CICS Transaction Server for z/OS

CICS での Java アプリケーション

バージョン *5* リリース *5*

CICS Transaction Server for z/OS

CICS での Java アプリケーション

バージョン *5* リリース *5*

注記

本書および本書で紹介する製品をご使用になる前に、 343 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 CICS Transaction Server for z/OS バージョン 5 リリース 5 (製品番号 5655-Y04) および新しい版で明記されていない限り、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。

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： CICS Transaction Server for z/OS
Java Applications in CICS
Version 5 Release 5

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

© Copyright IBM Corporation 1999, 2018.

目次

この PDF について v

第 1 章 CICS および Java 1
CICS における Java サポート 2
OSGi サービス・プラットフォーム 3
JVM サーバー・ランタイム環境 5
JVM プロファイル 8
JVM の構造 9
CICS タスクとスレッドの管理 12
共用クラス・キャッシュ 15
OSGi に準拠する Java アプリケーション 16
Liberty JVM サーバーでの Java アプリケーション 19
Java Web サービス 22

第 2 章 Java アプリケーションの開発 27
CICS について必要な知識 27
CICS トランザクション 28
CICS タスク 28
CICS アプリケーション・プログラム 28
CICS サービス 29
CICS での Java ランタイム環境 31
IBM CICS SDK for Java を使用したアプリケーションの開発 32
ターゲット・プラットフォームの更新 33
プラグイン・プロジェクトの作成 34
プラグイン・プロジェクトのマニフェスト・ファイルの更新 36
Java EE アプリケーションの作成 37
CICS バンドル・プロジェクトへのプロジェクトの追加 39
プロジェクト・ビルド・パスの更新 41
その他のツールを使用したアプリケーションの開発 41
共用 JVM に関する考慮事項 42
JCICS を使用した Java の開発 43
CICS 用 Java クラス・ライブラリー (JCICS) . . 43
データ・エンコード 47
JCICS API サービスと例 49
JCICS の使用 74
Java の制約事項 75
OSGi の使用に関するガイダンス 76
Liberty JVM サーバーで実行する Java アプリケーションの開発 79
開発環境のセットアップ 79
Java EE アプリケーションと Liberty アプリケーション 80
Liberty JVM サーバー内で実行するよう Java EE アプリケーションをマイグレーションする . . . 86
CICS プログラムからの Java EE アプリケーションへのリンク 87
Java Transaction API (JTA) 94
Java Persistence API (JPA) 95
Enterprise JavaBeans (EJB) 98
Java Message Service (JMS) 107
Java Management Extensions API (JMX) . . . 108
Java Authorization Contract for Containers (JACC) 110
Java Authentication Service Provider Interface for Containers (JASPIC). 111
Java EE コネクター・アーキテクチャー (JCA) 113
MicroProfile によるマイクロサービスの開発 129
Liberty Web サーバー・プラグイン 134
Liberty フィーチャー. 134
Java アプリケーションからのデータへのアクセス 148
Java からの構造化データとの対話 149
OSGi JVM サーバーで JZOS Toolkit API を使用する Java アプリケーションの開発 150
Java プログラムから IBM MQ へのアクセス . . 153
CICS Liberty JVM サーバーでの IBM MQ classes for JMS の使用 155
OSGi JVM サーバーでの IBM MQ classes for JMS の使用 160
OSGi JVM サーバーでの IBM MQ classes for Java の使用 165
CICS 内の Java アプリケーションからの接続 . . 166
JCA ローカル ECI サポート 167
JVM サーバーで実行する既存のアプリケーションのパッケージ化. 167
既存の Java プロジェクトのプラグイン・プロジェクトへの変換. 167
JAR ファイルの内容の OSGi プラグイン・プロジェクトへのインポート. 169
バイナリー JAR ファイルの OSGi プラグイン・プロジェクトへのインポート. 172

第 3 章 JVM サーバーへのアプリケーションのデプロイ 175
JVM サーバーにおける OSGi バンドルのデプロイ 175
CICS バンドル内の Java EE アプリケーションの Liberty JVM サーバーへのデプロイ 177
Liberty JVM サーバーへの Java EE アプリケーションの直接デプロイ 180
Liberty JVM サーバーへの共通ライブラリーのデプロイ 181
JVM サーバー内の Java アプリケーションの呼び出し 182
CICS の非 OSGi Java アプリケーションの配置 184

第 4 章 Java サポートのセットアップ 185
JVM プロファイルのロケーションの設定 186
Java のメモリー制限の設定. 187
z/OS UNIX ディレクトリーおよびファイルに対するアクセス権の CICS 領域への付与. 188

© Copyright IBM Corp. 1999, 2018

JVM サーバーのセットアップ 191
OSGi JVM サーバーの構成. 192
Liberty JVM サーバーの構成 196
Axis2 用の JVM サーバーの構成. 213
CICS セキュリティー・トークン・サービスのための JVM サーバーの構成 215
JVM プロファイルの検証およびプロパティー 216

第 5 章 Java アプリケーションの管理 245

JVM サーバーにおける OSGi バンドルの更新 . . 245
OSGi JVM サーバーの OSGi バンドルの更新 247
共通ライブラリーを含むバンドルの更新 . . . 250
OSGi ミドルウェア・バンドルの更新 252
JVM サーバーからの OSGi バンドルの除去 . . . 253
JVM サーバーへのアプリケーションの移動 . . . 254
Liberty JVM サーバーでの Java EE アプリケーションの更新 255
JVM サーバーのスレッド限度の管理 257
Recovery of resources in bundles 258
JVM stdout および stderr 出力をリダイレクトするための Java クラスの作成 261
出力リダイレクト・インターフェース 262
出力の予想される宛先 263
出力リダイレクト・エラーと内部エラーの処理 263

第 6 章 Java アプリケーションのセキュリティー. 265

OSGi アプリケーションに関するセキュリティーの構成 265
Liberty JVM サーバーに関するセキュリティーの構成 266
Liberty のエンジェル・プロセス 269
Liberty JVM サーバーでのユーザー認証 . . . 272
ユーザーに Liberty JVM サーバー内のアプリケーションを実行する権限を与える. 275
OAuth 2.0 を使用したアプリケーションの許可 277
SAF ロール・マッピングを使用した許可 . . . 281
LDAP レジストリーを使用した Liberty JVM サーバーのセキュリティーの構成 283
Java 鍵ストアを使用した Liberty JVM サーバー用の SSL (TLS) の構成. 287
RACF を使用した Liberty JVM サーバー用の SSL (TLS) の構成 288
Liberty JVM サーバーでの SSL (TLS) クライアント証明書認証のセットアップ 289
Java セキュリティー・マネージャーの有効化. . . 291

第 7 章 Java パフォーマンスの改善 295

Java ワークロードにおけるパフォーマンス・ゴールの判別 295
IBM Health Center を使用した Java アプリケーションの分析 296
ガーベッジ・コレクションおよびヒープ拡張 . . . 297
JVM サーバーのパフォーマンスの改善 299
JVM サーバーによるプロセッサー使用量の確認 300
JVM サーバーのストレージ所要量の計算 . . . 301
JVM サーバー・ヒープとガーベッジ・コレクションの調整 304
JVM サーバー始動環境の調整 306
JVM の Language Environment エンクレーブ・ストレージ 306
JVM サーバーの Language Environment ストレージ必要量の特定 308
DFHAXRO による JVM サーバーのエンクレーブの変更 313
z/OS 共用ライブラリー領域の調整 315

第 8 章 Java アプリケーションのトラブルシューティング 317

Java の診断 321
Liberty JVM サーバーおよび Java Web アプリケーションのトラブルシューティング 323
JVM 出力、ログ、ダンプ、およびトレースの場所の制御. 333
DD ステートメントを使用した JVM サーバー出力の JES への転送 334
JVM stdout および stderr ストリームのリダイレクト. 335
Java ダンプ・オプションの制御 338
JVM サーバーの CICS コンポーネント・トレース 338
JVM サーバーのトレースの活動化と管理 338
Java アプリケーションのデバッグ 339
CICS JVM プラグイン・メカニズム. 341

特記事項. 343

索引 349

この PDF について

この PDF では、CICS で Java アプリケーションとエンタープライズ Bean を作成して使用する方法について説明します。本書は、CICS の経験がほとんどなく、Java プログラムの開発と実行に必要な CICS の知識のみを求めている、経験のある Java アプリケーション・プログラマーを対象としています。 また、Java サポートに必要な CICS 要件について知りたい経験のある CICS ユーザーやシステム・プログラマーも対象としています。

本書で使用されている用語や表記について詳しくは、IBM Knowledge Center の CICS 資料で使用されている表記規則および用語を参照してください。

本書の作成日

本書は、2018 年 12 月 14 日に作成されました。

© Copyright IBM Corp. 1999, 2018

第 1 章 CICS および Java

企業で Java™ を使用する場合、CICS® によって、CICS 領域の制御下にある Java 仮想マシン (JVM) で Java アプリケーションを開発し、実行するためのツールおよびランタイム環境が提供されます。JVM サーバーで実行できる Java ワークロードは、zEnterprise® Application Assist Processor (zAAP) で実行できます。

Java アプリケーション開発

OSGi Service Platform に準拠する、再使用可能なモジュラー Java アプリケーションを作成できます。これらのアプリケーションは、CICS と他のプラットフォーム間での移植が容易であり、OSGi は、依存関係とバージョンの管理に細分性を提供します。

Java CICS (JCICS) API を使用すると、ファイルまたは一時記憶域キューからの読み取りなどの CICS サービスにアクセスするアプリケーションを作成できます。Java アプリケーションは、他の CICS アプリケーションにリンクでき、Db2® および IMS™ 内のデータにアクセスすることができます。 Java アプリケーションは、JVM サーバーで実行します。

Liberty プロファイルで提供されているさまざまな Web ツールを使用することにより、アプリケーションの Web プレゼンテーション層を作成できます。 CICS では、JSP ページと Web サーブレットを、同じ JVM サーバー上でアプリケーションとして実行できます。

Axis2 JVM サーバーにおける Web サービス

異機種混合環境でサービス・プロバイダーとサービス・リクエスターを処理するための Java Web サービスを作成できます。 Java Web サービスは JVM サーバーで実行され、SOAP 処理は Liberty JVM サーバーの Apache CXF Web サービス・エンジン、または JVM サーバーの Apache Axis2 Web サービス・エンジンによって実行されます。また、標準の Java API とアノテーションを使用して、Java Web サービスの作成、データ変換の実行、XML の処理、または構造化データの処理を行えます。

Java における CICS との接続

JCA (Java コネクター・アーキテクチャー) を使用することで、CICS Transaction Gateway を介して、外部の Java アプリケーションを CICS に接続できます。JCA は、外部の Java 環境から CICS アプリケーションを呼び出すための関連テクノロジーです。この方法で呼び出される CICS アプリケーションは Java で、または他のサポートされている言語で実装できます。

© Copyright IBM Corp. 1999, 2018

CICS における Java サポート

CICS は、CICS 領域の制御下にある Java 仮想マシン (JVM) で Java エンタープライズ・アプリケーションを開発し、実行するためのツールおよびランタイム環境を提供します。Java アプリケーションは、CICS サービスおよび他の言語で作成されたアプリケーションと対話できます。

z/OS® 上の Java は、Java アプリケーションを実行するための包括的なサポートを提供します。CICS は、IBM® 64-bit SDK for z/OS, Java Technology Edition, バージョン 8 を使用します。

SDK には、Java API のフルセットおよび 1 組の開発ツールをサポートする Java ランタイム環境が含まれています。汎用プロセッサーの生産性を高めて z/OS Java テクノロジー・ベースのアプリケーションのコンピューティングにかかる全体的なコストを削減することができるように、特定の z Systems® ハードウェアで特殊なプロセッサーを選択できます。 IBM® IBM Z Integrated Information Processor (zIIP) を使用することにより、CICS での Java ワークロードを含め、対象の Java ワークロードを実行するためのプロセッサー能力を追加できます。 Java Standard Edition Products on z/OS で、z/OS プラットフォーム上の Java に関する詳細情報を見つけ、64 ビット・バージョンの SDK をダウンロードすることができます。

CICS は Eclipse ベースのツール、および Java アプリケーション用の JVM サーバー・ランタイム環境を提供します。

IBM CICS SDK for Java

CICS Explorer® は、Eclipse ベースの統合開発環境 (IDE) 用に自由にダウンロードできます。CICS Explorer に含まれている IBM CICS SDK for Java は、OSGi Service Platform 仕様に従うアプリケーションの開発とデプロイをサポートします。OSGi Service Platform は、コンポーネント・モデルを使用してアプリケーションを開発し、それらのアプリケーションを OSGi バンドルとしてフレームワークにデプロイするためのメカニズムを提供します。*OSGi* バンドル は、アプリケーション・コンポーネントのデプロイメントの単位であり、バージョン管理情報、依存関係、およびアプリケーション・コードが入っています。OSGi の主な利点は、*OSGi* サービス と呼ばれる明確に定義されたインターフェースからのみアクセスされる再使用可能コンポーネントから、アプリケーションを作成できることです。また、Java アプリケーションのライフサイクルと依存関係をきめ細かく管理することもできます。

IBM CICS SDK for Java は、サポートされている任意のリリースの CICS 用に Java アプリケーションの開発をサポートします。 SDK には、CICS サービスにアクセスするためのクラス、および CICS 用のアプリケーション開発に取り掛かるためのサンプルから成る、Java CICS (JCICS) ライブラリーが組み込まれています。また、このツールを使用すると、既存の Java アプリケーションを OSGi に変換することもできます。

IBM CICS SDK for Java EE and Liberty は、オプションとして CICS Explorer と共に組み込まれており、CICS でデプロイできる CICS バンドルへの Liberty アプリケーションのパッケージ化をサポートします。

JVM サーバー

JVM サーバーは、CICS における Java アプリケーション用の戦略的なランタイム環境です。JVM サーバーは、単一の JVM でさまざまな Java アプリケーションからの多数の並行要求を処理できます。 JVM サーバーを使用することにより、CICS 領域で Java アプリケーションを実行するために必要な JVM の数が減ります。 JVM サーバーを使用するには、Java アプリケーションがスレッド・セーフになっていて、OSGi 仕様または Java EE 仕様に準拠していることが必要です。JVM サーバーには、次のような利点があります。

- 適格な Java ワークロードを zIIP プロセッサーで実行し、トランザクションのコストを削減できます。
- スレッド・セーフ Java プログラムや Web サービスなどの異なるタイプの作業を 1 つの JVM サーバーで実行できます。
- JVM サーバーを再始動することなく、OSGi フレームワークでアプリケーションのライフサイクルを管理できます。
- CICS とその他のプラットフォームとの間で、OSGi を使用してパッケージされた Java アプリケーションをより容易に移植できます。
- Liberty JVM サーバーの中に Java EE アプリケーションをデプロイできます。

  注**:** CICS の OSGi アプリケーションは Liberty JVM サーバーにインストールできますが、Liberty のサービスや機能はサポートされていないため、使用できません。

OSGi サービス・プラットフォーム

OSGi Service Platform は、コンポーネント・モデルを使用してアプリケーションを開発し、それらのアプリケーションを OSGi フレームワークにデプロイするためのメカニズムを提供します。OSGi アーキテクチャーは、Java アプリケーションの作成と管理に役立つ機能を備えた複数のレイヤーに分けられます。

OSGi フレームワークは、OSGi Service Platform 仕様の中核をなします。CICS は OSGi フレームワークの Equinox 実装環境を使用します。 OSGi フレームワークは、JVM サーバーの始動時に初期化されます。Java アプリケーションに OSGi を使用すると、次の主な利点があります。

- 新しい Java アプリケーション、および Java アプリケーションの新規バージョンを稼働中の実動システムにデプロイする際に JVM を再始動する必要はなく、また、その JVM にデプロイされている他の Java アプリケーションに影響が及ぼされることはありません。
- Java アプリケーションの移植可能性が向上し、リエンジニアリングが容易になり、要件の変化への適応性が高まります。
- Plain Old Java Object (POJO) プログラミング・モデルに従って、動的なライフサイクルを持つ 1 組の OSGi バンドルとしてアプリケーションをデプロイすることを選択できます。
- アプリケーション・バンドルの依存関係とバージョンの管理が容易になります。

OSGi アーキテクチャーには、以下のレイヤーがあります。

- モジュール・レイヤー

- ライフサイクル・レイヤー
- サービス・レイヤー

モジュール・レイヤー

デプロイメントの単位は OSGi バンドルです。モジュール・レイヤーでは、OSGi フレームワークがバンドルのモジュラー・アスペクトを処理します。OSGi フレームワークがこの処理を実行できるようにするメタデータは、バンドルのマニフェスト・ファイルで提供されます。

OSGi の主な利点の 1 つは、クラス・ローダー・モデルで、これは、マニフェスト・ファイルのメタデータを使用します。OSGi にはグローバル・クラスパスはありません。バンドルが OSGi フレームワークにインストールされると、それらのメタデータはモジュール・レイヤーによって処理され、宣言された外部依存関係は、インストールされた他のモジュールによって宣言されたエクスポートおよびバージョン情報に照らして調整されます。OSGi フレームワークは、すべての依存関係を解明し、バンドルごとに独立した必須のクラスパスを計算します。この方法により、以下の要件が満たされるようになるため、単純な Java クラス・ロードの短所が解決されます。

- 各バンドルは、それが明示的にエクスポートする Java パッケージからのみ可視である。
- 各バンドルが、明示的にそのパッケージ依存関係を宣言する。
- パッケージを特定のバージョンでエクスポートし、特定のバージョンで、または特定の範囲のバージョンからインポートできる。
- パッケージの複数のバージョンが異なるクライアントに対して同時に使用可能である。

ライフサイクル・レイヤー

OSGi におけるバンドルのライフサイクル管理レイヤーは、JVM のライフサイクルとは無関係に、バンドルを動的にインストール、開始、停止、およびアンインストールすることを可能にします。ライフサイクル・レイヤーは、バンドルが、その依存関係がすべて解決される場合のみ開始することを確実にし、実行時の ClassNotFoundException 例外の発生を減らします。未解決の依存関係がある場合、OSGi フレームワークは問題を報告し、バンドルを開始しません。

各バンドルは、バンドル・マニフェストで識別されるバンドル・アクティベーター・クラスを提供できます。フレームワークはバンドルの開始および停止イベントの一部として、そのクラスを呼び出します。

サービス・レイヤー

OSGi のサービス・レイヤーは、本来、非永続サービス・レジストリー・コンポーネントを使用してサービス指向アーキテクチャーをサポートします。バンドルはサービス・レジストリーにサービスを公開し、他のバンドルがそれらのサービスをサービス・レジストリーからディスカバーできます。これらのサービスは、バンドル相互間のコラボレーションの 1 次的な手段です。OSGi サービスは、1 つ以上の Java インターフェース名でサービス・レジストリーに公開される Plain Old Java Object (POJO) であり、オプションのメタデータがカスタム・プロパティー (名前/

値のペア) として保管されます。ディスカバーする側のバンドルは、インターフェース名によってサービス・レジストリーでサービスを検索することができ、カスタム・プロパティーに基づいて検索されるサービスを潜在的にフィルターに掛けることができます。

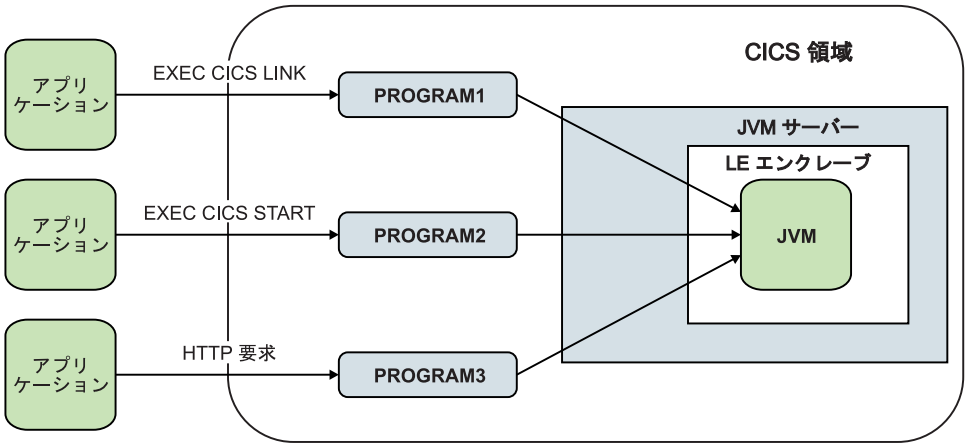
サービスは完全に動的であり、通常、それらのサービスを提供するバンドルと同じライフサイクルを持ちます。

JVM サーバー・ランタイム環境

JVM サーバー は、単一の JVM でさまざまな Java アプリケーションに対する多数の並行要求を処理できるランタイム環境です。 JVM サーバーを使用すると、OSGi フレームワークでスレッド・セーフ Java アプリケーションを実行し、Liberty で Web アプリケーションを実行し、Axis2 Web サービス・エンジンで Web サービス要求を処理することができます。

JVM サーバーは JVMSERVER リソースで表されます。JVMSERVER リソースを使用可能にすると、CICS は MVS™ にストレージを要求し、Language Environment® エンクレーブをセットアップし、そのエンクレーブで 64 ビット JVM を起動します。CICS は、JVMSERVER リソースで指定された JVM プロファイルを使用して、正しいオプションを持つ JVM を作成します。このプロファイルで、JVM オプションやシステム・プロパティーを指定したり、ネイティブ・ライブラリーを追加したりすることができます。例えば、Java アプリケーションから DB2® または IBM MQ にアクセスするためのネイティブ・ライブラリーを追加することができます。

JVM サーバーを使用する利点の 1 つは、同一 JVM でさまざまなアプリケーションに対して多数の要求を実行できることです。次の図では、3 つのアプリケーションが、別々のアクセス方式を使用して、CICS 領域内の 3 つの Java プログラムを同時に呼び出しています。各 Java プログラムは同じ JVM サーバーで実行されます。



Java アプリケーション

OSGi JVM サーバーで Java アプリケーションを実行するには、その Java アプリケーションがスレッド・セーフであり、1 つの CICS バンドル内の 1 つ以上の OSGi バンドルとしてパッケージされなければなりません。JVM サーバーは、OSGi

バンドルおよび、OSGi サービスを実行できる OSGi フレームワークを実装します。OSGi フレームワークは、サービスを登録し、バンドル相互間の依存関係とバージョンを管理します。OSGi は、フレームワーク内のすべてのクラスパス管理を処理するので、JVM サーバーの停止と再始動を行うことなく、Java アプリケーションを追加、更新、および除去できます。

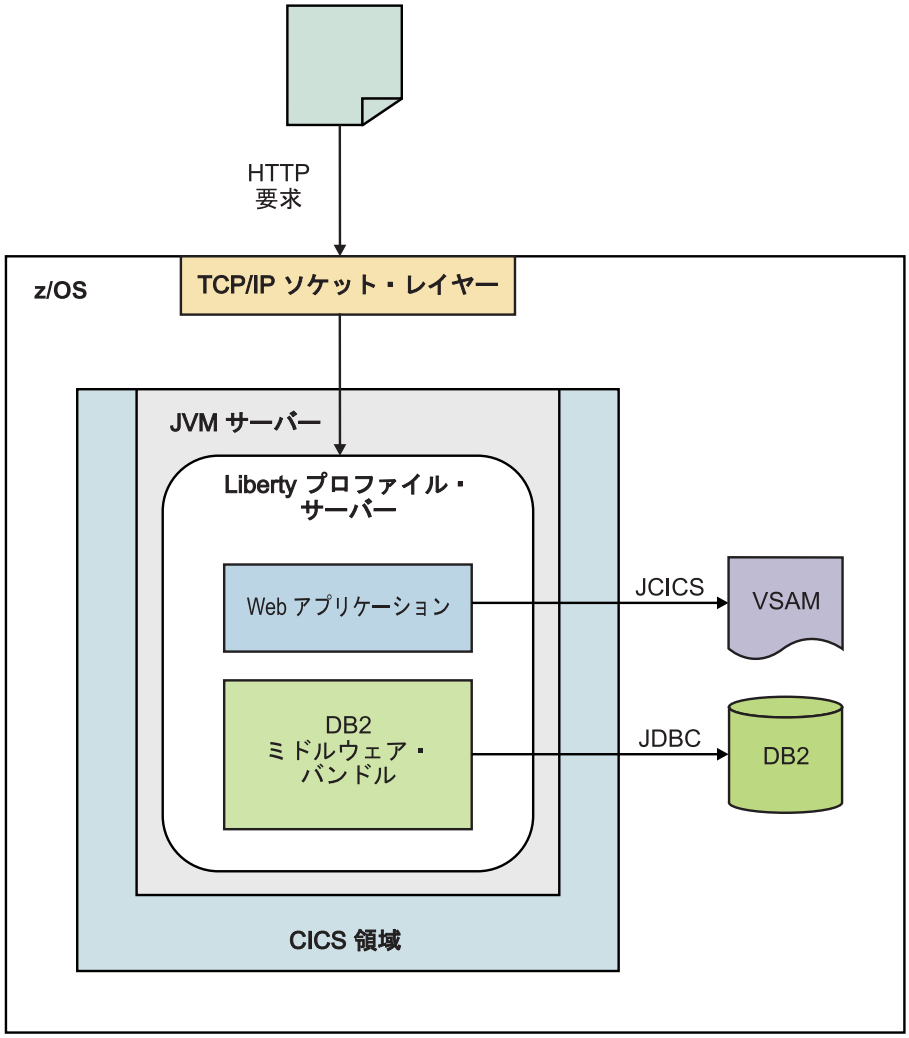
OSGi を使用してパッケージされる Java アプリケーションのデプロイメントの単位は、CICS バンドルです。 BUNDLE リソースは、アプリケーションを CICS に対して表現するもので、これを使用してアプリケーションのライフサイクルを管理することができます。IBM CICS SDK for Java は、CICS バンドル・プロジェクト内の OSGi バンドルを zFS にデプロイするためのサポートを提供します。

