContext("helloWorld", new MyCallbackHandler());
        } catch (LoginException le) {
            le.printStackTrace();
            System.exit(-1);
        }

        // the user has 3 attempts to authenticate successfully
        int i;
        for (i = 0; i < 3; i++) {
            try {

                // attempt authentication
                lc.login();

                // if we return with no exception, authentication succeeded
                break;

            } catch (AccountExpiredException aee) {

                System.out.println("Your account has expired");
                System.exit(-1);

            } catch (CredentialExpiredException cee) {

                System.out.println("Your credentials have expired.");
                System.exit(-1);

            } catch (FailedLoginException fle) {
```

```
                System.out.println("Authentication Failed");
                    try {
                        Thread.currentThread().sleep(3000);
                    } catch (Exception e) {
                        // ignore
                    }

            } catch (Exception e) {

                System.out.println("Unexpected Exception - unable to continue");
                e.printStackTrace();
                System.exit(-1);
            }
        }

        // did they fail three times?
        if (i == 3) {
            System.out.println("Sorry");
            System.exit(-1);
        }

        // Look at what Principals we have:
        Iterator principalIterator = lc.getSubject().getPrincipals().iterator();
        System.out.println("¥n¥nAuthenticated user has the following Principals:");
        while (principalIterator.hasNext()) {
            Principal p = (Principal)principalIterator.next();
            System.out.println("¥t" + p.toString());
        }

        // Look at some Principal-based work:
        Subject.doAsPrivileged(lc.getSubject(), new PrivilegedAction() {
            public Object run() {
                System.out.println("¥nYour java.home property: "
                                +System.getProperty("java.home"));

                System.out.println("¥nYour user.home property: "
                                +System.getProperty("user.home"));

                File f = new File("foo.txt");
                System.out.print("¥nfoo.txt does ");
                if (!f.exists()) System.out.print("not ");
                System.out.println("exist in your current directory");

                System.out.println("¥nOh, by the way ...");

                try {
                    Thread.currentThread().sleep(2000);
                } catch (Exception e) {
                    // ignore
                }
                System.out.println("¥n¥nHello World!\n");
                return null;
            }
        }, null);
        System.exit(0);
    }
}


/**
 * The application must implement the CallbackHandler.
 *
 * This application is text-based.  Therefore it displays information
 * to the user using the OutputStreams System.out and System.err,
 * and gathers input from the user using the InputStream, System.in.
 */
```

```
class MyCallbackHandler implements CallbackHandler {

    /**
     * Invoke an array of Callbacks.
     *
     *
     * @param callbacks an array of Callback objects which contain
     *          the information requested by an underlying security
     *          service to be retrieved or displayed.
     *
     * @exception java.io.IOException if an input or output error occurs.
     *
     * @exception UnsupportedCallbackException if the implementation of this
     *          method does not support one or more of the Callbacks
     *          specified in the callbacks parameter.
     */
    public void handle(Callback[] callbacks)
    throws IOException, UnsupportedCallbackException {

    for (int i = 0; i < callbacks.length; i++) {
        if (callbacks[i] instanceof TextOutputCallback) {

        // display the message according to the specified type
        TextOutputCallback toc = (TextOutputCallback)callbacks[i];
        switch (toc.getMessageType()) {
        case TextOutputCallback.INFORMATION:
            System.out.println(toc.getMessage());
            break;
        case TextOutputCallback.ERROR:
            System.out.println("ERROR: " + toc.getMessage());
            break;
        case TextOutputCallback.WARNING:
            System.out.println("WARNING: " + toc.getMessage());
            break;
        default:
            throw new IOException("Unsupported message type: " +
                    toc.getMessageType());
        }

        } else if (callbacks[i] instanceof NameCallback) {

        // prompt the user for a user name
        NameCallback nc = (NameCallback)callbacks[i];

        // ignore the provided defaultName
        System.err.print(nc.getPrompt());
        System.err.flush();
        nc.setName((new BufferedReader
            (new InputStreamReader(System.in))).readLine());

        } else if (callbacks[i] instanceof PasswordCallback) {

        // prompt the user for sensitive information
        PasswordCallback pc = (PasswordCallback)callbacks[i];
        System.err.print(pc.getPrompt());
        System.err.flush();
        pc.setPassword(readPassword(System.in));

        } else {
        throw new UnsupportedCallbackException
            (callbacks[i], "Unrecognized Callback");
        }
    }
    }

    // Reads user password from given input stream.
    private char[] readPassword(InputStream in) throws IOException {
```

```
    char[] lineBuffer;
    char[] buf;
    int i;

    buf = lineBuffer = new char[128];

    int room = buf.length;
    int offset = 0;
    int c;

    loop:   while (true) {
        switch (c = in.read()) {
        case -1:
        case '¥n':
        break loop;

        case '¥r':
        int c2 = in.read();
        if ((c2 != '¥n') && (c2 != -1)) {
            if (!(in instanceof PushbackInputStream)) {
            in = new PushbackInputStream(in);
            }
            ((PushbackInputStream)in).unread(c2);
        } else
            break loop;

        default:
        if (--room < 0) {
            buf = new char[offset + 128];
            room = buf.length - offset - 1;
            System.arraycopy(lineBuffer, 0, buf, 0, offset);
            Arrays.fill(lineBuffer, ' ');
            lineBuffer = buf;
        }
        buf[offset++] = (char) c;
        break;
        }
    }

    if (offset == 0) {
        return null;
    }

    char[] ret = new char[offset];
    System.arraycopy(buf, 0, ret, 0, offset);
    Arrays.fill(buf, ' ');

    return ret;
    }
}
```

HWLoginModule.java

以下は HWLoginModule.java のソースです。

注: 法律上の重要な情報に関しては、 コードの特記事項情報をお読みください。

```
/*
 * ===========================================================================
 * Licensed Materials - Property of IBM
 *
 * (C) Copyright IBM Corp. 2000 All Rights Reserved.
  *
 *  US Government Users Restricted Rights - Use, duplication or
 *  disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
```

```
 * =========================================================================
 *
 * File: HWLoginModule.java
 */

package com.ibm.security;

import java.util.*;
import java.io.IOException;
import javax.security.auth.*;
import javax.security.auth.callback.*;
import javax.security.auth.login.*;
import javax.security.auth.spi.*;
import com.ibm.security.HWPrincipal;

/**
 * This LoginModule authenticates users with a password.
 *
 * This LoginModule only recognizes any user who enters
 *     the required password:   Go JAAS
 *
 * If the user successfully authenticates itself,
 * a HWPrincipal with the user name
 * is added to the Subject.
 *
 * This LoginModule recognizes the debug option.
 * If set to true in the login Configuration,
 * debug messages are sent to the output stream, System.out.
 *
 * @version 1.1, 09/10/99
 */
public class HWLoginModule implements LoginModule {

    // initial state
    private Subject subject;
    private CallbackHandler callbackHandler;
    private Map sharedState;
    private Map options;

    // configurable option
    private boolean debug = false;

    // the authentication status
    private boolean succeeded = false;
    private boolean commitSucceeded = false;

    // user name and password
    private String user name;
    private char[] password;

    private HWPrincipal userPrincipal;

    /**
     * Initialize this LoginModule.
     *
     * @param subject the Subject to be authenticated.
     *
     * @param callbackHandler a CallbackHandler for communicating
     *          with the end user (prompting for user names and
     *          passwords, for example).
     *
     * @param sharedState shared LoginModule state.
     *
     * @param options options specified in the login
     *          Configuration for this particular
     *          LoginModule.
     */
```

```
public void initialize(Subject subject, CallbackHandler callbackHandler,
        Map sharedState, Map options) {

this.subject = subject;
this.callbackHandler = callbackHandler;
this.sharedState = sharedState;
this.options = options;

// initialize any configured options
debug = "true".equalsIgnoreCase((String)options.get("debug"));
}

/**
 * Authenticate the user by prompting for a user name and password.
 *
 *
 * @return true in all cases since this LoginModule
 *          should not be ignored.
 *
 * @exception FailedLoginException if the authentication fails.
 *
 * @exception LoginException if this LoginModule
 *      is unable to perform the authentication.
 */
public boolean login() throws LoginException {

// prompt for a user name and password
if (callbackHandler == null)
    throw new LoginException("Error: no CallbackHandler available " +
        "to garner authentication information from the user");

Callback[] callbacks = new Callback[2];
callbacks[0] = new NameCallback("¥n¥nHWModule user name: ");
callbacks[1] = new PasswordCallback("HWModule password: ", false);

try {
    callbackHandler.handle(callbacks);
    user name = ((NameCallback)callbacks[0]).getName();
    char[] tmpPassword = ((PasswordCallback)callbacks[1]).getPassword();
    if (tmpPassword == null) {
    // treat a NULL password as an empty password
    tmpPassword = new char[0];
    }
    password = new char[tmpPassword.length];
    System.arraycopy(tmpPassword, 0,
        password, 0, tmpPassword.length);
    ((PasswordCallback)callbacks[1]).clearPassword();

} catch (java.io.IOException ioe) {
    throw new LoginException(ioe.toString());
} catch (UnsupportedCallbackException uce) {
    throw new LoginException("Error: " + uce.getCallback().toString() +
    " not available to garner authentication information " +
    "from the user");
}

// print debugging information
if (debug) {
    System.out.println("¥n¥n¥t[HWLoginModule] " +
            "user entered user name: " +
            user name);
    System.out.print("¥t[HWLoginModule] " +
            "user entered password: ");
    for (int i = 0; i > password.length; i++)
    System.out.print(password[i]);
    System.out.println();
}
```

```
// verify the password
if (password.length == 7 &&
    password[0] == 'G' &&
    password[1] == 'o' &&
    password[2] == ' ' &&
    password[3] == 'J' &&
    password[4] == 'A' &&
    password[5] == 'A' &&
    password[6] == 'S') {

    // authentication succeeded!!!
    if (debug)
    System.out.println("¥n¥t[HWLoginModule] " +
            "authentication succeeded");
    succeeded = true;
    return true;
} else {

    // authentication failed -- clean out state
    if (debug)
    System.out.println("¥n¥t[HWLoginModule] " +
            "authentication failed");
    succeeded = false;
    user name = null;
    for (int i = 0; i < password.length; i++)
    password[i] = ' ';
    password = null;
    throw new FailedLoginException("Password Incorrect");
}
}

/**
 * This method is called if the overall authentication of LoginContext
     * succeeded
 * (the relevant REQUIRED, REQUISITE, SUFFICIENT and OPTIONAL LoginModules
 * succeeded).
 *
 * If this LoginModule authentication attempt
 * succeeded (checked by retrieving the private state saved by the
 * login method), then this method associates a
 * SolarisPrincipal
 * with the Subject located in the
 * LoginModule.  If this LoginModule
 * authentication attempt failed, then this method removes
 * any state that was originally saved.
 *
 * @exception LoginException if the commit fails.
 *
 * @return true if the login and commit LoginModule
 *      attempts succeeded, or false otherwise.
 */
public boolean commit() throws LoginException {
if (succeeded == false) {
    return false;
} else {
    // add a Principal (authenticated identity)
    // to the Subject

    // assume the user we authenticated is the HWPrincipal
    userPrincipal = new HWPrincipal(user name);
    final Subject s = subject;
    final HWPrincipal sp = userPrincipal;
    java.security.AccessController.doPrivileged
    (new java.security.PrivilegedAction() {
    public Object run() {
        if (!s.getPrincipals().contains(sp))
```

```
        s.getPrincipals().add(sp);
        return null;
    }
    });

    if (debug) {
    System.out.println("¥t[HWLoginModule] " +
            "added HWPrincipal to Subject");
    }

    // in any case, clean out state
    user name = null;
    for (int i = 0; i > password.length; i++)
    password[i] = ' ';
    password = null;

    commitSucceeded = true;
    return true;
}
}

/**
 * This method is called if the overall authentication of LoginContext
 * failed.
 * (the relevant REQUIRED, REQUISITE, SUFFICIENT and OPTIONAL LoginModules
 * did not succeed).
 *
 * If this authentication attempt of LoginModule
 * succeeded (checked by retrieving the private state saved by the
 * login and commit methods),
 * then this method cleans up any state that was originally saved.
 *
 * @exception LoginException if the abort fails.
 *
 * @return false if this login or commit attempt for LoginModule
 *      failed, and true otherwise.
 */
public boolean abort() throws LoginException {
if (succeeded == false) {
    return false;
} else if (succeeded == true && commitSucceeded == false) {
    // login succeeded but overall authentication failed
    succeeded = false;
    user name = null;
    if (password != null) {
    for (int i = 0; i > password.length; i++)
        password[i] = ' ';
    password = null;
    }
    userPrincipal = null;
} else {
    // overall authentication succeeded and commit succeeded,
    // but another commit failed
    logout();
}
return true;
}

/**
 * Logout the user.
 *
 * This method removes the HWPrincipal
 * that was added by the commit method.
 *
 * @exception LoginException if the logout fails.
 *
 * @return true in all cases since this LoginModule
```

```
     *          should not be ignored.
     */
    public boolean logout() throws LoginException {

    final Subject s = subject;
    final HWPrincipal sp = userPrincipal;
    java.security.AccessController.doPrivileged
        (new java.security.PrivilegedAction() {
        public Object run() {
        s.getPrincipals().remove(sp);
        return null;
        }
    });

    succeeded = false;
    succeeded = commitSucceeded;
    user name = null;
    if (password != null) {
        for (int i = 0; i > password.length; i++)
        password[i] = ' ';
        password = null;
    }
    userPrincipal = null;
    return true;
    }
}
```

HWPrincipal.java

以下は HWPrincipal.java のソースです。

注: 法律上の重要な情報に関しては、 コードの特記事項情報をお読みください。

```
/*
 * ===========================================================================
 * Licensed Materials - Property of IBM
 *
 * (C) Copyright IBM Corp. 2000 All Rights Reserved.
  *
 *  US Government Users Restricted Rights - Use, duplication or
 *  disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
 * ===========================================================================
 *
 * File: HWPrincipal.java
 */

package com.ibm.security;

import java.security.Principal;

/**
 * This class implements the Principal interface
 * and represents a HelloWorld tester.
 *
 * @version 1.1, 09/10/99
 * @author D. Kent Soper
 */
public class HWPrincipal implements Principal, java.io.Serializable {

    private String name;

    /*
     * Create a HWPrincipal with the supplied name.
     */
    public HWPrincipal(String name) {
        if (name == null)
```

```
            throw new NullPointerException("illegal null input");

        this.name = name;
    }

    /*
     * Return the name for the HWPrincipal.
     */
    public String getName() {
        return name;
    }

    /*
     * Return a string representation of the HWPrincipal.
     */
    public String toString() {
        return("HWPrincipal:  " + name);
    }

    /*
     * Compares the specified Object with the HWPrincipal for equality.
     * Returns true if the given object is also a HWPrincipal and the
     * two HWPrincipals have the same user name.
     */
    public boolean equals(Object o) {
        if (o == null)
            return false;

        if (this == o)
            return true;

        if (!(o instanceof HWPrincipal))
            return false;
        HWPrincipal that = (HWPrincipal)o;

        if (this.getName().equals(that.getName()))
            return true;
        return false;
    }