OSGi フレームワーク外部から Java アプリケーションにアクセスするには、PROGRAM リソースを使用して、アプリケーションが実行される JVM サーバー、および OSGi サービスの名前を識別してください。OSGi サービスは、CICS メイン・クラスを指します。

JVM サーバーにおける OSGi フレームワークの使用について詳しくは、OSGi に準拠する Java アプリケーションを参照してください。

Java Web アプリケーション

JVM サーバーでは、OSGi フレームワークでの Java アプリケーションの実行に加えて、WebSphere® Application Server Libertyの実行もサポートされます。プロファイルは Web アプリケーションを実行するための軽量のアプリケーション・サーバーです。Web アプリケーションは JCICS を使用して CICS 内のリソースとサービスにアクセスし、さらに DB2 内のデータにアクセスすることもできます。Liberty で実行されるアプリケーションは、CICS Web サポートを通してではなく、z/OS の TCP/IP ソケット・レイヤーを通してアクセスされます。



Java Web アプリケーションのデプロイメントは、開発者が Liberty のドロップイン・ディレクトリーに Web アーカイブ (WAR) ファイルまたはエンタープライズ・アプリケーション・アーカイブ (EAR) ファイルを直接デプロイできる Liberty モデルに従って行うことも、CICS バンドルを作成する CICS アプリケーション・モデルを使用して行うこともできます。CICS バンドルでは、ライフサイクル管理が提供され、OSGi バンドルや WAR ファイルなどの多数のコンポーネントを一緒に含む 1 つのアプリケーションをパッケージできます。

Web アプリケーションから OSGi バンドルにアクセスするには、Enterprise Bundle Archive (EBA) ファイルとしてアプリケーションをデプロイする必要があります。 EBA を開発するには、Rational® Application Developer を使用するか、または Eclipse IDE、IBM CICS SDK for Java、および WebSphere Application Server Developer Tools for Eclipse を組み合わせて使用することができます。後者のツール・セットは自由に使用可能ですが、IBM CICS SDK for Java を除いて IBM サポートの対象外となります。

Liberty の使用について詳しくは、Java Web アプリケーションを参照してください。

Web サービス

JVM サーバーを使用して、Web サービス・リクエスターおよび Web サービス・プロバイダーのアプリケーションの SOAP 処理を実行できます。 Java に基づく SOAP エンジンである Axis2 をパイプラインで使用する場合、SOAP 処理は JVM サーバーで実行されます。 Web サービスに JVM サーバーを使用する利点は、作業を zAAP プロセッサーにオフロードできることです。

Web サービスに対する JVM サーバーの使用について詳しくは、Java Web サービスを参照してください。

JVM プロファイル

JVM プロファイルは、JVM の特性を決定する Java ランチャー・オプションとシステム・プロパティーが入っているテキスト・ファイルです。任意の標準テキスト・エディターを使用して JVM プロファイルを編集することができます。

CICS が Java プログラムの実行要求を受け取ると、JVM プロファイルの名前が Java ランチャーに渡されます。 JVM プロファイルのオプションを使用して作成された JVM で、Java プログラムが実行されます。

CICS が使用する JVM プロファイルは、JVMPROFILEDIR システム初期設定パラメーターによって指定される z/OS UNIX システム・サービス・ディレクトリー内にあります。このディレクトリーには、CICS が JVM プロファイルを読み取るための適切な権限が必要です。

サンプル JVM プロファイル

CICS には、Java 環境の構成に役立ついくつかのサンプル JVM プロファイルが含まれています。これらのプロファイルは、CICS のインストール処理時にカスタマイズされます。これらのファイルは、CICS でデフォルトとして使用されるか、システム・プログラムに使用されます。

JVM プロファイルは、Java 用の CICS ランチャーで使用されるオプションをリストします。CICS に固有のオプションもあれば、JVM ランタイム環境に標準のオプションもあります。例えば、JVM プロファイルは、ストレージ・ヒープの初期サイズや拡張できる程度を制御します。また、プロファイルは、JVM によって作成されるメッセージやダンプ出力の宛先を定義することもできます。JVM プロファイルは JVMSERVER リソース定義内の JVMPROFILE 属性で指定されます。

これらのサンプルをコピーし、独自のアプリケーションに合わせてカスタマイズすることができます。 CICS で提供されるサンプル JVM プロファイルは、z/OS UNIX 上の `/usr/lpp/cicsts/cicsts55/JVMProfiles` ディレクトリーにあります。これらのサンプルをインストール・ディレクトリーから、**JVMPROFILEDIR** システム初期設定パラメーターで指定されたディレクトリーにコピーしてください。インストール場所にあるサンプル JVM プロファイルは、これらのファイルの変更を含む

APAR を適用すると、上書きされます。変更内容が失われないように、必ず、サンプルを別の場所にコピーしてから、独自のアプリケーション・クラスの追加またはオプションの変更を行ってください。

次の表は、各サンプル JVM プロファイルの主な特性をまとめています。

表 *1. CICS* で提供されるサンプル *JVM* プロファイル

JVM プロファイル	特性
DFHJVMAX.jvmprofile	Axis2 JVM サーバー用に提供されるサンプル・プロファイル。この JVM プロファイルは JVMSERVER リソースで指定されます。CICS は DFHJVMAX.jvmprofile を使用して JVM サーバーを初期化します。
DFHJVMST.jvmprofile	セキュリティー・トークン・サービスに関して JVM サーバー用に提供されているサンプル・プロファイル。この JVM プロファイルは JVMSERVER リソースで指定されます。CICS は DFHJVMST.jvmprofile を使用して JVM サーバーを初期化します。
DFHOSGI.jvmprofile	OSGi JVM サーバー用に提供されるサンプル・プロファイル。この JVM プロファイルは JVMSERVER リソースで指定されます。CICS は DFHOSGI.jvmprofile を使用して JVM サーバーを初期化します。
DFHWLP.jvmprofile	Liberty JVM サーバー用に提供されるサンプル・プロファイル。この JVM プロファイルは JVMSERVER リソースで指定されます。CICS は DFHWLP.jvmprofile を使用して Liberty JVM サーバーを初期化します。

JVM の構造

CICS で実行される JVM は、JVM プロファイルで定義される 1 組のクラスとクラスパスを使用し、64 ビット・ストレージを使用します。各 JVM が実行される Language Environment エンクレーブを調整すると、MVS ストレージを最も効率よく使用することができます。

IBM 64-bit SDK for z/OS, Java テクノロジー・エディション について詳しくは、IBM SDK, Java Technology Edition バージョン 7 の z/OS ユーザーズ・ガイドまたはIBM SDK, Java Technology Edition バージョン 8 の z/OS ユーザーズ・ガイドを参照してください。

JVM におけるクラスおよびクラスパス

CICS で実行される JVM では、さまざまなタイプのクラスまたはライブラリー・ファイルを使用できます。それには、原始クラス (システム・クラスと標準拡張クラス)、ネイティブ C DLL ライブラリー・ファイル、およびアプリケーション・クラスがあります。

JVM はこれらの各コンポーネントの目的を認識し、コンポーネントのロード方法を決定し、保管先を決定します。 JVM のクラスパスは、JVM プロファイル内のオプションによって定義され、(オプションとして) JVM プロパティー・ファイルで参照されます。

- 原始クラス は、JVM で基本サービスを提供する z/OS JVM コードです。原始クラスはシステム・クラスと標準拡張クラスに分類できます。
- z/OS UNIX では、ネイティブ *C* ダイナミック・リンク・ライブラリー *(DLL)* ファイル には拡張子 `.so` が付いています。 JVM の実行には何らかのライブラリーが必要であり、アプリケーション・コードまたはサービスによって追加のネ

イティブ・ライブラリーをロードすることができます。例えば、追加のネイティブ・ライブラリーには、Db2 JDBC ドライバーを使用する DLL ファイルを含めることができます。

- アプリケーション・クラス は、JVM で実行されるアプリケーション用のクラスであり、ユーザー作成アプリケーションに属するクラスを含みます。また、Java アプリケーション・クラスには、JCICS インターフェース・クラス、JDBC、JNDI などのリソースにアクセスするサービスを提供するための IBM 提供または他のベンダー提供のクラスも含まれます (JDBC と JNDI は CICS 用の標準 JVM セットアップに含まれていません)。クラス・キャッシュにロードされた Java アプリケーション・クラスは保持されるため、同じ JVM で実行される他のアプリケーションはこれらを再使用できます。

クラスまたはネイティブ・ライブラリーを指定できるクラスパスは、ライブラリー・パスと標準クラスパスです。

- ライブラリー・パス は、JVM が使用するネイティブ C ダイナミック・リンク・ライブラリー (DLL) ファイルを指定します。これには、アプリケーション・コードまたはサービスによってロードされる JVM および追加のネイティブ・ライブラリーの実行に必要なファイルが含まれます。各 DLL ファイルの 1 つのコピーのみがロードされ、すべての JVM がそのコピーを共用しますが、各 JVM には、その DLL 用に静的データ域の独自のコピーがあります。

  JVM の基本ライブラリー・パスは、**USSHOME** システム初期設定パラメーターと JVM プロファイルの **JAVA_HOME** オプションで指定されたディレクトリーを使用して自動的に作成されます。この基本ライブラリー・パスは、JVM プロファイルでは表示されません。このライブラリー・パスには、CICS が使用する JVM とネイティブ・ライブラリーを実行するのに必要なすべての DLL ファイルが含まれています。 **LIBPATH_SUFFIX** オプションまたは **LIBPATH_PREFIX** オプションを使用して、このライブラリー・パスを拡張できます。 **LIBPATH_SUFFIX** は、ライブラリー・パスの終わりに、IBM 提供のライブラリーの後に項目を追加します。**LIBPATH_PREFIX** は、項目を先頭に追加し、同じ名前である場合は IBM 提供のライブラリーの代わりにロードされます。問題判別の目的でそれを実行することが必要になる場合もあります。

  ライブラリー・パスに組み込むすべての DLL ファイルを、LP64 オプションを使用してコンパイルし、リンクしてください。基本ライブラリー・パスで提供される DLL ファイルおよび Db2 JDBC ドライバーなどのサービスで使用される DLL ファイルは、LP64 オプションを指定して作成されます。

- 標準クラスパス を OSGi が有効な JVM サーバーで使用しないでください。OSGi フレームワークは、アプリケーションを含む OSGi バンドル内の情報に基づいて、アプリケーションのクラスパスを自動的に判別するからです。 CICS の Axis2 環境など、OSGi 用に構成されていない JVM サーバーで使用するために、標準クラスパスが保持されます。標準クラスパスが使用されている Axis2 の場合など、例外的なシナリオでは、クラスパス・エントリーにワイルドカード接尾部を使用して、特定のディレクトリー内のすべての JAR ファイルを指定できます。

  また、CICS は、**USSHOME** システム初期設定パラメーターで指定されたディレクトリーの `/lib` サブディレクトリーを使用して、JVM の基本クラスパスを自動

的に作成します。このクラスパスには、CICS と JVM によって用意されている JAR ファイルが入ります。それは JVM プロファイルでは見られません。

システム・クラスと標準拡張クラス (原始クラス) は既に JVM のブート・クラスパスに含まれているので、これらをクラスパスに組み込む必要はありません。

JVM におけるストレージ・ヒープ

ランタイム JVM ストレージは単一の 64 ビット・ストレージ・ヒープによって管理されます。

各 JVM のヒープは、JVM の Language Environment エンクレーブにある 64 ビット・ストレージから割り振られます。各ヒープのサイズは、JVM プロファイル内のオプションによって決まります。

単一のストレージ・ヒープはヒープ と呼ばれ、場合によってはガーベッジ・コレクション・ヒープ と呼ばれます。その初期ストレージ割り振りは、JVM プロファイルの **-Xms** オプションによって設定され、その最大サイズは **-Xmx** オプションによって設定されます。

ヒープのサイズを調整すると、JVM の最適なパフォーマンスを実現することができます。JVM サーバー・ヒープとガーベッジ・コレクションの調整を参照してください。

JVM が構成される場所

JVM が必要な場合、JVM の CICS ランチャー・プログラムは MVS にストレージを要求し、Language Environment エンクレーブをセットアップし、その Language Environment エンクレーブで JVM を起動します。並行して実行される JVM 間の分離を確実にするために、各 JVM は独自の Language Environment エンクレーブ内で構成されます。

Language Environment エンクレーブは、Language Environment 事前初期設定モジュール CELQPIPI を使用して作成され、JVM は z/OS UNIX プロセスとして実行されます。したがって、JVM は、CICS Language Environment サービスではなく、MVS Language Environment サービスを使用します。JVM に使用されるストレージは、MVS Language Environment サービスの呼び出しによって取得された MVS 64 ビット・ストレージです。このストレージは、CICS アドレス・スペース内に置かれますが、CICS 動的ストレージ域 (DSA) には含まれません。

JVM の Language Environment エンクレーブは、その JVM のストレージ要件に応じて拡張することができます。Language Environment エンクレーブに CICS が使用する Language Environment ランタイム・オプションは、Language Environment エンクレーブのヒープ・ストレージの初期サイズおよび追加増分を制御します。

Language Environment エンクレーブに CICS が使用するランタイム・オプションを調整することができます。その結果、CICS がそのエンクレーブのために要求するストレージ量は、JVM プロファイルで指定されたストレージ量と可能なかぎり近くなります。したがって、MVS ストレージを最も効率よく使用することができます。ストレージの調整について詳しくは、JVM 用 Language Environment エンクレーブ・ストレージを参照してください。

JVM および z/OS 共用ライブラリー領域

共用ライブラリー領域は、アドレス・スペースがダイナミック・リンク・ライブラリー (DLL) ファイルを共用できるようにする z/OS 機能です。

この機能により、 CICS 領域は JVM に必要な DLL を共用できるようになり、各領域が DLL を個別にロードする必要はなくなります。これにより、MVS が使用する実際のストレージの量、および領域へのファイルのロード所要時間を大幅に削減できます。

共用ライブラリー領域用に予約されているストレージは、最初の JVM が領域で開始されるときにそれぞれの CICS 領域に割り振られます。割り振られるストレージの量は、z/OS の **SHRLIBRGNSIZE** パラメーターによって制御されます。共用ライブラリー領域に割り振られるストレージ量の調整について詳しくは、z/OS 共用ライブラリー領域の調整を参照してください。

CICS タスクとスレッドの管理

CICS はオープン・トランザクション環境 (OTE) を使用して JVM サーバーの作業を実行します。各タスクは、JVM サーバーのスレッドとして実行され、1 つの T8 TCB を使用して接続されます。OSGi を使用する主な利点は、OSGi フレームワーク内のアプリケーションが `ExecutorService` を使用して、CICS で追加タスクを非同期で実行するスレッドを作成できることです。CICS では、特殊な手段によってランナウェイ・タスクが処理されます。

CICS で JVM サーバーを有効にすると、JVM サーバーは言語環境プロセス・スレッドで実行されます。このスレッドは TP TCB の子です。各 CICS タスクは、1 つの T8 TCB を使用して JVM のスレッドに接続されます。 JVMSERVER リソースの `THREADLIMIT` 属性を設定すると、JVM サーバーから使用可能な T8 TCB の数を制御できます。

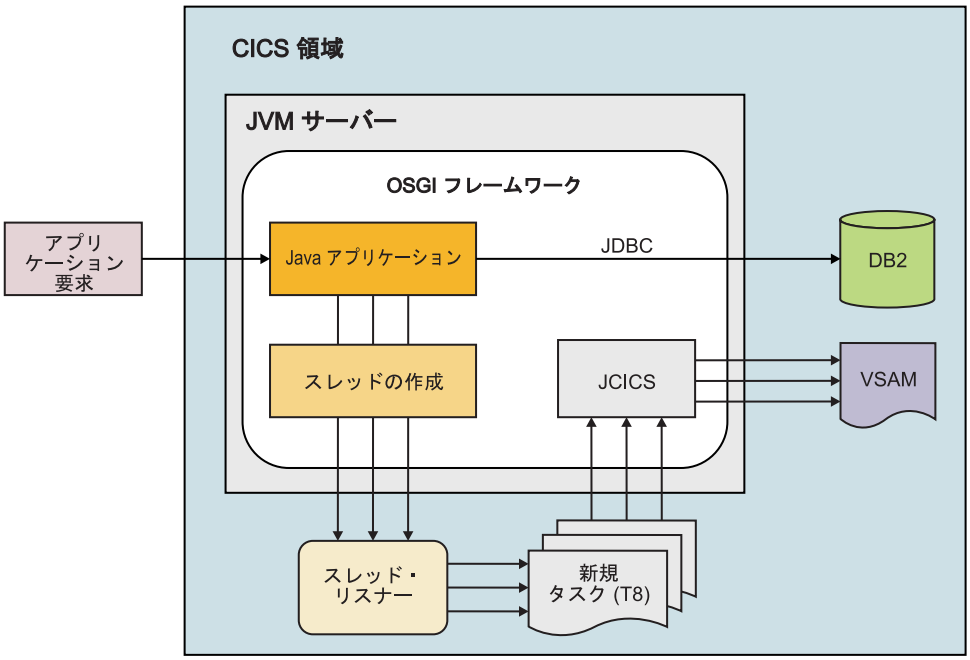
JVM サーバー用に作成される T8 TCB は、仮想プールに存在し、同じ CICS 領域で実行される別の JVM サーバーで再利用することはできません。 1 つの CICS 領域内に存在できる T8 TCB の最大数は、すべての JVM サーバーを合わせて 2000 です。また、特定の 1 つの JVM サーバーでの最大数は 256 です。

マルチスレッド・アプリケーション

OSGi フレームワークで実行される Java アプリケーションは、`ExecutorService` OSGi サービスを使用して CICS タスクを非同期に開始することもできます。 JVM サーバーは、始動時に `ExecutorService` を OSGi サービスとして登録します。`ExecutorService` では、CICS によって提供される実装が自動的に使用されます。この実装では、JCICS API を使用して CICS サービスにアクセスできるスレッドが作成されます。この方式により、アプリケーションは、スレッドを作成するために、特定の JCICS API メソッドを使用する必要がなくなります。ただし、アプリケーションは `CICSExecutorService` を使用して、別個の CICS 対応スレッドの処理を実行することもできます。

JVM サーバーが有効な場合は、このサーバーによって CJSL トランザクションが開始され、JVM サーバー・リスナーという長期実行タスクが作成されます。このリス

ナーは、アプリケーションからの新しいスレッド要求を待機し、CJSA トランザクションを実行して、T8 TCB でディスパッチされる CICS タスクを作成します。この処理は、以下の図で示されています。



高度なシナリオでは、アプリケーションは OSGi サービスを使用して、多数のスレッドを非同期で実行できます。これらのスレッドはすべて JCICS を介して CICS サービスにアクセスでき、T8 TCB 下で実行されます。

JVM サーバーの実行キー

Java プログラムは、正しい実行キーで実行される JVM を使用する必要があります。JVM サーバーは CICS キーで実行されます。JVM サーバーを使用するには、Java プログラムの `PROGRAM` リソースで、`EXECKEY` 属性が `CICS` に設定されなければなりません。CICS は、T8 TCB を使用して JVM を実行し、CICS キーの MVS ストレージを取得します。

ランナウェイ・タスク

CICS JVM サーバー・インフラストラクチャーでは、タスク・ランナウェイ検出メカニズムの使用がサポートされます。従来の CICS タスクとは異なり、T8 TCB で Java を実行中のタスクを終了させると、同じ JVM 内の他のワークロードに必ず影響します。Language Environment および JVM サーバーは、POSIX 準拠の環境で動作します。そのため、TCB/スレッドを終了させると、親プロセスも終了させられることになります。子プロセスもすべて突然終了させられるため、JVM 内のすべてのタスクが直ちに失敗することになります。

JVM サーバーで実行中のタスクが、変更された RUNAWAY 間隔を超える場合は、より制御された終了処理が発生します。これは従来の CICS の動作とは異なるので、ランナウェイ間隔を Java タスクに適用するかどうかや、どのような値を設定するかの判断が必要となります。

JVMSERVER 制御ランナウェイ処理

Java を実行中のタスクがランナウェイ間隔状態になると、JVMSERVER はその状態をインターセプトし、DISABLE PHASEOUT をトリガーします。新しい作業は JVM に入ることができず、既存の作業はそのままドレーンされます。その後、もしタスクがその処理を完了したなら、JVMSERVER は再び使用可能になり、新しい要求に使用できるようになります。多くの場合、Java を実行中のタスクがランナウェイ間隔値を超えたら、それはおそらく、例えば短ループ・アプリケーションのような不適切なアプリケーションであり、JVMSERVER の正常な PHASEOUT/RECYCLE の妨げとなります。アプリケーションが検出されると、別の間隔の後にランナウェイ・タイマーが再びトリガーされ、JVMSERVER DISABLE PHASEOUT が JVMSERVER DISABLE PURGE にエスカレートされます。残りのタスクは PURGE 処理され、ほとんどの場合は終了させられます。さらにランナウェイ間隔を超えた場合、JVMSERVER DISABLE は FORCEPURGE にエスカレートし、最終的には KILL にエスカレートして、実行中のタスクがすべて強制的に終了させられます。 JVMSERVER は再び ENABLED 状態にリサイクルされ、新しい要求に使用できるようになります。 JVMSERVER が DISABLE KILL 要求のレベルまでエスカレートされるような場合は、できるだけ早い機会に CICS をリサイクルすることをお勧めします。

変更されたランナウェイ間隔値

JVM サーバーで実行中のタスクのランナウェイ状態は、JVM サーバー全体の一時的な可用性の問題の原因になることがあります。このため、CICS は、構成されているランナウェイ間隔値を係数 10 で乗算して変更します (最大値の 45 分を上限とする)。この新しい値が、有効なランナウェイ間隔となります。ランナウェイ間隔が大きくなると、効率的でない (しかし機能はしている) アプリケーションのランナウェイ状態が検出される可能性が小さくなります。例えば、トランザクション定義で RUNAWAY=SYSTEM が指定され、ICVR システム初期設定パラメーターでデフォルト制限の 5000 ミリ秒が示されている場合、そのタスクが JVM サーバーで実行されるときの有効なランナウェイ間隔は 50000 ミリ秒となります。

ランナウェイ間隔値の設定

Liberty JVM サーバー、および CICSExecutorService から開始された OSGi JVM サーバー内の作業で使用される CJSA トランザクション定義では、デフォルトでランナウェイ検出がアクティブになっていて、システムの間隔に設定されています。これらのタスクにランナウェイ間隔が適用されないようにするには、ランナウェイ間隔を 0 または任意の別の値に設定した独自のトランザクション定義のもとで作業を実行します。 Liberty ワークロードは通常、URIMAP によって制御されます。一方 CICSExecutorService は、カスタマイズされたトランザクション定義を使用できる CICSTransactionRunnable インターフェースと CICSTransactionCallable インターフェースを備えています。

共用クラス・キャッシュ

Java 共用クラス・キャッシュを使用すると、JVM の起動時間を改善したり、全体的なストレージの使用量を削減したり、コンパイル・プロセスを最適化したりできます。クラス・キャッシュは、すべての JVM サーバー (OSGi、Liberty、およびクラスパスベース・ベース) で使用できます。

IBM® SDK, Java Technology Edition on z/OS® は、共用クラス・キャッシュをサポートします。クラス・キャッシュを JVM サーバーが使用できるようにし、そのサイズを設定するには、JVM コマンド行パラメーターを使用する必要があります。クラス・キャッシュの使用状況のモニターやクラス・キャッシュの破棄など、他の操作要件もまた、Java コマンドのオプションを使用して行います。

Java クラス・データ共用について詳しくは、クラス・データの共用を参照してください。

共用クラス・キャッシュには、以下の要素を含めることができます。

- Java クラス (アプリケーション・クラス、JVM サーバー・インフラストラクチャー、Java ブートストラップ・クラスなど)。
- Ahead-of-time (AOT) コンパイル済みコード。

クラス・キャッシュの有効化

クラス・キャッシュを有効にするには、JVM コマンド行パラメーターを使用します。例を以下に示します。

```
-Xshareclasses:name=cics.<group>
```

クラスを同じ領域内でのみ共用する場合、`<group>` に JVM プロファイル・シンボル `&applid;` を指定できます。あるいは、特定のタイプのすべての JVM サーバーが一般的なクラスを使用して接続する任意の ID を選択することもできます。共用の細分度はユーザー固有であり、ユーザーのニーズ、クラス・キャッシュのサイズ、および共用アプリケーションの数によって異なります。一般的な関数の多くの JVM サーバーがクラス・キャッシュ `name` を共用し、クラス・キャッシュのサイズをすべての使用目的に対応できる大きさにすることができます。

クラス・キャッシュの確認

z/OS UNIX コマンド `JAVA_HOME/bin/java -Xshareclasses:name=<named_cache>,printStats` を実行することによって、クラス・キャッシュがどの程度満杯かを確認できます。この照会では、`Cache is nn% full` というメッセージが返されます。詳細については、キャッシュに関する問題の処理を参照してください。

クラス・キャッシュ・サイズの設定または変更

1. JVM プロファイルを変更して、名前付きキャッシュ・サイズ `-Xscmx256M` を定義します。
2. キャッシュを使用しているすべての JVM サーバーをシャットダウンします。
3. `JAVA_HOME/bin/java -Xshareclasses:name=<named_cache>,destroy` を使用してキャッシュを削除します。
4. JVM サーバーを始動します。

この手順のステップ 2 と 3 の代わりに、z/OS を再始動することもできます。

OSGi に準拠する Java アプリケーション

CICS には、JVM サーバーで OSGi 仕様に従う Java アプリケーションを実行するために、OSGi フレームワークの Equinox 実装環境が組み込まれています。

OSGi Service Platform 仕様は、 3 ページの『OSGi サービス・プラットフォーム』 に記述されているように、動的なモジュラー Java アプリケーションを実行し、管理するためのフレームワークを提供します。JVM サーバーのデフォルト構成には、OSGi フレームワークの Equinox 実装環境が組み込まれています。JVM サーバーの OSGi フレームワークにデプロイされる Java アプリケーションは、OSGi を使用する利点、および CICS におけるアプリケーションの実行に固有のサービス品質が得られます。

次のいずれかの理由で Java アプリケーションを使用できます。

- トランザクションのコストを削減するために、zAAP で実行できる Java ワークロードを作成したい。
- 他のプラットフォームで OSGi を使用する Java アプリケーションを作成した経験があり、CICS で Java アプリケーションを作成したい。
- アプリケーションおよびそれらのアプリケーションが実行される JVM の可用性に影響を与えることなく、独立して再使用し、更新できる 1 組のモジュラー・コンポーネントとして、Java アプリケーションを提供したい。

OSGi に準拠する Java アプリケーションを効率よく開発し、デプロイし、管理するには、IBM CICS SDK for Java と CICS Explorer を使用する必要があります。

- IBM CICS SDK for Java は、既存の Eclipse 統合開発環境 (IDE) を拡張して、Java 開発者が CICS における Java アプリケーションを作成し、デプロイするのに役立つツールとサポートを提供します。既存の Java アプリケーションを OSGi バンドルに変換するには、このツールを使用します。
- CICS Explorer は、OSGi バンドルと OSGi サービス、およびそれらが実行される JVM サーバーのビューをシステム管理者に提供する、Eclipse ベースのシステム管理者ツールです。Java アプリケーションの使用可能化と使用不可化、フレームワーク内の OSGi バンドルとサービスの状況の確認、および JVM サーバーのパフォーマンスに関する予備統計の取得に、このツールを使用します。

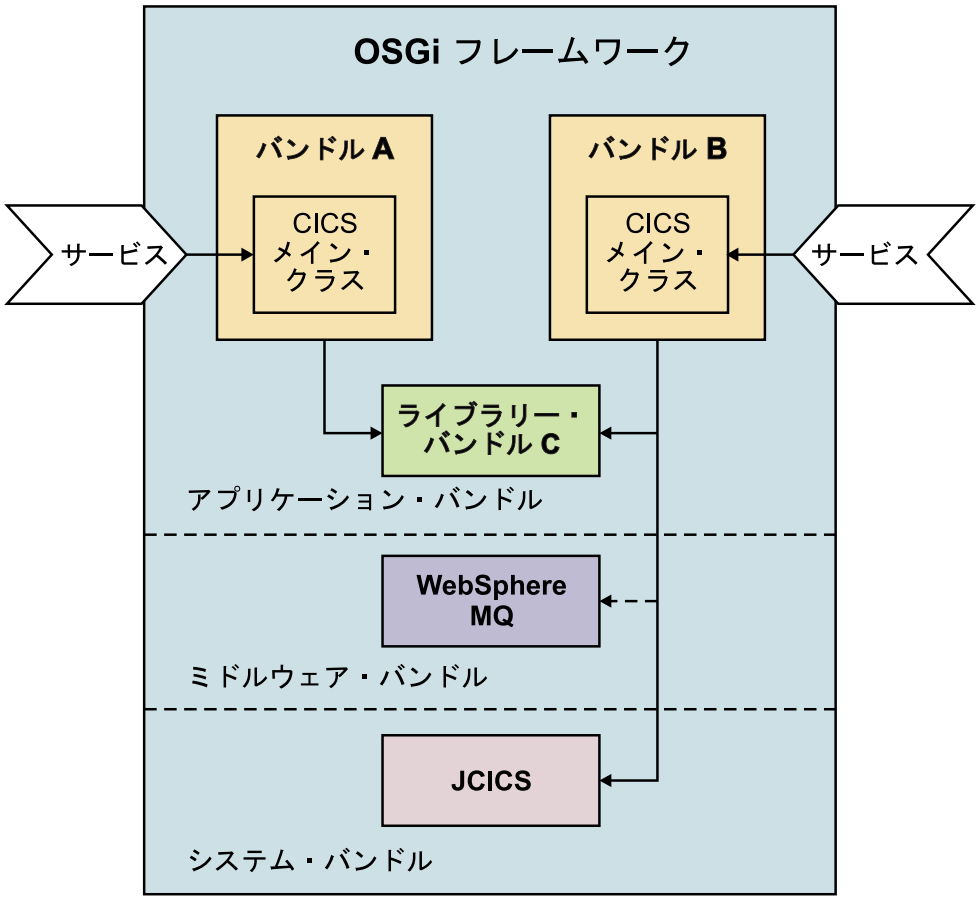
OSGi を使用する Java 開発者またはシステム管理者には、これらの自由に使用できるツールへのアクセス権限が必要です。

以下の例では、CICS において OSGi を使用する Java アプリケーションの実行方法を説明しています。

同一 JVM サーバー内の複数の Java アプリケーションの実行

JVM サーバーは、同一 JVM で複数の要求を同時に処理できます。したがって、同一アプリケーションを同時に複数回呼び出したり、同一 JVM サーバーで複数のアプリケーションを実行したりすることができます。

JVM サーバー間でアプリケーションを分ける方法が決定したら、OSGi モデルを使用して、複数のアプリケーションを 1 組の OSGi バンドルとしてコンポーネント化する方法を計画できます。また、アプリケーションにサービスを提供するために、フレームワークで必要な対応 OSGi バンドルを決定することも必要です。次の図に示されているように、OSGi フレームワークには、異なるタイプの OSGi バンドルを含むことができます。



アプリケーション・バンドル

アプリケーション・バンドルは、アプリケーション・コードを含む OSGi バンドルです。OSGi バンドルは、自己完結型であるか、フレームワーク内の他のバンドルに依存する場合があります。これらの依存関係はフレームワークによって管理され、未解決の依存関係がある OSGi バンドルはフレームワークで実行できません。CICS においてフレームワークの外部でアプリケーションにアクセスできるようにするために、OSGi バンドルは、CICS メイン・クラスを OSGi サービスとして宣言する必要があります。PROGRAM リソースが CICS メイン・クラスを指し示す場合、OSGi フレームワーク外部の他のアプリケーションは Java アプリケーションにアクセスできます。1 つ以上のアプリケーションの共通ライブラリーを含む OSGi バンドルがある場合、Java 開発者は、CICS メイン・クラスを宣言しないことを決定することがあります。この OSGi バンドルは、フレームワーク内の他の OSGi バンドルからのみ使用可能です。

Java アプリケーションのデプロイメント単位は、CICS バンドルです。CICS バンドルは、任意の数の OSGi バンドルを含むことができ、1 つ以上の JVM サーバーにデプロイできます。JVM サーバーの管理とは無関係に、アプリケーション・バンドルを追加、更新、および除去することができます。

ミドルウェア・バンドル
: ミドルウェア・バンドルは、WebSphere MQ への接続などのシステム・サービスを実装するためのクラスを含む OSGi バンドルです。もう 1 つの例は、ネイティブ・コードを含み、OSGi フレームワークに 1 回だけロードする必要がある OSGi バンドルです。ミドルウェア・バンドルは、クラスを使用するアプリケーションではなく、JVM サーバーのライフサイクルで管理されます。ミドルウェア・バンドルは、JVM サーバーの JVM プロファイルで指定され、JVM サーバーの始動時に CICS によってロードされます。

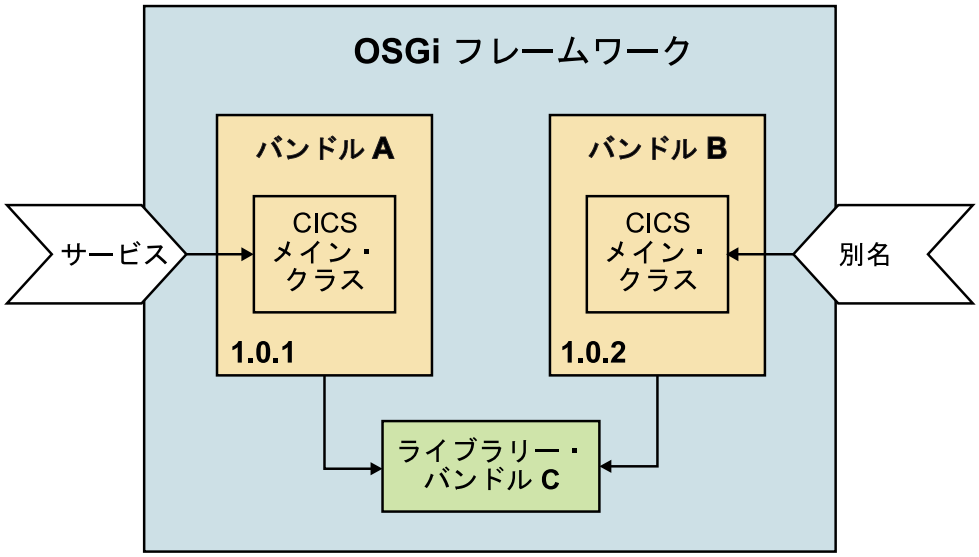
システム・バンドル
: システム・バンドルは、アプリケーションに主なサービスを提供するために CICS と OSGi フレームワーク間の対話を管理する OSGi バンドルです。主な例は、CICS サービスとリソースにアクセスできるようにする JCICS OSGi バンドルです。

Java アプリケーションの管理をシンプルにするために、以下のベスト・プラクティスに従ってください。

- アプリケーションを構成する密結合 OSGi バンドルを同一 CICS バンドルにデプロイします。密結合バンドルは、OSGi サービスを使用することなく、相互にクラスを直接エクスポートします。これらの OSGi バンドルをまとめて 1 つの CICS バンドルにデプロイして、一緒に更新し、管理します。
- アプリケーション間に依存関係が生じないようにします。代わりに、共通ライブラリーを別個の OSGi バンドルで作成し、独自の CICS バンドルで管理します。アプリケーションとは別個にライブラリーを更新できます。
- バンドル間の依存関係を作成する場合は、バージョンを使用する OSGi のベスト・プラクティスに従ってください。バージョンの範囲を使用すると、アプリケーションは、依存するバンドルへの両立可能な更新を許容できます。
- ツールがエラーを示していない場合でも、OSGi バンドルが使用するパッケージを常に明示的に宣言する必要があります。これを行うには、OSGi バンドル・マニフェストで `Import-Package` バンドル・ヘッダーを追加または更新します。Eclipse などのツールは、明示的な Import が必要なランタイム環境にとって適切ではない可能性のある **`javax.*`** パッケージの使用可能性について推測します。
- JVM サーバーの命名規則をセットアップし、システム・プログラマーと Java 開発者間でこの規則を合意します。
- シングルトン OSGi バンドルの使用は避けます。他のバンドルが従属しているシングルトン・バンドルを破棄すると、従属バンドルで障害が起こる可能性があります。

JVM サーバーにおける同一 Java アプリケーションの複数のバージョンの実行

OSGi フレームワークは、フレームワーク内で OSGi バンドルの複数バージョンの実行をサポートします。したがって、アプリケーションの使用可能状態を中断することなく、アプリケーションに段階的に更新を導入できます。同じ OSGi サービスの複数の実装形態をフレームワーク内にインストールすることができますが、そのサービスを呼び出した時には、バージョン・プロパティーが最も高いサービスが使用されます。CICS では、バージョン・プロパティーは、基礎となる OSGi バンドルから推定されます。そのため、同じ Java アプリケーションの複数バージョンを同時に JVM サーバーで実行する必要があり、別々のバージョンの OSGi バンドルに同じ CICS メイン・クラスがある場合は、いずれかの CICS メイン・クラスの定義で別名を使用する必要があります。別名を CICS メイン・クラスの宣言で指定すると、特定のバージョンのアプリケーション用の OSGi サービスとして OSGi フレームワークで登録されます。この別名を別の PROGRAM リソースで指定して、そのバージョンのアプリケーションを使用可能にします。



Liberty JVM サーバーでの Java アプリケーション

CICS では、軽量の Java サーブレットおよび JavaServer Pages を実行できる Java EE アプリケーション・サーバーが提供されています。開発者は、CICS 仕様に含まれる Liberty の豊富な機能を使用することにより、CICS 用の Java EE アプリケーションを作成できます。アプリケーション・サーバーは JVM サーバーの中で実行され、WebSphere Application Server Liberty に基づいて作成されています。

Liberty は、短時間で起動し、さまざまなプラットフォームで実行可能なアプリケーション開発用の軽量アプリケーション・サーバーです。これは、Java 開発者がアプリケーションの開発とテストの時間を短縮できるよう最適化されており、それにより Web サーバーを構成および開始するために必要な労力を最小限に抑えることができます。 Java 開発者は、無料で利用できる Eclipse のさまざまなツールを使用

することにより、アプリケーションと Web サーバーをまとめてパッケージ化して、デプロイメントを簡素化します。使用可能な Web サービス・サポートには、Java API for RESTful Web Services (JAX-RS) や Java API for XML Web Services (JAX-WS) などがあります。 Liberty について詳しくは、Liberty の概要を参照してください。

Liberty は CICS と共にインストールされ、JVM サーバー内のアプリケーション・サーバーとして実行されます。Liberty JVM サーバーは、Liberty で使用可能な機能のサブセットをサポートしています。 OSGi アプリケーション、Java サーブレット、および JSP ページを実行することができます。サポートされる機能について詳しくは、Liberty の機能を参照してください。

Liberty JVM サーバーとそれに関連するツールを使用する理由としては、以下のことが考えられます。

- 3270 画面を Web ブラウザーおよび RESTful クライアントで置き換えて、CICS アプリケーションの表示インターフェースを最新化する必要がある。
- Java 標準ベースの開発ツールを使用して、既存の他の CICS アプリケーションと共に Web クライアントをパッケージ化、併置、および管理する必要がある。
- WebSphere Application Server で既に Liberty アプリケーションを使用しており、それらを CICS で実行するために移植する必要がある。
- CICS において、既に Jetty または類似のサーブレット・エンジンを使用しており、Liberty に基づくアプリケーション・サーバーにマイグレーションする必要がある。
- データ・ソース定義を使用して、Java から Db2 データベースにアクセスする必要がある。Defining the CICS Db2 connectionを参照してください。
- Java Transaction API (JTA) を使用することにより、タイプ 4 JDBC データベース・ドライバーを介したリモート・リソース・マネージャーの更新と CICS リカバリー可能リソースの更新を調整する必要がある。
- JAX-RS を使用して、REST (Representational State Transfer) 原則に準拠するサービスを開発する必要がある。
- JAX-WS を使用して、標準のアノテーション・ベースのモデル・サポートにより、アプリケーションを開発する必要がある。
- JMS を介してセキュアなメッセージを送受信する Java EE アプリケーションを開発する必要がある。

Liberty アプリケーションにおける CICS 例外処理

Liberty アプリケーションは、JCICS API など、複数の異なるトランザクション API を使用できます。大部分の Liberty コンポーネント (EJB を除く) では、Java Transactions API (JTA) を明示的に使用して、これらの API 全体でトランザクションを調整する必要があります。例えば、JCICS とリモート JDBC アクティビティーで、アプリケーション・コードで発行された例外の後にロールバックする必要がある場合、JDBC 接続と対話する前に JTA トランザクションを開始する必要があります。

CICS は、Liberty でホストされる単純なサーブレットのための自動の rollback-CICS-transactions-on-Exception ポリシーを実施します。このポリシーによって、通常のサーブレットから例外がスローされると、CICS トランザクションが確

実にロールバックされます。JCICS API を使用する単純なサーブレット向けの基本的なトランザクション統合を提供する場合はこれで十分ですが、発生する可能性のある、より複雑なシナリオにはこのポリシーで対処できないものがあります。

例えば、rollback-CICS-transactions-on-Exception ポリシーは、リモート JDBC 接続やリモート JCA 接続などの他の非 CICS リソース・アダプターと統合されません。CICS と他のリソース・マネージャーの間のトランザクションを調整する必要がある場合、JTA を使用してトランザクションを明示的に調整する必要があります。これにより、Liberty、CICS、およびリモート・トランザクション・マネージャーは、トランザクションをコミットするか、ロールバックするかを連携してネゴシエーションするようになります。

rollback-CICS-transactions-on-Exception ポリシーは、単純なサーブレットに使用できますが、Liberty 環境で使用可能な拡張性ポイントの全範囲には使用できません。他のプラグイン、コールバック、および拡張ポイントを活用する上級ユーザーの場合、例外をスローした場合の CICS トランザクションの自動ロールバックが実行されない可能性があります。そのようなコンポーネントからスローされる例外に対して予測可能なトランザクション性が必要な場合、JTA を使用してトランザクションを調整します。あるいは、明示的な JCICS Abend を発行し、アプリケーションによって検出されたエラーに応じて、CICS に CICS トランザクションのロールバックを強制します。

Liberty アプリケーションの CICS タスク

Liberty アプリケーションで JCICS API や他の CICS リソース (タイプ 2 接続の JDBC DataSource など) を使用できるようにするために、要求は CICS タスクの下で実行する必要があります。CICS は、要求のタイプに応じたさまざまなタイミングで、アプリケーション要求のためのタスクを作成します。HTTP 要求の場合、Liberty アプリケーションが起動される前にタスクが作成されます。メッセージ駆動型 Bean (MDB)、インバウンド JCA、リモート EJB など、その他のタイプの要求の場合、タスクは必要に応じて作成されます。

アプリケーションが CICS と対話しない場合、非 HTTP 要求に対して CICS タスクは作成されません。

CICS タスクが作成されると、CICS は以下のアクションを実行します。

- CICS トランザクションのセキュリティー検査が実行されます。
- タスクに対する CICS モニターが開始します。
- タスクに対する CICS トレースが開始します。
- Java スレッドの名前が変更され、CICS タスク番号とトランザクション ID が含まれるようになります。

非 HTTP アプリケーションの場合、JCICS API または タイプ 2 接続の JDBC DataSource が初めて使用されたときに、これらのアクションが実行されます。アプリケーションが CICS と対話しない場合、CICS モニターおよびトランザクション・セキュリティーは実行されません。

キャッチされていない Java 例外に対する CICS 異常終了処理は、CICS タスクがない場合は適用されません。JCICS API またはタイプ 2 接続の JDBC DataSource

が使用される前にアプリケーションによって例外がスローされると、AJ05 異常終了は発生しません。

Java Web サービス

CICS には、Java Web サービスを実行するための Axis2 テクノロジーが組み込まれています。Axis2 は、Apache Foundation からのオープン・ソース Web サービス・エンジンであり、Java 環境で SOAP メッセージを処理するために CICS に提供されています。

Axis2 は、複数の Web サービス仕様をサポートする、Web サービス SOAP エンジンの Java の実装です。また、Axis2 で実行できる Java アプリケーションの作成方法を記述するプログラミング・モデルも提供します。Axis2 は、Java 環境で Web サービスを処理するために CICS で提供されているため、zAAP プロセッサーへの適格な Java 処理のオフロードをサポートします。

JVM サーバーは、既存の Web サービスを変更することなく、SOAP パイプラインでインバウンドおよびアウトバウンド SOAP メッセージを処理するための Axis2 の実行をサポートします。ただし、Java アプリケーションから Web サービスを作成し、同じ JVM サーバーで実行することもできます。JVM サーバーの Axis2 リポジトリーにアプリケーションをデプロイすることによって、Java アプリケーションと SOAP 処理の両方が、zEnterprise Application Assist Processor (zAAP) で実行できるようになります。

次のいずれかの理由で Java Web サービスを使用できます。

- 他のプラットフォームで Axis2 Web サービスの経験があり、CICS で Web サービスを作成したい。
- 標準の Java API を使用して、Axis2 に統合される Java データ・バインディングを作成したい。
- CICS Web サービス・アシスタントで処理するのが困難な複雑な WSDL 文書がある。

以下の例では、Web サービスで Java を使用する方法を説明しています。

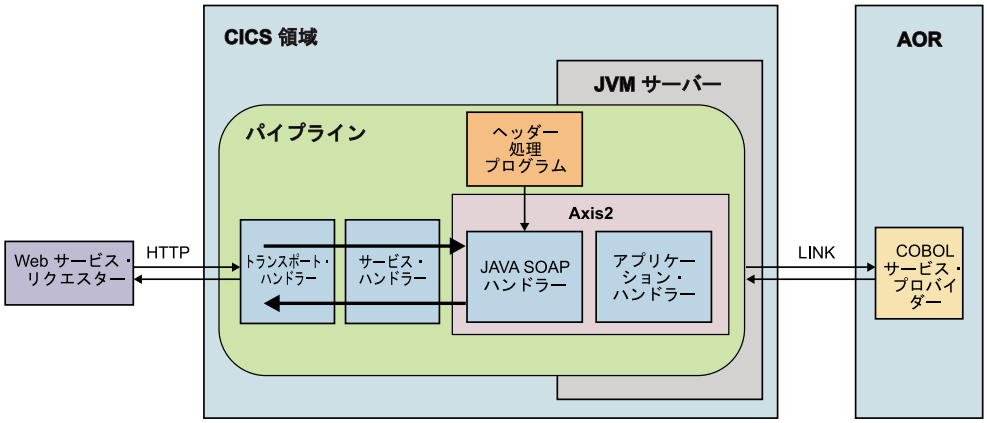
JVM サーバーにおける SOAP メッセージの処理

Web サービス・パイプラインで発生する大部分の SOAP 処理は、SOAP ハンドラーおよびアプリケーション・ハンドラーによって実行されます。オプションとして、JVM サーバーでこの SOAP 処理を実行し、zAAP を使用して作業を実行できます。 COBOL、C、C++、または PL/I で作成される Web サービス・アプリケーションを引き続き使用できます。

既存の Web サービスがある場合、JVM サーバーを使用するようにパイプラインの構成を更新できます。Web サービスを変更する必要はありません。パイプラインが SOAP ヘッダー処理プログラムを使用する場合、Axis2 プログラミング・モデルを使用することによって、Java でそのプログラムを再作成するのが最善の方法です。このヘッダー処理プログラムは、追加のデータ変換を行うことなく、Java オブジェクトを Axis2 と共用できます。例えば、COBOL のヘッダー処理プログラムがある