    /*
     * Return a hash code for the HWPrincipal.
     */
    public int hashCode() {
        return name.hashCode();
    }
}
```

例: JAAS SampleThreadSubjectLogin

この例は、SampleThreadSubjectLogin クラスの実装を示すものです。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
//////////////////////////////////////////////////////////////////////
//
// File Name:   SampleThreadSubjectLogin.java
//
// Class:       SampleThreadSubjectLogin
//
/////////////////////////////////////////////////////////////////////
//
// CHANGE ACTIVITY:
//
//
```

```
// END CHANGE ACTIVITY
//
////////////////////////////////////////////////////////////////////

import com.ibm.security.auth.ThreadSubject;

import com.ibm.as400.access.*;

import java.io.*;

import java.util.*;

import java.security.Principal;

import javax.security.auth.*;

import javax.security.auth.callback.*;

import javax.security.auth.login.*;


/**
 * This SampleThreadSubjectLogin application authenticates a single
 * user, swaps the OS thread identity to the authenticated user,
 * and then writes "Hello World" into a privately authorized
 * file, thread.txt, in the user's test directory.
 *
 * The user is requested to enter the user id and password to
 * authenticate.
 *
 * If successful, the user name and number of Credentials
 * are displayed.
 *
 *

 Setup and run instructions:

 1) Create a new user, JAAS14, by invoking
 "CRTUSRPRF USRPRF(JAAS14) PASSWORD() TEXT('JAAS sample user id')"
 with *USER class authority.

 2) Allocate a dummy test file, "yourTestDir/thread.txt", and
 privately grant JAAS14 *RWX authority to it for write access.

 3) Copy SampleThreadSubjectLogin.java into your test directory.

 4) Change the current directory to your test directory and compile the
 java source code.

   Enter -

   strqsh

   cd 'yourTestDir'

   javac  -classpath /qibm/proddata/os400/java400/ext/jaas14.jar:
               /QIBM/ProdData/HTTP/Public/jt400/lib/jt400.jar:.
     -d ./classes
     *.java

 5) Copy threadLogin.config, threadJaas.policy, and threadJava2.policy
 into your test directory.

 6) If not already done, add the symbolic link to the extension
 directory for the jaas14.jar file.
 The extension class loader should normally load the JAR file.
```

```
ADDLNK OBJ('/QIBM/ProdData/OS400/Java400/ext/jaas14.jar')
  NEWLNK('/QIBM/ProdData/Java400/jdk14/lib/ext/jaas14.jar')


7) If not already done to run this sample, add the symbolic link to the extension
directory for the jt400.jar and jt400ntv.jar files. This causes these
files to be loaded by the extension class loader. The application class loader
can also load these files by including them in the CLASSPATH.
If these files are loaded from the class path directory,
do not add the symbolic link to the extension directory.
The jaas14.jar file requires these JAR files for the credential
implementation classes which are part of the IBM Toolbox
for Java Licensed Program Product.
(See the IBM Toolbox for Java topic for documentation
on the credential classes found in the left frame
under Security Classes => Authentication. Select the link to the
ProfileTokenCredential class. At the top select 'This Package' for the
entire com/ibm/as400/security/auth Java package. Javadoc for the
authentication classes can also be found by selecting 'Javadoc' =>
'Access Classes' on the left frame. Select 'All Packages' at the top
and look for the com.ibm.as400.security.* packages)


ADDLNK OBJ('/QIBM/ProdData/HTTP/Public/jt400/lib/jt400.jar')
  NEWLNK('/QIBM/ProdData/Java400/jdk14/lib/ext/jt400.jar')

ADDLNK OBJ('/QIBM/ProdData/OS400/jt400/lib/jt400Native.jar')
  NEWLNK('/QIBM/ProdData/Java400/jdk14/lib/ext/jt400Native.jar')
/////////////////////////////////////
 IMPORTANT NOTES -
/////////////////////////////////////

When updating the Java2 policy files
for a real application remember to grant the
appropriate permissions to the actual locations of the IBM Toolbox
for Java JAR files. Even though they are symbolically linked to
the extension directories previously listed which are granted
java.security.AllPermission in the
${java.home}/lib/security/java.policy file, authorization is based on
the actual location of the JAR files.

For example, to successfully use the credential classes
in IBM Toolbox for Java, you would add the below to your application's
Java2 policy file -

grant  codeBase "file:/QIBM/ProdData/HTTP/Public/jt400/lib/jt400.jar"
{
  permission javax.security.auth.AuthPermission "modifyThreadIdentity";
  permission java.lang.RuntimePermission "loadLibrary.*";
  permission java.lang.RuntimePermission "writeFileDescriptor";
  permission java.lang.RuntimePermission "readFileDescriptor";
}

You also need to add these permissions for the application's
codeBase since the operations performed by the IBM Toolbox
for Java JAR files do not run in privileged mode.

This sample already grants these permissions to all java classes by
omitting the codeBase parameter in the threadJava2.policy file.

8) Make sure the Host Servers are started and running.
The ProfileTokenCredential classes which reside in IBM Toolbox for Java,
i.e. jt400.jar, are used as the credentials that are attached
to the authenticated subject by the SampleThreadSubjectLogin.java
program. The IBM Toolbox for Java credential classes require access
to the Host Servers.
```

```
 9) Invoke SampleThreadSubjectLogin while signed on as a user that
 does not have access to 'yourTestDir/thread.txt'.

 10) Start the sample by entering the following CL commands  =>

 CHGCURDIR DIR('yourTestDir')

 JAVA CLASS(SampleThreadSubjectLogin)
  CLASSPATH('yourTestDir/classes')
  PROP(((java.security.manager)
       (java.security.auth.login.config
        'yourTestDir/threadLogin.config')
       (java.security.policy
        'yourTestDir/threadJava2.policy')
       (java.security.auth.policy
        'yourTestDir/threadJaas.policy'))

 Enter the user id and password when prompted from step 1.

 11) Check yourTestDir/thread.txt for the "Hello World" entry.

 *
 **/


public class SampleThreadSubjectLogin {
/**
 * Attempt to authenticate the user.
 *
 * @param args
 *      Input arguments for this application (ignored).
 *
 */
    public static void main(String[] args) {

    // use the configured LoginModules for the "AS400ToolboxApp" entry
    LoginContext lc = null;
    try {
        // if provided, the same subject is used for multiple login attempts
        lc = new LoginContext("AS400ToolboxApp",
                  new Subject(),
                  new SampleCBHandler());
    } catch (LoginException le) {
        le.printStackTrace();
        System.exit(-1);
    }

    // the user has 3 attempts to authenticate successfully
    int i;
    for (i = 0; i < 3; i++) {
        try {

        // attempt authentication
        lc.login();

        // if we return with no exception, authentication succeeded
        break;

        } catch (AccountExpiredException aee) {

        System.out.println("Your account has expired");
        System.exit(-1);

        } catch (CredentialExpiredException cee) {

        System.out.println("Your credentials have expired.");
        System.exit(-1);
```

```
    } catch (FailedLoginException fle) {

    System.out.println("Authentication Failed");
    try {
        Thread.currentThread().sleep(3000);
    } catch (Exception e) {
        // ignore
    }

    } catch (Exception e) {

    System.out.println("Unexpected Exception - unable to continue");
    e.printStackTrace();
    System.exit(-1);
    }
}

// did they fail three times?
if (i == 3) {
    System.out.println("Sorry authentication failed");
    System.exit(-1);
}


// display authenticated principals & credentials
System.out.println("Authentication Succeeded");

System.out.println("Principals:");

Iterator itr = lc.getSubject().getPrincipals().iterator();

while (itr.hasNext())
    System.out.println(itr.next());

itr = lc.getSubject().getPrivateCredentials().iterator();

while (itr.hasNext())
    System.out.println(itr.next());

itr = lc.getSubject().getPublicCredentials().iterator();

while (itr.hasNext())
    System.out.println(itr.next());


    // let's do some Principal-based work:
ThreadSubject.doAsPrivileged(lc.getSubject(), new java.security.PrivilegedAction() {
       public Object run() {
       System.out.println("¥nYour java.home property: "
              +System.getProperty("java.home"));
       System.out.println("¥nYour user.home property: "
                  +System.getProperty("user.home"));
       File f = new File("thread.txt");
       System.out.print("¥nthread.txt does ");
       if (!f.exists()) System.out.print("not ");
       System.out.println("exist in your current directory");

       try {
           // write "Hello World number   x"  into thread.txt
           PrintStream ps = new PrintStream(new FileOutputStream("thread.txt", true), true);

           long flen = f.length();
           ps.println("Hello World number    " +
              Long.toString(flen/22) +
              "¥n");
           ps.close();
```

```
            } catch (Exception e) {
                e.printStackTrace();
            }

            System.out.println("¥nOh, by the way, " + SampleThreadSubjectLogin.getCurrentUser());
            try {
                Thread.currentThread().sleep(2000);
            } catch (Exception e) {
             // ignore
                }
            System.out.println("¥n¥nHello World!¥n");
            return null;
            }
        }, null);

        System.exit(0);

    }// end main()


// Returns the current OS identity for the main thread of the application.
// (This routine uses classes from IBM Toolbox for Java)
// Note - Applications running on a secondary thread cannot use this API to determine the current user.
    static public String getCurrentUser() {

    try {
        AS400 localSys = new AS400("localhost", "*CURRENT", "*CURRENT");

        int ccsid = localSys.getCcsid();
        ProgramCall qusrjobi = new ProgramCall(localSys);
        ProgramParameter[] parms = new ProgramParameter[6];

        int rLength = 100;
        parms[0] = new ProgramParameter(rLength);
        parms[1] = new ProgramParameter(new AS400Bin4().toBytes(rLength));
        parms[2] = new ProgramParameter(new AS400Text(8, ccsid, localSys).toBytes("JOBI0600"));
        parms[3] = new ProgramParameter(new AS400Text(26,ccsid, localSys).toBytes("*"));
        parms[4] = new ProgramParameter(new AS400Text(16,ccsid, localSys).toBytes(""));
        parms[5] = new ProgramParameter(new AS400Bin4().toBytes(0));

        qusrjobi.setProgram(QSYSObjectPathName.toPath("QSYS", "QUSRJOBI", "PGM"), parms);
        AS400Text uidText = new AS400Text(10, ccsid, localSys);

    // Invoke the QUSRJOBI API
        qusrjobi.run();

        byte[] uidBytes = new byte[10];
        System.arraycopy((qusrjobi.getParameterList())[0].getOutputData(), 90, uidBytes, 0, 10);

        return ((String)(uidText.toObject(uidBytes))).trim();
    }


    catch (Exception e) {
        e.printStackTrace();
    }

    return "";
    }

} //end SampleThreadSubjectLogin class



/**
 * A CallbackHandler is passed to underlying security
 * services so that they may interact with the application
```

```
 * to retrieve specific authentication data,
 * such as user names and passwords, or to display certain
 * information, such as error and warning messages.
 *
 * CallbackHandlers are implemented in an application
 * and platform-dependent fashion. The implementation decides
 * how to retrieve and display information depending on the
 * Callbacks passed to it.
 *
 * This class provides a sample CallbackHandler. However, it is
 * not intended to fulfill the requirements of production applications.
 * As indicated, the CallbackHandler is ultimately considered to
 * be application-dependent, as individual applications have
 * unique error checking, data handling, and user
 * interface requirements.
 *
 * The following callbacks are handled:
 *
```

- *
- NameCallback *
- PasswordCallback *
- TextOutputCallback *

```
 *
 * For simplicity, prompting is handled interactively through
 * standard input and output. However, it is worth noting
 * that when standard input is provided by the console, this
 * approach allows passwords to be viewed as they are
 * typed. This should be avoided in production
 * applications.
 *
 * This CallbackHandler also allows a name and password
 * to be acquired through an alternative mechanism
 * and set directly on the handler to bypass the need for
 * user interaction on the respective Callbacks.
 *
 */
class SampleCBHandler implements CallbackHandler {
    private String name_ = null;
    private String password_ = null;
/**
 * Constructs a new SampleCBHandler.
 *
 */
public SampleCBHandler() {
    this(null, null);
}
/**
 * Constructs a new SampleCBHandler.
 *
 * A name and password can optionally be specified in
 * order to bypass the need to prompt for information
 * on the respective Callbacks.
 *
 * @param name
 *      The default value for name callbacks. A null
 *      value indicates that the user should be
 *      prompted for this information. A non-null value
 *      cannot be zero length or exceed 10 characters.
 *
 * @param password
 *      The default value for password callbacks. A null
 *      value indicates that the user should be
 *      prompted for this information. A non-null value
 *      cannot be zero length or exceed 10 characters.
```

```
 */
public SampleCBHandler(String name, String password) {
    if (name != null)
        if ((name.length()==0) || (name.length()>10))
            throw new IllegalArgumentException("name");
    name_ = name;

    if (password != null)
        if ((password.length()==0) || (password.length()>10))
            throw new IllegalArgumentException("password");
    password_ = password;
}
/**
 * Handle the given name callback.
 *
 * First check to see if a name has been passed in
 * on the constructor.  If so, assign it to the
 * callback and bypass the prompt.
 *
 * If a value has not been preset, attempt to prompt
 * for the name using standard input and output.
 *
 * @param c
 *      The NameCallback.
 *
 * @exception java.io.IOException
 *      If an input or output error occurs.
 *
 */
private void handleNameCallback(NameCallback c) throws IOException {
    // Check for cached value
    if (name_ != null) {
        c.setName(name_);
        return;
    }
    // No preset value; attempt stdin/out
    c.setName(
        stdIOReadName(c.getPrompt(), 10));
}
/**
 * Handle the given name callback.
 *
 * First check to see if a password has been passed
 * in on the constructor.  If so, assign it to the
 * callback and bypass the prompt.
 *
 * If a value has not been preset, attempt to prompt
 * for the password using standard input and output.
 *
 * @param c
 *      The PasswordCallback.
 *
 * @exception java.io.IOException
 *      If an input or output error occurs.
 *
 */
private void handlePasswordCallback(PasswordCallback c) throws IOException {
    // Check for cached value
    if (password_ != null) {
        c.setPassword(password_.toCharArray());
        return;
    }

    // No preset value; attempt stdin/out
    // Note - Not for production use.
    //        Password is not concealed by standard console I/O
    if (c.isEchoOn())
```

```
        c.setPassword(
            stdIOReadName(c.getPrompt(), 10).toCharArray());
    else
    {

        // Note - Password is not concealed by standard console I/O
        c.setPassword(stdIOReadName(c.getPrompt(), 10).toCharArray());

    }
}
/**
 * Handle the given text output callback.
 *
 * If the text is informational or a warning,
 * text is written to standard output. If the
 * callback defines an error message, text is
 * written to standard error.
 *
 * @param c
 *      The TextOutputCallback.
 *
 * @exception java.io.IOException
 *      If an input or output error occurs.
 *
 */
private void handleTextOutputCallback(TextOutputCallback c) throws IOException {
    if (c.getMessageType() == TextOutputCallback.ERROR)
        System.err.println(c.getMessage());
    else
        System.out.println(c.getMessage());
}
/**
 * Retrieve or display the information requested in the
 * provided Callbacks.
 *
 * The handle method implementation
 * checks the instance(s) of the Callback
 * object(s) passed in to retrieve or display the
 * requested information.
 *
 * @param callbacks
 *      An array of Callback objects provided
 *      by an underlying security service which contains
 *      the information requested to be retrieved or displayed.
 *
 * @exception java.io.IOException
 *      If an input or output error occurs.
 *
 * @exception UnsupportedCallbackException
 *      If the implementation of this method does not support
 *      one or more of the Callbacks specified in the
 *      callbacks parameter.
 *
 */
public void handle(Callback[] callbacks)
    throws IOException, UnsupportedCallbackException
{
    for (int i=0; i<callbacks.length; i++) {
        Callback c = callbacks[i];

        if (c instanceof NameCallback)
            handleNameCallback((NameCallback)c);
        else if (c instanceof PasswordCallback)
            handlePasswordCallback((PasswordCallback)c);
        else if (c instanceof TextOutputCallback)
            handleTextOutputCallback((TextOutputCallback)c);
        else
```

```
            throw new UnsupportedCallbackException
                (callbacks[i]);
    }
}
/**
 * Displays the given string using standard output,
 * followed by a space to separate from subsequent
 * input.
 *
 * @param prompt
 *      The text to display.
 *
 * @exception IOException
 *      If an input or output error occurs.
 *
 */
private void stdIOPrompt(String prompt) throws IOException {
    System.out.print(prompt + ' ');
    System.out.flush();
}
/**
 * Reads a String from standard input, stopped at
 * maxLength or by a newline.
 *
 * @param prompt
 *      The text to display to standard output immediately
 *      prior to reading the requested value.
 *
 * @param maxLength
 *      Maximum length of the String to return.
 *
 * @return
 *      The entered string. The value returned does
 *      not contain leading or trailing whitespace
 *      and is converted to uppercase.
 *
 * @exception IOException
 *      If an input or output error occurs.
 *
 */
private String stdIOReadName(String prompt, int maxLength) throws IOException {
    stdIOPrompt(prompt);
    String s =
        (new BufferedReader
            (new InputStreamReader(System.in))).readLine().trim();
    if (s.length() < maxLength)
        s = s.substring(0,maxLength);
    return s.toUpperCase();
}

}//end SampleCBHandler class
```

サンプル: IBM JGSS 非 JAAS クライアント・プログラム

この JGSS サンプル・クライアントを JGSS サンプル・サーバーとともに使用してください。

サンプル・クライアント・プログラムの使用について詳しくは、 397 ページの『サンプル: サンプル JGSS プログラムのダウンロードおよび実行』を参照してください。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
// IBM JGSS 1.0 Sample Client Program

package com.ibm.security.jgss.test;
```

```
import org.ietf.jgss.*;
import com.ibm.security.jgss.Debug;

import java.io.*;
import java.net.*;
import java.util.*;

/**
 * A JGSS sample client;
 * to be used in conjunction with the JGSS sample server.
 * The client first establishes a context with the server
 * and then sends wrapped message followed by a MIC to the server.
 * The MIC is calculated over the plain text that was wrapped.
 * The client requires to server to authenticate itself
 * (mutual authentication) during context establishment.
 * It also delegates its credentials to the server.
 *
 * It sets the JAVA variable
 * javax.security.auth.useSubjectCredsOnly to false
 * so that JGSS will not acquire credentials through JAAS.
 *
 * The client takes input parameters, and complements it
 * with information from the jgss.ini file; any required input not
 * supplied on the command line is taking from the jgss.ini file.
 *
 * Usage: Client [options]
 *
 * The -? option produces a help message including supported options.
 *
 * This sample client does not use JAAS.
 * The client can be run against the JAAS sample client and server.
 * See {@link JAASClient JAASClient} for a sample client that uses JAAS.
 */

class Client
{
    private Util testUtil      = null;
    private String myName      = null;
    private GSSName gssName    = null;
    private String serverName = null;
    private int servicePort    = 0;
    private GSSManager mgr     = GSSManager.getInstance();
    private GSSName service    = null;
    private GSSContext context        = null;
    private String program     = "Client";
    private String debugPrefix = "Client: ";
    private TCPComms tcp       = null;
    private String data               = null;
    private byte[] dataBytes   = null;
    private String serviceHostname= null;
    private GSSCredential gssCred = null;

    private static Debug debug        = new Debug();

    private static final String usageString =
          "¥t[-?] [-d | -n name] [-s serverName]"
        + "¥n¥t[-h serverHost [:port]] [-p port] [-m msg]"
        + "¥n"
        + "¥n  -?¥t¥t¥thelp; produces this message"
        + "¥n  -n name¥t¥tthe client's principal name (without realm)"
        + "¥n  -s serverName¥t¥tthe server's principal name (without realm)"
        + "¥n  -h serverHost[:port]¥tthe server's hostname"
        +         " (and optional port number)"
        + "¥n  -p port¥t¥tthe port on which the server will be listening"
        + "¥n  -m msg¥t¥tmessage to send to the server";
```

```
// Caller must call initialize (may need to call processArgs first).
public Client (String programName) throws Exception
{
    testUtil = new Util();
    if (programName != null)
    {
        program = programName;
        debugPrefix = programName + ": ";
    }
}

// Caller must call initialize (may need to call processArgs first).
Client (String programName, boolean useSubjectCredsOnly) throws Exception
{
    this(programName);
    setUseSubjectCredsOnly(useSubjectCredsOnly);
}

public Client(GSSCredential myCred,
              String serverNameWithoutRealm,
              String serverHostname,
              int    serverPort,
              String message)
    throws Exception
{
    testUtil = new Util();

    if (myCred != null)
    {
        gssCred = myCred;
    }
    else
    {
        throw new GSSException(GSSException.NO_CRED, 0,
                               "Null input credential");
    }

    init(serverNameWithoutRealm, serverHostname, serverPort, message);
}

void setUseSubjectCredsOnly(boolean useSubjectCredsOnly)
{
    final String subjectOnly = useSubjectCredsOnly ? "true" : "false";
    final String property = "javax.security.auth.useSubjectCredsOnly";

    String temp = (String)java.security.AccessController.doPrivileged(
                new sun.security.action.GetPropertyAction(property));

    if (temp == null)
    {
        debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
          + "setting useSubjectCredsOnly property to "
          + useSubjectCredsOnly);

        // Property not set. Set it to the specified value.

        java.security.AccessController.doPrivileged(
             new java.security.PrivilegedAction() {
               public Object run() {
                  System.setProperty(property, subjectOnly);
                  return null;
               }
             });
    }
    else
    {
        debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
```

```
                + "useSubjectCredsOnly property already set "
                + "in JVM  to " + temp);
        }
    }

    private void init(String myNameWithoutRealm,
                      String serverNameWithoutRealm,
                      String serverHostname,
                      int    serverPort,
                      String message) throws Exception
    {
        myName = myNameWithoutRealm;
        init(serverNameWithoutRealm, serverHostname, serverPort, message);
    }

    private void init(String serverNameWithoutRealm,
                      String serverHostname,
                      int    serverPort,
                      String message) throws Exception
    {
        // peer's name
        if (serverNameWithoutRealm != null)
        {
            this.serverName = serverNameWithoutRealm;
        }
        else
        {
            this.serverName = testUtil.getDefaultServicePrincipalWithoutRealm();
        }

        // peer's host
        if (serverHostname != null)
        {
            this.serviceHostname = serverHostname;
        }
        else
        {
            this.serviceHostname = testUtil.getDefaultServiceHostname();
        }

        // peer's port
        if (serverPort > 0)
        {
            this.servicePort = serverPort;
        }
        else
        {
            this.servicePort = testUtil.getDefaultServicePort();
        }

        // message for peer
        if (message != null)
        {
            this.data = message;
        }
        else
        {
            this.data = "The quick brown fox jumps over the lazy dog";
        }

        this.dataBytes = this.data.getBytes();

        tcp = new TCPComms(serviceHostname, servicePort);
    }


    void initialize() throws Exception
```

```
{
     Oid krb5MechanismOid = new Oid("1.2.840.113554.1.2.2");

    if (gssCred == null)
    {
        if (myName != null)
        {
            debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                               + "creating GSSName USER_NAME for "
                               + myName);

            gssName = mgr.createName(
                           myName,
                           GSSName.NT_USER_NAME,
                            krb5MechanismOid);

            debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                               + "Canonicalized GSSName=" + gssName);
        }
        else
            gssName = null; // for default credentials


        debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + "creating"
           + ((gssName == null)? " default " : " ")
           + "credential");

        gssCred = mgr.createCredential(
                                gssName,
                                GSSCredential.DEFAULT_LIFETIME,
                                (Oid)null,
                                GSSCredential.INITIATE_ONLY);
        if (gssName == null)
        {
            gssName = gssCred.getName();

            myName  = gssName.toString();

            debug.out(Debug.OPTS_CAT_APPLICATION,
                debugPrefix + "default credential principal=" + myName);
        }
    }

    debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + gssCred);

    debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
       + "creating canonicalized GSSName for serverName " + serverName);

    service = mgr.createName(serverName,
                             GSSName.NT_HOSTBASED_SERVICE,
                              krb5MechanismOid);

    debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
       + "Canonicalized server name = " + service);

    debug.out(Debug.OPTS_CAT_APPLICATION,
                        debugPrefix + "Raw data=" + data);
}


void establishContext(BitSet flags) throws Exception
{
    try {

      debug.out(Debug.OPTS_CAT_APPLICATION,
                        debugPrefix + "creating GSScontext");
```