場合、データが Java から COBOL に変換された後、再び戻す必要があります。これにより、SOAP 処理のパフォーマンスが低下する場合があります。

次の図に示されているシナリオは、Web サービス・プロバイダーである COBOL アプリケーションの例です。要求は、Java をサポートするように構成されているパイプラインで処理されます。SOAP ハンドラーおよびアプリケーション・ハンドラーは、Axis2 によって処理され、JVM サーバーで実行される Java プログラムです。アプリケーション・ハンドラーは、データを XML から COBOL に変換し、アプリケーションにリンクします。



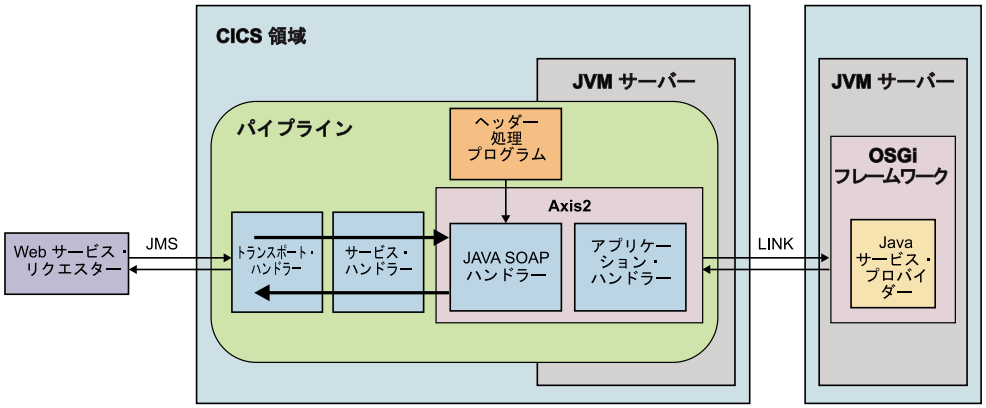
環境を計画する際には、JVM サーバーに 1 組の専用領域を使用するようにしてください。この例では、COBOL アプリケーションが実行されているアプリケーション専有領域 (AOR) は、JVM サーバーが実行される CICS 領域とは別です。ワークロード管理を使用すると、例えば **EXEC CICS LINK** ではアプリケーション・ハンドラーから、インバウンド要求では Web サービス・リクエスターからのように、ワークロードのバランスを取ることができます。

CICS Web サービス・アシスタントからの出力を使用する Java アプリケーションの作成

言語構造を解釈し、CICS Web サービス・アシスタントによって生成されるデータ・バインディングを使用する Java アプリケーションを作成できます。Web サービス・アシスタントは、WSDL から言語構造を作成し、言語構造から WSDL を作成することができます。また、このアシスタントは、SOAP 処理時に XML とターゲット言語間でデータを変換する方法を記述する Web サービス・バインディングも作成します。

アシスタントを使用して言語構造を生成する場合は、IBM Record Generator for Java または Rational J2C ツールを使用して言語構造を処理して、Java クラスを生成できます。これらのツールを使用すると、Java 開発者は他の CICS アプリケーションと対話できます。この例では、これらのツールを使用して、CICS が XML からデータを変換した後にインバウンド SOAP メッセージを処理できる Java アプリケーションを作成することができます。詳しくは、Java からの構造化データとの対話を参照してください。

次の図に示されているシナリオは、Web サービス・プロバイダーである Java アプリケーションの例です。SOAP 処理は、JVM サーバーで Axis2 によって処理されます。アプリケーション・ハンドラーがリンクする Java アプリケーションは、1 つ以上の OSGi バンドルとしてパッケージされ、デプロイされ、JVM サーバーで実行されます。



この方法の利点は、データ・バインディングが Web サービス・アシスタントによって生成されたため、CICS では Web サービスは WEBSERVICE リソースによって表されることです。CICS で統計、リソース管理、およびその他の機能を使用して、Web サービスを管理できます。欠点は、Java 開発者が、慣れていないプログラミング言語の言語構造を処理しなければならないことです。

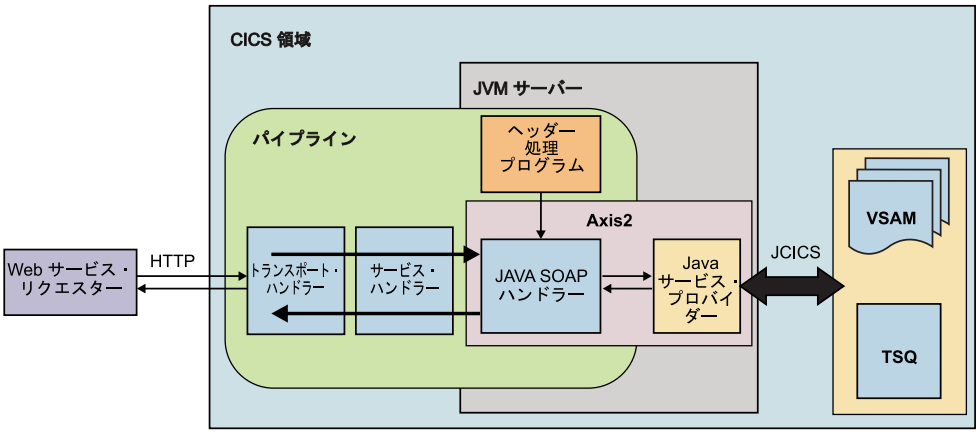
このタイプのアプリケーション用の環境を計画する場合は、アプリケーションを実行するために別個の JVM サーバーを使用してください。

- さまざまなワークロードに合わせて JVM サーバーをより効率よく管理し、調整することができます。
- インバウンド要求および **EXEC CICS LINK** でワークロード管理を使用して、ワークロードのバランスを取り、環境を拡大することができます。
- CICS における OSGi サポートを利用して、Java アプリケーションを管理できます。

Java データ・バインディングを使用する Java アプリケーションの作成

SOAP メッセージの XML を生成し、解析する Java アプリケーションを作成できます。Java API は、XML を処理するために標準の Java ライブラリーを提供します。例えば、Java Architecture for XML Binding (JAXB) を使用して、Java データ・バインディングを作成し、Java API for XML Web Services (JAX-WS) ライブラリーを使用して、XML を生成し、解析することができます。これらのライブラリーを使用すると、アプリケーションは、SOAP パイプライン処理と同じ JVM サーバーの Axis2 で実行できます。

次の図に示されているシナリオは、Web サービス・プロバイダーであり、JVM サーバー内の Axis2 SOAP エンジンによって処理される Java アプリケーションの例です。



Java アプリケーションは Java データ・バインディングを使用し、Java SOAP ハンドラーと対話するので、アプリケーション・ハンドラーはありません。この例では、Web サービス・リクエスターは HTTP を使用して CICS 領域に接続しますが、JMS も使用することができます。Java アプリケーションは JCICS を使用して CICS サービス (この例では VSAM ファイルと一時記憶域キュー) にアクセスします。

この方法の利点は、Java 開発者が使い慣れたテクノロジーを使用してアプリケーションを作成することです。また、Java 開発者は、Web サービス・アシスタントが処理できない複雑な WSDL 文書を使用して、バインディングを作成できます。ただし、この方法には、次のようないくつかの制限があります。

- このタイプのアプリケーションには WS-Security を使用できません。したがって、セキュリティーを使用したい場合は、SSL を使用して接続を保護してください。
- パイプライン処理でユーザー ID のコンテキスト切り替えが行われません。要求でユーザー ID を変更するには、URIMAP リソースを使用してください。
- Web サービス・アシスタントからの Web サービス・バインディングを使用しないため、WEBSERVICE リソースがありません。
- アプリケーションが Web サービス・リクエスターである場合、パイプライン処理はバイパスされます。したがって、パイプラインで使用可能なサービス品質が得られません。

CICS 領域にワークロード管理を実装する場合は、このタイプのワークロードの経路指定方法を計画する必要があります。Java アプリケーションは SOAP 処理と同じ JVM サーバーで実行されるため、CICS は経路指定を行う機会を提供しません。ただし、経路指定が必要な場合は、JAX-WS アプリケーションで他のプログラムに対して分散プログラム・リンクを実装できます。

第 2 章 Java アプリケーションの開発

CICS サービスを使用し、CICS 制御下で実行される Java アプリケーション・プログラムを作成できます。IBM CICS SDK for Java を使用すると、JCICS クラス・ライブラリーを使用して CICS リソースにアクセスし、他の言語で作成されるプログラムと対話するアプリケーションを開発できます。さまざまなプロトコルおよびテクノロジー (Web サービスや CICS Transaction Gateway など) を使用して、Java プログラムに接続することもできます。

CICS は、Java アプリケーションをサポートするためのツールやランタイム環境を提供します。IBM CICS SDK for Java は、Eclipse ベースのツールであり、Java アプリケーションを開発し、CICS にデプロイするためのサポートを提供します。CICS リソースやサービスにアクセスするアプリケーションを開発するための JCICS クラス・ライブラリーが含まれています。例えば、VSAM ファイル、一時データ・キュー、および一時記憶にアクセスできます。また、JCICS を使用して、他の言語 (COBOL や C など) で作成された CICS アプリケーションにリンクすることもできます。IBM CICS SDK for Java には、CICS 用の Java アプリケーションの開発の初心者が開始できるようにする一連のサンプルが含まれています。

CICS について必要な知識

CICS は、ユーザーが要求によってアプリケーションを実行するためのサービスを提供するトランザクション処理のサブシステムです。多数のユーザーが、同じファイルとプログラムを使用する同じアプリケーションを同時に実行する要求を実行依頼することが可能です。 CICS は、リソースの共用、データの保全性、実行の優先順位付けを管理する一方で、短い応答時間を維持します。

CICS アプリケーションは、製品オーダーの処理や会社の給与計算の準備などの業務を連携して実行する、関連したプログラムの集合です。CICS アプリケーションは、CICS 制御下で実行され、CICS サービスとインターフェースを使用してプログラムとファイルにアクセスします。

CICS アプリケーションを実行するには、トランザクション 要求を実行依頼します。 CICS では、トランザクションという用語に特別な意味があります。CICS での意味と、業界でより一般的に使用される意味との違いについては、 28 ページの『CICS トランザクション』を参照してください。トランザクションの実行は、必要な機能を実装する 1 つ以上のアプリケーション・プログラムの実行で構成されます。

CICS 用の Java アプリケーションを開発するには、CICS プログラム、トランザクション、およびタスク間の関係を理解する必要があります。これらの用語は、CICS 資料全体で使用され、多くのプログラミング・コマンドで表示されます。また、ランタイム環境において CICS が Java アプリケーションを処理する方法も理解しておく必要があります。

© Copyright IBM Corp. 1999, 2018

CICS トランザクション

トランザクションは、単一の要求によって開始される 1 つの処理です。

要求は通常、ユーザーによって端末で行われます。ただし、Web ページから、リモート・ワークステーション・プログラムから、または別の CICS 領域のアプリケーションから行われる場合があります。もしくは、事前定義された時点に自動的にトリガーされる場合もあります。CICS Web サポートの概念と構造およびCICS 外部インターフェースの概要で、CICS トランザクションのさまざまな実行方法を説明します。

単一トランザクションは、1 つ以上のアプリケーション・プログラム で構成されます。これらのアプリケーション・プログラムが実行されると、必要な処理を実行します。

ただし、CICS ではトランザクション という用語は、単一イベントと、同じタイプの他のすべてのトランザクションの両方を意味するのに使用されます。CICS に対し、それぞれのトランザクション・タイプを、TRANSACTION リソース定義を使用して記述します。この定義により、トランザクション・タイプに名前 (トランザクション ID、すなわち TRANSID) が指定され、実行される作業に関する複数の項目が CICS に指示されます。例えば、最初にどのプログラムを呼び出すか、トランザクションの実行全体でどの種類の認証が必要であるかなどです。

トランザクションを実行するには、その TRANSID を CICS に対して送信します。CICS は、TRANSACTION 定義に記録された情報を使用して正しい実行環境を確立し、最初のプログラムを開始します。

トランザクション という用語は、リカバリー単位、または CICS で作業単位 と呼ばれるものを記述するために、IT 業界で広く使用されています。一般に、これはリカバリー可能な完全な論理オペレーションです。プログラムされたコマンド、またはシステム障害の発生によって、トランザクション全体をコミットまたはバックアウトすることができます。多くの場合、CICS トランザクションの有効範囲は単一の作業単位でもありますが、CICS 資料を読むときは、意味の違いを認識する必要があります。

CICS タスク

タスクは、トランザクションを実行する単一のインスタンスです。

CICS では、タスク という用語に特別の意味があります。CICS はトランザクションの実行要求を受け取ると、トランザクション・タイプの実行のこの *1* つのインスタンス に関連した新規タスクを開始します。すなわち、CICS タスクは、通常は特定のユーザーのために、データの独自の専用セットを使用した、トランザクションの 1 つの実行です。また、タスクをスレッド と見なすこともできます。タスクは、優先順位と準備度にしたがって CICS によってディスパッチ されます。トランザクションが完了すると、タスクは終了します。

CICS アプリケーション・プログラム

Java プログラムでは、CICS 用の Java クラス・ライブラリー (JCICS) を使用して、CICS サービスにアクセスし、他の言語で作成されたアプリケーション・プログラムにリンクすることができます。

CICS アプリケーション・プログラムは COBOL、C、C++、Java、PL/I、またはアセンブラー言語で作成できます。大部分の処理ロジックは標準言語ステートメントで表されますが、CICS サービスを要求するには、アプリケーションは提供されたアプリケーション・プログラミング・インターフェースを使用します。

COBOL、C、C++、PL/I、またはアセンブラー・プログラムは、**EXEC CICS** アプリケーション・プログラミング・インターフェースまたは C++ クラス・ライブラリーを使用できます。Java プログラムは JCICS クラス・ライブラリーを使用します。JCICS については、 43 ページの『CICS 用 Java クラス・ライブラリー (JCICS)』 で説明しています。

CICS サービス

Java プログラムは、JCICS プログラミング・インターフェースを介して、データ管理サービス、通信サービス、作業単位サービス、プログラム・サービス、および診断サービスの各 CICS サービスにアクセスできます。

CICS サービス・マネージャーの名称には、通常は「管理」または「制御」という語が含まれています (例えば、「端末管理」、「プログラム制御」など)。これらの用語は、CICS 資料で幅広く使用されています。

データ管理サービス

CICS が提供するデータ管理サービスは、次のとおりです。

- 仮想記憶アクセス方式 (VSAM) データ・セットにアクセスする際の、保全性のあるレコード・レベル共用。データのバックアウト (トランザクション障害またはシステム障害の場合)、または順方向リカバリー (メディア障害の場合) をサポートするために、CICS はアクティビティーをログに記録します。 CICS ファイル制御は、VSAM データを管理します。

  また、CICS は 2 つの専有ファイル構造も実装し、それらを操作するためのコマンドを提供します。

  一時記憶

  一時記憶 (TS) は、複数のトランザクションからデータを容易に使用可能にする手段です。データは、プログラムからの要求に応じて作成されるキューに保持されます。キューには順次にアクセスするか、または項目番号でアクセスすることができます。

  一時記憶域キューは、メインメモリーに常駐することも、ストレージ・デバイスに書き込むこともできます。

  一時記憶域キューは、名前付きのスクラッチパッドと見なすことができます。

  一時データ

  一時データ (TD) も複数のトランザクションから使用可能であり、キューに保持されます。ただし、TS キューとは異なり、TD キューは事前定義する必要があり、順次にしか読み取れません。各項目は、読み取られるとキューから除去されます。

  一時データ・キューは常にデータ・セットに書き込まれます。一時データ・キューは、特定数の項目が書き込まれると、特定トランザクション

を開始するトリガーの役目をするように定義できます。 例えば、起動したトランザクションによってそのキューを処理できます。

- データベース製品とのインターフェースを使用した、他のデータベース (Db2 を含む) 内のデータへのアクセス。

通信サービス

CICS は、SNA と TCP/IP プロトコルを使用して、さまざまな端末 (ディスプレイ、プリンター、およびワークステーション) へのアクセスを可能にするコマンドを備えています。CICS 端末管理により、SNA ネットワークおよび TCP/IP ネットワークを管理できます。

拡張プログラム間通信機能 (APPC) コマンドを使用して、SNA プロトコルを使用してリモート・システム内の他のプログラムを開始し、通信するプログラムを作成できます。CICS APPC は、ピアツーピア分散アプリケーション・モデルを実装します。

次の CICS 専有通信サービスが提供されます。

機能シップ
: リモート CICS 領域で既存のものとして定義されるリソース (ファイル、キュー、およびプログラム) にアクセスするプログラム要求は、自動的に CICS によって専有領域に転送されます。

分散プログラム・リンク **(DPL)**
: リモート CICS 領域で既存のものとして定義されるプログラムに対するプログラム・リンク要求は、自動的に専有領域に転送されます。CICS は、分散アプリケーションの保全性を維持するためのコマンドを提供します。

非同期処理
: CICS は、プログラムが同じ CICS 領域またはリモート CICS 領域内の別のトランザクションを開始し、オプションとしてデータをそのトランザクションに渡すことを可能にするコマンドを提供します。新しいトランザクションは、新しいタスク内で独立してスケジュールされます。この機能は、他のソフトウェア製品によって提供される *fork* 操作に似ています。

トランザクション・ルーティング
: リモート CICS 領域で既存のものとして定義されるトランザクションを実行する要求は、自動的に専有領域に転送されます。ユーザーへの応答は、要求を受け取った領域に返されます。

作業単位サービス

CICS がトランザクションを実行する新しいタスクを作成すると、新しい作業単位 (UOW) が自動的に開始されます BEGIN コマンドは必要ないため、CICS はこのコマンドを提供しません。CICS トランザクションは常にトランザクション内で実行されます。

CICS は、実行されたリカバリー可能な作業をコミットまたはロールバックするために SYNCPOINT コマンドを提供します。同期点が完了すると、CICS は自動的に別

の作業単位を開始します。SYNCPOINT コマンドを発行せずにプログラムを終了すると、CICS は暗黙的な同期点を取り、トランザクションをコミットしようとします。

コミットの有効範囲には、リカバリー可能として定義されたすべての CICS リソース、および CICS によって提供されたインターフェースを使用してインタレストを登録した他のすべてのリソース・マネージャーが含まれます。

プログラム・サービス

CICS は、プログラムが別のプログラムにリンクするか、制御を転送してから戻ることを可能にするコマンドを提供します。

診断サービス

CICS が提供するコマンドを使用して、プログラムをトレースし、ダンプを作成できます。

CICS での Java ランタイム環境

CICS は、スレッド・セーフ Java アプリケーションを実行するための JVM サーバー環境を提供します。スレッド・セーフではないアプリケーションは、JVM サーバーを利用することはできません。

JVM サーバーは、単一の JVM で複数のタスクを実行できるランタイム環境です。この環境はそれぞれの Java タスクで必要な仮想ストレージの量を削減し、CICS が多数のタスクを同時に実行できるようにします。

CICS タスクは、同じ JVM サーバー・プロセス内のスレッドとして並列で実行されます。 JVM は、複数のアプリケーションを同時に実行している可能性のあるすべての CICS タスクにより共用されます。すべての静的データおよび静的クラスも共用されます。このため、CICS で JVM サーバーを使用するには、Java アプリケーションがスレッド・セーフである必要があります。各スレッドは T8 TCB で実行され、JCICS API を使用して CICS サービスにアクセスできます。

アプリケーション内で System.exit() メソッドを使用しないでください。このメソッドにより、JVM サーバーと CICS の両方がシャットダウンし、アプリケーションの状態と可用性に影響を与えます。

マルチスレッド・アプリケーション

新しいスレッドを開始したり、スレッドを開始するライブラリーを呼び出すためにアプリケーション・コードを作成できます。アプリケーションにスレッドを作成する場合、OSGi レジストリーから汎用 ExecutorService を使用する方式が推奨されています。アプリケーションが JVM サーバーで実行されている場合には、ExecutorService は自動的に CICSExecutorService を使用して CICS スレッドを作成します。この方式により、アプリケーションを他の環境に移植することが容易になり、特定の JCICS API メソッドを使用する必要がなくなります。

しかし、CICS に固有のアプリケーションを作成している場合は、JCICS API 内の CICSExecutorService クラスを使用して、新しいスレッドを要求することができます。

どちらの方法を選択した場合にも、新しく作成されたスレッドは CICS タスクとして実行され、CICS サービスにアクセスできます。 JVM サーバーが使用不可にされると、CICS は、JVM で実行中の CICS タスクすべてが終了するまで待機します。ExecutorService クラスまたは CICSExecutorService クラスを使用することにより CICS は実行中のタスクを認識するので、アプリケーションの作業完了後に JVM サーバーをシャットダウンさせることが可能になります。

JCICS オブジェクトは、そのオブジェクトを作成したタスクでのみ使用してください。それらのオブジェクトをタスク間で共用しようとすると、予測不能な結果をもたらす可能性があります。

CICS ExecutorService の使用の詳細については、 46 ページの『スレッド』を参照してください。

JVM サーバーの始動とシャットダウン

静的データは JVM サーバーで実行中のすべてのスレッドで共用されるため、静的データを初期化して JVM のシャットダウン時に適切な状態にするための OSGi バンドル・アクティベーター・クラスを作成できます。JVM サーバーは、例えば、JVM の構成を変更したり、問題を修正したりするために、管理者が使用不可にするまで実行されます。バンドル・アクティベーター・クラスを提供することにより、アプリケーションにとって適切な状態を確実に設定できます。 CICS にはタイムアウトがあります。このタイムアウトは、JVM サーバーの開始または停止を続行するまでにこれらのクラスが完了するのを待機する時間を指定します。開始クラスと終了クラスで JCICS を直接使用することはできません。ただし、開発者が CICSExecutorService.runAsCICS() API を使用して、アクティベーターから新しい JCICS 対応スレッドを開始することは可能です。 JCICS コマンドはすべて、インストール・コマンドを実行したユーザー ID の権限で実行されます。そのため、管理者は、バンドル・アクティベーターをインストールする前に、その中で使用されるリソースを把握しておく必要があります。

IBM CICS SDK for Java を使用したアプリケーションの開発

CICS Explorer には、IBM CICS SDK for Java およびオプションで IBM CICS SDK for Java EE and Liberty が含まれています。IBM CICS SDK for Java は、OSGi および Web プロジェクトのサポートを含め、Java アプリケーションを開発して CICS に配置するための環境を提供します。

IBM CICS SDK for Java を使用して、OSGi 仕様に準拠するように、新しいアプリケーションを作成したり、既存の Java アプリケーションを再パッケージ化したりできます。 OSGi には、コンポーネント・モデルを使用してアプリケーションを開発し、それらのアプリケーションを OSGi バンドルとして 1 つのフレームワークにデプロイするためのメカニズムが用意されています。*OSGi バンドル* は、アプリケーションのデプロイメントの単位であり、バージョン情報、依存関係、およびアプリケーション・コードが入っています。OSGi の主な利点は、*Java パッケージ* と呼ばれる明確に定義されたインターフェースを介してのみアクセスされる再使用可能コンポーネントから、アプリケーションを作成できることです。その後、OSGi サービスを使用して、Java パッケージにアクセスできます。また、Java アプリケー

ションのライフサイクルと依存関係をきめ細かく管理することもできます。 OSGi を使用したアプリケーションの開発については、OSGi Allianceを参照してください。

IBM CICS SDK for Java を使用して、サポートされるリリースの CICS で動作する Java アプリケーションを開発できます。異なるリリースの CICS は、別々のバージョンの Java をサポートします。また、JCICS API も、CICS のより多くの機能をサポートするように後のリリースで拡張されています。。間違ったクラスの使用を防ぐ目的で、IBM CICS SDK for Java には、ターゲット・プラットフォームまたはプロジェクト・ライブラリーをセットアップする機能が用意されています。どのリリースの CICS 用に開発するかを定義すると、IBM CICS SDK for Java は、使用できない Java クラスを自動的に非表示にします。

Liberty JVM サーバーを使用している場合、IBM CICS SDK for Java は、動的 Web プロジェクトおよび OSGi アプリケーション・プロジェクトの処理に役立ちます。JCICS を使用して CICS サービスにアクセスする、最新の Web レイヤーおよびビジネス・ロジックを備えたアプリケーションを作成できます。別の OSGi バンドルからのコードに Web アプリケーションでアクセスする必要がある場合、OSGi アプリケーション・プロジェクト (EBA ファイル) としてそれをデプロイする必要があります。アプリケーション・マニフェストの中にもう一方の OSGi バンドルを含めるか、共通ライブラリーとして Liberty `bundle_repository` の中にもう一方のバンドルをインストールする必要があります。アプリケーションへの入り口点を提供し、それを URL として Web ブラウザーに公開するには、Web 使用可能 OSGi バンドル (WAB ファイル) を EBA ファイルに含める必要があります。

ターゲット・プラットフォームの更新

サード・パーティーの Java クラスを Eclipse 開発環境のターゲット・プラットフォームに追加できます。

始める前に

サード・パーティーの Java クラスが入っている JAR ファイルを OSGi プラグインとして使用できることと、そのファイルがローカル・ワークステーションにコピーされていることを確認します。 **CICS TS V5.5** と **Java EE** および **Liberty** ターゲット・プラットフォームが既に構成されていることを確認します。

このタスクについて

CICS Explorer Software Development Kit (SDK) には、CICS API や Web API を使用するために必要な Java クラスしか用意されていません。その他のインターフェースのサポートを追加するには、サード・パーティーの JAR が入っている OSGi プラグインを Eclipse ターゲット・プラットフォームに追加する必要があります。そうすれば、そのターゲット・プラットフォームを使用するすべてのアプリケーションで、エクスポートしたパッケージを使用できるようになります。

手順

1. Eclipse で「ウィンドウ」 > 「設定」 > 「ターゲット・プラットフォーム」を選択します。

2. ターゲット定義から **CICS TS V5.5** と **Java EE** および **Liberty** ターゲット・プラットフォームを選択します。
3. 「編集」をクリックし、「ロケーション」タブにある「追加」をクリックします。サード・パーティーのバンドル JAR が入っているディレクトリーを表示します。
4. 「次へ」をクリックします。 OSGi プラグインの内容が表示されます。
5. 「終了」 >をクリックし、再び「終了」 >をクリックしてから、「**OK**」をクリックします。

タスクの結果

OSGi 環境を更新して、Java アプリケーション開発に必要なサード・パーティーの OSGi バンドルと CICS の OSGi バンドルを組み込むことができました。

次のタスク

Java アプリケーションを CICS JVM サーバーにデプロイして、サード・パーティーの JAR を、OSGi ミドルウェア・バンドルとして追加するか、Liberty 共用バンドル・リポジトリーに追加します。詳しくは、OSGi ミドルウェア・バンドルの更新およびserver.xml の手動調整を参照してください。

プラグイン・プロジェクトの作成

CICS Java アプリケーションを、OSGi 仕様に準拠する Eclipse プラグイン・プロジェクトとして作成します。OSGi サービス・プラットフォームは、コンポーネント・モデルを使ってアプリケーションを開発し、それらのアプリケーションを OSGi バンドルとしてフレームワークに配置するためのメカニズムを提供します。

プラグイン・プロジェクトは OSGi バンドルであり、CICS Java アプリケーションで必要とされるすべてのファイルと成果物が含まれます。プラグイン・プロジェクトは、CICS バンドル・プロジェクトに組み込まれてから、ホスト・システムにエクスポートされます。

始める前に

ターゲット・プラットフォームを設定する必要があります。詳しくは、ターゲット・プラットフォームの更新を参照してください。

このタスクについて

このタスクでは、新しいプラグイン・プロジェクトを作成します。特に記載されている場合を除き、設定はデフォルト値のままにしておくことができます。プロジェクトを作成する場合、マニフェストを編集して、JCICS API の依存関係を追加する必要があります。

手順

1. Eclipse メニュー・バーで、「ファイル」 > 「新規」 > 「プロジェクト **(Project)**」をクリックして、「新規プロジェクト」ウィザードを開きます。

2. 表示されるリストから、「プラグイン・プロジェクト **(Plug-in Project)**」を選択し、「次へ」をクリックして「新規プラグイン・プロジェクト (New Plug-in Project)」ウィザードを開きます。
3. 「プロジェクト名」フィールドに、プロジェクトの名前を入力します (例えば `com.ibm.cics.example.accounting`)。「ターゲット・プラットフォーム」セクションで、「**OSGi** フレームワーク **(an OSGi framework)**」を選択して、メニューから「標準」を選択します。「次へ **(Next)**」をクリックします。 「コンテンツ (Content)」ペインが表示されます。
4. 「バージョン」フィールドで、バージョン番号の最後から 『.qualifier』 を削除します。
5. 「実行環境 **(Execution Environment)**」フィールドで、CICS ランタイム・ターゲット・プラットフォームの実行環境に一致する Java のレベルを選択します (例えば「**JavaSE-1.7**」)。
6. 「アクティベーターを生成 **(Generate an activator)**」チェック・ボックスのチェック・マークを外して、「終了」をクリックします。 「パッケージ・エクスプローラー (Package Explorer)」ビューで、新しいプラグイン・プロジェクトが作成されます。
7. 必須: ここでプラグイン・マニフェスト・ファイルを編集して、`JCICS` および `com.ibm.record` API の依存関係を追加する必要があります。これらのステップを実行しない場合、バンドルをエクスポートしてインストールできますが、動作しません。
   a. 「パッケージ・エクスプローラー」ビューで、プロジェクト名を右クリックして、「プラグイン・ツール」 > 「マニフェストを開く」をクリックします。 マニフェスト・ファイルがマニフェスト・エディターで開きます。
   b. 「依存関係 **(Dependencies)**」タブを選択して、「インポートされたパッケージ (Imported Packages)」セクションで、「追加」をクリックします。 「パッケージの選択 (Package Selection)」ダイアログが開きます。
   c. パッケージ `com.ibm.cics.server` を選択して、「**OK**」をクリックします。 パッケージが「インポートされたパッケージ (Imported Packages)」リストに表示されます。
   d. オプション: アプリケーションで必要な場合には、前述のステップを繰り返して、以下のパッケージをインストールします。

      `com.ibm.record`
      VisualAge に付属の Java レコード・フレームワークから IByteBuffer を使用するレガシー・プログラム用の Java API。以前は `dfjcics.jar` ファイル内にありました。
   e. 「ファイル」 > 「保管」を選択してマニフェスト・ファイルを保管します。

タスクの結果

JCICS API の依存関係を含む新しいプラグイン・プロジェクトが作成されました。

次のタスク

これで、CICS Java アプリケーションを作成できます。 CICS 用の Java アプリケーションの開発が初めての場合は、IBM CICS SDK for Java とともに提供される JCICS のサンプルを使用して開始することができます。

注: アプリケーションを開発したら、CICS-MainClass 宣言をマニフェスト・ファイルに追加し、アプリケーションで使用されるクラスを宣言する必要があります。詳しくは、関連リンクを参照してください。

プラグインの開発について詳しくは、Eclipse のヘルプ文書の *Plug-in Development Environment (PDE)* の『*User Guide*』セクションを参照してください。

Java アプリケーションが終了したら、CICS バンドル内のそのアプリケーションを zFS にデプロイする必要があります。CICS バンドルは、1 つ以上のプラグインを含むことができ、CICS のアプリケーションの配置の単位です。

プラグイン・プロジェクトのマニフェスト・ファイルの更新

JCICS アプリケーションを開発する場合、または既存のアプリケーションをプラグイン・プロジェクトにパッケージ化する場合には、プロジェクトのマニフェスト・ファイルを更新して `CICS-MainClass` ヘッダーを含める必要があります。

このタスクについて

`CICS-MainClass` ヘッダーは、 `LINK`、`START`、または `RUN` コマンド、あるいはトランザクション初期プログラムで呼び出し可能なクラスを宣言するために使用します。 CICS メインクラスを宣言する OSGi バンドルに、「遅延」アクティベーション・ポリシーを使用しないでください。CICS は、OSGi フレームワークで OSGi バンドルが開始されると、すぐにそれらをアクティブにします。この宣言は、手動でマニフェスト・ファイルに追加する必要があります。

手順

1. マニフェスト・ファイルがエディターでまだ開かれていない場合は、「パッケージ・エクスプローラー」ビューでプロジェクト名を右クリックして、「プラグイン・ツール」>「マニフェストを開く」をクリックします。 マニフェスト・ファイルがマニフェスト・エディターで開きます。
2. 「**MANIFEST.MF**」タブを選択します。ファイルの内容が表示されます。
3. 次の宣言をマニフェスト・ファイルに追加します。 `CICS-MainClass:`*`packagename.classname`* ここで:

   packagename
   完全修飾 Java パッケージ名です。

   classname
   アプリケーションで使用されるクラスの名前です。複数のクラスを使用する場合、*`packagename.classname`* エレメントをコンマで区切って繰り返します。

   `CICS-MainClass` ヘッダーでは、別名を使用できます。例えば、宣言 `CICS-MainClass: examples.hello.HelloCICSWorld; alias=greeting` は、別名 `greeting` を `CICS-MainClass examples.hello.HelloCICSWorld` に割り当てま

す。プログラムを CICS に定義する際は、クラス名の代わりに別名 greeting を使用します。同じプログラムの複数のバージョンがあり、それぞれが同じクラス名を持つ場合には、別名が便利です。 別名を使用することによって、異なるバージョンを識別できます。

以下の例は、HelloCICSWorld および HelloWorld クラスに CICS-MainClass ヘッダーを設定したマニフェスト・ファイルを示しています。

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Hello Plug-in
Bundle-SymbolicName: com.ibm.cics.server.examples.hello
Bundle-Version 1.0.0
Bundle-RequiredExecutionEnvironment: JavaSE-1.7
Import-Package: com.ibm.cics.core.bundle,
  com.ibm.cics.core.model.builders,
  com.ibm.cics.server;version="[1.300.0,2.0.0)"
CICS-MainClass: examples.hello.HelloCICSWorld,
  examples.hello.HelloWorld
```

4. クラス宣言をすべて追加した後、「ファイル」>「保管」を選択してマニフェスト・ファイルを保管します。

タスクの結果

これで、プラグイン・プロジェクトを CICS バンドルに追加して、zFS に配置できるようになりました。CICS バンドルは、1 つ以上のプラグインを含むことができ、CICS のアプリケーションの配置の単位です。

次のタスク

CICS バンドル・プロジェクトを作成します。CICS Explorer 製品資料内の『CICS バンドル・プロジェクトの作成』を参照してください。

Java EE アプリケーションの作成

CICS Explorer および IBM CICS SDK for Java のヘルプには、アプリケーションを開発して配置する以下のステップの実行方法が詳細に説明されています。

手順

1. Java 開発用のターゲット・プラットフォームをセットアップします。 の関連するステップを参照してください。

   ターゲット・プラットフォームを使用すると、アプリケーション開発において必ず CICS のターゲット・リリースに適した Java クラスのみが使用されるようにすることができます。

2. Java アプリケーション開発用の OSGi バンドル・プロジェクトまたはプラグイン・プロジェクトを作成します。
   a. プロジェクトのデフォルト・バージョンは 1.0.0.qualifier です。「バージョン」フィールドで、「.qualifier」を使用する必要がない場合はバージョン番号の最後から .qualifier を削除します。または、日時タイムスタンプなどの意味のあるものに設定します。

ベスト・プラクティスを使用して Java アプリケーションを開発します。例えば、OSGi バンドルどうしの従属関係を構成するには、Require-Bundle ではなく Import-Package/Export-Package を設定します。

3. CICS 用の Java アプリケーションの開発に慣れていない場合は、IBM CICS SDK for Java で提供されるサンプルを使用して開発に取り掛かることができます。 OSGi Java アプリケーションで JCICS を使用するには、com.ibm.cics.server パッケージをインポートする必要があります。
4. オプション: Liberty では、アプリケーション・プレゼンテーション層を開発するために、動的 Web アプリケーション (WAR) または Web 使用可能 OSGi バンドル・プロジェクト (WAB) を作成します。 動的 Web プロジェクト内でサーブレットや JSP ページを作成できます。また、WAR ファイルの場合、Liberty API バンドルにアクセスできるように、Liberty ライブラリーをビルド・パスに追加する必要もあります。詳細については、 79 ページの『開発環境のセットアップ』を参照してください。
5. 以下のようにして、配置用にアプリケーションをパッケージ化します。
   a. Web 使用可能 OSGi バンドル・プロジェクト (WAB) をデプロイする場合は、OSGi アプリケーション・プロジェクト (EBA) を作成します。
   b. EBA、EAR ファイルまたは Web アプリケーション (WAR ファイル) を参照するための、1 つ以上の CICS バンドル・プロジェクトを作成します。CICS バンドルは、CICS におけるアプリケーションのデプロイメント単位です。一緒に更新および管理される複数の Web アプリケーションを 1 つの CICS バンドル・プロジェクトに入れてください。アプリケーションの配置先の JVMSERVER リソースの名前を知っている必要があります。

      CICS リソースも、PROGRAM、URIMAP、および TRANSACTION リソースなどの CICS バンドル・プロジェクトに追加できます。これらのリソースは、Java アプリケーションで動的にインストールされて管理されます。
   c. オプション: アプリケーションを CICS プラットフォームにデプロイする場合は、CICS バンドルを参照するアプリケーション・プロジェクトを作成します。 アプリケーションにより、CICSplex 全体にわたってアプリケーションをデプロイおよび管理するための単一管理ポイントが CICS に提供されます。詳細については、仕組み: アプリケーションを参照してください。
   d. ツールがエラーを示していない場合でも、OSGi バンドルが使用するパッケージを常に明示的に宣言する必要があります。これを行うには、OSGi バンドル・マニフェストで **Import-Package** バンドル・ヘッダーを追加または更新します。Eclipse などのツールは、明示的な Import が必要なランタイム環境にとって適切ではない可能性のある **javax.*** パッケージの使用可能性について推測します。
6. アプリケーション・プロジェクトまたは CICS バンドル・プロジェクトをエクスポートすることにより、Java アプリケーションを zFS にデプロイします。あるいは、配置用にソース・リポジトリーにプロジェクトを保管することもできます。

タスクの結果

これで、IBM CICS SDK for Java を使用したアプリケーションの開発とエクスポートが正常に完了しました。

次のタスク

アプリケーションを JVM サーバーにインストールします。 CICS 内にリソースを作成する権限がない場合は、システム・プログラマーまたは管理者にアプリケーションの作成を依頼します。システム・プログラマーまたは管理者には、エクスポートしたバンドルがある場所とターゲット JVM サーバーの名前を伝える必要があります。

CICS バンドル・プロジェクトへのプロジェクトの追加

CICS バンドル・プロジェクトを作成すると、マニフェスト・ファイルが `META-INF` ディレクトリーに作成されます。マニフェスト・ファイルを編集して、各種タイプのプロジェクト (動的 Web プロジェクト、エンタープライズ・アプリケーション・プロジェクト、OSGi アプリケーション・プロジェクト、または OSGi バンドル・プロジェクト) を 1 つ以上組み込むことができます。組み込まれたプロジェクトは、ソースであっても、事前ビルドされていても構いません。CICS バンドル・プロジェクトをエクスポートすると、組み込まれているすべてのプロジェクトが、zFS 上のCICS バンドルに含まれます。

始める前に

このタスクでは、CICS バンドルにプロジェクトの詳細を追加する方法について説明します。 CICS バンドル・プロジェクトを作成していない場合、CICS Explorer 製品資料内の『CICS バンドル・プロジェクトの作成』を参照してください。

このタスクについて

プロジェクトの詳細を CICS バンドルに追加できます。これを行うには、各種ウィザード (「動的 **Web** プロジェクトの組み込み」、「エンタープライズ・アプリケーション・プロジェクトの組み込み」、「**OSGi** アプリケーション・プロジェクトの組み込み」、または「**OSGi** バンドル・プロジェクトの組み込み」) のいずれかを使用します。ウィザードは、追加されるプロジェクトの詳細を含めるようにバンドル・マニフェスト・ファイルを更新し、プロジェクトを指す `.warbundle`、`.earbundle`、`.ebabundle`、または `.osgibundle` というファイル拡張子の付いたリソース・ファイルを作成します。

注: OSGi アプリケーションに含まれていない OSGi バンドルを CICS バンドル・プロジェクトに追加するには、出力フォルダーのロケーションを含む `build.properties` ファイルが必要です。`build.properties` ファイルの内容の例を以下に示します。