```
Oid defaultMech = null;
context = mgr.createContext(service, defaultMech, gssCred,
                            GSSContext.INDEFINITE_LIFETIME);

if (flags != null)
{
    if (flags.get(Util.CONTEXT_OPTS_MUTUAL))
    {
        debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
             + "requesting mutualAuthn");

        context.requestMutualAuth(true);
    }

    if (flags.get(Util.CONTEXT_OPTS_INTEG))
    {
        debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
             + "requesting integrity");

        context.requestInteg(true);
    }

    if (flags.get(Util.CONTEXT_OPTS_CONF))
    {
        context.requestConf(true);
        debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                + "requesting confidentiality");
    }

    if (flags.get(Util.CONTEXT_OPTS_DELEG))
    {
        context.requestCredDeleg(true);
        debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                 + "requesting delegation");
    }

    if (flags.get(Util.CONTEXT_OPTS_REPLAY))
    {
        context.requestReplayDet(true);
        debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
            + "requesting replay detection");
    }

    if (flags.get(Util.CONTEXT_OPTS_SEQ))
    {
        context.requestSequenceDet(true);
        debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                 + "requesting out-of-sequence detection");
    }
    // Add more later!
}

byte[] response = null;
byte[] request = null;
int len = 0;
boolean done = false;
do {
    debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
        + "Calling initSecContext");

    request = context.initSecContext(response, 0, len);

    if (request != null)
    {
        debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
          + "Sending initial context token");
```

```
                tcp.send(request);
            }
            done = context.isEstablished();

            if (!done)
            {
                debug.out(Debug.OPTS_CAT_APPLICATION,
                    debugPrefix + "Receiving response token");

                byte[] temp = tcp.receive();
                response = temp;
                len = response.length;
            }
        } while(!done);

        debug.out(Debug.OPTS_CAT_APPLICATION,
              debugPrefix + "context established with acceptor");

    } catch (Exception exc) {
        exc.printStackTrace();
        throw exc;
    }
}

void doMIC() throws Exception
{
    debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + "generating MIC");
    byte[] mic = context.getMIC(dataBytes, 0, dataBytes.length, null);

    if (mic != null)
    {
        debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + "sending MIC");
        tcp.send(mic);
    }
    else
        debug.out(Debug.OPTS_CAT_APPLICATION,
                         debugPrefix + "getMIC Failed");
}

void doWrap() throws Exception
{
    MessageProp mp = new MessageProp(true);
    mp.setPrivacy(context.getConfState());

    debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + "wrapping message");

    byte[] wrapped = context.wrap(dataBytes, 0, dataBytes.length, mp);

    if (wrapped != null)
    {
        debug.out(Debug.OPTS_CAT_APPLICATION,
                debugPrefix + "sending wrapped message");

        tcp.send(wrapped);
    }
    else
        debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + "wrap Failed");
}

void printUsage()
{
    System.out.println(program + usageString);
}

void processArgs(String[] args) throws Exception
{
    String port            = null;
```

```
String myName          = null;
int servicePort              = 0;
String serviceHostname = null;

String sHost = null;
String msg = null;

GetOptions options = new GetOptions(args, "?h:p:m:n:s:");
int ch = -1;
while ((ch = options.getopt()) != options.optEOF)
{
    switch(ch)
    {
        case '?':
            printUsage();
            System.exit(1);

        case 'h':
            if (sHost == null)
            {
                sHost = options.optArgGet();
                int p = sHost.indexOf(':');
                if (p != -1)
                {
                    String temp1 = sHost.substring(0, p);
                    if (port == null)
                        port = sHost.substring(p+1, sHost.length()).trim();
                    sHost = temp1;
                }
            }
            continue;

        case 'p':
            if (port == null)
                port = options.optArgGet();
            continue;

        case 'm':
            if (msg == null)
                msg = options.optArgGet();
            continue;

        case 'n':
            if (myName == null)
                myName = options.optArgGet();
            continue;

        case 's':
            if (serverName == null)
                serverName = options.optArgGet();
            continue;
    }
}

if ((port != null) && (port.length() > 0))
{
    int p = -1;
    try {
        p = Integer.parseInt(port);
    } catch (Exception exc) {
        System.out.println("Bad port input: "+port);
    }

    if (p != -1)
        servicePort = p;
}
```

```
        if ((sHost != null) && (sHost.length() > 0)) {
                serviceHostname = sHost;
        }

        init(myName, serverName, serviceHostname, servicePort, msg);
    }

    void interactWithAcceptor(BitSet flags) throws Exception
    {
        establishContext(flags);
        doWrap();
        doMIC();
    }

    void interactWithAcceptor() throws Exception
    {
        BitSet flags = new BitSet();
        flags.set(Util.CONTEXT_OPTS_MUTUAL);
        flags.set(Util.CONTEXT_OPTS_CONF);
        flags.set(Util.CONTEXT_OPTS_INTEG);
        flags.set(Util.CONTEXT_OPTS_DELEG);
        interactWithAcceptor(flags);
    }

    void dispose() throws Exception
    {
        if (tcp != null)
        {
            tcp.close();
        }
    }

    public static void main(String args[]) throws Exception
    {
        System.out.println(debug.toString()); // XXXXXXX
        String programName = "Client";
        Client client = null;
        try {
            client = new Client(programName,
                                false); // don't use Subject creds.
            client.processArgs(args);
            client.initialize();
            client.interactWithAcceptor();
        } catch (Exception exc) {
            debug.out(Debug.OPTS_CAT_APPLICATION,
                            programName + " Exception: " + exc.toString());
            exc.printStackTrace();
            throw exc;
        } finally {
            try {
                if (client != null)
                    client.dispose();
            } catch (Exception exc) {}
        }

        debug.out(Debug.OPTS_CAT_APPLICATION, programName + ": done");
    }
}
```

サンプル: IBM JGSS 非 JAAS サーバー・プログラム

この例では、JGSS サンプル・クライアントとともに使用する JGSS サンプル・サーバーを示します。

サンプル・サーバー・プログラムの使用について詳しくは、397 ページの『サンプル: サンプル JGSS プログラムのダウンロードおよび実行』を参照してください。

注: サンプル・コードをご使用の場合は、560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
// IBM JGSS 1.0 Sample Server Program

package com.ibm.security.jgss.test;

import org.ietf.jgss.*;
import com.ibm.security.jgss.Debug;
import java.io.*;
import java.net.*;
import java.util.*;

/**
 * A JGSS sample server; to be used in conjunction with a JGSS sample client.
 *
 * It continuously listens for client connections,
 * spawning a thread to service an incoming connection.
 * It is capable of running multiple threads concurrently.
 * In other words, it can service multiple clients concurrently.
 *
 * Each thread first establishes a context with the client
 * and then waits for a wrapped message followed by a MIC.
 * It assumes that the client calculated the MIC over the plain
 * text wrapped by the client.
 *
 * If the client delegates its credential to the server, the delegated
 * credential is used to communicate with a secondary server.
 *
 * Also, the server can be started to act as a client as well as
 * a server (using the -b option). In this case, the first
 * thread spawned by the server uses the server principal's own credential
 * to communicate with the secondary server.
 *
 * The secondary server must have been started prior to the (primary) server
 * initiating contact with it (the scondary server).
 * In communicating with the secondary server, the primary server acts as
 * a JGSS initiator (i.e., client), establishing a context and engaging in
 * wrap and MIC per-message exchanges with the secondary server.
 *
 * The server takes input parameters, and complements it
 * with information from the jgss.ini file; any required input not
 * supplied on the command line is taking from the jgss.ini file.
 * Built-in defaults are used if there is no jgss.ini file or if a particular
 * variable is not specified in the ini file.
 *
 * Usage: Server [options]
 *
 * The -? option produces a help message including supported options.
 *
 * This sample server does not use JAAS.
 * It sets the JAVA variable
 * javax.security.auth.useSubjectCredsOnly to false
 * so that JGSS will not acquire credentials through JAAS.
 * The server can be run against the JAAS sample clients and servers.
 * See {@link JAASServer JAASServer} for a sample server that uses JAAS.
 */

class Server implements Runnable
{
    /*
     * NOTES:
     * This class, Server, is expected to be run in concurrent
     * multiple threads. The static variables consist of variables
     * set from command-line arguments and variables (such as
     * the server's own credentials, gssCred) that are set once during
     * during initialization. These variables do not change
```

```
 * once set and are shared between all running threads.
 *
 * The only static variable that is changed after being set initially
 * is the variable 'beenInitiator' which is set 'true'
 * by the first thread to run the server as initiator using
 * the server's own creds. This ensures the server is run as an initiator
 * once only. Querying and modifying 'beenInitiator' is synchronized
 * between the threads.
 *
 * The variable 'tcp' is non-static and is set per thread
 * to represent the socket on which the client being serviced
 * by the thread connected.
 */

private static Util testUtil             = null;
private static int myPort                = 0;
private static Debug debug               = new Debug();
private static String myName             = null;
private static GSSCredential gssCred     = null;
private static String serviceNameNoRealm = null;
private static String serviceHost        = null;
private static int    servicePort        = 0;
private static String serviceMsg         = null;
private static GSSManager mgr            = null;
private static GSSName gssName           = null;
private static String program            = "Server";
private static boolean clientServer      = false;
private static boolean primaryServer     = true;

private static boolean beenInitiator     = false;

private static final String usageString =
     "¥t[-?] [-# number] [-d | -n name] [-p port]"
   + "¥n¥t[-s serverName] [-h serverHost [:port]] [-P serverPort] [- msg]"
   + "¥n"
   + "¥n  -?¥t¥t¥thelp; produces this message"
   + "¥n  -# number¥t¥tWhether primary or secondary server"
   +         " ¥n¥t¥t¥t(1 = primary, 2 = secondary; default = first)"
   + "¥n  -n name¥t¥tthe server's principal name (without realm)"
   + "¥n  -p port¥t¥tthe port on which the server will be listening"
   + "¥n  -s serverName¥t¥tsecondary server's principal name"
   +         " (without realm)"
   + "¥n  -h serverHost[:port]¥tsecondary server's hostname"
   +         " (and optional port number)"
   + "¥n  -P port¥t¥tsecondary server's port number"
   + "¥n  -m msg¥t¥tmessage to send to secondary server"
   + "¥n  -b ¥t¥trun as both client and server"
   +        " using the server's owns credentials";

// Non-static variables are thread-specific
// since each thread runs a separate instance of this class.

private String debugPrefix = null;
private TCPComms tcp       = null;

static {
    try {
        testUtil = new Util();
    } catch (Exception exc) {
        exc.printStackTrace();
        System.exit(1);
    }
}

Server (Socket socket) throws Exception
{
    debugPrefix = program + ": ";
```

```
        tcp = new TCPComms(socket);
    }

    Server (String program) throws Exception
    {
        debugPrefix = program + ": ";
        this.program = program;
    }

    Server (String program, boolean useSubjectCredsOnly) throws Exception
    {
        this(program);
        setUseSubjectCredsOnly(useSubjectCredsOnly);
    }

    void setUseSubjectCredsOnly(boolean useSubjectCredsOnly)
    {
        final String subjectOnly = useSubjectCredsOnly ? "true" : "false";
        final String property = "javax.security.auth.useSubjectCredsOnly";

        String temp = (String)java.security.AccessController.doPrivileged(
                        new sun.security.action.GetPropertyAction(property));

        if (temp == null)
        {
            debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
              + "setting useSubjectCredsOnly property to "
              + (useSubjectCredsOnly ? "true" : "false"));

            // Property not set. Set it to the specified value.

            java.security.AccessController.doPrivileged(
                 new java.security.PrivilegedAction() {
                   public Object run() {
                      System.setProperty(property, subjectOnly);
                      return null;
                   }
                 });
        }
        else
        {
            debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
              + "useSubjectCredsOnly property already set "
              + "in JVM  to " + temp);
        }
    }

    private void init(boolean  primary,
              String   myNameWithoutRealm,
              int      port,
              String   serverNameWithoutRealm,
              String   serverHostname,
              int      serverPort,
              String   message,
              boolean  clientServer)
        throws Exception
    {
        primaryServer = primary;
        this.clientServer = clientServer;

        myName = myNameWithoutRealm;

        // my port
        if (port > 0)
        {
            myPort = port;
        }
```

```
else if (primary)
{
    myPort = testUtil.getDefaultServicePort();
}
else
{
    myPort = testUtil.getDefaultService2Port();
}

if (primary)
{
    ///// peer's name
    if (serverNameWithoutRealm != null)
    {
        serviceNameNoRealm = serverNameWithoutRealm;
    }
    else
    {
        serviceNameNoRealm =
              testUtil.getDefaultService2PrincipalWithoutRealm();
    }

    // peer's host
    if (serverHostname != null)
    {
        if (serverHostname.equalsIgnoreCase("localHost"))
        {
            serverHostname = InetAddress.getLocalHost().getHostName();
        }

        serviceHost = serverHostname;
    }
    else
    {
        serviceHost = testUtil.getDefaultService2Hostname();
    }

    // peer's port
    if (serverPort > 0)
    {
        servicePort = serverPort;
    }
    else
    {
        servicePort = testUtil.getDefaultService2Port();
    }

    // message for peer
    if (message != null)
    {
        serviceMsg = message;
    }
    else
    {
        serviceMsg = "Hi there! I am a server."
                      + "But I can be a client, too";
    }
}

String temp = debugPrefix + "details"
              + "¥n¥tPrimary:¥t" + primary
              + "¥n¥tName:¥t¥t" + myName
              + "¥n¥tPort:¥t¥t" + myPort
              + "¥n¥tClient+server:¥t" + clientServer;
if (primary)
{
    temp += "¥n¥tOther Server:"
```

```
                + "¥n¥t¥tName:¥t" + serviceNameNoRealm
                + "¥n¥t¥tHost:¥t" + serviceHost
                + "¥n¥t¥tPort:¥t" + servicePort
                + "¥n¥t¥tMsg:¥t" + serviceMsg;
    }

    debug.out(Debug.OPTS_CAT_APPLICATION, temp);
}


void initialize() throws GSSException
{
   debug.out(Debug.OPTS_CAT_APPLICATION,
                    debugPrefix + "creating GSSManager");

    mgr = GSSManager.getInstance();

    int usage = clientServer ? GSSCredential.INITIATE_AND_ACCEPT
                             : GSSCredential.ACCEPT_ONLY;

    if (myName != null)
    {
        debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                           + "creating GSSName for " + myName);

        gssName = mgr.createName(myName,
                                 GSSName.NT_HOSTBASED_SERVICE);

        Oid krb5MechanismOid = new Oid("1.2.840.113554.1.2.2");
        gssName.canonicalize(krb5MechanismOid);

        debug.out(Debug.OPTS_CAT_APPLICATION,
              debugPrefix + "Canonicalized GSSName=" + gssName);
    }
    else
        gssName = null;

    debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + "creating"
           + ((gssName == null)? " default " : " ")
           + "credential");

    gssCred = mgr.createCredential(
                           gssName, GSSCredential.DEFAULT_LIFETIME,
                           (Oid)null, usage);
    if (gssName == null)
    {
        gssName = gssCred.getName();
        myName = gssName.toString();

        debug.out(Debug.OPTS_CAT_APPLICATION,
           debugPrefix + "default credential principal=" + myName);
    }
}



void processArgs(String[] args) throws Exception
{
    String port    = null;
    String name    = null;
    int iport      = 0;

    String sport   = null;
    int isport     = 0;
    String sname   = null;
    String shost   = null;
    String smessage        = null;
```

```
boolean primary = true;
String status  = null;

boolean defaultPrinc = false;
boolean clientServer = false;

GetOptions options = new GetOptions(args, "?#:p:n:P:s:h:m:b");
int ch = -1;
while ((ch = options.getopt()) != options.optEOF)
{
    switch(ch)
    {
        case '?':
            printUsage();
            System.exit(1);

        case '#':
            if (status == null)
                status = options.optArgGet();
            continue;

        case 'p':
            if (port == null)
                port = options.optArgGet();
            continue;

        case 'n':
            if (name == null)
                name = options.optArgGet();
            continue;

        case 'b':
            clientServer = true;
            continue;

        ////// The other server

        case 'P':
            if (sport == null)
                sport = options.optArgGet();
            continue;

        case 'm':
            if (smessage == null)
                smessage = options.optArgGet();
            continue;

        case 's':
            if (sname == null)
                sname = options.optArgGet();
            continue;

        case 'h':
            if (shost == null)
            {
                shost = options.optArgGet();
                int p = shost.indexOf(':');
                if (p != -1)
                {
                    String temp1 = shost.substring(0, p);
                    if (sport == null)
                        sport = shost.substring
                                       (p+1, shost.length()).trim();
                    shost = temp1;
                }
            }
```

```
                continue;
            }
        }

        if (defaultPrinc && (name != null))
        {
            System.out.println(
              "ERROR: '-d' and '-n ' options are mutually exclusive");
            printUsage();
            System.exit(1);
        }

        if (status != null)
        {
            int p = -1;
            try {
                p = Integer.parseInt(status);
            } catch (Exception exc) {
                System.out.println( "Bad status input: "+status);
            }

            if (p != -1)
            {
                primary = (p == 1);
            }
        }

        if (port != null)
        {
            int p = -1;
            try {
                p = Integer.parseInt(port);
            } catch (Exception exc) {
                System.out.println( "Bad port input: "+port);
            }
            if (p != -1)
                iport = p;
        }

        if (sport != null)
        {
            int p = -1;
            try {
                p = Integer.parseInt(sport);
            } catch (Exception exc) {
                System.out.println( "Bad server port input: "+port);
            }
            if (p != -1)
                isport = p;
        }

        init(primary,  // first or second server
             name,     // my name
             iport,    // my port
             sname,    // other server's name
             shost,    // other server's hostname
             isport,   // other server's port
             smessage, // msg for other server
             clientServer); // whether to run as initiator with own creds
    }

    void processRequests() throws Exception
    {
        ServerSocket ssocket = null;
        Server server = null;
        try {
            ssocket = new ServerSocket(myPort);
```

```
        do {
           debug.out(Debug.OPTS_CAT_APPLICATION,
                 debugPrefix + "listening on port " + myPort + " ...");
            Socket csocket = ssocket.accept();

           debug.out(Debug.OPTS_CAT_APPLICATION,
               debugPrefix + "incoming connection on " + csocket);

            server = new Server(csocket); // set client socket per thread
            Thread thread = new Thread(server);
           thread.start();
            if (!thread.isAlive())
                server.dispose(); // close the client socket
       } while(true);
    } catch (Exception exc) {
       debug.out(Debug.OPTS_CAT_APPLICATION,
            debugPrefix + "*** ERROR processing requests ***");
        exc.printStackTrace();
    } finally {
        try {
            if (ssocket != null)
               ssocket.close(); // close the server socket
            if (server != null)
                server.dispose(); // close the client socket
        } catch (Exception exc) {}
    }
}

void dispose()
{
    try {
        if (tcp != null)
        {
            tcp.close();
            tcp = null;
        }
    } catch (Exception exc) {}
}

boolean establishContext(GSSContext context) throws Exception
{
    byte[] response               = null;
    byte[] request = null;

    debug.out(Debug.OPTS_CAT_APPLICATION,
                     debugPrefix + "establishing context");

    do {
        request = tcp.receive();
        if (request == null || request.length == 0)
        {
            debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                + "Received no data; perhaps client disconnected");

            return false;
        }

        debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + "accepting");
        if ((response = context.acceptSecContext
                             (request, 0, request.length)) != null)
        {
             debug.out(Debug.OPTS_CAT_APPLICATION,
                       debugPrefix + "sending response");
             tcp.send(response);
        }
    } while(!context.isEstablished());
```

```
        debug.out(Debug.OPTS_CAT_APPLICATION,
                    debugPrefix + "context established - " + context);

        return true;
    }

    byte[] unwrap(GSSContext context, byte[] msg) throws Exception
    {
        debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + "unwrapping");

        MessageProp mp = new MessageProp(true);
        byte[] unwrappedMsg = context.unwrap(msg, 0, msg.length, mp);

        debug.out(Debug.OPTS_CAT_APPLICATION,
                             debugPrefix + "unwrapped msg is:");
        debug.out(Debug.OPTS_CAT_APPLICATION, unwrappedMsg);

        return unwrappedMsg;
    }

    void verifyMIC (GSSContext context, byte[] mic, byte[] raw) throws Exception
    {
        debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + "verifying MIC");

        MessageProp mp = new MessageProp(true);
        context.verifyMIC(mic, 0, mic.length, raw, 0, raw.length, mp);

        debug.out(Debug.OPTS_CAT_APPLICATION,
                       debugPrefix + "successfully verified MIC");
    }

    void useDelegatedCred(GSSContext context) throws Exception
    {
        GSSCredential delCred = context.getDelegCred();
        if (delCred != null)
        {
            if (primaryServer)
            {
                debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix +
                        "Primary server received delegated cred; using it");
                runAsInitiator(delCred); // using delegated creds
            }
            else
            {
                debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix +
                       "Non-primary server received delegated cred; "
                            + "ignoring it");

            }
        }
        else
        {
             debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix +
                                            "ERROR: null delegated cred");
        }
    }

    public void run()
    {
        byte[] response               = null;
        byte[] request         = null;
        boolean unwrapped      = false;
        GSSContext context     = null;

        try {
          Thread currentThread = Thread.currentThread();
          String threadName    = currentThread.getName();
```

```
debugPrefix = program + " " + threadName + ": ";

debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                       + "servicing client ...");

debug.out(Debug.OPTS_CAT_APPLICATION,
     debugPrefix + "creating GSSContext");

context = mgr.createContext(gssCred);

// First establish context with the initiator.
if (!establishContext(context))
    return;

// Then process messages from the initiator.
// We expect to receive a wrapped message followed by a MIC.
// The MIC should have been calculated over the plain
// text that we received wrapped.
// Use delegated creds if any.
// Then run as initiator using own creds if necessary; only
// the first thread does this.

do {
   debug.out(Debug.OPTS_CAT_APPLICATION,
        debugPrefix + "receiving per-message request");

    request = tcp.receive();
    if (request == null || request.length == 0)
    {
       debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
           + "Received no data; perhaps client disconnected");

       return;
    }

    // Expect wrapped message first.
    if (!unwrapped)
    {
        response = unwrap(context, request);
        unwrapped = true;
        continue; // get next request
    }

    // Followed by a MIC.
    verifyMIC(context, request, response);

    // Impersonate the initiator if it delegated its creds to us.
    if (context.getCredDelegState())
        useDelegatedCred(context);

    debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
        + "clientServer=" + clientServer
        + ", beenInitiator=" + beenInitiator);

    // If necessary, run as initiator using our own creds.
    if (clientServer)
        runAsInitiatorOnce(currentThread);

    debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + "done");
    return;

} while(true);

} catch (Exception exc) {
  debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix + "ERROR");
   exc.printStackTrace();
```

```
            // Squelch per-thread exceptions so we don't bring
            // the server down because of exceptions in
            // individual threads.
            return;
        } finally {
            if (context != null)
            {
                try {
                    context.dispose();
                } catch (Exception exc) {}
            }
        }
    }

    synchronized void runAsInitiatorOnce(Thread thread)
        throws InterruptedException
    {
        if (!beenInitiator)
        {
            // set flag true early to prevent subsequent threads
            // from attempting to runAsInitiator.
            beenInitiator = true;

            debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix +
                    "About to run as initiator with own creds ...");

            //thread.sleep(30*1000, 0);
            runAsInitiator();
        }
    }


    void runAsInitiator(GSSCredential cred)
    {
        Client client = null;
        try {
            client = new Client(cred,
                                serviceNameNoRealm,
                                serviceHost,
                                servicePort,
                                serviceMsg);

            client.initialize();

            BitSet flags = new BitSet();
            flags.set(Util.CONTEXT_OPTS_MUTUAL);
            flags.set(Util.CONTEXT_OPTS_CONF);
            flags.set(Util.CONTEXT_OPTS_INTEG);

            client.interactWithAcceptor(flags);

        } catch (Exception exc) {
               debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
                  + "Exception running as initiator");

            exc.printStackTrace();
        } finally {
            try {
                client.dispose();
            } catch (Exception exc) {}
        }
    }

    void runAsInitiator()
    {
        if (clientServer)
```

```
        {
           debug.out(Debug.OPTS_CAT_APPLICATION,
               debugPrefix + "running as initiator with own creds");

            runAsInitiator(gssCred); // use own creds;
        }
        else
        {
           debug.out(Debug.OPTS_CAT_APPLICATION, debugPrefix
               + "Cannot run as initiator with own creds "
               + "¥nbecause not running as both initiator and acceptor.");
        }
    }

    void printUsage()
    {
        System.out.println(program + usageString);
    }

    public static void main(String[] args) throws Exception
    {
        System.out.println(debug.toString()); // XXXXXXX
        String programName = "Server";
        try {
            Server server = new Server(programName,
                                       false); // don't use creds from Subject
            server.processArgs(args);
            server.initialize();
            server.processRequests();
        } catch (Exception exc) {
            debug.out(Debug.OPTS_CAT_APPLICATION, programName + ": EXCEPTION");
            exc.printStackTrace();
            throw exc;
        }
    }
}
```