```
source.. = src/
output.. = bin/
bin.includes = META-INF/
```

手順

1. 「パッケージ・エクスプローラー」ビューで、更新するバンドル・プロジェクトを右クリックし、「新規」 > 「その他」をクリックして新規ウィザードを開きます。
2. CICS リソース・フォルダーを展開し、「動的 **Web** プロジェクトの組み込み」、「エンタープライズ・アプリケーション・プロジェクトの組み込み」、「**OSGi** アプリケーション・プロジェクトの組み込み」、または「**OSGi** バンドル・プロジェクトの組み込み」をクリックします。「次へ」をクリックします。 ウィザードが開き、ワークスペース内の当該タイプのプロジェクトが表示されます。ウィザードには、選択したバンドル・プロジェクト内にあるビルドされたプロジェクト (例えば、JAR、EAR、EBA、および WAR の各ファイル) もすべて表示されます。
3. バンドルに含めるプロジェクトをクリックします。 プロジェクトをクリックすると、ウィザードにはシンボル名と、該当する場合にはバージョンが表示されます。プロジェクト上にカーソルを置くと、それがビルドされたプロジェクトであるか、またはソース・プロジェクトであるかを識別できます。
4. オプション: OSGi プロジェクトの場合、含めるバージョンまたはバージョン範囲を指定します。
   - 「このバージョンを使用する **(Use this version)**」を選択して、「バージョン」フィールドに示されているような方法で、選択した OSGi プロジェクトの特定のバージョンを指定します。
   - 「バージョン範囲を使用する **(Use version range)**」を選択して、選択した OSGi プロジェクトをエクスポートするときに含める、その OSGi プロジェクトの定義済みのバージョン範囲の最上位バージョンを指定します。デフォルトでは、バージョン範囲には、選択した OSGi プロジェクトのバージョンから次の最上位のメジャー・バージョンまでが含まれます。フィールドとボタンを使用して、別の範囲を指定することもできます。
5. 「**JVM** サーバー」フィールドに、アプリケーション・コンポーネントを実行する JVM サーバーの名前を入力します。
6. オプション: プロジェクト名に基づいて、作成するリソース・ファイルの名前が生成され、それがウィザードに表示されます。ファイル名を変更する場合は、「戻る」ボタンを使用できます。
7. 「完了 **(Finish)**」をクリックします。

タスクの結果

プロジェクトのリソース・ファイルがバンドル・プロジェクトに追加され、マニフェスト・ファイルが更新されます。これらのステップを繰り返して、プロジェクトをさらに CICS バンドル・プロジェクトに追加することができます。

次のタスク

アプリケーションの CICS バンドル・プロジェクトにリソースを追加できます。例えば、Java アプリケーションを CICS 内の他のアプリケーションで使用できるようにするプログラムを作成できます。

CICS Explorer 製品資料内の『CICS バンドルのデプロイ』で説明されているように、CICS バンドルを z/OS UNIX ファイル・システムにデプロイできます。CICS

バンドル・プロジェクトを zFS にエクスポートすると、アプリケーションで必要なすべてのファイルと成果物がコンパイルされてエクスポートされます。

または、CICS バンドル・プロジェクトをクラウド・スタイルのアプリケーション・プロジェクトにパッケージし、CICS プラットフォームにデプロイできます。アプリケーション・プロジェクトを使用すると、アプリケーションを構成するすべての CICS バンドル・プロジェクトをまとめてグループ化し、それを単一ステップでデプロイおよびインストールすることができます。詳しくは、CICS Explorer 製品資料内の『CICS アプリケーション・バインディング・プロジェクトの作成』を参照してください。

プロジェクト・ビルド・パスの更新

プロジェクト・ビルド・パスの更新方法。

このタスクについて

動的 Web プロジェクトを使用する場合は、デプロイメント用の WAR ファイル・アーカイブが作成されます。この場合、Eclipse の OSGi フレームワークは使用されないため、サード・パーティーの JAR ファイルをプロジェクト・ビルド・パスに追加する必要があります。この例では、IBM MQ JAR ファイルを使用します。

手順

1. Eclipse で Web プロジェクトを選択し、「ビルド・パス」 > 「ビルド・パスの構成」を右クリックします。これにより、「Java のビルド・パス」ウィンドウが表示されます。
2. CICS および Liberty ライブラリーを追加し、「ライブラリーの追加」 > 「**Liberty JVM** サーバー」 > 「次へ」 > 「終了」をクリックします。
3. 「外部 **JAR** の追加」をクリックし、前にダウンロードした IBM MQ JAR ファイルが配置されているディレクトリーにナビゲートします。アプリケーションで使用するインポートに応じて、以下の JAR ファイルを選択します。
   - com.ibm.mq.jar
   - com.ibm.mq.jmql.jar
   - com.ibm.mq.headers.jar

   注**:** ステップ 3 は、IBM MQ のみを使用する場合のオプションです。

タスクの結果

プロジェクトのビルド・パスに、CICS と IBM MQ の両方の API を使用する Web アプリケーションを開発するための適切なインターフェースが含まれるようになりました。

その他のツールを使用したアプリケーションの開発

IBM CICS SDK for Java は、CICS Java アプリケーションの 1 次開発環境ですが、この SDK の使用は必須ではありません。CICS Java プログラムを作成、パッケージ化、およびデプロイするための独自のスクリプトとツールを作成できます。

CICS には、複数の .jar ファイルが付属して配布されており、Java アプリケーションはこれらのファイルに対してコンパイルする必要があります。 CICS ライセンスを使用すると、アプリケーション開発の目的で、これらの .jar ファイルを CICS インストール・ディレクトリーからコピーできます。これらのファイルをまとめたものが JCICS ライブラリーであり、CICS インストール zFS ライブラリーの /lib ディレクトリー内に配置されます。

アプリケーション開発の .jar ファイルは、以下のとおりです

- com.ibm.cics.server.jar
- com.ibm.cics.server.pipeline.jar
- com.ibm.cics.server.invocation.jar
- com.ibm.cics.server.invocation.binding.jar
- com.ibm.cics.server.invocation.annotations.jar
- com.ibm.record.jar

場合によっては、新しい Java API およびメソッドが、PTF によって CICS に追加されることがあります。更新によって導入された API を使用する場合は、コピーされた .jar ファイルをリフレッシュするメカニズムを組み込む必要があります。新しいリリースの CICS をインストールする場合は、フルリフレッシュが必要です。

要確認**:** Java コンパイル・セットアップだけでなく独自のスクリプトを使用する場合は、OSGi のパッケージ化、Java EE コンポーネント、および CICS バンドル・リソースを考慮する必要があります。IBM CICS SDK for Java は、これらの問題を単純化するように設計されています。

共用 JVM に関する考慮事項

CICS 内で実行する Java アプリケーションを開発する際は、JVM 内の共用リソースを変更すると、その変更がすべての実行中アプリケーションおよびスレッドによって検出される場合があることに注意してください。開発するアプリケーションにおいて、他のアプリケーションが依存する JVM を予期しない状態のままにしないようにしてください。

考慮すべき重要な点は次のとおりです。

- アプリケーションがデフォルトのタイム・ゾーンをリセットする場合、同じ JVM サーバーを使用する他のアプリケーションは新しいデフォルトのタイム・ゾーンを使用することになりますが、この変更が他のアプリケーションにとって予期されないものである可能性があります。
- 開発するアプリケーションでは **System.exit()** を使用しないでください。**System.exit()** を使用すると、JVM サーバーと CICS の両方がシャットダウンします。
- アプリケーションがスレッド・セーフになるようにしてください。アプリケーションの間で共用する静的変数を慎重に調べて、アプリケーションが相互に悪影響を与えないことを確認する必要があります。固有性を確実にする標準的なパターンは、スレッド・ローカル変数を使用することです。

- オブジェクトが静的変数で参照される場合、それらのオブジェクトはガーベッジ・コレクションの候補にはなりません。JVM サーバーでは、システム・プログラマーが JVM サーバーを使用不可にするまで、すべてのアプリケーションの静的状態が持続します。
- Db2 には異なる複数のアプリケーションが接続する可能性があります。したがって、Db2 での処理が完了したら、接続を閉じることがベスト・プラクティスです。これは、後でタスクが完了した時点で接続が削除される場合にも適用されます。
- `java.net` パッケージに含まれるクラスを使用して作成されたソケットは CICS ドメインのソケットではないため、CICS で管理またはモニターすることはできません。

JCICS を使用した Java の開発

CICS Java クラス・ライブラリー (JCICS) を使用して CICS サービスにアクセスする Java アプリケーションを作成できます。JCICS は、Java において、CICS でサポートされている他の言語 (COBOL など) に提供される **EXEC CICS** アプリケーション・プログラミング・インターフェースに相当するものです。

JCICS を使用すると、CICS リソースにアクセスする Java アプリケーションを作成し、他の言語で作成されたプログラムと統合することができます。**EXEC CICS** API の大部分の機能がサポートされます。このライブラリーは、`com.ibm.cics.server.jar` ファイルで CICS および IBM CICS SDK for Java に提供されています。

保守作業または開発作業で API の追加、API の削除、バグ修正が発生すると、`com.ibm.cics.server` パッケージ (JCICS) のバージョン番号が増えます。リリースの境界では、バージョンの増分は保証されていません。バージョン、およびこれらのバージョンが適用される CICS リリースについては、パッケージ com.ibm.cics.serverを参照してください。

インポートには、アプリケーションの最小サポート・レベルから (そのレベルを含む) 次の大きな API 変更まで (含まない) の互換可能な範囲を宣言しておくことをお勧めします。例: Import-Package: `com.ibm.cics.server;version="[1.600.0,2.0.0)"`

CICS 用 Java クラス・ライブラリー (JCICS)

JCICS では、**EXEC CICS** API コマンドのほとんどの機能がサポートされます。

JCICS クラスは、クラス定義から生成される Javadoc で詳しく説明されています。Javadoc は、JCICS Javadoc 情報から入手できます。

JavaBeans

JCICS の一部のクラスは JavaBeans として使用できます。つまり、Eclipse などのアプリケーション開発ツールでカスタマイズし、直列化し、JavaBeans API を使用して操作することができます。

以下の JavaBeans が JCICS で使用可能です。

- プログラム

- ESDS
- KSDS
- RRDS
- TDQ
- TSQ
- AttachInitiator
- EnterRequest

これらの Bean はイベントを定義しません。プロパティーとメソッドで構成されます。次の 3 つの方法のいずれかで実行時にインスタンス化することができます。

- クラス自体の new メソッドを呼び出す。この方法が優先されます。
- プロパティー値を手動で設定して、クラスの名前の Beans.instantiate() を呼び出す。
- 設計時に設定されたプロパティー値を使用して、`.ser` ファイルから Beans.instantiate() を呼び出す。

最初の 2 つのオプションのどちらかを選択する場合、プロパティー値 (CICS リソースの名前を含めて) は、実行時に適切な set メソッドを呼び出すことによって設定されなければなりません。

ライブラリー構造

各 JCICS ライブラリー・コンポーネントは、4 つのカテゴリー (インターフェース、クラス、例外、エラー) のいずれかに分類されます。

インターフェース

一部のインターフェースは、1 組の定数を定義するために提供されます。例えば、TerminalSendBits インターフェースは、`java.util.BitSet` の構成に使用できる 1 組の定数を提供します。

クラス

指定されたクラスは、大部分の JCICS 機能を提供します。`API` クラスは抽象クラスであり、ABEND と例外を除いて、CICS API の一部に対応する、すべてのクラスに共通する初期化を提供します。例えば、`Task` クラスは、CICS タスクに対応する 1 組のメソッドと変数を提供します。

エラーと例外

Java 言語は、クラス `Throwable` のサブクラスとして例外とエラーの両方を定義します。JCICS は、`Error` のサブクラスとして `CicsError` を定義します。`CicsError` は、重大エラーに使用される他のすべての CICS エラー・クラスのスーパークラスです。

JCICS は、`Exception` のサブクラスとして `CicsException` を定義します。`CicsException` は、(CICS QIDERR 条件を表す、`InvalidQueueIdException` などの `CicsConditionException` クラスを含む) すべての CICS 例外クラスのスーパークラスです。

詳しくは、 54 ページの『エラー処理と異常終了』を参照してください。

CICS リソース

プログラムや一時記憶域キューなどの CICS リソースは、該当する Java クラスのインスタンスによって表され、リソースの名前などの各種プロパティーの値によって識別されます。

CICS リソースは、CICS Explorer、CEDA トランザクション、または CICSPlex® SM WUI を使用して定義します。暗黙的なリモート・アクセスを使用するには、リモート・リソースを指すリソースをローカルで定義します。

CICS リソースの定義について詳しくは、CICS リソースを参照してください。

データを渡すための引数

チャネルとコンテナーを使用するか、通信域 (COMMAREA) を使用して、プログラム間でデータを受け渡すことができます。

COMMAREA を使用する場合、一度に渡すデータは 32 KB に制限されます。チャネルとコンテナーを使用する場合、プログラム間で 32 KB より多く渡すことができます。COMMAREA またはチャネル、およびその他のすべてのパラメーターは、引数として該当するメソッドに渡されます。

メソッドの多くは多重定義されています。すなわち、バージョンが異なれば、取る引数の数か、引数のタイプのどちらかが異なります。引数がないか、最小限の必須引数があるメソッドや、すべての引数があるメソッドがあります。例えば、`Program` クラスには以下の各種の link() メソッドが含まれています。

`link()`
このメソッドは、COMMAREA を使用してデータを受け渡したり、他のオプションを指定したりせず、単純なリンクを行います。

`link(com.ibm.cics.server.CommAreaHolder)`
このメソッドは、COMMAREA を使用してデータを受け渡すのみで、他のオプションを指定せず、単純なリンクを行います。

`link(com.ibm.cics.server.CommAreaHolder, int)`
このメソッドは、COMMAREA を使用してデータを受け渡し、DATALENGTH 値を使用して COMMAREA 内のデータの長さを指定することにより、分散リンクを行います。

`link(com.ibm.cics.server.Channel)`
このメソッドは、チャネルを使用して 1 つ以上のコンテナー内のデータを受け渡すことによって、リンクを行います。

直列化可能クラス

JCICS 直列化可能クラスのリスト。

- AddressResource
- AttachInitiator
- CommAreaHolder
- EnterRequest
- ESDS
- ファイル

- KeyedFile
- KSDS
- NameResource
- Program
- RemotableResource
- Resource
- RRDS
- StartRequest
- SynchronizationResource
- SyncLevel
- TDQ
- TSQ
- TSQType

Task.out および Task.err

Java 関連の CICS タスクごとに、CICS は、標準出力および標準エラー・ストリームとして使用できる、2 つの Java `PrintWriters` クラスを自動的に作成します。標準出力と標準エラー・ストリームは、`out` と `err` と呼ばれる、`Task` クラス内のパブリック・フィールドです。

CICS タスクが端末 (この場合、端末は基本機構 と呼ばれます) から駆動される場合、CICS は標準出力および標準エラー・ストリームをタスクの端末にマップします。

タスクに基本機構としての端末がない場合、標準出力および標準エラー・ストリームは System.out および System.err に送信されます。

スレッド

JVM サーバー環境では、OSGi フレームワークで実行されているアプリケーションは ExecutorService を使用して、CICS タスクで非同期的に実行するスレッドを作成できます。

CICS には、Java ExecutorService インターフェースの実装が用意されています。この実装では、JCICS API を使用して CICS サービスにアクセスできるスレッドが作成されます。 JVM サーバーは、始動時に CICS ExecutorService を OSGi サービスとして登録します。Java Thread クラスではなくこのサービスを使用して、JCICS を使用できるタスクを作成してください。

CICS で提供される ExecutorService は、スレッドを作成するためにアプリケーションで使用できるように、OSGi フレームワークにおいて優先順位が高いものとして登録されます。アプリケーションは通常、サービスをフィルタリングして特定の実装を使用するのでない限り、最も高い優先順位の ExecutorService を使用します。

アプリケーションにスレッドを作成する場合、OSGi レジストリーから汎用 ExecutorService を使用する方式が推奨されています。アプリケーションが JVM サーバーで実行されているときに、OSGi レジストリーは自動的に CICS

ExecutorService を使用して CICS スレッドを作成します。この方式により、アプリケーションは実装環境とは分離されるので、JCICS API メソッドを使用してスレッドを作成する必要がなくなります。

ただし、CICS に固有のアプリケーションを作成している場合は、JCICS API 内の CICSExecutorService クラスを使用して、新しいスレッドを要求できます。

CICSExecutorService

このクラスは java.util.concurrent.ExecutorService インターフェースを実装します。 CICSExecutorService クラスには runAsCICS() という名前の静的メソッドがあり、このメソッドを使用して、Java オブジェクトの Runnable または Callable を、新しい JCICS 対応スレッドでの実行対象として送信できます。runAsCICS() メソッドは、OSGi レジストリー検索を実行し、アプリケーションの CICSExecutorService のインスタンスを取得するユーティリティー・メソッドです。

親 CICS スレッドから spawn される作業の場合、新しい CICS タスクが作成され、親から継承される user ID と transaction ID のタスクの下で実行されます。非 CICS スレッドから spawn される作業の場合、デフォルトの CJSA transaction ID とデフォルトの CICS user ID が使用されます。自分で選択した transaction ID の下で新しいタスクが実行されるよう保証するには、Runnable または Callable オブジェクトが CICSTransactionRunnable または CICSTransactionRunnable インターフェースを実装している必要があります。

```
CICSExecutorService.runAsCICS(Runnable runnable)
CICSExecutorService.runAsCICS(Callable callable)
```

制約事項

OSGi フレームワーク (Axis2 Java プログラムなど) で実行されていないアプリケーションの場合は、ExecutorService を使用できないため、初期アプリケーション・スレッドのみで JCICS にアクセスできます。さらに、以下のいずれかのアクションを行う前に、初期スレッド以外のすべてのスレッドが終了している必要があります。

- com.ibm.cics.server.Program クラスでの link メソッド
- com.ibm.cics.server.TerminalPrincipalFacility クラスでの setNextTransaction (String) メソッド
- com.ibm.cics.server.TerminalPrincipalFacility クラスでの setNextCOMMAREA (byte[]) メソッド
- com.ibm.cics.server.Task クラスでの commit() メソッド
- com.ibm.cics.server.Task クラスでの rollback() メソッド
- com.ibm.cics.server クラスから `AbendException` 例外を返す。

データ・エンコード

JVM は文字エンコードに CICS とは異なるコード・ページを使用できます。CICS では常に EBCDIC コード・ページを使用しなければなりませんが、JVM では

ASCII などの別のエンコード方式を使用できます。JCICS API を使用するアプリケーションを開発する際には、使用するエンコード方式が正しいものであることを確認する必要があります。

JCICS API は、基礎となっている JVM ではなく CICS 領域で指定されているコード・ページを使用します。そのために、JVM で別のファイル・エンコード方式を使用する場合は、アプリケーションが別のコード・ページを処理することが必要になります。CICS が使用しているコード・ページを判別できるように、CICS には以下の Java プロパティーが用意されています。

- **com.ibm.cics.jvmserver.supplied.ccsid** プロパティーは、CICS 領域に指定されているコード・ページを返します。デフォルトでは、JCICS API は文字エンコードにこのコード・ページを使用します。ただし、この値は JVM サーバー構成でオーバーライドできます。
- **com.ibm.cics.jvmserver.override.ccsid** プロパティーは、JVM プロファイル内のオーバーライド値を返します。JCICS API は文字エンコードに、CICS 領域で使用されているコード・ページではなく、この値で示されたコード・ページを使用します。
- **com.ibm.cics.jvmserver.local.ccsid** プロパティーは、JVM サーバーで JCICS API が文字エンコードに使用しているコード・ページを返します。

Java アプリケーションにこれらのプロパティーを設定して、JCICS のエンコード方式を変更することはできません。コード・ページを変更するには、システム管理者に、JVM プロファイルを更新して JVM システム・プロパティー **-Dcom.ibm.cics.jvmserver.override.ccsid** を追加するよう依頼する必要があります。

エンコードの例

java.lang.String パラメーターを入力として受け入れる JCICS メソッドはいずれも、データが CICS に渡される前に、正しいコード・ページで自動的にエンコードされます。同様に、JCICS API から返されるすべての java.lang.String 値も、正しいコード・ページでエンコードされます。JCICS API は、ほとんどのクラスにヘルパー・メソッドを提供しています。これらのヘルパー・メソッドはアプリケーションのために、ストリングやデータを処理してコード・ページを判別し、設定します。

アプリケーションが String.getBytes() メソッドまたは new String(byte[] bytes) メソッドを使用する場合、そのアプリケーションは必ず正しいエンコード方式を使用する必要があります。これらのメソッドをアプリケーションで使用する場合、以下のように Java プロパティーを使用すれば、データを正しくエンコードすることができます。

```
String.getBytes(System.getProperty("com.ibm.cics.jvmserver.local.ccsid"))
String(bytes, System.getProperty("com.ibm.cics.jvmserver.local.ccsid"))
```

以下の例に、アプリケーションが COMMAREA からフィールドを読み取る場合の JCICS エンコードの使用方法を示します。

```
public static void main(CommAreaHolder ca)
{
    //Convert first 8 bytes of ca into a String using JCICS encoding
    String str=new String(ca.getValue(), 0, 8, System.getProperty("com.ibm.cics.jvmserver.local.ccsid"));
}
```

JCICS API サービスと例

CICS は、Java アプリケーション用のさまざまな API とサービスをサポートしています。 EXEC CICS API を介して非 Java プログラムで使用可能なサービスの多くは、Java SDK で提供される標準の Java SE API とともに、JCICS API を介して Java プログラムで使用可能です。

以下のトピックでは、JCICS サービス、および Java 例外処理との統合について詳しく説明しています。 Liberty JVM サーバーでは、他の JEE API を使用できます。詳しくは、 79 ページの『Liberty JVM サーバーで実行する Java アプリケーションの開発』を参照してください。

Java プログラムでの CICS 例外処理

CICS で発生した問題を処理するために、CICS の ABEND および例外は Java 例外処理体系に統合されています。

通常のすべての CICS ABEND は単一の Java 例外 `AbendException` にマップされる一方、各 CICS 条件は個別の Java 例外にマップされます。これにより、Java の ABEND 処理モデルは他のプログラミング言語と同じようになります。つまり、すべての ABEND について単一のハンドラーに制御が与えられ、そのハンドラーは特定の ABEND を照会してから、処理内容を決定する必要があります。

条件を表す例外は、CICS 自体によってキャッチされると ABEND になります。

Java 例外処理は、他の言語の ABEND および条件処理に完全に統合されるため、ABEND は、言語に依存しない標準的な方法で、Java プログラムと非 Java プログラム間に波及することができます。条件は ABEND にマップされてから、その条件の原因となるか、またはその条件を検出したプログラムを終了します。

ただし、他のプログラミング言語の ABEND 処理モデルには、次のような複数の相違点があります。これらの相違点は、Java 例外処理体系の性質や、Java API の基礎となる一部のテクノロジーの実装に起因します。

- 他のプログラミング言語では処理できない ABEND を、Java プログラムでキャッチできます。これらの ABEND は通常、同期点処理時に発生します。これらの ABEND が Java アプリケーションを中断しないようにするために、チェックなし例外の拡張にマップされます。したがって、これらの ABEND の宣言もキャッチも必要ありません。
- プログラム終了などの複数の内部 CICS イベントも、Java 例外にマップされるので、Java アプリケーションでキャッチできます。また、正常な状態を中断しないように、これらのイベントはチェックなし例外の拡張にマップされ、キャッチも宣言も必要ありません。

例外の以下の 3 つのクラス階層が CICS に関連しています。

1. CicsError は java.lang.Error の拡張で、AbendError および UnknownCicsError の基本クラスです。

2. CicsRuntimeException は java.lang.RuntimeException の拡張で、次のクラスにより拡張されます。

   AbendCancelException
   CICS ABEND CANCEL を表します。

   AbendException
   通常の CICS ABEND を表します。

   EndOfProgramException
   リンク先プログラムが通常どおりに終了したことを示します。

3. CicsException は java.lang.Exception の拡張で、次のサブクラスがあります。

   CicsConditionException
   すべての CICS 条件の基本クラス。

CICS エラー処理コマンド**:**

Java ではどのように **EXEC CICS** エラー処理コマンドがサポートされるかについて説明します。

49 ページの『Java プログラムでの CICS 例外処理』で説明されているように、CICS 条件処理は Java 例外処理体系に統合されています。 Java では同等の **EXEC CICS** コマンドが以下のようにサポートされます。

HANDLE ABEND
CICS でサポートされる任意の言語のプログラムによって生成された ABEND を処理するには、catch 節に AbendException を指定して、Java の try-catch ステートメントを使用します。

HANDLE CONDITION
PGMIDERR などの特定の条件を処理するには、適切な例外の名前を指定した catch 節を使用します。この場合は、InvalidProgramException です。あるいは、すべての CICS 条件をキャッチする場合は、CicsConditionException を指定した catch 節を使用します。

IGNORE CONDITION
このコマンドは、Java アプリケーションでは適切ではありません。

POP HANDLE および **PUSH HANDLE**
これらのコマンドは、Java アプリケーションでは適切ではありません。CICS ABEND および条件を表すために使用される Java 例外は、スコープ内の catch ブロックによってキャッチされます。

CICS 条件**:**

Java での条件処理モデルは、他の CICS プログラミング言語でのモデルとは異なります。

COBOL では、条件ごとに条件処理ラベルを定義できます。その条件が CICS コマンドの処理中に発生する場合、制御はラベルに転送されます。

C および C++ では、条件に例外処理ラベルを定義することはできません。条件を検出するために、各 CICS コマンドの後に、EIB 内の RESP フィールドが検査されなければなりません。

Java では、CICS コマンドから返された条件はいずれも Java 例外にマップされます。すべての CICS コマンドを try-catch ブロックに組み込んで、条件ごとに特定の処理を行うか、特定の例外が適切でない場合は、単一の null catch 節を持つことができます。または、条件を波及させて、より大きいスコープで catch 節によって処理できるようにすることができます。

Java Web アプリケーションでの CICS 例外処理

Liberty Web コンテナーが CICS アプリケーションで発生する問題に対処するために、CICS ABEND および例外が Java 例外処理体系に組み込まれています。 Web アプリケーションで処理されない Java 例外は Web コンテナーによってキャッチされ、サーブレット例外処理プロセスを起動します。この処理の一部として、コミットされていない CICS 作業単位は CICS によってロールバックされます。

HttpServlet インターフェースを拡張するすべての Java Web アプリケーションは、HttpServlet インターフェースに定義されているように、`IOException` や `ServletException` 以外のすべてのチェック例外を処理する必要があります。チェック例外は、未処理の CICS 条件を表す `com.ibm.cics.server.CicsConditionException` のサブクラスのすべてに含まれます。したがって、CICS 条件をキャッチするすべての例外処理コードは、作業単位をロールバックする必要があり、例に示されているように、Task オブジェクトで rollback() メソッドを使用して明示的に同期点ロールバックを呼び出すか、`AbendException` をスローするエラー条件をすべて識別する必要があります。

```
try
{
    TSQ tsqQ = new TSQ();
    tsqQ.setName("tsq1");
    tsqQ.writeString("input data");

} catch (IOErrorException e) {
    // Log error
    try
    {
        Task.getTask().rollback();

        } catch (InvalidRequestException e1) {
            throw new RuntimeException(e1);
        }
    }
}
```

Java のチェックなし例外は、java.lang.RuntimeException のサブクラスであり、Web アプリケーションを含む任意の Java アプリケーションでスローすることができます。これらのチェックなし例外には、`com.ibm.cics.server.AbendException` および `com.ibm.cics.server.AbendCancelException` が含まれます、したがって、`AbendException` をスローするか、トランザクションの異常終了を処理しない Web アプリケーションは、サーブレット例外処理プロセスおよび関連する作業単位のロールバック処理を起動します。

Java Transaction API (JTA) を使用して作業単位をコミットする Web アプリケーションは、Liberty トランザクション・マネージャーの制御に従ってコミットされます。詳細については、 94 ページの『Java Transaction API (JTA)』を参照してください。

JCICS 例外マッピング

Java では、CICS コマンドによって戻される条件は、Java 例外にマップされます。

表 2. *Java* 例外マッピング

CICS 条件	**Java** 例外
ALLOCERR	AllocationErrorException
CBIDERR	InvalidControlBlockIdException
CCSIDERR	CCSIDErrorException
CHANNELERR	ChannelErrorException
CONTAINERERR	ContainerErrorException
DISABLED	FileDisabledException ResourceDisabledException
DSIDERR	FileNotFoundException
DSSTAT	DestinationStatusChangeException
DUPKEY	DuplicateKeyException
DUPREC	DuplicateRecordException
END	EndException
ENDDATA	EndOfDataException
ENDFILE	EndOfFileException
ENDINPT	EndOfInputIndicatorException
ENQBUSY	ResourceUnavailableException
ENVDEFERR	InvalidRetrieveOptionException
EOC	EndOfChainIndicatorException
EODS	EndOfDataSetIndicatorException
EOF	EndOfFileIndicatorException
ERROR	ErrorException
EXPIRED	TimeExpiredException
FILENOTFOUND	FileNotFoundException
FUNCERR	FunctionErrorException
IGREQID	InvalidREQIDPrefixException
IGREQCD	InvalidDirectionException
ILLOGIC	LogicException
INBFMH	InboundFMHException
INVERRTERM	InvalidErrorTerminalException
INVEXITREQ	InvalidExitRequestException
INVLDC	InvalidLDCException
INVMPSZ	InvalidMapSizeException
INVPARTNSET	InvalidPartitionSetException
INVPARTN	InvalidPartitionException
INVREQ	InvalidRequestException
INVTSREQ	InvalidTSRequestException
IOERR	IOErrorException

表 2. *Java* 例外マッピング (続き)

CICS 条件	**Java** 例外
ISCINVREQ	ISCInvalidRequestException
ITEMERR	ItemErrorException
JIDERR	InvalidJournalIdException
LENGERR	LengthErrorException
MAPERROR	MapErrorException
MAPFAIL	MapFailureException
NAMEERROR	NameErrorException
NODEIDERR	InvalidNodeIdException
NOJBUFSP	NoJournalBufferSpaceException
NONVAL	NotValidException
NOPASSBKRD	NoPassbookReadException
NOPASSBKWR	NoPassbookWriteException
NOSPACE	NoSpaceException
NOSPOOL	NoSpoolException
NOSTART	StartFailedException
NOSTG	NoStorageException
NOTALLOC	NotAllocatedException
NOTAUTH	NotAuthorisedException
NOTFINISHED	NotFinishedException
NOTFND	RecordNotFoundException NotFoundException
NOTOPEN	NotOpenException
OPENERR	DumpOpenErrorException
OVERFLOW	MapPageOverflowException
PARTNFAIL	PartitionFailureException
PGMIDERR	InvalidProgramIdException
QBUSY	QueueBusyException
QIDERR	InvalidQueueIdException
QZERO	QueueZeroException
RDATT	ReadAttentionException
RETPAGE	ReturnedPageException
ROLLEDBACK	RolledBackException
RTEFAIL	RouteFailedException
RTESOME	RoutePartiallyFailedException
SELNERR	DestinationSelectionErrorException
SESSBUSY	SessionBusyException
SESSIONERR	SessionErrorException
SIGNAL	InboundSignalException
SPOLBUSY	SpoolBusyException
SPOLERR	SpoolErrorException

表 2. *Java* 例外マッピング (続き)

CICS 条件	**Java** 例外
STRELERR	STRELERRException
SUPPRESSED	SuppressedException
SYMBOLERR	SymbolErrorException
SYSBUSY	SystemBusyException
SYSIDERR	InvalidSystemIdException
TASKIDERR	InvalidTaskIdException
TCIDERR	TCIDERRException
TEMPLATERR	TemplateErrorException
TERMERR	TerminalException
TERMIDERR	InvalidTerminalIdException
TOKENERR	TokenErrorException
TRANSIDERR	InvalidTransactionIdException
TSIOERR	TSIOErrorException
UNEXPIN	UnexpectedInformationException
USERIDERR	InvalidUserIdException
WRBRK	WriteBreakException
WRONGSTAT	WrongStatusException

注: CICS コマンド WEB RECEIVE が、受信されたデータが非 HTTP メッセージ (TYPE=HTTPNO の設定による) であることを示す場合、`NonHttpDataException` が `getContent()` によってスローされます。

エラー処理と異常終了

Java プログラムから ABEND を開始するには、Task.abend() メソッドまたは Task.forceAbend() メソッドのいずれかを呼び出す必要があります。

メソッド	**JCICS** クラス	**EXEC CICS** コマンド
abend()、forceAbend()	タスク	ABEND

ABEND

Java プログラムから ABEND を開始するには、Task.abend() メソッドのいずれかを呼び出します。これにより、アベンド条件が CICS で設定され、AbendException がスローされます。AbendException が上位レベルのアプリケーション・オブジェクト内でキャッチされないか、呼び出し側プログラムに登録されている ABEND ハンドラー (ある場合) によって処理される場合、CICS はトランザクションを終了し、ロールバックします。

各種 abend() メソッドは次のとおりです。

- abend (String *abcode*)。ABEND に ABEND コード *abcode* が設定されます。
- abend (String *abcode*、boolean *dump*)。ABEND に ABEND コード *abcode* が設定されます。**dump** パラメーターが false である場合、ダンプは取られません。

- abend()。ABEND コードもダンプもない ABEND が生じます。

ABEND CANCEL

処理できない ABEND を開始するには、Task.forceAbend() メソッドのいずれかを呼び出します。上記のように、これにより、AbendCancelException がスローされ、Java プログラムでキャッチされます。これを行う場合は、**ABEND_CANCEL** 処理を完了するために例外を再スローする必要があります。その結果、制御が CICS に戻ると、CICS はトランザクションを終了し、ロールバックします。通知目的の AbendCancelException のキャッチのみを行ってから、再スローしてください。

各種 forceAbend() メソッドは次のとおりです。

- forceAbend (String *abcode*)。**ABEND CANCEL** に ABEND コード *abcode* が設定されます。
- forceAbend (String *abcode*、boolean *dump*)。**ABEND CANCEL** に ABEND コード *abcode* が設定されます。**dump** パラメーターが false である場合、ダンプは取られません。
- forceAbend()。ABEND コードもダンプもない **ABEND CANCEL** が生じます。

APPC マップ式会話

APPC 非マップ式会話は、JCICS API からサポートされません。

APPC マップ式会話:

メソッド	**JCICS** クラス	**EXEC CICS** コマンド
initiate()	AttachInitiator	ALLOCATE、CONNECT PROCESS
converse()	Conversation	CONVERSE
get*() メソッド	Conversation	EXTRACT ATTRIBUTES
get*() メソッド	Conversation	EXTRACT PROCESS
free()	Conversation	FREE
issueAbend()	Conversation	ISSUE ABEND
issueConfirmation()	Conversation	ISSUE CONFIRMATION
issueError()	Conversation	ISSUE ERROR
issuePrepare()	Conversation	ISSUE PREPARE
issueSignal()	Conversation	ISSUE SIGNAL
receive()	Conversation	RECEIVE
send()	Conversation	SEND
flush()	Conversation	WAIT CONVID

基本マッピング・サポート (BMS)

基本マッピング・サポート (BMS) は、CICS プログラムと端末装置間のアプリケーション・プログラミング・インターフェースです。JCICS は、BMS アプリケーション・プログラミング・インターフェースの一部をサポートします。

メソッド	**JCICS** クラス	**EXEC CICS** コマンド
sendControl()	TerminalPrincipalFacility	SEND CONTROL

メソッド	**JCICS** クラス	**EXEC CICS** コマンド
sendText()	TerminalPrincipalFacility	SEND TEXT
	サポート対象外	SEND MAP、RECEIVE MAP

チャネルとコンテナーの例

コンテナー とは、プログラム間で情報を受け渡すために設計されたデータのブロックのことです。コンテナーは、チャネル と呼ばれる集合にグループ化されます。この資料では、Java アプリケーションでチャネルとコンテナーを使用する方法について説明し、いくつかのサンプル・コードを示します。

チャネルとコンテナーの概要、および非 Java アプリケーションでのチャネル使用の手引きについては、チャネルによるプログラム間データ転送を参照してください。Java プログラムが既存の CICS アプリケーション・データにアクセスできるようにするツールについては、 149 ページの『Java からの構造化データとの対話』を参照してください。

表 3 では、チャネルとコンテナーに対する JCICS サポートを実装するクラスとメソッドをリストしています。

表 3. チャネルとコンテナーに対する *JCICS* サポート

メソッド	**JCICS** クラス	**EXEC CICS** コマンド
containerIterator()	チャネル	STARTBROWSE CONTAINER
createContainer()	チャネル	
delete()	チャネル	DELETE CHANNEL
deleteContainer()	チャネル	DELETE CONTAINER CHANNEL
getContainer()	チャネル	
getContainerCount()	チャネル	QUERY CHANNEL
getName()	チャネル	
delete()	Container (コンテナー)	DELETE CONTAINER CHANNEL
get()	Container (コンテナー)	GET CONTAINER CHANNEL
getLength()	Container (コンテナー)	GET CONTAINER CHANNEL NODATA
getDatatype()	Container (コンテナー)	
getName()	Container (コンテナー)	
put()	Container (コンテナー)	PUT CONTAINER CHANNEL
getOwner()	ContainerIterator	
hasNext()	ContainerIterator	
next()	ContainerIterator	GETNEXT CONTAINER BROWSETOKEN
remove()	ContainerIterator	
link()	プログラム	LINK
setNextChannel()	TerminalPrincipalFacility	RETURN CHANNEL
issue()	StartRequest	START CHANNEL

表 3. チャネルとコンテナーに対する *JCICS* サポート (続き)

メソッド	**JCICS** クラス	**EXEC CICS** コマンド
createChannel()	タスク	
getCurrentChannel()	タスク	ASSIGN CHANNEL
containerIterator()	タスク	STARTBROWSE CONTAINER

CICS 条件 CHANNELERR の場合、結果として ChannelErrorException がスローされます。 CONTAINERERR CICS 条件の結果は ContainerErrorException になります。 CCSIDERR CICS 条件の結果は CCSIDErrorException になります。

JCICS におけるチャネルとコンテナーの作成**:**

チャネルを作成するには、Task クラスの createChannel() メソッドを使用します。

以下に例を示します。