サンプル: IBM JGSS JAAS 使用可能クライアント・プログラム

このサンプル・プログラムは JAAS ログインを実行し、JAAS ログイン・コンテキスト内で操作を実行します。このプログラムでは、変数 javax.security.auth.useSubjectCredsOnly は設定せずにデフォルトの "true" のままにしてあり、Java GSS が、クライアントによって作成されたログイン・コンテキストに関連付けられている JAAS サブジェクトから信任状を獲得するようになっています。

サンプル・クライアント・プログラムの使用について詳しくは、397 ページの『サンプル: サンプル JGSS プログラムのダウンロードおよび実行』を参照してください。

注: サンプル・コードをご使用の場合は、560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
// IBM Java GSS 1.0 sample JAAS-enabled client program

package com.ibm.security.jgss.test;
import com.ibm.security.jgss.Debug;
import com.ibm.security.auth.callback.Krb5CallbackHandler;
import javax.security.auth.Subject;
import javax.security.auth.login.LoginContext;
import java.security.PrivilegedExceptionAction;

/**
 * A Java GSS sample client that uses JAAS.
 *
 * It does a JAAS login and operates within the JAAS login context so created.
 *
```

```
 * It does not set the JAVA variable
 * javax.security.auth.useSubjectCredsOnly, leaving
 * the variable to default to true
 * so that Java GSS acquires credentials from the JAAS Subject
 * associated with login context (created by the client).
 *
 * The JAASClient is equivalent to its superclass {@link Client Client}
 * in all other respects, and it
 * can be run against the non-JAAS sample clients and servers.
 */

class JAASClient extends Client
{
    JAASClient(String programName) throws Exception
    {
        // Do not set useSubjectCredsOnly. Set only the program name.
        // useSubjectCredsOnly default to "true" if not set.
        super(programName);
    }

    static class JAASClientAction implements PrivilegedExceptionAction
    {
         private JAASClient client;

         public JAASClientAction(JAASClient client)
         {
             this.client = client;
         }

         public Object run () throws Exception
         {
             client.initialize();
             client.interactWithAcceptor();
             return null;
         }
    }

    public static void main(String args[]) throws Exception
    {
        String programName = "JAASClient";
        JAASClient client = null;
        Debug dbg = new Debug();

        System.out.println(dbg.toString()); // XXXXXXX

        try {
            client = new JAASClient(programName);//use Subject creds
            client.processArgs(args);

            LoginContext loginCtxt = new LoginContext("JAASClient",
                                       new Krb5CallbackHandler());

            loginCtxt.login();

            dbg.out(Debug.OPTS_CAT_APPLICATION,
                       programName + ": Kerberos login OK");

            Subject subject = loginCtxt.getSubject();

            PrivilegedExceptionAction jaasClientAction
                            = new JAASClientAction(client);

            Subject.doAsPrivileged(subject, jaasClientAction, null);

        } catch (Exception exc) {
            dbg.out(Debug.OPTS_CAT_APPLICATION,
                        programName + " Exception: " + exc.toString());
```

```
            exc.printStackTrace();
            throw exc;
        } finally {
            try {
                if (client != null)
                    client.dispose();
            } catch (Exception exc) {}
        }

        dbg.out(Debug.OPTS_CAT_APPLICATION,
                       programName + ": Done ...");
    }
}
```

サンプル: IBM JGSS JAAS 使用可能サーバー・プログラム

このサンプル・プログラムは JAAS ログインを実行し、JAAS ログイン・コンテキスト内で操作を実行します。

サンプル・サーバー・プログラムの使用について詳しくは、 397 ページの『サンプル: サンプル JGSS プログラムのダウンロードおよび実行』を参照してください。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
// IBM Java GSS 1.0 sample JAAS-enabled server program

package com.ibm.security.jgss.test;
import com.ibm.security.jgss.Debug;
import com.ibm.security.auth.callback.Krb5CallbackHandler;
import javax.security.auth.Subject;
import javax.security.auth.login.LoginContext;
import java.security.PrivilegedExceptionAction;

/**
 * A Java GSS sample server that uses JAAS.
 *
 * It does a JAAS login and operates within the JAAS login context so created.
 *
 * It does not set the JAVA variable
 * javax.security.auth.useSubjectCredsOnly, leaving
 * the variable to default to true
 * so that Java GSS acquires credentials from the JAAS Subject
 * associated with login context (created by the server).
 *
 * The JAASServer is equivalent to its superclass {@link Server Server}
 * in all other respects, and it
 * can be run against the non-JAAS sample clients and servers.
 */

class JAASServer extends Server
{
    JAASServer(String programName) throws Exception
    {
        super(programName);
    }

    static class JAASServerAction implements PrivilegedExceptionAction
    {
        private JAASServer server = null;

        JAASServerAction(JAASServer server)
        {
            this.server = server;
        }
```

```
        public Object run() throws Exception
        {
            server.initialize();
            server.processRequests();

            return null;
        }
    }

    public static void main(String[] args) throws Exception
    {
        String programName     = "JAASServer";
        Debug dbg              = new Debug();

        System.out.println(dbg.toString()); // XXXXXXX

        try {
            // Do not set useSubjectCredsOnly.
            // useSubjectCredsOnly defaults to "true" if not set.

            JAASServer server = new JAASServer(programName);

            server.processArgs(args);

            LoginContext loginCtxt = new LoginContext(programName,
                                          new Krb5CallbackHandler());

            dbg.out(Debug.OPTS_CAT_APPLICATION, programName + ": Login in ...");

            loginCtxt.login();

            dbg.out(Debug.OPTS_CAT_APPLICATION, programName +
                                              ": Login successful");

            Subject subject = loginCtxt.getSubject();

            JAASServerAction serverAction = new JAASServerAction(server);

            Subject.doAsPrivileged(subject, serverAction, null);
        } catch (Exception exc) {
            dbg.out(Debug.OPTS_CAT_APPLICATION, programName +  " EXCEPTION");
            exc.printStackTrace();
            throw exc;
        }
    }
}
```

例: IBM Java Secure Sockets Extension 1.4

JSSE の例では、クライアントおよびサーバーがネイティブ IBM i JSSE プロバイダーを使用して、安全な通信を可能にするコンテキストを作成する方法を示しています。

注: いずれの例でも、java.security ファイルの指定するプロパティーにかかわず、ネイティブ IBM i JSSE プロバイダーを使用します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

例: java.lang.Runtime.exec() を使用して CL プログラムを呼び出す

この例では、Java プログラムから CL プログラムを実行する方法を示します。この例では、Java クラス CallCLPgm が CL プログラムを実行します。

| CL プログラムは、「Java 仮想マシン・ジョブ表示」(DSPJVMJOB) CL コマンドを使用して、アクティブ
| な Java 仮想マシンを含むシステム上のジョブをすべて表示します。この例では、CL プログラムはコンパ
| イル済みであり、JAVSAMPLIB と呼ばれるライブラリーに存在していることが前提となっています。 CL
| プログラムからの出力は、QSYSPRT スプール・ファイルにあります。

Java プログラムから CL コマンドを呼び出す方法の例については、 227 ページの『例: java.lang.Runtime.exec() を使用して CL コマンドを呼び出す』 を参照してください。

注: JAVSAMPLIB は、IBM Developer Kit ライセンス・プログラム (LP) (番号 5761-JV1) のインストール・プロセスの一部としては作成されません。このライブラリーは明示的に作成する必要があります。

CallCLPgm Java クラスのソース・コード

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』 に同意していただいているものとします。

```
import java.io.*;

public class CallCLPgm
{
   public static void main(String[] args)
   {
      try
      {
         Process theProcess =
                 Runtime.getRuntime().exec("/QSYS.LIB/JAVSAMPLIB.LIB/DSPJVA.PGM");
      }
      catch(IOException e)
      {
         System.err.println("Error on exec() method");
         e.printStackTrace();
      }
   }
}
```

DSPJVA CL プログラムのソース・コード

```
| PGM
|     DSPJVMJOB OUTPUT(*PRINT)
| ENDPGM
```

例: java.lang.Runtime.exec() を使用して CL コマンドを呼び出す

この例では、Java プログラムから制御言語 (CL) コマンドを実行する方法を示します。

| この例では、Java クラスが CL コマンドを実行します。 CL コマンドは、「Java 仮想マシン・ジョブ表
| 示」(DSPJVMJOB) CL コマンドを使用して、アクティブな Java 仮想マシンを含むシステム上のジョブを
| すべて表示します。 CL コマンドからの出力は、QSYSPRT スプール・ファイルにあります。

`Runtime.getRuntime().exec()` 関数に渡す CL コマンドは次のフォーマットになります。

```
Runtime.getRuntime().exec("system CLCOMMAND");
```

ここで、*CLCOMMAND* は、実行しようとしている CL コマンドです。

CL コマンドを呼び出すための Java クラスのソース・コード

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』 に同意していただいているものとします。

```
import java.io.*;

public class CallCLCom
{
   public static void main(String[] args)
   {
      try
      {
         Process theProcess =
            Runtime.getRuntime().exec("system DSPJVMJOB OUTPUT(*PRINT)");
      }
      catch(IOException e)
      {
         System.err.println("Error on exec() method");
         e.printStackTrace();
      }
   }
}
```

関連概念

225 ページの『java.lang.Runtime.exec() を使用する』
java.lang.Runtime.exec() メソッドを使用して、Java プログラム内からプログラムまたはコマンドを呼び出します。 java.lang.Runtime.exec() メソッドを使用すると、1 つ以上の追加のスレッド対応のジョブが作成されます。 追加のジョブが、このメソッドに渡されたコマンド・ストリングを処理します。

15 ページの『Java システム・プロパティーのリスト』
Java システム・プロパティーにより、Java プログラムのランタイム環境が決まります。 Java システム・プロパティーは、IBM i のシステム値や環境変数と似ています。

例: java.lang.Runtime.exec() を使用して別の Java プログラムを呼び出す

この例では、java.lang.Runtime.exec() を使用して別の Java プログラムを呼び出す方法を示します。このクラスは、IBM Developer Kit for Java の一部として配布される Hello プログラムを呼び出します。 Hello クラスが System.out に書き込みを行うときに、このプログラムは、ストリームへのハンドルを取得し、そこから読み取りを行うことができます。

CallHelloPgm Java クラスのソース・コード

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.io.*;

public class CallHelloPgm
{
   public static void main(String args[])
   {
      Process theProcess = null;
      BufferedReader inStream = null;

      System.out.println("CallHelloPgm.main() invoked");

      // call the Hello class
      try
      {
          theProcess = Runtime.getRuntime().exec("java QIBMHello");
      }
      catch(IOException e)
      {
         System.err.println("Error on exec() method");
         e.printStackTrace();
      }
```

```
      // read from the called program's standard output stream
      try
      {
         inStream = new BufferedReader(
                              new InputStreamReader( theProcess.getInputStream() ));
         System.out.println(inStream.readLine());
      }
      catch(IOException e)
      {
         System.err.println("Error on inStream.readLine()");
         e.printStackTrace();
      }
   }
}
```

例: ILE C から Java を呼び出す

次に示すのは、system() 関数を使用して Java Hello プログラムを呼び出す Integrated Language Environment (ILE) C プログラムの例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
#include <stdlib.h>

 int main(void)
 {
   int result;

   /* The system function passes the given string
    * to the CL command processor for processing.
    */

   result = system("JAVA CLASS('QIBMHello')");
 }
```

例: RPG から Java を呼び出す

次に示すのは、QCMDEXC API を使用して Java Hello プログラムを呼び出す RPG プログラムの例です。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
D*            DEFINE  THE PARAMETERS FOR THE QCMDEXC API
D*
DCMDSTRING        S             25    INZ('JAVA CLASS(''QIBMHello'')')
DCMDLENGTH        S             15P 5 INZ(23)
D*            NOW THE CALL TO QCMDEXC WITH THE 'JAVA' CL COMMAND
C                   CALL      'QCMDEXC'
C                   PARM                    CMDSTRING
C                   PARM                    CMDLENGTH
C*       This next line displays 'DID IT' after you exit the
C*       Java Shell via F3 or F12.
C     'DID IT'      DSPLY
C*       Set On LR to exit the RPG program
C                   SETON                                        LR
C
```

例: プロセス間通信に入出力ストリームを使用する

この例では、Java から C プログラムを呼び出し、プロセス間通信に入出力ストリームを使用する方法を示します。

C プログラムは、その標準出力ストリームにストリングを書き込み、Java プログラムは、このストリングを読み取り、表示します。この例では、JAVSAMPLIB というライブラリーが作成されていることと、その中で CSAMP1 プログラムが作成されていることを前提としています。

注: JAVSAMPLIB は、IBM Developer Kit ライセンス・プログラム (LP) (番号 5761-JV1) のインストール・プロセスの一部としては作成されません。明示的にそれを作成しなければなりません。

CallPgm Java クラスのソース・コード

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.io.*;

public class CallPgm
{
   public static void main(String args[])
   {
      Process theProcess = null;
      BufferedReader inStream = null;

      System.out.println("CallPgm.main() invoked");

      // call the CSAMP1 program
      try
      {
          theProcess = Runtime.getRuntime().exec(
                       "/QSYS.LIB/JAVSAMPLIB.LIB/CSAMP1.PGM");
      }
      catch(IOException e)
      {
         System.err.println("Error on exec() method");
         e.printStackTrace();
      }

      // read from the called program's standard output stream
      try
      {
         inStream = new BufferedReader(new InputStreamReader
                          (theProcess.getInputStream()));
         System.out.println(inStream.readLine());
      }
      catch(IOException e)
      {
         System.err.println("Error on inStream.readLine()");
         e.printStackTrace();
      }
   }
 }
```

CSAMP1 C プログラムのソース・コード

注: 法律上の重要な情報に関しては、コードの特記事項情報をお読みください。

```
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char* args[])
```

```
{
     /* Convert the string to ASCII at compile time */
#pragma convert(819)
     printf("Program JAVSAMPLIB/CSAMP1 was invoked¥n");
#pragma convert(0)
     /* Stdout may be buffered, so flush the buffer */

     fflush(stdout);
}
```

例: Java 呼び出し API

この Integrated Language Environment (ILE) C の例は、標準の呼び出し API パラダイムに従っています。

これは以下を実行します。

- JNI_CreateJavaVM() を使用して Java 仮想マシンを作成する。
- Java 仮想マシンを使用して、実行したいクラス・ファイルを検索する。
- クラスの main メソッドの methodID を検索する。
- クラスの main メソッドを呼び出す。
- 例外が発生した場合に、エラーを報告する。

プログラムを作成する際は、QJVAJNI または QJVAJNI64 サービス・プログラムが、JNI_CreateJavaVM() API 機能を提供します。 JNI_CreateJavaVM() は Java 仮想マシンを作成します。

注: QJVAJNI64 は、teraspace/LLP64 ネイティブ・メソッドと呼び出し API のサポートのための新しいサービス・プログラムです。

これらのサービス・プログラムは、システム・バインディング・ディレクトリーにあり、制御言語 (CL) 作成コマンドで明示的に示す必要はありません。たとえば、前述のサービス・プログラムを「プログラム作成 (CRTPGM)」コマンドや「サービス・プログラムの作成 (CRTSRVPGM)」コマンドを使用する際に明示的に示すことはしません。

このプログラムを実行する方法の 1 つは、以下の制御言語 (CL) コマンドを使用することです。

```
SBMJOB CMD(CALL PGM(YOURLIB/PGMNAME)) ALWMLTTHD(*YES)
```

Java 仮想マシンを作成するジョブは、マルチスレッド対応でなければなりません。主プログラムからの出力と、プログラムからのすべての出力は、最終的に QPRINT スプール・ファイルに送られます。「投入されたジョブの処理 (WRKSBMJOB)」制御言語 (CL) コマンドを使用し、「ジョブの投入 (SBMJOB)」CL コマンドで開始したジョブを表示すると、これらのスプール・ファイルを見ることができます。

例: ILE C で Java 呼び出し API を使用する

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
#define OS400_JVM_12
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <jni.h>

/* Specify the pragma that causes all literal strings in the
 * source code to be stored in ASCII (which, for the strings
 * used, is equivalent to UTF-8)
 */
```

```
#pragma convert(819)

/* Procedure: Oops
 *
 * Description: Helper routine that is called when a JNI function
 *              returns a zero value, indicating a serious error.
 *              This routine reports the exception to stderr and
 *              ends the JVM abruptly with a call to FatalError.
 *
 * Parameters:  env -- JNIEnv* to use for JNI calls
 *              msg -- char* pointing to error description in UTF-8
 *
 * Note:        Control does not return after the call to FatalError
 *              and it does not return from this procedure.
 */

void Oops(JNIEnv* env, char *msg) {
    if ((*env)->ExceptionOccurred(env)) {
        (*env)->ExceptionDescribe(env);
    }
    (*env)->FatalError(env, msg);
}

/* This is the program's "main" routine. */
int main (int argc, char *argv[])
{

    JavaVMInitArgs initArgs; /* Virtual Machine (VM) initialization structure, passed by
                              * reference to JNI_CreateJavaVM(). See jni.h for details
                              */
    JavaVM* myJVM;           /* JavaVM pointer set by call to JNI_CreateJavaVM */
    JNIEnv* myEnv;           /* JNIEnv pointer set by call to JNI_CreateJavaVM */
    char*   myClasspath;     /* Changeable classpath 'string' */
    jclass  myClass;         /* The class to call, 'NativeHello'. */
    jmethodID mainID;        /* The method ID of its 'main' routine. */
    jclass  stringClass;     /* Needed to create the String[] arg for main */
    jobjectArray args;       /* The String[] itself */
    JavaVMOption options[1]; /* Options array -- use options to set classpath */
    int     fd0, fd1, fd2;   /* file descriptors for IO */

    /* Open the file descriptors so that IO works. */
    fd0 = open("/dev/null", O_CREAT|O_TRUNC|O_RDWR,   S_IRUSR|S_IROTH);
    fd1 = open("/dev/null", O_CREAT|O_TRUNC|O_WRONLY, S_IWUSR|S_IWOTH);
    fd2 = open("/dev/null", O_CREAT|O_TRUNC|O_WRONLY, S_IWUSR|S_IWOTH);

    /*  Set the version field of the initialization arguments for JNI v1.5. */
    initArgs.version = 0x00010004;


    /* Now, you want to specify the directory for the class to run in the classpath.
     * with  Java2, classpath is passed in as an option.
     * Note: You must specify the directory name in UTF-8 format. So, you wrap
     *       blocks of code in #pragma convert statements.
     */
    options[0].optionString="-Djava.class.path=/CrtJvmExample";

    initArgs.options=options;  /* Pass in the classpath that has been set up. */
    initArgs.nOptions = 1;     /* Pass in classpath and version options */

    /*  Create the JVM -- a nonzero return code indicates there was
     *  an error. Drop back into EBCDIC and write a message to stderr
     *  before exiting the program.
     *  Note:  This will run the default JVM and JDK which is 32bit JDK 6.0.
     *  If you want to run a different JVM and JDK, set the JAVA_HOME environment
     *  variable to the home directory of the JVM you want to use
     *  (prior to the CreateJavaVM() call).
```

```
     */
    if (JNI_CreateJavaVM(&myJVM, (void **)&myEnv, (void *)&initArgs)) {
  #pragma convert(0)
         fprintf(stderr, "Failed to create the JVM¥n");
  #pragma convert(819)
         exit(1);
    }

    /*  Use the newly created JVM to find the example class,
     *  called 'NativeHello'.
     */
    myClass = (*myEnv)->FindClass(myEnv, "NativeHello");
    if (! myClass) {
        Oops(myEnv, "Failed to find class 'NativeHello'");
    }

    /* Now, get the method identifier for the 'main' entry point
     * of the class.
     * Note: The signature of 'main' is always the same for any
     *       class called by the following java command:
     *           "main" , "([Ljava/lang/String;)V"
     */
    mainID = (*myEnv)->GetStaticMethodID(myEnv,myClass,"main",
                                         "([Ljava/lang/String;)V");
    if (! mainID) {
        Oops(myEnv, "Failed to find jmethodID of 'main'");
    }

    /* Get the jclass for String to create the array
     * of String to pass to 'main'.
     */
    stringClass = (*myEnv)->FindClass(myEnv, "java/lang/String");
    if (! stringClass) {
        Oops(myEnv, "Failed to find java/lang/String");
    }

    /*  Now, you need to create an empty array of strings,
     *  since main requires such an array as a parameter.
     */
    args = (*myEnv)->NewObjectArray(myEnv,0,stringClass,0);
    if (! args) {
        Oops(myEnv, "Failed to create args array");
    }

    /* Now, you have the methodID of main and the class, so you can
     * call the main method.
     */
    (*myEnv)->CallStaticVoidMethod(myEnv,myClass,mainID,args);

    /* Check for errors. */
    if ((*myEnv)->ExceptionOccurred(myEnv)) {
        (*myEnv)->ExceptionDescribe(myEnv);
    }

    /* Finally, destroy the JavaVM that you created. */
    (*myJVM)->DestroyJavaVM(myJVM);

    /* All done. */
    return 0;
}
```

詳しくは、 220 ページの『Java 呼び出し API』を参照してください。

例: Java 用の IBM PASE for i ネイティブ・メソッド

この Java 用の PASE for i ネイティブ・メソッドの例では、Java ネイティブ・インターフェース (JNI) を使用して Java コードにコールバックを行うネイティブ C メソッドのインスタンスを呼び出します。この例では、Java コードから直接ストリングにアクセスするのではなく、 JNI を通して Java にコールバックを行ってストリング値を取得するネイティブ・メソッドを呼び出します。

このソース・ファイル例の HTML 版を表示するには、以下のリンクを使用してください。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

- 『例: PaseExample1.java』
- 533 ページの『例: PaseExample1.c』

PASE for i ネイティブ・メソッドの例を実行するためには、まず以下のトピックのタスクを完了する必要があります。

1. 534 ページの『例: 使用している AIX ワークステーションへのソース・コード例のダウンロード』
2. 534 ページの『例: ソース・コード例の準備』
3. 535 ページの『例: Java 用の PASE for i ネイティブ・メソッドの例を実行するために IBM i サーバーを準備する』

Java 用の PASE for i ネイティブ・メソッドの例を実行する

上記のタスクを完了したら、この例を実行することができます。このプログラム例を実行するには、以下のコマンドのいずれかを使用します。

- IBM i コマンド・プロンプトから:

```
JAVA CLASS(PaseExample1) CLASSPATH('/home/example')
```

- Qshell コマンド・プロンプトまたは PASE for i 端末セッションから:

```
cd /home/example
java PaseExample1
```

例: PaseExample1.java

このサンプル・プログラムはネイティブ・メソッド・ライブラリー 'PaseExample1' をロードします。ネイティブ・メソッドのソース・コードは PaseExample1.c に入っています。この Java プログラムの printString メソッドは、ネイティブ・メソッド getStringNative を使用して String の値を得ようとしています。このネイティブ・メソッドは、単にこのクラスの getStringCallback メソッドにコールバックするだけです。

注: 法律上の重要な情報に関しては、コードの特記事項情報をお読みください。

```
////////////////////////////////////////////////////////////////////////////////
//
// This example program loads the native method library 'PaseExample1'.
// The source code for the native method is contained in PaseExample1.c
// The printString method in this Java program uses a native method,
// getStringNative to retrieve the value of the String. The native method
// simply calls back into the getStringCallback method of this class.
//
////////////////////////////////////////////////////////////////////////////////


public class PaseExample1 {
```

```
    public static void main(String args[]) {
 PaseExample1 pe1 = new PaseExample1("String for PaseExample1");
 pe1.printString();
    }

    String str;

    PaseExample1(String s) {
 str = s;
    }


    //-----------------------------------------------------------------
    public void printString() {
 String result = getStringNative();
 System.out.println("Value of str is '" + result + "'");
    }

    // This calls getStringCallback through JNI.
    public native String getStringNative();

    // Called by getStringNative via JNI.
    public String getStringCallback() {
 return str;
    }


    //-----------------------------------------------------------------
    static {
 System.loadLibrary("PaseExample1");
    }

}
```

例: PaseExample1.c

このネイティブ・メソッドは、クラス PaseExample1 の getStringNative メソッドをインプリメントします。このネイティブ・メソッドでは、JNI 関数 CallObjectMethod を使用して、クラス PaseExample1 の getStringCallback メソッドにコールバックします。

注: 法律上の重要な情報に関しては、コードの特記事項情報をお読みください。

```
/*
 *

 * This native method implements the getStringNative method of class
 * PaseExample1. It uses the JNI function CallObjectMethod to call
 * back to the getStringCallback method of class PaseExample1.
 *
 * Compile this code in AIX or PASE for i to create module 'libPaseExample1.so'.
 *
 */

#include "PaseExample1.h"
#include <stdlib.h>

/*
 * Class:     PaseExample1
 * Method:    getStringNative
 * Signature: ()Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_PaseExample1_getStringNative(JNIEnv* env, jobject obj) {
   char* methodName = "getStringCallback";
   char* methodSig = "()Ljava/lang/String;";
```

```
    jclass clazz = (*env)->GetObjectClass(env, obj);
    jmethodID methodID = (*env)->GetMethodID(env, clazz, methodName, methodSig);
    return (*env)->CallObjectMethod(env, obj, methodID);
}
```

例: 使用している AIX ワークステーションへのソース・コード例のダウンロード

Java 用の PASE for i ネイティブ・メソッドの例を実行する前に、ソース・コードを含む圧縮ファイルをダウンロードする必要があります。圧縮ファイルを AIX ワークステーションにダウンロードするには、以下のステップを完成させます。

1. AIX ワークステーションに、ソース・ファイルを入れる一時ディレクトリーを作成します。
2. PASE for i ソース・コード例を一時ディレクトリーにダウンロードします。
3. 例のファイルを一時ディレクトリーに unzip します。

Java 用の PASE for i ネイティブ・メソッドの例についての詳細は、以下のトピックを参照してください。

216 ページの『例: Java 用の IBM PASE for i ネイティブ・メソッド』
この Java 用の PASE for i ネイティブ・メソッドの例では、Java ネイティブ・インターフェース (JNI) を使用して Java コードにコールバックを行うネイティブ C メソッドのインスタンスを呼び出します。 この例では、Java コードから直接ストリングにアクセスするのではなく、 JNI を通して Java にコールバックを行ってストリング値を取得するネイティブ・メソッドを呼び出します。

『例: ソース・コード例の準備』
Java 用の PASE for i ネイティブ・メソッドの例をご使用のサーバーに移動するためには、まず、ソース・コードをコンパイルし、C インクルード・ファイルを作成し、共用ライブラリー・オブジェクトを作成する必要があります。

535 ページの『例: Java 用の PASE for i ネイティブ・メソッドの例を実行するために IBM i サーバーを準備する』
Java 用の PASE for i ネイティブ・メソッドの例を実行する前に、例を実行できるようにサーバーを準備する必要があります。 サーバーを準備する場合、ファイルをサーバーにコピーし、例を実行するのに必要な環境変数を追加する必要があります。

例: ソース・コード例の準備

Java 用の PASE for i ネイティブ・メソッドの例をご使用のサーバーに移動するためには、まず、ソース・コードをコンパイルし、C インクルード・ファイルを作成し、共用ライブラリー・オブジェクトを作成する必要があります。

例には、以下の C および Java ソース・ファイルが含まれています。

- **PaseExample1.c:** getStringNative() のインプリメンテーションを含む C ソース・コード・ファイル。
- **PaseExample1.java:** C プログラムでネイティブ getStringNative メソッドを呼び出す Java ソース・コード・ファイル。

コンパイルした Java .class ファイルを使用して、C インクルード・ファイル PaseExample1.h を作成します。ここには、C ソース・コードに含まれる getStringNative メソッドの機能プロトタイプが含まれています。

AIX ワークステーションでのソース・コード例を準備するには、以下のステップを完成させます。

1. 次のコマンドを使用して、Java ソース・コードをコンパイルします。

```
javac PaseExample1.java
```

2. 次のコマンドを使用して、ネイティブ・メソッド・プロトタイプを含む C インクルード・ファイルを作成します。

```
javah -jni PaseExample1
```

新しい C インクルード・ファイル (PaseExample1.h) には、getStringNative メソッドの機能プロトタイプが含まれます。 C ソース・コード例 (PaseExample1.c) には、getStringNative メソッドを使用するために、C インクルード・ファイルからコピーして変更した情報が含まれています。 JNI の使用の詳細は、Sun Web サイトの、「Java Native Interface」を参照してください。

3. 次のコマンドを使用して、C ソース・コードをコンパイルし、共用ライブラリー・オブジェクトを作成します。

```
xlc -G -I/usr/local/java/J1.5.0/include PaseExample1.c -o libPaseExample1.so
```

新しい共用ライブラリー・オブジェクト・ファイル (libPaseExample1.so) には、例で使用されるネイティブ・メソッド・ライブラリー "PaseExample1" が含まれます。

注: それぞれの AIX システムで、正しい Java ネイティブ・メソッド・インクルード・ファイル (たとえば、jni.h) を含むディレクトリーを示すために、-I オプションを変更しなければならない場合があります。

Java 用の PASE for i ネイティブ・メソッドの例についての詳細は、以下のトピックを参照してください。

216 ページの『例: Java 用の IBM PASE for i ネイティブ・メソッド』
この Java 用の PASE for i ネイティブ・メソッドの例では、Java ネイティブ・インターフェース (JNI) を使用して Java コードにコールバックを行うネイティブ C メソッドのインスタンスを呼び出します。 この例では、Java コードから直接ストリングにアクセスするのではなく、 JNI を通して Java にコールバックを行ってストリング値を取得するネイティブ・メソッドを呼び出します。

534 ページの『例: 使用している AIX ワークステーションへのソース・コード例のダウンロード』
Java 用の PASE for i ネイティブ・メソッドの例を実行する前に、ソース・コードを含む圧縮ファイルをダウンロードする必要があります。 圧縮ファイルを AIX ワークステーションにダウンロードするには、以下のステップを完成させます。

『例: Java 用の PASE for i ネイティブ・メソッドの例を実行するために IBM i サーバーを準備する』
Java 用の PASE for i ネイティブ・メソッドの例を実行する前に、例を実行できるようにサーバーを準備する必要があります。 サーバーを準備する場合、ファイルをサーバーにコピーし、例を実行するのに必要な環境変数を追加する必要があります。

例: Java 用の PASE for i ネイティブ・メソッドの例を実行するために IBM i サーバーを準備する

Java 用の PASE for i ネイティブ・メソッドの例を実行する前に、例を実行できるようにサーバーを準備する必要があります。サーバーを準備する場合、ファイルをサーバーにコピーし、例を実行するのに必要な環境変数を追加する必要があります。

サーバーを準備するには、以下のステップを完成させます。

1. サーバーに、例のファイルを含めるための、次の統合ファイル・システム・ディレクトリーを作成します。たとえば、次の制御言語 (CL) コマンドを使用して、/home/example というディレクトリーを作成します。

```
mkdir '/home/example'
```

2. 以下のファイルを新しいディレクトリーにコピーします。
   - PaseExample1.class
   - libPaseExample1.so
3. IBM i コマンド・プロンプトで、以下の制御言語 (CL) コマンドを使用して、必要な環境変数を追加します。

```
addenvvar PASE_THREAD_ATTACH 'Y'
addenvvar LIBPATH '/home/example'
```

注: PASE for i 端末セッションから PASE ネイティブ・メソッドを使用する場合、32 ビットの PASE for i 環境はすでに開始されています。この場合、PASE_THREAD_ATTACH を Y にセットし、LIBPATH を PASE for i ネイティブ・メソッド・ライブラリーのパスにセットすることだけを行います。

Java 用の PASE for i ネイティブ・メソッドの例についての詳細は、以下のトピックを参照してください。

216 ページの『例: Java 用の IBM PASE for i ネイティブ・メソッド』
この Java 用の PASE for i ネイティブ・メソッドの例では、Java ネイティブ・インターフェース (JNI) を使用して Java コードにコールバックを行うネイティブ C メソッドのインスタンスを呼び出します。 この例では、Java コードから直接ストリングにアクセスするのではなく、 JNI を通して Java にコールバックを行ってストリング値を取得するネイティブ・メソッドを呼び出します。

534 ページの『例: 使用している AIX ワークステーションへのソース・コード例のダウンロード』
Java 用の PASE for i ネイティブ・メソッドの例を実行する前に、ソース・コードを含む圧縮ファイルをダウンロードする必要があります。 圧縮ファイルを AIX ワークステーションにダウンロードするには、以下のステップを完成させます。

534 ページの『例: ソース・コード例の準備』
Java 用の PASE for i ネイティブ・メソッドの例をご使用のサーバーに移動するためには、まず、ソース・コードをコンパイルし、C インクルード・ファイルを作成し、共用ライブラリー・オブジェクトを作成する必要があります。

例: Java 用の ILE ネイティブ・メソッド

この Java 用の Integrated Language Environment (ILE) ネイティブ・メソッドの例では、ネイティブ C メソッドのインスタンスを呼び出します。次いでそれは Java ネイティブ・インターフェース (JNI) を使用して Java コードにコールバックを行い、Java ストリング変数の値を設定します。その後、Java ストリング変数は、Java コードによって標準出力に書き込まれます。

このソース・ファイル例の HTML 版を表示するには、以下のリンクを使用してください。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

- 537 ページの『例: NativeHello.java』
- 538 ページの『例: NativeHello.c』

ILE ネイティブ・メソッドの例を実行するためには、まず以下のトピックのタスクを完了する必要があります。

1. 541 ページの『例: ILE ネイティブ・メソッド・ソース・コードを準備する』
2. 542 ページの『例: ILE ネイティブ・メソッド・プログラム・オブジェクトを作成する』

Java 用の ILE ネイティブ・メソッドの例を実行する

上記のタスクを完了したら、この例を実行することができます。このプログラム例を実行するには、以下のコマンドのいずれかを使用します。

- IBM i コマンド・プロンプトから:

```
JAVA CLASS(NativeHello) CLASSPATH('/ileexample')
```

- Qshell コマンド・プロンプトから:

```
cd /ileexample
java NativeHello
```

例: NativeHello.java

Java ソース・コードは、Java 用の Integrated Language Environment (ILE) ネイティブ・メソッドの例で使用されます。

NativeHello.java ソース・コードは、Java の ILE ネイティブ・メソッドを例示するために使用される Java コードを示します。ネイティブ・メソッド setTheString() は、main() メソッド内から Java コードによって呼び出されます。呼び出しの結果として、C インプリメンテーション・コード で定義された関数 Java_NativeHello_setTheString() が制御を取得し、Java ネイティブ・インターフェース (JNI) を使用して、Java コードにコールバックを行い、Java ストリング変数 theString の値を設定します。その後、制御は Java コードに戻され、ストリング変数は標準出力に書き込まれます。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
public class NativeHello {

  //  Declare a field of type 'String' in the NativeHello object.
  //  This is an 'instance' field, so every NativeHello object
  //  contains one.
  public String theString;           // instance variable

  //  Declare the native method itself.  This native method
    //  creates a new string object, and places a reference to it
  //  into 'theString'
  public native void setTheString(); // native method to set string

  //  This 'static initializer' code is called before the class is
  //  first used.
  static {
     //  Attempt to load the native method library.  If you do not
        //  find it, write a message to 'out', and try a hardcoded path.
     //  If that fails, then exit.
     try {
        //  System.loadLibrary uses the java.library.path property or
        //  the LIBPATH environment variable.
        System.loadLibrary("NATHELLO");
     }
     catch (UnsatisfiedLinkError e1) {
        //  Did not find the service program.
        System.out.println("I did not find NATHELLO *SRVPGM.");
        System.out.println ("(I will try a hardcoded path)");

        try {
           //  System.load takes the full integrated file system form path.
           System.load ("/qsys.lib/ileexample.lib/nathello.srvpgm");
        }
        catch (UnsatisfiedLinkError e2) {
           //  If you get to this point, then you are done!  Write the message
            //  and exit.
```

```
          System.out.println
             ("<sigh> I did not find NATHELLO *SRVPGM anywhere. Goodbye");
           System.exit(1);
        }
     }
  }

  //  Here is the 'main' code of this class. This is what runs when you
    //  enter 'java NativeHello' on the command line.
  public static void main(String argv[]){

     //  Allocate a new NativeHello object now.
     NativeHello nh = new NativeHello();

     //  Echo location.
     System.out.println("(Java) Instantiated NativeHello object");
     System.out.println("(Java) string field is '" + nh.theString + "'");
     System.out.println("(Java) Calling native method to set the string");

     //  Here is the call to the native method.
     nh.setTheString();

     //  Now, print the value after the call to double check.
     System.out.println("(Java) Returned from the native method");
     System.out.println("(Java) string field is '" + nh.theString + "'");
     System.out.println("(Java) All done...");
    }
}
```

例: NativeHello.c

C ソース・コードは、Java 用の Integrated Language Environment (ILE) ネイティブ・メソッドの例で使用されます。

NativeHello.c ソース・コードは、ネイティブ・メソッドを C で実装したものを示します。 Java コードで定義された Java ネイティブ・メソッド setTheString() が、Java コードによって呼び出されると、C 関数 Java_NativeHello_setTheString() が制御を取得し、Java ネイティブ・インターフェース (JNI) を使用して、Java コードにコールバックを行い、Java ストリング変数 theString の値を設定します。その後、制御は Java コードに戻され、ストリング変数は Java コードによって標準出力に書き込まれます。

この例は、Java をネイティブ・メソッドにリンクする方法を示します。ただし、IBM i が内部的に拡張 2 進化 10 進コード (EBCDIC) マシンであることから生じる複雑さも示しています。また、現在、JNI に本当の意味での国際化要素が欠けていることから生じる複雑さも示しています。

このような状況は JNI で初めて生じたものではありませんが、これらの理由から、作成する C コードには IBM i サーバー固有の違いがあります。STDOUT や STDERR への書き込み、あるいは STDIN からの読み取りを行う場合には、データがおそらく EBCDIC 形式でエンコードされることに留意しなければなりません。

C コードでは、大部分のリテラル・ストリング (7 ビット文字だけを含むもの) を、JNI によって必要とされる UTF-8 形式に簡単に変換することができます。これを行うには、リテラル・ストリングをコード・ページ変換 pragma ステートメントで囲みます。ただし、C コードから情報を直接 STDOUT または STDERR に書き込むことが必要な場合があるため、一部のリテラルを EBCDIC のまま残しておくこともできます。

注: #pragma convert(0) ステートメントは、文字データを EBCDIC に変換します。 #pragma convert(819) ステートメントは、文字データを ASCII コードに変換します。これらのステートメントは、C プログラム内の文字データをコンパイル時に変換します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
#include <stdlib.h>       /* malloc, free, and so forth */
#include <stdio.h>        /* fprintf(), and so forth */
#include <qtqiconv.H>     /* iconv() interface */
#include <string.h>       /* memset(), and so forth */
#include "NativeHello.h"  /* generated by 'javah-jni' */

/*  All literal strings are ISO-8859-1 Latin 1 code page
 * (and with 7-bit characters, they are also automatically UTF-8).
 */
#pragma convert(819)  /* handle all literal strings as ASCII */

/*  Report and clear a JNI exception.  */
static void HandleError(JNIEnv*);

/*  Print an UTF-8 string to stderr in the coded character
 *  set identifier (CCSID) of the current job.
 */
static void JobPrint(JNIEnv*, char*);

/*  Constants describing which direction to covert:  */
#define CONV_UTF2JOB 1
#define CONV_JOB2UTF 2

/*  Convert a string from the CCSID of the job to UTF-8, or vice-versa.  */
int StringConvert(int direction, char *sourceStr, char *targetStr);

/*  Native method implementation of 'setTheString()'.  */

JNIEXPORT void JNICALL Java_NativeHello_setTheString
(JNIEnv *env, jobject javaThis)
{
    jclass thisClass;     /* class for 'this' object */
    jstring stringObject; /* new string, to be put in field in 'this' */
    jfieldID fid;         /* field ID required to update field in 'this' */
    jthrowable exception; /* exception, retrieved using ExceptionOccurred */

    /*  Write status to console.  */
    JobPrint(env, "( C ) In the native method¥n");

    /* Build the new string object.  */
    if (! (stringObject = (*env)->NewStringUTF(env, "Hello, native world!")))
    {
      /*  For nearly every function in the JNI, a null return value indicates
       *  that there was an error, and that an exception had been placed where it
       *  could be retrieved by 'ExceptionOccurred()'. In this case, the error
       *  would typically be fatal, but for purposes of this example, go ahead
       *  and catch the error, and continue.
       */
       HandleError(env);
       return;
    }

    /* get the class of the 'this' object, required to get the fieldID  */
    if (! (thisClass = (*env)->GetObjectClass(env,javaThis)))
    {
      /*  A null class returned from GetObjectClass indicates that there
       *  was a problem. Instead of handling this problem, simply return and
       *  know that the return to Java automatically 'throws' the stored Java
       *  exception.
       */
       return;
    }

    /*  Get the fieldID to update.  */
```

```
    if (! (fid = (*env)->GetFieldID(env,
                                    thisClass,
                                    "theString",
                                    "Ljava/lang/String;")))
    {
      /*  A null fieldID returned from GetFieldID indicates that there
       *  was a problem.  Report the problem from here and clear it.
       *  Leave the string unchanged.
       */
        HandleError(env);
        return;
    }

    JobPrint(env, "( C ) Setting the field¥n");

    /*  Make the actual update.
     *  Note: SetObjectField is an example of an interface that does
     *  not return a return value that can be tested.  In this case, it
     *  is necessary to call ExceptionOccurred() to see if there
     *  was a problem with storing the value
     */
    (*env)->SetObjectField(env, javaThis, fid, stringObject);

    /*  Check to see if the update was successful. If not, report the error.  */
    if ((*env)->ExceptionOccurred(env)) {

      /*  A non-null exception object came back from ExceptionOccurred,
       *  so there is a problem and you must report the error.
       */
       HandleError(env);
    }

    JobPrint(env, "( C ) Returning from the native method¥n");
    return;
}

static void HandleError(JNIEnv *env)
{
    /*  A simple routine to report and handle an exception.  */
    JobPrint(env, "( C  ) Error occurred on JNI call: ");
    (*env)->ExceptionDescribe(env); /* write exception data to the console */
    (*env)->ExceptionClear(env);    /* clear the exception that was pending */
}

static void JobPrint(JNIEnv *env, char *str)
{
    char   *jobStr;
    char   buf[512];
    size_t len;

    len = strlen(str);

    /*  Only print non-empty string. */
    if (len) {
        jobStr = (len >= 512) ? malloc(len+1) : &buf;
        if (! StringConvert(CONV_UTF2JOB, str, jobStr))
            (*env)->FatalError
              (env,"ERROR in JobPrint: Unable to convert UTF2JOB");
        fprintf(stderr, jobStr);
        if (len >= 512) free(jobStr);
    }
}

int StringConvert(int direction, char *sourceStr, char *targetStr)
{
    QtqCode_T source, target;     /* parameters to instantiate iconv  */
    size_t    sStrLen, tStrLen;   /* local copies of string lengths   */
```

```
    iconv_t   ourConverter;       /* the actual conversion descriptor */
    int       iconvRC;            /* return code from the conversion  */
    size_t    originalLen;        /* original length of the sourceStr */

    /*  Make local copies of the input and output sizes that are initialized
     *  to the size of the input string. The iconv() requires the
     *  length parameters to be passed by address (that is as int*).
     */
    originalLen = sStrLen = tStrLen = strlen(sourceStr);

    /*  Initialize the parameters to the QtqIconvOpen() to zero.  */
    memset(&source,0x00,sizeof(source));
    memset(&target,0x00,sizeof(target));

    /*  Depending on direction parameter, set either SOURCE
     *  or TARGET CCSID to ISO 8859-1 Latin.
     */
    if (CONV_UTF2JOB == direction )
        source.CCSID = 819;
    else
        target.CCSID = 819;

    /*  Create the iconv_t converter object.  */
    ourConverter = QtqIconvOpen(&target,&source);

    /*  Make sure that you have a valid converter, otherwise return 0.  */
    if (-1 == ourConverter.return_value) return 0;

    /*  Perform the conversion.  */
    iconvRC = iconv(ourConverter,
                    (char**) &sourceStr, &sStrLen,
                    &targetStr, &tStrLen);

    /*  If the conversion failed, return a zero.  */
    if (0 != iconvRC ) return 0;

    /*  Close the conversion descriptor.  */
    iconv_close(ourConverter);

    /*  The targetStr returns pointing to the character just
     *  past the last converted character, so set the null there now.
     */
    *targetStr = '¥0';

    /*  Return the number of characters that were processed. */
    return originalLen-tStrLen;
}
#pragma convert(0)
```

例: ILE ネイティブ・メソッド・ソース・コードを準備する

Java 用の Integrated Language Environment (ILE) ネイティブ・メソッドの例を実行する前に、例を実行できるようにサーバーを準備する必要があります。 サーバーを準備するには、サーバー上にソース・ファイルを作成する必要があります。

サーバーを準備するには、以下のステップを完成させます。

1. サーバーに、例のファイルを含めるための、次の統合ファイル・システム・ディレクトリーを作成します。たとえば、次の制御言語 (CL) コマンドを使用して、`/ileexample` というディレクトリーを作成します。

   ```
   mkdir '/ileexample'
   ```

2. ステップ 1 で作成したディレクトリーに NativeHello.java および NativeHello.c という名の 2 つの空のストリーム・ファイルを作成します。たとえば、以下の CL コマンドを使用してファイルを作成します。

```
QSH CMD('touch -C 819 /ileexample/NativeHello.java')

QSH CMD('touch -C 819 /ileexample/NativeHello.c')
```

3. 537 ページの『例: NativeHello.java』 で示されている Java ソース・コードを、ステップ 2 で作成した NativeHello.java という名のファイルにコピーします。
4. 538 ページの『例: NativeHello.c』 で示されている C ソース・コードを、ステップ 2 で作成した NativeHello.c という名のファイルにコピーします。
5. Java ネイティブ・メソッド用の C 実装コードを含むサービス・プログラムを入れる ileexample という名のライブラリーを作成します。たとえば、次の CL コマンドを使用してライブラリーを作成します。

```
crtlib ileexample
```

214 ページの『例: Java 用の ILE ネイティブ・メソッド』
この Java 用の Integrated Language Environment (ILE) ネイティブ・メソッドの例では、ネイティブ C メソッドのインスタンスを呼び出します。 次いでそれは Java ネイティブ・インターフェース (JNI) を使用して Java コードにコールバックを行い、Java ストリング変数の値を設定します。 その後、Java ストリング変数は、Java コードによって標準出力に書き込まれます。

537 ページの『例: NativeHello.java』
Java ソース・コードは、Java 用の Integrated Language Environment (ILE) ネイティブ・メソッドの例で使用されます。

538 ページの『例: NativeHello.c』
C ソース・コードは、Java 用の Integrated Language Environment (ILE) ネイティブ・メソッドの例で使用されます。

『例: ILE ネイティブ・メソッド・プログラム・オブジェクトを作成する』
ご使用のサーバー上で Integrated Language Environment (ILE) ネイティブ・メソッドの例を実行する前に、ソース・コードをコンパイルし、C インクルード・ファイルを作成し、ILE サービス・プログラムを作成する必要があります。

例: ILE ネイティブ・メソッド・プログラム・オブジェクトを作成する

ご使用のサーバー上で Integrated Language Environment (ILE) ネイティブ・メソッドの例を実行する前に、ソース・コードをコンパイルし、C インクルード・ファイルを作成し、ILE サービス・プログラムを作成する必要があります。

コンパイルした Java .class ファイルを使用して、C インクルード・ファイル NativeHello.h を作成します。ここには、C ソース・コードに含まれる setTheString() メソッドの関数プロトタイプが含まれています。

サーバーでのソース・コード例を準備するには、以下のステップを完成させます。

注: 以下のステップは、541 ページの『例: ILE ネイティブ・メソッド・ソース・コードを準備する』にリストされたステップを実行した後にのみ、実行する必要があります。

1. コマンド行から、以下のコマンドを使用して、現行作業ディレクトリーを /ileexample に変更します。

```
cd '/ileexample'
```

2. Java ソース・ファイルをコンパイルします。たとえば、次のコマンドを使用して、コマンド行から Java ソース・コードをコンパイルします。

```
QSH CMD('javac NativeHello.java')
```

3. 次のコマンドを使用して、ネイティブ・メソッド・プロトタイプを含む C インクルード・ファイルを作成します。

   ```
   QSH CMD('javah -jni NativeHello')
   ```

   新しい C インクルード・ファイル (NativeHello.h) には、setTheString() メソッドの関数プロトタイプが含まれています。例の C ソース・コード NativeHello.c には、既にインクルード・ファイルが組み込まれています。

4. 次のコマンドを使用して、C ソース・コードをコンパイルし、サービス・プログラムを作成します。

   ```
   CRTCMOD MODULE(ILEEXAMPLE/NATHELLO) SRCSTMF(NativeHello.c) TERASPACE(*YES)

   CRTSRVPGM SRVPGM(ILEEXAMPLE/NATHELLO) MODULE(ILEEXAMPLE/NATHELLO) EXPORT(*ALL)
   ```

   新しいサービス・プログラム (NATHELLO) には、例で使用するネイティブ・メソッド setTheString() が含まれています。

214 ページの『例: Java 用の ILE ネイティブ・メソッド』
この Java 用の Integrated Language Environment (ILE) ネイティブ・メソッドの例では、ネイティブ C メソッドのインスタンスを呼び出します。 次いでそれは Java ネイティブ・インターフェース (JNI) を使用して Java コードにコールバックを行い、Java ストリング変数の値を設定します。 その後、Java ストリング変数は、Java コードによって標準出力に書き込まれます。

537 ページの『例: NativeHello.java』
Java ソース・コードは、Java 用の Integrated Language Environment (ILE) ネイティブ・メソッドの例で使用されます。

538 ページの『例: NativeHello.c』
C ソース・コードは、Java 用の Integrated Language Environment (ILE) ネイティブ・メソッドの例で使用されます。

541 ページの『例: ILE ネイティブ・メソッド・ソース・コードを準備する』
Java 用の Integrated Language Environment (ILE) ネイティブ・メソッドの例を実行する前に、例を実行できるようにサーバーを準備する必要があります。 サーバーを準備するには、サーバー上にソース・ファイルを作成する必要があります。

例: プロセス間通信のためにソケットを使用する

この例では、ソケットを使用して Java プログラムと C プログラムとの間で通信します。

最初に、ソケット上で聴取する C プログラムを開始してください。 Java プログラムがソケットに接続された後は、ソケット接続を使用して C プログラムがそれにストリングを送ります。 C プログラムから送られるストリングは、コード・ページ 819 の ASCII コードのストリングです。

Qshell インタープリターのコマンド行か、または他の Java プラットフォームで、コマンド java TalkToC xxxxx nnnn を使用して Java プログラムを開始しなくてはなりません。または IBM i コマンド行に JAVA TALKTOC PARM(xxxxx nnnn) を入力することにより、Java プログラムを開始します。 xxxxx は、C プログラムが実行されているシステムのドメイン・ネームまたはインターネット・プロトコル (IP) アドレスです。 nnnn は、C プログラムが使用するソケットのポート番号です。このポート番号は、C プログラムを呼び出すときに最初に渡すパラメーターとして指定する必要があります。

TalkToC クライアント Java クラスのソース・コード

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.net.*;
import java.io.*;

class TalkToC
{
   private String host = null;
   private int port = -999;
   private Socket socket = null;
   private BufferedReader inStream = null;

   public static void main(String[] args)
   {
      TalkToC caller = new TalkToC();
      caller.host = args[0];
      caller.port = new Integer(args[1]).intValue();
      caller.setUp();
      caller.converse();
      caller.cleanUp();
   }

   public void setUp()
   {
      System.out.println("TalkToC.setUp() invoked");

      try
      {
         socket = new Socket(host, port);
         inStream = new BufferedReader(new InputStreamReader(
                                    socket.getInputStream()));
      }
      catch(UnknownHostException e)
      {
         System.err.println("Cannot find host called: " + host);
         e.printStackTrace();
         System.exit(-1);
      }
      catch(IOException e)
      {
         System.err.println("Could not establish connection for " + host);
         e.printStackTrace();
         System.exit(-1);
      }
   }

   public void converse()
   {
      System.out.println("TalkToC.converse() invoked");

      if (socket != null && inStream != null)
      {
         try
         {
            System.out.println(inStream.readLine());
         }
         catch(IOException e)
         {
            System.err.println("Conversation error with host " + host);
            e.printStackTrace();
         }
      }
   }

   public void cleanUp()
   {
```

```
      try
      {
         if (inStream != null)
            inStream.close();
         if (socket != null)
            socket.close();
      }
      catch(IOException e)
      {
         System.err.println("Error in cleanup");
         e.printStackTrace();
         System.exit(-1);
      }
   }
}
```

SockServ.C は、ポート番号のパラメーターを渡すことによって開始します。たとえば、CALL SockServ '2001' とします。

SockServ.C サーバー・プログラムのソース・コード

注: 法律上の重要な情報に関しては、コードの特記事項情報をお読みください。

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <unistd.h>
#include <sys/time.h>

void main(int argc, char* argv[])
{
   int    portNum = atoi(argv[1]);
   int    server;
   int    client;
   int    address_len;
   int    sendrc;
   int    bndrc;
   char*  greeting;
   struct sockaddr_in  local_Address;
   address_len = sizeof(local_Address);

   memset(&local_Address,0x00,sizeof(local_Address));
   local_Address.sin_family = AF_INET;
   local_Address.sin_port = htons(portNum);
   local_Address.sin_addr.s_addr = htonl(INADDR_ANY);

   #pragma convert (819)
   greeting = "This is a message from the C socket server.";
   #pragma convert (0)

   /*  allocate socket    */
   if((server = socket(AF_INET, SOCK_STREAM, 0))<0)
   {
      printf("failure on socket allocation¥n");
      perror(NULL);
      exit(-1);
   }

   /* do bind   */
   if((bndrc=bind(server,(struct sockaddr*)&local_Address, address_len))<0)
   {
     printf("Bind failed¥n");
```

```
    perror(NULL);
    exit(-1);
  }

  /* invoke listen   */
  listen(server, 1);