```
Task t=Task.getTask();
Channel custData = t.createChannel("Customer_Data");
```

createChannel メソッドに提供されるストリングは、Channel オブジェクトが CICS に認識されている名前です (この名前は、CICS 命名規則に準拠するために、16 文字までスペースで埋め込まれます)。

チャネルに新しいコンテナーを作成するには、Channel createContainer() メソッドを使用します。以下に例を示します。

```
Container custRec = custData.createContainer("Customer_Record");
```

createContainer() メソッドに提供されるストリングは、Container オブジェクトが CICS に認識されている名前です この名前は、CICS 命名規則に準拠するために、必要に応じて 16 文字までスペースで埋め込まれます。同じ名前のコンテナーがこのチャネルに既に存在する場合、ContainerErrorException がスローされます。

コンテナーへのデータの書き込み**:**

Container オブジェクトにデータを書き込むには、Container.put() メソッドを使用します。

データは、ストリングとしてコンテナーに追加できます。例:

```
          String custNo = "00054321";
string[] custRecIn = custNo.putString();
custRec.put(custRecIn);
```

または

```
custRec.putString("00054321");
```

別のプログラムまたはタスクへのチャネルの受け渡し**:**

プログラム・リンクでチャネルを渡すには、Program クラスの link() メソッドを使用します。

```
programX.link(custData);
```

プログラム戻り呼び出しで次のチャネルを設定するには、TerminalPrincipalFacility クラスの setNextChannel() メソッドを使用します。

```
terminalPF.setNextChannel(custData);
```

START 要求でチャネルを渡すには、StartRequest クラスの issue メソッドを使用します。

```
startrequest.issue(custData);
```

現行チャネルの受け取り:

プログラムが現行チャネルを明示的に受け取る必要はありません。ただし、プログラムは現行チャネルを現行タスクから取得することができます。

プログラムが現行タスクから現行チャネルを取得する場合、タスクはコンテナーを名前で取り出すことができます。

```
Task t = Task.getTask();
Channel custData = t.getCurrentChannel();

if (custData != null) {
    Container custRec = custData.getContainer("Customer_Record");
} else {
    System.out.println("There is no Current Channel");
}
```

コンテナーからのデータの取得:

コンテナー内のデータをバイト配列に読み取るには、Container.get() メソッドを使用します。

```
byte[] custInfo = custRec.get();
```

現行チャネルのブラウズ:

チャネルが渡される JCICS プログラムは、そのチャネルを明示的に受け取ることなく、すべての Container オブジェクトにアクセスできます。

これを行うには、ContainerIterator オブジェクトを使用します ContainerIterator クラスは java.util.Iterator インターフェースを実装します。Task オブジェクトが現行タスクからインスタンス化される場合、その containerIterator() メソッドは、現行チャネルの Iterator を戻し、現行チャネルがない場合は null を戻します。以下に例を示します。

```
Task t = Task.getTask();
ContainerIterator ci = t.containerIterator();

while (ci.hasNext()) {
    Container custData = ci.next();
    // Process the container...
}
```

チャネルとコンテナーの例:

この例は、PAYR という名前の COBOL サーバー・プログラムを呼び出す、Payroll と呼ばれる Java クラスの抜粋を示しています。Payroll クラスはチャネルとそのコンテナーを処理するために、JCICS の com.ibm.cics.server.Channel クラスと com.ibm.cics.server.Container クラスを使用します。

```
import com.ibm.cics.server.*;

public class Payroll
{
    ...
    Task t=Task.getTask();

    // create the payroll_2004 channel
    Channel payroll_2004 = t.createChannel("payroll-2004");

    // create the employee container
    Container employee = payroll_2004.createContainer("employee");

    // put the employee name into the container
    employee.putString("John Doe");

    // create the wage container
    Container wage = payroll_2004.createContainer("wage");

    // put the wage into the container
    wage.putString("2000");

    // Link to the PAYROLL program, passing the payroll_2004 channel
    Program p = new Program();
    p.setName("PAYR");
    p.link(payroll_2004);

    // Get the status container which has been returned
    Container status = payroll_2004.getContainer("status");

    if (status != null)
    {
        // Get the status information
        byte[] payrollStatus = status.get();
    }

    ...
}
```

図 *1. JCICS* の *com.ibm.cics.server.Channel* および *com.ibm.cics.server.Container* クラスを使用して、*COBOL* サーバー・プログラムにチャネルを渡す *Java* クラス

診断サービス

JCICS アプリケーション・プログラミング・インターフェースは、以下の CICS トレースおよびダンプ・コマンドをサポートします。

メソッド	**JCICS** クラス	**EXEC CICS** コマンド
	サポート対象外	DUMP
enterTrace()	EnterRequest	ENTER

文書サービス

このセクションでは、DOCUMENT アプリケーション・プログラミング・インターフェースにおけるコマンドの JCICS サポートについて説明します。

`Document` クラスは **EXEC CICS DOCUMENT** API にマップします。

`Document` クラスのデフォルトの引数なしのコンストラクターが、CICS 内に新しい文書を作成します。コンストラクター `Document(byte[] docToken)` は、以前に作成された既存の文書の文書トークンを受け入れます。例えば、別のプログラムが文書を作成して、その文書トークンを COMMAREA またはコンテナー内の Java アプリケーションに渡すことができます。

`DocumentLocation` クラスのコンストラクターは、EXEC CICS DOCUMENT API の AT および TO キーワードにマップされます。

SymbolList クラスの setter と getter は、**`EXEC CICS DOCUMENT`** API の SYMBOLLIST、LENGTH、DELIMITER、および UNESCAPE キーワードにマップされます。

メソッド	**JCICS** クラス	**`EXEC CICS`** コマンド
create*()	Document	DOCUMENT CREATE
append*()	Document	DOCUMENT INSERT
insert*()	Document	DOCUMENT INSERT
addSymbol()	Document	DOCUMENT SET
setSymbolList()	Document	DOCUMENT SET
retrieve*()	Document	DOCUMENT RETRIEVE
get*()	Document	DOCUMENT

環境サービス

CICS 環境サービスは、アプリケーション・プログラムに関連する CICS データ域、パラメーター、およびリソース属性にアクセスできるようにします。

JCICS サポートに相当する **`EXEC CICS`** コマンドとオプションは次のとおりです。

- ADDRESS
- ASSIGN
- INQUIRE SYSTEM
- INQUIRE TASK
- INQUIRE TERMINAL/NETNAME

ADDRESS:

`ADDRESS` API コマンド・オプションには次のサポートが提供されます。

`EXEC CICS ADDRESS` コマンドの詳細については、ADDRESSを参照してください。

ACEE

アクセス制御環境エレメント (ACEE) は、CICS ユーザーがサインオンするときに外部セキュリティー・マネージャーによって作成されます。このオプションは JCICS ではサポートされません。

COMMAREA

COMMAREA には、コマンドで渡されるユーザー・データが入っています。COMMAREA ポインターは、**`CommAreaHolder`** 引数によって、リンクされたプログラムに自動的に渡されます。詳しくは、 45 ページの『データを渡すための引数』を参照してください。

CWA 共通作業域 (CWA) には、タスク間で共用可能なグローバル・ユーザー・データが入っています。CWA のコピーは、Region クラスの getCWA() メソッドを使用して取得できます。

EIB EIB フィールド には、最後に実行された CICS コマンドに関する情報が入っています。EIB 値へのアクセスは、該当するオブジェクトのメソッドによって提供されます。例えば、次のとおりです。

eibtrnid
Task クラスの getTransactionName() メソッドによって戻されます。

eibaid
TerminalPrincipalFacility クラスの getAIDbyte() メソッドによって戻されます。

eibcposn
Cursor クラスの getRow() および getColumn() メソッドによって戻されます。

TCTUA
端末管理テーブル・ユーザー域 (TCTUA) には、CICS トランザクションを起動する端末 (基本機構) に関連したユーザー・データが入っています。この領域は、アプリケーション・プログラム間で情報を渡すのに使用されますが、関係するアプリケーション・プログラムに同じ端末が関連付けられている場合のみです。TCTUA の内容は、TerminalPrincipalFacility クラスの getTCTUA() メソッドを使用して取得できます。

TWA トランザクション作業域 (TWA) には、CICS タスクに関連したユーザー・データが入っています。この領域は、アプリケーション・プログラム間で情報を渡すのに使用されますが、同じタスク内にある場合のみです。TWA のコピーは、Task クラスの getTWA() メソッドを使用して取得できます。

ASSIGN:

ASSIGN API コマンド・オプションには次のサポートが提供されます。

このコマンドについて詳しくは、ASSIGNを参照してください。

メソッド	**JCICS** クラス
getABCODE()	AbendException
getApplicationContext()	タスク
getAPPLID()	領域
getCurrentChannel()	タスク
getCWA()	領域
getName()	TerminalPrincipalFacility または ConversationPrincipalFacility
getFCI()	タスク
getNetName()	TerminalPrincipalFacility または ConversationPrincipalFacility
getPrinSysid()	TerminalPrincipalFacility または ConversationPrincipalFacility

メソッド	**JCICS** クラス
getProgramName()	タスク
getQNAME()	タスク
getSTARTCODE()	タスク
getSysid()	領域
getTCTUA()	TerminalPrincipalFacility
getTERMCODE()	TerminalPrincipalFacility
getTWA()	タスク
getUSERID() 、Task.getUSERID()	Task、TerminalPrincipalFacility または ConversationPrincipalFacility

その他の ASSIGN オプションはサポートされません。

INQUIRE SYSTEM:

INQUIRE SYSTEM SPI オプションに対するサポートが提供されます。

メソッド	**JCICS** クラス
getAPPLID()	領域
getSYSID()	領域

その他の **INQUIRE SYSTEM** オプションはサポートされません。

INQUIRE TASK:

INQUIRE TASK API コマンド・オプションには次のサポートが提供されます。

メソッド	**JCICS** クラス
getSTARTCODE()	タスク
getTransactionName()	タスク
getUSERID()	タスク

FACILITY
タスクの基本機構で getName() メソッドを呼び出すことによって、タスクの基本機構の名前を見つけることができます。基本機構は、現行の Task オブジェクトで getPrincipalFacility() メソッドを呼び出すことによって見つけることができます。

FACILITYTYPE
Java instanceof 演算子を使用して、戻されたオブジェクト参照のクラスを確認することによって、機構のタイプを判別できます。

その他の INQUIRE TASK オプションはサポートされません。

INQUIRE TERMINAL および **INQUIRE NETNAME:**

INQUIRE TERMINAL および **INQUIRE NETNAME** SPI オプションには次のサポートが提供されます。

メソッド	**JCICS** クラス
getUSERID()	Terminal、ConversationalPrincipalFacility
Terminal.getUser()	Terminal、ConversationalPrincipalFacility

現行の Task オブジェクトで、またはタスクの基本機構を表すオブジェクトで getUSERID() メソッドを呼び出すことでも、USERID 値を見つけることができます。

他の **INQUIRE TERMINAL** または **INQUIRE NETNAME** オプションはサポートされません。

ファイル・サービス

JCICS は、CICS ファイルおよび索引のタイプごとに **EXEC CICS** API コマンドにマップされるクラスおよびメソッドを提供します。

Java プログラムが既存の CICS アプリケーション・データにアクセスできるようにするツールについては、Java からの構造化データとの対話を参照してください。

CICS は、次のタイプのファイルをサポートします。

- キー順データ・セット (KSDS)
- 入力順データ・セット (ESDS)
- 相対レコード・データ・セット (RRDS)

KSDS および ESDS ファイルには、代替 (または 2 次) 索引を備えることができます。CICS は、2 次索引から RRDS ファイルへのアクセスをサポートしません。2 次索引は、それ自体が別々の KSDS ファイルである、つまり別々の FD 項目を持つ場合と同じように CICS によって扱われます。

KSDS、ESDS (1 次索引)、および ESDS (2 次索引) ファイルのアクセスにはいくつかの相違点があります。すなわち、常に共通インターフェースを使用できるとは限りません。

どのタイプのファイルでもレコードの読み取り、更新、削除、およびブラウズを行うことができますが、例外的に ESDS ファイルからはレコードを削除することはできません。

データ・セットについて詳しくは、VSAM data sets: KSDS, ESDS, RRDS を参照してください。

データを読み取る Java コマンドは、**EXEC CICS** コマンドの SET オプションに相当するもののみをサポートします。戻されるデータは、CICS ストレージから Java オブジェクトに自動的にコピーされます。

ファイル制御に関連する Java インターフェースには、5 つのカテゴリーがあります。

ファイル
: 他のファイル・クラスのスーパークラス。すべてのファイル・クラスに共通のメソッドが含まれます。

KeyedFile
1 次索引を使用してアクセスされる KSDS ファイル、2 次索引を使用してアクセスされる KSDS ファイル、および 2 次索引を使用してアクセスされる ESDS ファイルに共通のインターフェースが含まれます。

KSDS
KSDS ファイルに固有のインターフェースが含まれます。

ESDS 相対バイト・アドレス (RBA、1 次索引) または拡張相対バイト・アドレス (XRBA) からアクセスされる ESDS ファイルに固有のインターフェースが含まれます。RBA ではなく、XRBA を使用するには、setXRBA(true) メソッドを発行します。

RRDS
相対レコード番号 (RRN、1 次索引) からアクセスされる RRDS ファイルに固有のインターフェースが含まれます。

ファイルごとに、作動可能な 2 つのオブジェクト、つまり、File オブジェクトと FileBrowse オブジェクトがあります。 File オブジェクトはファイル自体を表し、次の API オペレーションを実行するメソッドで使用できます。

- DELETE
- READ
- REWRITE
- UNLOCK
- WRITE
- STARTBR

File オブジェクトは、必要なファイル・クラスを明示的に開始するユーザー・アプリケーションによって作成されます。FileBrowse オブジェクトは、ファイル上のブラウズ・オペレーションを表します。特定のファイルに対して常に複数のアクティブ・ブラウズが可能であり、各ブラウズは REQID で区別されます。メソッドは、次の API オペレーションを実行するために FileBrowse オブジェクトに対してインスタンス化することができます。

- ENDBR
- READNEXT
- READPREV
- RESETBR

FileBrowse オブジェクトは、ユーザー・アプリケーションによって明示的にインスタンス化されません。STARTBR オペレーションを実行するメソッドによって作成され、ユーザー・クラスに戻されます。

以下の表では、JCICS クラスとメソッドが、CICS ファイルおよび索引のタイプごとに **EXEC CICS** API コマンドにどのようにマップされるかを示しています。これらの表では、JCICS クラスとメソッドは `class.method()` の形式で示されます。例えば、KeyedFile.read() は、KeyedFile クラスの read() メソッドを参照しています。

最初の表では、キー付きファイルのクラスとメソッドを示しています。

表 4. キー付きファイルのクラスとメソッド

KSDS 1 次または **2** 次索引のクラスとメソッド	**ESDS 2** 次索引のクラスとメソッド	**CICS File API** コマンド
KeyedFile.read()	KeyedFile.read()	**READ**
KeyedFile.readForUpdate()	KeyedFile.readForUpdate()	**READ UPDATE**
KeyedFile.readGeneric()	KeyedFile.readGeneric()	**READ GENERIC**
KeyedFile.rewrite()	KeyedFile.rewrite()	**REWRITE**
KSDS.write()	KSDS.write()	**WRITE**
KSDS.delete()		**DELETE**
KSDS.deleteGeneric()		**DELETE GENERIC**
KeyedFile.unlock()	KeyedFile.unlock()	**UNLOCK**
KeyedFile.startBrowse()	KeyedFile.startBrowse()	**START BROWSE**
KeyedFile.startGenericBrowse()	KeyedFile.startGenericBrowse()	**START BROWSE GENERIC**
KeyedFileBrowse.next()	KeyedFileBrowse.next()	**READNEXT**
KeyedFileBrowse.previous()	KeyedFileBrowse.previous()	**READPREV**
KeyedFileBrowse.reset()	KeyedFileBrowse.reset()	**RESET BROWSE**
FileBrowse.end()	FileBrowse.end()	**END BROWSE**

次の表は、キー付きでないファイルのクラスとメソッドを示しています。ESDS と RRDS は 1 次索引によってアクセスされます。

ESDS 1 次索引のクラスとメソッド	**RRDS 1** 次索引のクラスとメソッド	**CICS File API** コマンド
ESDS.read()	RRDS.read()	**READ**
ESDS.readForUpdate()	RRDS.readForUpdate()	**READ UPDATE**
ESDS.rewrite()	RRDS.rewrite()	**REWRITE**
ESDS.write()	RRDS.write()	**WRITE**
	RRDS.delete()	**DELETE**
KeyedFile.unlock()	RRDS.unlock()	**UNLOCK**
ESDS.startBrowse()	RRDS.startBrowse()	**START BROWSE**
ESDS_Browse.next()	RRDS_Browse.next()	**READNEXT**
ESDS_Browse.previous()	RRDS_Browse.previous()	**READPREV**
ESDS_Browse.reset()	RRDS_Browse.reset()	**RESET BROWSE**
FileBrowse.end()	FileBrowse.end()	**END BROWSE**
ESDS.setXRBA()		

ファイルに書き込まれるデータは、Java バイト配列でなければなりません。

データはファイルから RecordHolder オブジェクトに読み取られます。ストレージは CICS によって提供され、プログラムの終わりに自動的に解放されます。

File メソッドには **KEYLENGTH** 値を指定する必要はありません。使用される長さは、渡されるキーの実際の長さです。FileBrowse オブジェクトが作成されると、

startBrowse メソッドで指定されたキーの長さが入っています。この長さは、そのオブジェクトに対する以降のブラウズ要求で CICS に渡されます。

ブラウズ・オペレーションに **REQID** を指定する必要はありません。各ブラウズ・オブジェクトには、固有の REQID が含まれ、そのブラウズ・オブジェクトに対する以降のすべてのブラウズ要求に自動的に使用されます。

HTTP サービスおよび TCP/IP サービス

`HttpHeader`、`NameValueData`、および `FormField` クラスの getter は、HTTP ヘッダー、名前と値のペア、および該当する API コマンドのフォーム・フィールド値を戻します。

メソッド	**JCICS** クラス	**`EXEC CICS`** コマンド
get*()	CertificateInfo	EXTRACT CERTIFICATE / EXTRACT TCPIP
get*()	HttpRequest	EXTRACT WEB
getHeader()	HttpRequest	WEB READ HTTPHEADER
getFormField()	HttpRequest	WEB READ FORMFIELD
getContent()	HttpRequest	WEB RECEIVE
getQueryParm()	HttpRequest	WEB READ QUERYPARM
startBrowseHeader()	HttpRequest	WEB STARTBROWSE HTTPHEADER
getNextHeader()	HttpRequest	WEB READNEXT HTTPHEADER
endBrowseHeader()	HttpRequest	WEB ENDBROWSE HTTPHEADER
startBrowseFormField()	HttpRequest	WEB STARTBROWSE FORMFIELD
getNextFormField()	HttpRequest	WEB READNEXT FORMFIELD
endBrowseFormField()	HttpRequest	WEB ENDBROWSE FORMFIELD
startBrowseQueryParm()	HttpRequest	WEB STARTBROWSE QUERYPARM
getNextQueryParm()	HttpRequest	WEB READNEXT QUERYPARM
endBrowseQueryParm()	HttpRequest	WEB ENDBROWSE QUERYPARM
writeHeader()	HttpResponse	WEB WRITE
getDocument()	HttpResponse	WEB RETRIEVE
getCurrentDocument()	HttpResponse	WEB RETRIEVE
sendDocument()	HttpResponse	WEB SEND

注: HttpRequest オブジェクトを取得するには、メソッド get HttpRequestInstance() を使用してください。

CICS Web サポートによって処理される各着信 HTTP 要求には、HTTP ヘッダーが含まれています。要求で POST HTTP verb を使用する場合、文書データも含まれます。CICS Web サポートによって生成される各応答 HTTP 要求には、HTTP ヘッダーと文書データが含まれています。

これを処理するために、JCICS は次の Web および TCP/IP サービスを提供します。

HTTP ヘッダー
: HttpRequest クラスを使用して HTTP ヘッダーを調べることができます。

GET モードの HTTP では、クライアントが HTTP フォームに入力し、送信ボタンを選択した場合、照会ストリングが送信されます。

SSL CICS Web サポートは、TcpipRequest クラスを提供します。これは、要求を送信したクライアントに関する詳細情報と、SSL サポートに関する基本情報を取得するために、HttpRequest によって拡張されます。SSL 証明書が提供される場合、CertificateInfo クラスを使用して詳細に調べることができます。

文書 文書がサーバーに公開される (HTTP POST) 場合、CICS 文書として提供されます。HttpRequest クラスの getDocument() メソッドを呼び出すことによって、この文書にアクセスできます。既存文書の処理の詳細については、59 ページの『文書サービス』を参照してください。

要求から生じる HTTP クライアント Web コンテンツを提供するために、サーバー・プログラマーは、Document Services API を使用して CICS 文書を作成し、sendDocument() メソッドを呼び出す必要があります。

CICS Web サポートについて詳しくは、CICS Web サポートを参照してください。JCICS Web クラスについて詳しくは、JCICS Javadoc 情報を参照してください。

プログラム・サービス

JCICS は、CICS プログラム制御コマンド LINK、RETURN、および INVOKE APPLICATION をサポートします。

Java プログラムが既存の CICS アプリケーション・データにアクセスできるようにするツールについては、Java からの構造化データとの対話を参照してください。

表 5 には、CICS プログラム制御コマンドにマップされるメソッドと JCICS クラスがリストされています。

表 *5.* メソッド、*JCICS* クラス、および *CICS* コマンド間の関係：

EXEC CICS コマンド	**JCICS** クラス	**JCICS** メソッド
LINK	プログラム	link()
RETURN	TerminalPrincipalFacility	setNextTransaction() 、setNextCOMMAREA() 、setNextChannel()
INVOKE APPLICATION	アプリケーション	invoke()

LINK

link() メソッドを使用して、CICS に対して定義される別のプログラムに制御を移動することができます。ターゲット・プログラムは、CICS でサポートされる任意の言語にすることができます。

RETURN

このコマンドの疑似会話型の側面のみがサポートされます。return に対する CICS 呼び出しを行う必要はありません。アプリケーションは通常どおり終了できます。疑似会話型機能は、TerminalPrincipalFacility クラスのメソッドでサポートされます。setNextTransaction() は、RETURN の TRANSID オプションの使用に相当します。setNextCOMMAREA() は、COMMAREA オプションの使用に相当します。一方、setNextChannel() は、CHANNEL オプションの使用に

相当します。これらのメソッドは、プログラムの実行中にいつでも呼び出すことができ、プログラムが終了するときに有効になります。

INVOKE
いずれか 1 つのプログラム・エントリー・ポイントに対応する操作を指定することで、アプリケーションを呼び出すことができます。アプリケーション・エントリー・ポイント・プログラムの名前を知っておく必要はなく、プログラムがパブリック/プライベートのどちらであるかも無関係です。

注: 指定される COMMAREA の長さは、CICS の LENGTH 値として使用されます。 COMMAREA が任意の 2 つの CICS サーバー (製品/バージョン/リリースの任意の組み合わせ) 間で渡される場合、この値は 24 KB を超えてはなりません。この制限により、ヘッダーに COMMAREA およびスペースを使用できます。

スケジューリング・サービス

JCICS は、CICS スケジューリング・サービスをサポートします。これにより、タスク用に保管されたデータを取り出し、インターバル制御要求を取り消し、指定された時間にタスクを開始することができます。

メソッド	**JCICS** クラス	**EXEC CICS** コマンド
cancel()	StartRequest	CANCEL
retrieve()	タスク	RETRIEVE
issue()	StartRequest	START

Task.retrieve() メソッドによって取り出される内容を定義するには、java.util.BitSet オブジェクトを使用します。com.ibm.cics.server.RetrieveBits クラスは、BitSet オブジェクトで設定できる次のビットを定義します。

- RetrieveBits.DATA
- RetrieveBits.RTRANSID
- RetrieveBits.RTERMID
- RetrieveBits.QUEUE

これらは、**EXEC CICS** RETRIEVE コマンドのオプションに対応します。

Task.retrieve() メソッドは、RetrieveBits の設定に応じて、単一の呼び出しで最大 4 つの情報を取り出します。DATA、RTRANSID、RTERMID および QUEUE データは、RetrievedData オブジェクトに置かれ、このオブジェクトは RetrievedDataHolder オブジェクトに保持されます。次の例では、データと transid を取り出します。