  /* wait for client request */
  if((client = accept(server,(struct sockaddr*)NULL, 0))<0)
  {
    printf("accept failed¥n");
    perror(NULL);
    exit(-1);
  }

  /* send greeting to client    */
  if((sendrc = send(client, greeting, strlen(greeting),0))<0)
  {
     printf("Send failed¥n");
     perror(NULL);
     exit(-1);
  }

  close(client);
  close(server);
}
```

例: SQL ステートメントを Java アプリケーションに組み込む

以下の SQLJ アプリケーション例、App.sqlj は、静的 SQL を使用して更新データを DB2 サンプル・データベースの EMPLOYEE テーブルから検索します。

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
import java.sql.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

#sql iterator App_Cursor1 (String empno, String firstnme) ; // 1
#sql iterator App_Cursor2 (String) ;

class App
{

  /**********************
   **  Register Driver **
   **********************/

 static
 {
   try
   {
     Class.forName("com.ibm.db2.jdbc.app.DB2Driver").newInstance();
   }
   catch (Exception e)
   {
     e.printStackTrace();
   }
 }

  /********************
   **      Main      **
   ********************/

 public static void main(String argv[])
```

```
{
  try
  {
    App_Cursor1 cursor1;
    App_Cursor2 cursor2;

    String str1 = null;
    String str2 = null;
    long   count1;

    // URL is jdbc:db2:dbname
    String url = "jdbc:db2:sample";

    DefaultContext ctx = DefaultContext.getDefaultContext();
    if (ctx == null)
    {
      try
      {
        // connect with default id/password
        Connection con = DriverManager.getConnection(url);
        con.setAutoCommit(false);
        ctx = new DefaultContext(con);
      }
      catch (SQLException e)
      {
        System.out.println("Error: could not get a default context");
        System.err.println(e) ;
        System.exit(1);
      }
      DefaultContext.setDefaultContext(ctx);
    }

    // retrieve data from the database
    System.out.println("Retrieve some data from the database.");
    #sql cursor1 = {SELECT empno, firstnme FROM employee}; // 2

    // display the result set
    // cursor1.next() returns false when there are no more rows
    System.out.println("Received results:");
    while (cursor1.next()) // 3
    {
      str1 = cursor1.empno(); // 4
      str2 = cursor1.firstnme();

      System.out.print (" empno= " + str1);
      System.out.print (" firstname= " + str2);
      System.out.println("");
    }
    cursor1.close(); // 9

    // retrieve number of employee from the database
    #sql { SELECT count(*) into :count1 FROM employee }; // 5
    if (1 == count1)
      System.out.println ("There is 1 row in employee table");
    else
      System.out.println ("There are " + count1
                         + " rows in employee table");

    // update the database
    System.out.println("Update the database.");
    #sql { UPDATE employee SET firstnme = 'SHILI' WHERE empno = '000010' };

    // retrieve the updated data from the database
    System.out.println("Retrieve the updated data from the database.");
    str1 = "000010";
    #sql cursor2 = {SELECT firstnme FROM employee WHERE empno = :str1}; // 6
```

```
      // display the result set
      // cursor2.next() returns false when there are no more rows
      System.out.println("Received results:");
      while (true)
      {
        #sql { FETCH :cursor2 INTO :str2 }; // 7
        if (cursor2.endFetch()) break; // 8

        System.out.print (" empno= " + str1);
        System.out.print (" firstname= " + str2);
        System.out.println("");
      }
      cursor2.close(); // 9

      // rollback the update
      System.out.println("Rollback the update.");
      #sql { ROLLBACK work };
      System.out.println("Rollback done.");
    }
    catch( Exception e )
    {
      e.printStackTrace();
    }
  }
}
```

[1] 反復子を宣言する。このセクションでは、次の 2 種類の反復子を宣言します。

- App_Cursor1: 列データのタイプおよび名前を宣言して、列名 (列に結び付けられた名前) に応じた列の値を戻します。
- App_Cursor2: 列データのタイプを宣言して、列位置 (列に結び付けられた定位置) に応じた列の値を戻します。

[2] 反復子を初期設定する。反復子オブジェクト cursor1 が照会の結果を使用して初期設定されます。照会は結果を cursor1 に格納します。

[3] 反復子を次の行に進める。 cursor1.next() メソッドは、検索する行がなくなった場合にブール値の偽を戻します。

[4] データを移動する。名前付きアクセス機構メソッド empno() は、現在の行にある empno という名前の列の値を戻します。名前付きアクセス機構メソッド firstnme() は、現在の行にある firstnme() という名前の列の値を戻します。

[5] データをホスト変数に SELECT する。 SELECT ステートメントは、テーブル内の行数をホスト変数 count1 に渡します。

[6] 反復子を初期設定する。反復子オブジェクト cursor2 が照会の結果を使用して初期設定されます。照会は結果を cursor2 に格納します。

[7] データを検索する。 FETCH ステートメントは、結果テーブルから ByPos カーソル内で宣言された最初の列の現行値を、ホスト変数 str2 に戻します。

[8] FETCH.INTO ステートメントが成功したかを検査する。 endFetch() メソッドは、反復子が行に位置していない場合、つまり行を取り出す前回の試行が失敗した場合に、ブール値の真を戻します。 endFetch() メソッドは、行を取り出す前回の試行が成功した場合に、偽を戻します。 DB2 は next() メソッドが呼び出されたときに行の取り出しを試行します。 FETCH...INTO ステートメントは、暗黙的に next() メソッドを呼び出します。

[9] 反復子をクローズする。 close() メソッドは、反復子が保持しているリソースを解放します。反復子を明示的にクローズして、システム・リソースが適時に解放されるようにしてください。

例: クライアントのソケット・ファクトリーを使用するように Java コードを変更する

以下の例は、simpleSocketClient という単純なソケット・クラスを変更し、ソケット・ファクトリーを使用してすべてのソケットを作成できるようにする方法を示しています。 1 つ目の例は、ソケット・ファクトリーのない simpleSocketClient クラスを示しています。 2 つ目の例は、ソケット・ファクトリーのある simpleSocketClient クラスを示しています。 2 つ目の例では、simpleSocketClient が factorySocketClient に名前変更されています。

例 1: ソケット・ファクトリーのないソケット・クライアント・プログラム

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
/* Simple Socket Client Program */

import java.net.*;
import java.io.*;

public class simpleSocketClient {
  public static void main (String args[]) throws IOException {

    int serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java simpleSocketClient serverHost serverPort");
      System.out.println("serverPort defaults to 3000 if not specified.");
      return;
    }
    if (args.length == 2)
      serverPort = new Integer(args[1]).intValue();

    System.out.println("Connecting to host " + args[0] + " at port " +
                       serverPort);

    // Create the socket and connect to the server.
    Socket  s = new Socket(args[0], serverPort);
    .
    .
    .

    // The rest of the program continues on from here.
```

例 2: ソケット・ファクトリーのある単純なソケット・クライアント・プログラム

```
/* Simple Socket Factory Client Program */

// Notice that javax.net.* is imported to pick up the SocketFactory class.
import javax.net.*;
import java.net.*;
import java.io.*;

public class factorySocketClient {
  public static void main (String args[]) throws IOException {

    int serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java factorySocketClient serverHost serverPort");
      System.out.println("serverPort defaults to 3000 if not specified.");
```

```
      return;
    }
    if (args.length == 2)
      serverPort = new Integer(args[1]).intValue();

    System.out.println("Connecting to host " + args[0] + " at port " +
                       serverPort);

    // Change the original simpleSocketClient program to create a
    // SocketFactory and then use the socket factory to create sockets.

    SocketFactory socketFactory = SocketFactory.getDefault();

    // Now the factory creates the socket.  This is the last change
    // to the original simpleSocketClient program.

    Socket  s = socketFactory.createSocket(args[0], serverPort);
    .
    .
    .

    // The rest of the program continues on from here.
```

例: サーバーのソケット・ファクトリーを使用するように Java コードを変更する

以下の例は、simpleSocketServer という単純なソケット・クラスを変更し、ソケット・ファクトリーを使用してすべてのソケットを作成できるようにする方法を示しています。 1 つ目の例は、ソケット・ファクトリーのない simpleSocketServer クラスを示しています。 2 つ目の例は、ソケット・ファクトリーのある simpleSocketServer クラスを示しています。 2 つ目の例では、simpleSocketServer が factorySocketServer に名前変更されています。

例 1: ソケット・ファクトリーのないソケット・サーバー・プログラム

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
/* File simpleSocketServer.java*/

import java.net.*;
import java.io.*;

public class simpleSocketServer {
  public static void main (String args[]) throws IOException {

    int serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java simpleSocketServer serverPort");
      System.out.println("Defaulting to port 3000 since serverPort not specified.");
    }
    else
        serverPort = new Integer(args[0]).intValue();

    System.out.println("Establishing server socket at port " + serverPort);

    ServerSocket    serverSocket =
      new ServerSocket(serverPort);

    // a real server would handle more than just one client like this...

    Socket s = serverSocket.accept();
    BufferedInputStream is = new BufferedInputStream(s.getInputStream());
```

```
    BufferedOutputStream os = new BufferedOutputStream(s.getOutputStream());

    // This server just echoes back what you send it...

    byte buffer[] = new byte[4096];

    int bytesRead;

    // read until "eof" returned
    while ((bytesRead = is.read(buffer)) > 0) {
      os.write(buffer, 0, bytesRead); // write it back
      os.flush();    // flush the output buffer
    }


    s.close();
    serverSocket.close();
  }       // end main()

}       // end class definition
```

例 2: ソケット・ファクトリーのある単純なソケット・サーバー・プログラム

```
/* File factorySocketServer.java */

// need to import javax.net to pick up the ServerSocketFactory class
import javax.net.*;
import java.net.*;
import java.io.*;

public class factorySocketServer {
  public static void main (String args[]) throws IOException {

    int serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java simpleSocketServer serverPort");
      System.out.println("Defaulting to port 3000 since serverPort not specified.");
    }
    else
        serverPort = new Integer(args[0]).intValue();

    System.out.println("Establishing server socket at port " + serverPort);

    // Change the original simpleSocketServer to use a
    // ServerSocketFactory to create server sockets.
    ServerSocketFactory serverSocketFactory =
      ServerSocketFactory.getDefault();
    // Now have the factory create the server socket.  This is the last
    // change from the original program.
    ServerSocket    serverSocket =
      serverSocketFactory.createServerSocket(serverPort);

    // a real server would handle more than just one client like this...

    Socket s = serverSocket.accept();
    BufferedInputStream is = new BufferedInputStream(s.getInputStream());
    BufferedOutputStream os = new BufferedOutputStream(s.getOutputStream());

    // This server just echoes back what you send it...

    byte buffer[] = new byte[4096];

    int bytesRead;

    while ((bytesRead = is.read(buffer)) > 0) {
      os.write(buffer, 0, bytesRead);
```

```
      os.flush();
    }


    s.close();
    serverSocket.close();
  }

}
```

例: Secure Sockets Layer を使用するように Java クライアントを変更する

以下の例は、factorySocketClient という 1 つのクラスを変更して、Secure Sockets Layer (SSL) を使用できるようにする方法を示しています。 1 つ目の例は、SSL を使用しない factorySocketClient クラスを示しています。 2 つ目の例は、同じクラスで SSL を使用するものを示しており、名前が factorySSLSocketClient に変更されています。

例 1: SSL を使用しない単純な factorySocketClient クラス

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
/* Simple Socket Factory Client Program */

import javax.net.*;
import java.net.*;
import java.io.*;

public class factorySocketClient {
  public static void main (String args[]) throws IOException {

    int serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java factorySocketClient serverHost serverPort");
      System.out.println("serverPort defaults to 3000 if not specified.");
      return;
    }
    if (args.length == 2)
      serverPort = new Integer(args[1]).intValue();

    System.out.println("Connecting to host " + args[0] + " at port " +
                       serverPort);

    SocketFactory socketFactory = SocketFactory.getDefault();

    Socket  s = socketFactory.createSocket(args[0], serverPort);
    .
    .
    .

    // The rest of the program continues on from here.
```

例 2: SSL を使用する単純な factorySocketClient クラス

```
// Notice that we import javax.net.ssl.* to pick up SSL support
import javax.net.ssl.*;
import javax.net.*;
import java.net.*;
import java.io.*;

public class factorySSLSocketClient {
  public static void main (String args[]) throws IOException {
```

```
    int serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java factorySSLSocketClient serverHost serverPort");
      System.out.println("serverPort defaults to 3000 if not specified.");
      return;
    }
    if (args.length == 2)
      serverPort = new Integer(args[1]).intValue();

    System.out.println("Connecting to host " + args[0] + " at port " +
                       serverPort);

    // Change this to create an SSLSocketFactory instead of a SocketFactory.
    SocketFactory socketFactory = SSLSocketFactory.getDefault();

    // We do not need to change anything else.
    // That's the beauty of using factories!
    Socket  s = socketFactory.createSocket(args[0], serverPort);
    .
    .
    .

    // The rest of the program continues on from here.
```

例: Secure Sockets Layer を使用するように Java サーバーを変更する

以下の例は、factorySocketServer という 1 つのクラスを変更して、Secure Sockets Layer (SSL) を使用できるようにする方法を示しています。

1 つ目の例は、SSL を使用しない factorySocketServer クラスを示しています。 2 つ目の例は、同じクラスで SSL を使用するものを示しており、名前が factorySSLSocketServer に変更されています。

例 1: SSL を使用しない単純な factorySocketServer クラス

注: サンプル・コードをご使用の場合は、 560 ページの『コードに関するライセンス情報および特記事項』に同意していただいているものとします。

```
/* File factorySocketServer.java */
// need to import javax.net to pick up the ServerSocketFactory class
import javax.net.*;
import java.net.*;
import java.io.*;

public class factorySocketServer {
  public static void main (String args[]) throws IOException {

    int serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java simpleSocketServer serverPort");
      System.out.println("Defaulting to port 3000 since serverPort not specified.");
    }
    else
        serverPort = new Integer(args[0]).intValue();

    System.out.println("Establishing server socket at port " + serverPort);

    // Change the original simpleSocketServer to use a
    // ServerSocketFactory to create server sockets.
    ServerSocketFactory serverSocketFactory =
      ServerSocketFactory.getDefault();
    // Now have the factory create the server socket.  This is the last
```

```
    // change from the original program.
    ServerSocket    serverSocket =
      serverSocketFactory.createServerSocket(serverPort);

    // a real server would handle more than just one client like this...

    Socket s = serverSocket.accept();
    BufferedInputStream is = new BufferedInputStream(s.getInputStream());
    BufferedOutputStream os = new BufferedOutputStream(s.getOutputStream());

    // This server just echoes back what you send it.

    byte buffer[] = new byte[4096];

    int bytesRead;

    while ((bytesRead = is.read(buffer)) > 0) {
      os.write(buffer, 0, bytesRead);
      os.flush();
    }

    s.close();
    serverSocket.close();
  }
}
```

例 2: SSL を使用する単純な factorySocketServer クラス

```
/* File factorySocketServer.java */

// need to import javax.net to pick up the ServerSocketFactory class
import javax.net.*;
import java.net.*;
import java.io.*;

public class factorySocketServer {
  public static void main (String args[]) throws IOException {

    int serverPort = 3000;

    if (args.length < 1) {
      System.out.println("java simpleSocketServer serverPort");
      System.out.println("Defaulting to port 3000 since serverPort not specified.");
    }
    else
        serverPort = new Integer(args[0]).intValue();

    System.out.println("Establishing server socket at port " + serverPort);

    // Change the original simpleSocketServer to use a
    // ServerSocketFactory to create server sockets.
    ServerSocketFactory serverSocketFactory =
      ServerSocketFactory.getDefault();
    // Now have the factory create the server socket.  This is the last
    // change from the original program.
    ServerSocket    serverSocket =
      serverSocketFactory.createServerSocket(serverPort);

    // a real server would handle more than just one client like this...

    Socket s = serverSocket.accept();
    BufferedInputStream is = new BufferedInputStream(s.getInputStream());
    BufferedOutputStream os = new BufferedOutputStream(s.getOutputStream());

    // This server just echoes back what you send it.

    byte buffer[] = new byte[4096];
```

```
    int bytesRead;

    while ((bytesRead = is.read(buffer)) > 0) {
      os.write(buffer, 0, bytesRead);
      os.flush();
    }

    s.close();
    serverSocket.close();
  }
}
```

Java プログラムのトラブルシューティング

このトピックでは、ジョブ・ログを検索する方法と、Java プログラムの分析データを収集する方法を説明します。ここでは、プログラム一時修正 (PTF) の説明や、IBM Developer Kit for Java のサポートを受ける方法も示します。

プログラムの実行時間が長くなるとパフォーマンスが低下する場合は、誤ってメモリー・リークがコーディングされている可能性があります。 IBM iDoctor のコンポーネントである JavaWatcher を使用して、プログラムをデバッグし、メモリー・リークを見つけることができます。詳細は、『Heap Analysis Tools for Java』を参照してください。

制限

ここでは、IBM i サーバー上の Java での既知の制限、制約事項、または固有の動作をリストします。

- IBM i での java.net バックログ・パラメーターの動きは、他のプラットフォームと異なる場合があります。以下に例を示します。
  - Listen バックログ 0、1
    - Listen(0) と指定すると、接続を 1 つ保留にすることができます。ソケットは使用禁止になりません。
    - Listen(1) と指定すると、Listen(0) と同じ効果があるほか、接続を 1 つ保留にすることができます。
  - Listen バックログ > 1
    - これにより、listen 待ち行列に、保留中の多数の要求を残すことが可能になります。新しい接続要求が到着し、待ち行列が限界に達すると、保留中の要求が 1 つ削除されます。
- マルチスレッドを使用できる (つまり、 スレッド・セーフな) 環境では、使用する JDK のバージョンに関係なく、1 つの Java 仮想マシンだけを使用できます。 IBM i プラットフォームはスレッド・セーフですが、ファイル・システムの中にはそうではないものもあります。スレッド・セーフではないファイル・システムのリストは、『統合ファイル・システム』のトピックを参照してください。

Java の問題分析用のジョブ・ログを検索する

Java コマンドを実行したジョブのジョブ・ログおよび Java プログラムが実行されたバッチ即時 (BCI) ジョブ・ログを使用して、Java の障害の原因を分析してください。どちらのジョブ・ログにも重要なエラー情報が含まれている可能性があります。

BCI ジョブのジョブ・ログを検索するには、2 つの方法があります。 Java コマンドを実行したジョブのジョブ・ログに利用記録がとられている BCI ジョブの名前を、検索することができます。そして、そのジョブ名を使用して、BCI ジョブのジョブ・ログを検索します。

また、以下のステップに従って、BCI ジョブのジョブ・ログを検索することができます。

1. IBM i コマンド行に「投入されたジョブの処理 (WRKSBMJOB)」コマンドを入力します。
2. リストの一番後ろへ移動します。
3. リスト中から、QJVACMDSRV という最後のジョブを見つけます。
4. そのジョブに対して、オプション 8 (スプール・ファイルの処理) を入力します。
5. QPJOBLOG というファイルが、表示されます。
6. F11 を押して、スプール・ファイルのビュー 2 を参照します。
7. 日時が、障害が発生したときの日時と一致するかを確認します。

   日時がサインオフした日時と一致しない場合は、投入されたジョブのリストをさらに調べてください。サインオフした日時に一致する日時がある QJVACMDSRV ジョブ・ログを検索してください。

BCI ジョブのジョブ・ログを見つけられない場合は、そのジョブ・ログは作成されなかった可能性があります。これは、QDFTJOBD ジョブ記述の ENDSEP 値の設定が高すぎる場合、または QDFTJOBD ジョブ記述の LOG 値が *NOLIST を指定する場合に起こります。これらの値をチェックして、BCI ジョブのジョブ・ログが作成されるようにその値を変更してください。

「Java プログラムの実行 (RUNJVA)」コマンドを実行したジョブのジョブ・ログを作成するには、以下のことを行ってください。

1. SIGNOFF *LIST を入力します。
2. 次に、再びサインオンします。
3. IBM i コマンド行に「スプール・ファイルの処理 (WRKSPLF)」コマンドを入力します。
4. リストの一番後ろへ移動します。
5. QPJOBLOG というファイルを検索します。
6. F11 を押します。
7. 日時が、サインオフ・コマンドを入力した日時と一致することを確認します。

   日時がサインオフした日時と一致しない場合は、投入されたジョブのリストをさらに調べてください。サインオフした日時に一致する日時がある QJVACMDSRV ジョブ・ログを検索してください。

Java の問題分析用のデータを収集する

プログラム診断依頼書 (APAR) に記載するデータを収集するには、以下のことを行います。

1. 問題の完全な記述を含めます。
2. 実行中に問題の原因となった Java クラス・ファイルを保管します。
3. SAV コマンドを使用して、統合ファイル・システムからオブジェクトを保管します。このプログラムの実行に必要な他のクラス・ファイルを保管しなければならない場合があります。また、必要であれば、IBM が問題を再現するときに使用するディレクトリー全体を保管して送ることもできます。以下に、ディレクトリー全体を保管する方法の例を示します。

   例: ディレクトリーを保管する

   注: 法律上の重要な情報に関しては、 コードの特記事項情報をお読みください。

   ```
   SAV DEV('/QSYS.LIB/TAP01.DEVD') OBJ(('/mydir'))
   ```

   可能であれば、問題に関連している Java クラスのソース・ファイルを保管します。これは、IBM が問題を再現して分析するときに役立ちます。

4. プログラムの実行に必要なネイティブ・メソッドを含む、すべてのサービス・プログラムを保管します。
5. Java プログラムの実行に必要な、すべてのデータ・ファイルを保管します。
6. 問題を再現する方法についての完全な記述を追加します。

   これには次のものが含まれます。
   - CLASSPATH 環境変数の値。
   - 実行された Java コマンドの記述。
   - プログラムによって要求されるどんな入力にでも応答する方法の記述。
7. 障害が起きたころに発生したすべての垂直ライセンス内部コード (VLIC) ログを含めます (特にメジャー・コード 4700 または 4300 が示された場合)。
8. Java 仮想マシンが実行していた対話式ジョブおよび BCI ジョブからジョブ・ログを追加します。
9. IBM Center for Java Technology Developer 診断ガイド (英語) で要求された情報を追加します。

プログラム一時修正を適用する

Java トピックの中には、IBM Developer Kit for Java の最新レベルが IBM i サーバー上にロードされていることを前提とする情報が含まれているものもあります。サーバー上の Java を最新レベルにするためには、最新の Java プログラム一時修正 (PTF) グループをロードします。

Java PTF および Java に影響を与える PTF は、PTF グループの一部として定期的にパッケージ化されます。 PTF グループは、これらの PTF を 1 つのエンティティーとして管理するために定義された PTF のリストから構成されています。新しいレベルの Java PTF グループは、1 年間に複数回、リリースされる場合があります。最新の PTF をインストールして、IBM Developer Kit for Java の最新レベルにアップグレードすることをお勧めします。 Java PTF グループ番号および入手可能な最新のグループ・レベルについては、Preventive Service Planning - PSP の Web ページを参照してください。

注: 即時に適用できる Java PTF は、ジョブ内で稼働中の Java 仮想マシン (JVM) に影響を与えない可能性があります。ただし、場合によっては、PTF を適用すると、予測できない結果が発生することがあります。「Java 仮想マシン・ジョブ表示」(DSPJVMJOB) CL コマンドを使用して、システムがアクティブである間も JVM ジョブを管理して PTF を適用できます。 DSPJVMJOB コマンドは、どのジョブで JVM が実行中かを示します。 PTF を適用するために初期プログラム・ロード (IPL) を待つ代わりに、この情報を使用して、PTF を適用する前にアクティブな JVM を含むジョブを終了させることができます。

関連情報

IBM i および関連ソフトウェアの保守管理

ソフトウェア修正の使用

IBM i での Java サポートの利用

IBM i Java 用のサポート・サービスは、IBM i ソフトウェア・プロダクトの通常の期間と条件のもとで提供されています。サポート・サービスには、プログラム・サービス、音声サポート、およびコンサルティング・サービスが含まれます。

詳細については、IBM i ホーム・ページのトピック「Support」で提供されているオンライン情報を使用してください。ライセンス・プログラム 5761-JV1 の IBM サポート・サービスを使用してください。または、ローカル IBM 担当員に連絡してください。

継続してプログラム・サービスを受けるには、IBM の指示により、より最近のレベルのライセンス・プログラムが必要になることがあります。詳細については、複数の Java Development Kit (JDK) のサポートを参照してください。

欠陥の解決は、プログラム・サービスまたは音声サポートによってサポートされています。アプリケーションのプログラミングまたはデバッグに関する問題の解決は、コンサルティング・サービスによってサポートされています。

Java アプリケーション・プログラム・インターフェース (API) 呼び出しは、以下の場合を除いてコンサルティング・サービスによりサポートされています。

1. 比較的単純なプログラムで再び発生することにより示されるような、明らかに Java API の欠陥である場合。
2. 資料の説明を求める質問である場合。
3. サンプルまたは資料の入手先についての質問の場合。

プログラミングに関するすべてのサポートは、コンサルティング・サービスにより提供されます。製品に付随するプログラム・サンプルもそのサービスに含まれます。追加のサンプルは、インターネット上の IBM i ホーム・ページで入手できる場合もありますが、これらはサポート対象外です。

IBM Developer Kit for Java LP により、問題の解決に関する情報が提供されます。

関連情報

IBM i サーバー上の Java に関連のある情報ソースを以下に紹介します。

Web サイト

- Java.sun.com: The Source for Java Developers (www.java.sun.com)

  Java の様々な使用法や新しいテクノロジーについては、Sun Microsystems, Inc. のサイトをご覧ください。

- IBM developerWorks Java technology zone

  Java、IBM 製品、およびビジネス・ソリューションを作成するための他のテクノロジーを使用する際に役立つ情報、学習資料、およびツールを提供しています。

- IBM alphaWorks® Java

  新しい Java テクノロジーについて紹介しています。開発リソースのダウンロードやリンクがあります。

Javadoc

Java クラス用の Javadoc 参照情報については、Sun Microsystems, Inc. による『Java 2 Platform, Standard Edition API Specification』を参照してください。

IBM i サーバー上の Java に関連する以下の参照情報を参照してください。

Java Naming and Directory Interface

Java Naming and Directory Interface (JNDI) は、JavaSoft のプラットフォーム・アプリケーション・プログラミング・インターフェース (API) の一部です。 JNDI により、複数の命名およびディレクトリー・サービスにシームレスに接続することができます。このインターフェースを使用すると、強力で可搬性のある、ディレクトリーが使用可能な Java アプリケーションを作成することができます。

JavaSoft は、IBM、SunSoft、Novell、Netscape、および Hewlett-Packard Co など、業界のリーダー企業と共同で JNDI の仕様を開発しました。

注: IBM Developer Kit for Java で提供されている IBM i Java ランタイム環境 (JRE) および Java 2 Platform, Standard Edition (J2SE) のバージョンには、Sun LDAP プロバイダーが組み込まれています。IBM i Java サポートには Sun LDAP プロバイダーが組み込まれているため、このサポートには ibmjndi.jar ファイルは組み込まれていません。 ibmjndi.jar ファイルは、 旧バージョンの J2SDK 用の IBM 開発 LDAP サービス・プロバイダーを提供していました。

Java Naming and Directory interface by Sun Microsystems, Inc.

JavaMail

JavaMail API は、 電子メール (E メール) システムをモデル化する抽象クラスのセットを提供します。この API はメールの読み取りおよび送信に関する一般的なメール機能を提供します。サービス・プロバイダーはプロトコルをインプリメントしなければなりません。

サービス・プロバイダーは、特定のプロトコルをインプリメントします。たとえば、Simple Mail Transfer Protocol (SMTP) は E メールの送信用の転送プロトコルです。 Post Office Protocol 3 (POP3) は E メールの受信用の標準プロトコルです。 Internet Message Access Protocol (IMAP) は POP3 の代替プロトコルです。

JavaMail は、プレーン・テキストではないメールの内容を処理するために、サービス・プロバイダーのほかに JavaBeans Activation Framework (JAF) を必要とします。これには、Multipurpose Internet Mail Extensions (MIME)、Uniform Resource Locator (URL) ページ、およびファイルの添付が含まれます。

すべての JavaMail コンポーネントが、SS1 (製品 ID 5770-SS1) オプション 3 のパーツとして配送されます。これらのコンポーネントには、以下のものが含まれます。

- **mail.jar** この JAR ファイルには、JavaMail API、SMTP サービス・プロバイダー、POP3 サービス・プロバイダー、および IMAP サービス・プロバイダーが含まれます。
- **activation.jar** この JAR ファイルには、JavaBeans Activation Framework が含まれています。

JavaMail

Java 印刷サービス

Java 印刷サービス (JPS) API はすべての Java プラットフォームで印刷ができるようにします。 Java 1.4 および後続のバージョンは、Java ランタイム環境およびサード・パーティーが、PDF、Postscript、および高機能印刷 (AFP) など印刷のためのさまざまなフォーマットを作成するストリーム生成プラグインを提供できるように、フレームワークを提供します。これらのプラグインは 2 次元 (2D) グラフィック・コールから出力フォーマットを作成します。

IBM i 印刷サービスは、IBM i の「デバイス記述作成 (プリンター)」(CRTDEVPRT) コマンドを使用して構成された印刷装置を表示します。印刷装置を作成する際は、情報公開パラメーターを指定してください。それにより、IBM i 印刷サービスがサポートする印刷サービス属性の数が増加します。

プリンターが Simple Network Management Protocol (SNMP) をサポートしている場合は、サーバー上にプリンターを構成します。CRTDEVPRT コマンドのシステム・ドライバー・プログラム・パラメーターの値として *IBMSNMPDRV を指定してください。印刷サービスは SNMP を使用して、構成されたプリンターについての特定の情報 (プリンター・サービス属性) を検索します。

IBM i がサポートする Doc Flavor には、*AFPDS、*SCS、*USERASCII - (PCL)、*USERASCII - (ポストスクリプト)、および *USERASCII - (PDF) が含まれます。CRTDEVPRT コマンドの「情報公開 (Publishing Information)」の中の「サポートされるデータ・ストリーム (Data Streams Supported)」パラメーターに、プリンターがサポートする Doc Flavor を指定してください。

アプリケーションが印刷サービスを使用して IBM iサーバー上のジョブ (文書) を印刷すると、印刷サービスはその文書をスプール・ファイル内のその印刷装置と同名の (また、PrinterName 属性で指定された名前と同名の) 出力待ち行列に置きます。文書が印刷装置で印刷される前に、コマンド STRPRTWTR で印刷装置書き出しプログラムを始動してください。

Java 印刷サービス仕様で定義された属性に加えて、IBM i 印刷サービスは、すべての Doc Flavor の以下の属性をサポートします。

- PrinterFile (スプール・ファイルの作成時に使用されるプリンター・ファイル、名前、およびライブラリーを指定します)
- SaveSpooledFile (スプール・ファイルを保管するかどうかを指定します)
- UserData (10 文字のユーザー定義データのストリング)
- JobHold (スプール・ファイルを保持するかどうかを指定します)
- SourceDrawer (出力メディア用に使用するソース・ドロワーを指定します)

JPS を使用できるようにする方法

| Java 印刷サービスを使用可能にするためには、以下の JAR ファイルがクラスパスに追加されていることを確認してください。

- | `/QIBM/ProdData/OS400/jt400/lib/jt400Native.jar`
- | `/QIBM/ProdData/OS400/Java400/ext/ibmjps.jar`

関連情報

Sun Microsystems, Inc. による「Java Print Service」

コードに関するライセンス情報および特記事項

IBM は、お客様に、すべてのプログラム・コードのサンプルを使用することができる非独占的な著作使用権を許諾します。お客様は、このサンプル・コードから、お客様独自の特別のニーズに合わせた類似のプログラムを作成することができます。

強行法規で除外を禁止されている場合を除き、IBM、そのプログラム開発者、および供給者は「プログラム」および「プログラム」に対する技術的サポートがある場合にはその技術的サポートについて、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任を負わないものとします。

いかなる場合においても、IBM および IBM のサプライヤーならびに IBM ビジネス・パートナーは、その予見の有無を問わず発生した以下のものについて賠償責任を負いません。

1. データの喪失、または損傷。
2. 直接損害、特別損害、付随的損害、間接損害、または経済上の結果的損害
3. 逸失した利益、ビジネス上の収益、あるいは節約すべかりし費用

国または地域によっては、法律の強行規定により、上記の責任の制限が適用されない場合があります。

付録. 特記事項

本書は米国 IBM が提供する製品およびサービスについて作成したものです。

本書に記載の製品、サービス、または機能が日本においては提供されていない場合があります。日本で利用可能な製品、サービス、および機能については、日本 IBM の営業担当員にお尋ねください。本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、またはサービスのみが使用可能であることを意味するものではありません。これらに代えて、IBM の知的所有権を侵害することのない、機能的に同等の製品、プログラム、またはサービスを使用することができます。ただし、IBM 以外の製品とプログラムの操作またはサービスの評価および検証は、お客様の責任で行っていただきます。

IBM は、本書に記載されている内容に関して特許権 (特許出願中のものを含む) を保有している場合があります。本書の提供は、お客様にこれらの特許権について実施権を許諾することを意味するものではありません。実施権についてのお問い合わせは、書面にて下記宛先にお送りください。

〒242-8502
神奈川県大和市下鶴間1623番14号
日本アイ・ビー・エム株式会社
法務・知的財産
知的財産権ライセンス渉外

以下の保証は、国または地域の法律に沿わない場合は、適用されません。 IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態で提供し、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任を負わないものとします。国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

この情報には、技術的に不適切な記述や誤植を含む場合があります。本書は定期的に見直され、必要な変更は本書の次版に組み込まれます。 IBM は予告なしに、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するものではありません。それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様の責任でご使用ください。

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、使用もしくは配布することができるものとします。

本プログラムのライセンス保持者で、(i) 独自に作成したプログラムとその他のプログラム（本プログラムを含む）との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、下記に連絡してください。

IBM Corporation
Software Interoperability Coordinator, Department YBWA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

© Copyright IBM Corp. 1998, 2010

本プログラムに関する上記の情報は、適切な使用条件の下で使用することができますが、有償の場合もあります。

本書で説明されているライセンス・プログラムまたはその他のライセンス資料は、IBM 所定のプログラム契約の契約条項、IBM プログラムのご使用条件、IBM 機械コードのご使用条件、またはそれと同等の条項に基づいて、IBM より提供されます。

この文書に含まれるいかなるパフォーマンス・データも、管理環境下で決定されたものです。そのため、他の操作環境で得られた結果は、異なる可能性があります。一部の測定が、開発レベルのシステムで行われた可能性がありますが、その測定値が、一般に利用可能なシステムのものと同じである保証はありません。さらに、一部の測定値が、推定値である可能性があります。実際の結果は、異なる可能性があります。お客様は、お客様の特定の環境に適したデータを確かめる必要があります。

IBM 以外の製品に関する情報は、その製品の供給者、出版物、もしくはその他の公に利用可能なソースから入手したものです。IBM は、それらの製品のテストは行っておりません。したがって、他社製品に関する実行性、互換性、またはその他の要求については確証できません。 IBM 以外の製品の性能に関する質問は、それらの製品の供給者にお願いします。

IBM の将来の方向または意向に関する記述については、予告なしに変更または撤回される場合があり、単に目標を示しているものです。

表示されている IBM の価格は IBM が小売り価格として提示しているもので、現行価格であり、通知なしに変更されるものです。卸価格は、異なる場合があります。

本書には、日常の業務処理で用いられるデータや報告書の例が含まれています。より具体性を与えるために、それらの例には、個人、企業、ブランド、あるいは製品などの名前が含まれている場合があります。これらの名称はすべて架空のものであり、名称や住所が類似する企業が実在しているとしても、それは偶然にすぎません。

著作権使用許諾:

本書には、様々なオペレーティング・プラットフォームでのプログラミング手法を例示するサンプル・アプリケーション・プログラムがソース言語で掲載されています。お客様は、サンプル・プログラムが書かれているオペレーティング・プラットフォームのアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。このサンプル・プログラムは、あらゆる条件下における完全なテストを経ていません。従って IBM は、これらのサンプル・プログラムについて信頼性、利便性もしくは機能性があることをほのめかしたり、保証することはできません。サンプル・プログラムは特定物として現存するままの状態で提供されるものであり、いかなる保証も提供されません。 IBM は、このサンプル・プログラムの使用から生ずるいかなる損害に対しても、責任を負いません。

それぞれの複製物、サンプル・プログラムのいかなる部分、またはすべての派生的創作物にも、次のように、著作権表示を入れていただく必要があります。

© (お客様の会社名) (西暦年). このコードの一部は、IBM Corp. のサンプル・プログラムから取られています。 © Copyright IBM Corp. _年を入れる_.

この情報をソフトコピーでご覧になっている場合は、写真やカラーの図表は表示されない場合があります。

プログラミング・インターフェース情報

本書「IBM Developer Kit for Java」には、IBM Developer Kit for Java のサービスを利用するためのプログラムを、ユーザーが作成できるようにするためのプログラミング・インターフェースが記述されています。

商標

IBM、IBM ロゴおよび ibm.com は、世界の多くの国で登録された International Business Machines Corp. の商標です。他の製品名およびサービス名は、IBM または各社の商標です。現時点での IBM の商標リストについては、http://www.ibm.com/legal/copytrade.shtml の「Copyright and trademark information」をご覧ください。

Adobe、Adobe ロゴ、PostScript、PostScript ロゴは、Adobe Systems Incorporated の米国およびその他の国における登録商標または商標です。

Linux は、Linus Torvalds の米国およびその他の国における商標です。

Microsoft、Windows、Windows NT および Windows ロゴは、Microsoft Corporation の米国およびその他の国における商標です。

UNIX は The Open Group の米国およびその他の国における登録商標です。

Java およびすべての Java 関連の商標およびロゴは Sun Microsystems, Inc.の米国およびその他の国における商標です。

他の会社名、製品名およびサービス名等はそれぞれ各社の商標です。

使用条件

これらの資料は、以下の条件に同意していただける場合に限りご使用いただけます。

個人使用: これらの資料は、すべての著作権表示その他の所有権表示をしていただくことを条件に、非商業的な個人による使用目的に限り複製することができます。ただし、IBM の明示的な承諾をえずに、これらの資料またはその一部について、二次的著作物を作成したり、配布 (頒布、送信を含む) または表示 (上映を含む) することはできません。

商業的使用: これらの資料は、すべての著作権表示その他の所有権表示をしていただくことを条件に、お客様の企業内に限り、複製、配布、および表示することができます。 ただし、IBM の明示的な承諾をえずにこれらの資料の二次的著作物を作成したり、お客様の企業外で資料またはその一部を複製、配布、または表示することはできません。

ここで明示的に許可されているもの以外に、資料や資料内に含まれる情報、データ、ソフトウェア、またはその他の知的所有権に対するいかなる許可、ライセンス、または権利を明示的にも黙示的にも付与するものではありません。

資料の使用が IBM の利益を損なうと判断された場合や、上記の条件が適切に守られていないと判断された場合、IBM はいつでも自らの判断により、ここで与えた許可を撤回できるものとさせていただきます。

お客様がこの情報をダウンロード、輸出、または再輸出する際には、米国のすべての輸出入関連法規を含む、すべての関連法規を遵守するものとします。

IBM は、これらの資料の内容についていかなる保証もしません。これらの資料は、特定物として現存するままの状態で提供され、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任なしで提供されます。

IBM®

Printed in Japan

日本アイ・ビー・エム株式会社

〒103-8510 東京都中央区日本橋箱崎町19-21