```
BitSet bs = new BitSet();
bs.set(RetrieveBits.DATA, true);
bs.set(RetrieveBits.RTRANSID, true);
RetrievedDataHolder rdh = new RetrievedDataHolder();
t.retrieve(bs, rdh);
byte[] inData = rdh.value.data;
String transid = rdh.value.transId;
```

直列化サービス

JCICS は、タスクによるリソースの使用をスケジュールできる CICS 直列化サービスをサポートします。

メソッド	**JCICS** クラス	**EXEC CICS** コマンド
dequeue()	SynchronizationResource	DEQ
enqueue()、tryEnqueue()	SynchronizationResource	ENQ

ストレージ・サービス

CICS サービスを使用した明示的なストレージ管理 (**EXEC CICS** GETMAIN など) はサポートされません。標準の Java ストレージ管理機能で、タスク専用ストレージのニーズを十分に満たすことができます。

タスク間のデータの共用は、CICS リソースを使用して行われなければなりません。

名前は一般的に、Java ストリングまたはバイト配列として表されます。これらが必要な長さであることを確認する必要があります。

スレッドとタスクのサンプル

CICS Java アプリケーションが OSGi または Liberty 環境内で実行されている場合、`CICSExecutorService` を使用することにより、CICS タスク/トランザクションごとにそれぞれ異なるスレッドで作業を実行することができます。

Java 実行可能オブジェクトまたは呼び出し可能オブジェクトを実行プログラム・サービスに実行依頼します。実行依頼されたアプリケーション・コードは、新規 CICS タスク下の別個のスレッドで実行されます。 Java から作成された通常のスレッドとは異なり、実行プログラムで制御されるスレッドは JCICS API および CICS サービスにアクセスできます。 CICS OSGi または Liberty 環境では、標準の OSGi API を使用して `CICSExecutorService` を検索することができます。あるいは、JCICS API 便利メソッド `CICSExecutorService.runAsCICS()` を使用することもできます。このメソッドは、自動的にサービスを検索して、実行可能オブジェクトまたは呼び出し可能オブジェクトを実行依頼します。

注: Liberty での非 HTTP 要求に CICS タスクが作成されるのは、タイプ 2 の接続で JCICS または JDBC データ・ソースが最初に呼び出された場合のみです。

以下の例では、アプリケーション・コードの実行可能な部分を `CICSExecutorService` に実行依頼する Java クラスの抜粋を示します。アプリケーション・コードは単純に CICS TSQ への書き込みを行います。

```
public class ExecutorTest
{
    public static void main(String[] args)
    {
        // Inline the new Runnable class
        class CICSJob implements CICSTransactionRunnable
        {
            public void run()
            {
                // Create a temporary storage queue
                TSQ test_tsq = new TSQ();
                test_tsq.setType(TSQType.MAIN);

                // Set the TSQ name
                test_tsq.setName("TSQWRITE");

                // Write to the temporary storage queue
                // Use the CICS region local CCSID so it is readable
                String test_string = "Hello from a non CICS Thread - "+ threadId;

                try
```

```
            {
                test_tsq.writeItem(test_string.getBytes(System.getProperty("com.ibm.cics.jvmserver.local.ccsid")));
            }
            catch (Exception e)
            {
                e.printStackTrace();
            }
        }

        @Override
        public String getTranid()
        {
            // *** This transaction id should be installed and available ***
            return "IJSA";
        }
    }

    // Create and run the new CICSJob Runnable
    Runnable task = new CICSJob();
    CICSExecutorService.runAsCICS(task);
  }
}
```

一時記憶域キュー・サービス

JCICS は、CICS 一時記憶コマンド DELETEQ TS、READQ TS、および WRITEQ TS をサポートします。

JCICS メソッドと EXEC CICS コマンド間の対話

Java プログラムが既存の CICS アプリケーション・データにアクセスできるようにするツールについては、Java からの構造化データとの対話を参照してください。

表 6 は、CICS 一時記憶コマンドにマップされるメソッドと JCICS クラスをリストしています。

表 *6.* メソッド、*JCICS* クラスおよび *CICS* コマンド間の関係

メソッド	**JCICS** クラス	**EXEC CICS** コマンド
delete()	TSQ	DELETEQ TS
readItem()、 readNextItem()	TSQ	READQ TS
writeItem()、 rewriteItem()、 writeItemConditional()、 rewriteItemConditional()	TSQ	WRITEQ TS

DELETEQ TS

TSQ クラスの delete() メソッドを使用して一時記憶域キュー (TSQ) を削除することができます。

READQ TS

CICS INTO オプションは Java プログラムではサポートされません。TSQ クラスの readItem() および readNextItem() メソッドを使用して、TSQ から特定の項目を読み取ることができます。これらのメソッドは、引数の 1 つとして ItemHolder オブジェクトを取ります。これには、バイト配列で読み取られるデータが含まれます。このバイト配列のストレージは、CICS によって作成され、プログラムの終わりにガーベッジ・コレクションされます。

WRITEQ TS

Java バイト配列で一時記憶域キューに書き込まれるデータを提供する必要があります。NOSPACE 条件が検出される場合、writeItem() および rewriteItem() メソッドは中断し、データをキューに書き込むためのスペースが使用可能になるまで待機します。writeItemConditional() および rewriteItemConditional() メソッドは、NOSPACE 条件の場合に中断しませんが、この条件を NoSpaceException としてアプリケーションに即時に戻します。

端末サービス

JCICS は、以下の CICS 端末サービス・コマンドをサポートします。

メソッド	**JCICS** クラス	**EXEC CICS** コマンド
converse()	TerminalPrincipalFacility	CONVERSE
	サポート対象外	HANDLE AID
receive()	TerminalPrincipalFacility	RECEIVE
send()	TerminalPrincipalFacility	SEND
	サポート対象外	WAIT TERMINAL

タスクに基本機能として端末が割り振られている場合、CICS は、標準出力と標準エラー・ストリームとして使用できる 2 つの Java PrintWriter コンポーネントを自動的に作成します。これらのコンポーネントはタスク端末にマップされます。 `out` および `err` という名前の 2 つのストリームは Task オブジェクトのパブリック・ファイルであり、System.out や System.err と同様に使用できます。

端末に送られるデータは、Java バイト配列で提供されなければなりません。データは端末から DataHolder オブジェクトに読み込まれます。CICS では返されるデータ用のストレージが提供されます。これは、プログラムの終了時に割り振り解除されます。

データと **XML** の間の変換

JCICS は、データから XML への変換とその逆の変換を実行するための API コマンドをサポートしています。それらのコマンドは、**EXEC CICS TRANSFORM DATATOXML** コマンドおよび **TRANSFORM XMLTODATA** コマンドと同等の機能を提供します。

メソッド	**JCICS** クラス	**EXEC CICS** コマンド
SetName	XmlTransform	TRANSFORM DATATOXML, TRANSFORM XMLTODATA
dataToXML	Transform	TRANSFORM DATATOXML
xmltoData	Transform	TRANSFORM XMLTODATA
setChannel	TransformInput	TRANSFORM DATATOXML, TRANSFORM XMLTODATA

メソッド	**JCICS** クラス	**EXEC CICS** コマンド
setDataContainer	TransformInput	TRANSFORM DATATOXML TRANSFORM XMLTODATA
setElementName	TransformInput	TRANSFORM DATATOXML, TRANSFORM XMLTODATA
setElementNamespace	TransformInput	TRANSFORM DATATOXML, TRANSFORM XMLTODATA
setNsContainer	TransformInput	TRANSFORM XMLTODATA
setTypeName	TransformInput	TRANSFORM DATATOXML, TRANSFORM XMLTODATA
setTypeNamespace	TransformInput	TRANSFORM DATATOXML, TRANSFORM XMLTODATA
setXmlContainer	TransformInput	TRANSFORM DATATOXML, TRANSFORM XMLTODATA
setXmltransform	TransformInput	TRANSFORM DATATOXML, TRANSFORM XMLTODATA
getElementName	TransformOutput	TRANSFORM DATATOXML, TRANSFORM XMLTODATA
getElementNamespace	TransformOutput	TRANSFORM DATATOXML, TRANSFORM XMLTODATA
getTypeName	TransformOutput	TRANSFORM DATATOXML, TRANSFORM XMLTODATA
getTypeNamespace	TransformOutput	TRANSFORM DATATOXML, TRANSFORM XMLTODATA

一時データ・キュー・サービス

JCICS は、CICS 一時データ・コマンド DELETEQ TD、READQ TD、および WRITEQ TD をサポートします。INTO オプションを除くすべてのオプションがサポートされます。

JCICS メソッドと EXEC CICS コマンド間の対話

Java プログラムが既存の CICS アプリケーション・データにアクセスできるようにするツールについては、Java からの構造化データとの対話を参照してください。

表 7 は、CICS 一時データ・コマンドにマップされるメソッドと JCICS クラスをリストしています。

表 *7. メソッド、JCICS クラスおよび CICS コマンド間の関係*

メソッド	**JCICS** クラス	**EXEC CICS** コマンド
delete()	TDQ	DELETEQ TD
readData()、 readDataConditional()	TDQ	READQ TD
writeData()	TDQ	WRITEQ TD

DELETEQ TD

TDQ クラスの delete() メソッドを使用して一時データ・キュー (TDQ) を削除することができます。

READQ TD

CICS INTO オプションは Java プログラムではサポートされません。TDQ クラスの readData() または readDataConditional() メソッドを使用して TDQ から読み取ることができます。これらのメソッドは、バイト配列で読み取られるデータが入っている DataHolder オブジェクトのインスタンスをパラメーターとして取ります。このバイト配列のストレージは、CICS によって作成され、プログラムの終わりにガーベッジ・コレクションされます。

readDataConditional() メソッドは、CICS NOSUSPEND ロジックを駆動します。QBUSY 条件が検出されると、QueueBusyException として即時にアプリケーションに戻されます。

readData() メソッドは、別のタスクによって使用中のレコードにアクセスしようとするときに、コミットされたレコードがそれ以上ない場合は中断します。

WRITEQ TD

Java バイト配列で TDQ に書き込まれるデータを提供する必要があります。

作業単位 (UOW) サービス

JCICS は、CICS SYNCPOINT サービスをサポートします。

表 *8. UOW サービスの JCICS と EXEC CICS コマンド間の関係*

メソッド	**JCICS** クラス	**EXEC CICS** コマンド
commit()、rollback()	タスク	**SYNCPOINT**

Liberty JVM サーバーでは、UOW 同期点処理を Java Transaction API (JTA) を使用して制御できます。詳しくは、 94 ページの『Java Transaction API (JTA)』を参照してください。

Web サービスの例

JCICS は、アプリケーション内で Web サービスを操作するために使用できるすべての API コマンドをサポートします。

メソッド	**JCICS** クラス	**EXEC CICS** コマンド
invoke()	WebService	INVOKE WEBSERVICE
create()	SoapFault	SOAPFAULT CREATE
addFaultString()	SoapFault	SOAPFAULT ADD FAULTSTRING
addSubCode()	SoapFault	SOAPFAULT ADD SUBCODESTR
delete()	SoapFault	SOAPFAULT DELETE
create()	WSAEpr	WSAEPR CREATE
delete()	WSAContext	WSACONTEXT DELETE
set*()	WSAContext	WSACONTEXT BUILD
get*()	WSAContext	WSACONTEXT GET

次の例に、JCICS を使用して Web サービス要求を作成する方法を示します。

```
Channel requesterChannel = Task.getTask().createChannel("TestRequester");
Container appData = requesterChannel.createContainer("DFHWS-DATA");
byte[] exampleData = "ExampleData".getBytes();
appData.put(exampleData);

WebService requester = new WebService();
requester.setName("MyWebservice");
requester.invoke(requesterChannel, "myOperationName");

byte[] response = appData.get();
```

Web サービス要求で送受信されるアプリケーション・データを処理する際に、構造化データを処理している場合、IBM Record Generator for Java などのツールを使用してクラスを生成できます。 149 ページの『Java からの構造化データとの対話』を参照してください。また、Java を使用して XML の生成とコンシュームを直接行うこともできます。

JCICS の使用

JCICS ライブラリーのクラスは、Java クラスと同じ方法で使用します。アプリケーションは必要なタイプの参照を宣言し、クラスの新しいインスタンスが `new` 演算子を使用して作成されます。

基礎の CICS リソースの名前を指定するために、setName メソッドを使用して CICS リソースに名前を付けます。リソースを作成した後、標準の Java 構成体を使用してオブジェクトを操作します。宣言されたオブジェクトのメソッドを通常の方法で呼び出します。クラスごとにサポートされるメソッドの詳細は、提供される Javadoc で入手可能です。

CICS は、PROGRAM リソースの JVMCLASS 属性で指定されるクラスで、main(CommAreaHolder) の署名を使用するメソッドに制御を渡そうとします。このメソッドが見つからない場合、CICS は、main(String[]) メソッドを呼び出そうとします。

詳しくは、Java restrictions およびJCICS Javadoc 情報を参照してください。

この例では、TSQ オブジェクトを作成し、作成したばかりの一時記憶域キュー・オブジェクトで delete メソッドを呼び出し、キューが空の場合はスローされた例外をキャッチする方法を示しています。

```
// Define a package name for the program
package unit_test;

// Import the JCICS package
import com.ibm.cics.server.*;

// Declare a class for a CICS application
public class JCICSTSQ
{
    // The main method is called when the application runs
    public static void main(CommAreaHolder cah)
    {
        try
        {
            // Create and name a Temporary Storage queue object
            TSQ tsq = new TSQ();
            tsq.setName("JCICSTSQ");

            // Delete the queue if it exists
            try
            {
                tsq.delete();
            }
            catch(InvalidQueueIdException e)
            {
                // Absorb QIDERR
                System.out.println("QIDERR ignored!");
            }

            // Write an item to the queue
            String transaction = Task.getTask().getTransactionName();
            String message = "Transaction name is - " + transaction;
            tsq.writeItem(message.getBytes());
        }
        catch(Throwable t)
        {
            System.out.println("Unexpected Throwable: " + t.toString());
        }

        // Return from the application
        return;
    }
}
```

Java の制約事項

CICS 用 Java アプリケーションを開発する際は、プログラマーが注意すべき制約事項がいくつかあります。

CICS で使用される Java アプリケーションには、以下の制約事項が適用されます。

- System.exit() メソッドを使用しないでください。JVM サーバーでアプリケーションが実行されているときにこのメソッドを使用すると、JVM サーバーが終了し、CICS が静止するため、データ不整合につながる場合があります。 Java セキュリティー・ポリシーを使用して System.exit() の使用を禁止してください。関連情報については、Java セキュリティー・マネージャーの有効化を参照してください。
- JCICS API 呼び出し: これらの呼び出しを OSGi バンドルのアクティベーター・クラスで使用することはできません。

  注**:** OSGi バンドル・アクティベーターを実行する Java スレッドは、JCICS 対応ではありません。開発者が CICSExecutorService.runAsCICS() API を使用して、アクティベーターから新しい JCICS 対応スレッドを開始することは可能です。 JCICS コマンドはすべて、インストール・コマンドを実行したユーザー ID の権限で実行されます。そのため、管理者は、OSGi バンドル・アクティベーターをインストールする前に、その中で使用されるリソースを把握しておく必要があります。runAsCICS() API については、 69 ページの『スレッドとタスクのサンプル』でさらに詳しく説明しています。
- OSGi バンドル・アクティベーターで使用される start メソッドおよび stop メソッド: これらのメソッドは、適切な時間内に戻す必要があります。
- スレッド間で JCICS オブジェクトを共用しないでください。JCICS オブジェクトに対するインスタンス・メソッドの呼び出しが許可されるのは、そのオブジェクトを作成したスレッドのみです。
- CICS Java プログラムでファイナライザーを使用しないでください。ファイナライザーを推奨しない理由の解説については、Troubleshooting and supportを参照してください。

OSGi の使用に関するガイダンス

OSGi アプリケーションを開発する際は、いくつもの考慮事項があります。

依存関係の定義

OSGi バンドルで別の OSGi バンドルからの Java パッケージを使用する場合、2 つのバンドル間のインターフェースを明示的に表現する必要があります。別のバンドルからのパッケージを使用するバンドルでは、`manifest.mf` 内の **`Import-Package`** ステートメントにそのパッケージを追加する必要があります。パッケージを提供するバンドルでは、`manifest.mf` 内の **`Export-Package`** ステートメントに、提供するパッケージを追加する必要があります。両方の OSGi バンドルを環境にデプロイすると、この依存関係を解決できます。

JRE 拡張 (**`javax.*`** など) を含め、OSGi バンドルで使用されるすべてのパッケージは、明示的にインポートされる必要があります。これは、ランタイムがこれらのパッケージをブート委任などの他の手段で検出する場合にも適用されます。デフォルトでは、コア **`java.*`** パッケージのみが使用可能になると想定してください。

依存関係を表現するには、別の手段もあります。その 1 つは、バンドル・ヘッダー **`Require-Bundle`** です。ただし、**`Require-Bundle`** はより粗視化されており、ユーザーを特定のバンドルに関連付けます。**`Require-Bundle`** を使用すると、アーキテクチ

ャーに柔軟性がなくなり、パッケージを独立してバージョン管理する機能も制限されます。

JRE クラスの可視性、ブート委任、および system.packages.extra

OSGi では、コアとなる JRE パッケージ/クラス (`java.*`) のロードは常にブートストラップ・クラス・ローダーに委任されます。システム内に JRE は 1 つのみ存在することが前提となっているため、明示的な依存関係ステートメントは必要ありません。そのため、`java.*` 依存関係をバンドル・マニフェストに追加する必要は一切ありません。ただし、JRE の他の部分では、これらのパッケージを必要とするアプリケーション・バンドルで Import-Package ステートメントをコーディングする必要があります。例えば、ベンダー固有の拡張である `javax.*`、`com.sun.*`、および `com.ibm.*` にはインポートが必要です。その理由は、これらはブートストラップ・クラス・ローダーに委任されるのではなく、OSGi システムの一部として扱われるためです。

OSGi フレームワークは、既知の拡張パッケージをシステムに自動的に公開するシステム・バンドルを提供します。OSGi バンドルで提供される他のすべてのパッケージと同様に、アプリケーション・バンドルは Import ステートメントを組み込むことによって、自身の依存関係を登録します。この手法の利点は、新しいコードが含まれる OSGi バンドルをインストールすることによって、拡張パッケージを新しい実装で置き換えられることです。

このプロセスの例外となるのは、特殊な OSGi プロパティーを使用して特定のパッケージがブート委任リストに追加されている場合です。この方法は (パッケージにアクセスするために Import ステートメントが必要にならないため) 便利ではあるものの、OSGi の柔軟性が制限されてしまうため、ベスト・プラクティスであるとはみなされません。ベンダー固有の拡張パッケージの中には、OSGi 実装で自動的にシステム・バンドルに追加されないものもあります。そのような場合は、パッケージは本来 JRE から利用可能になるという前提に基づいて、`-Dorg.osgi.framework.system.packages.extra` プロパティーを使用してパッケージをシステム・バンドルに追加し、アプリケーションの Import で解決できるようにします。

バンドル・アクティベーター

バンドル・アクティベーターは、OSGi バンドルに含まれる **`BundleActivator`** インターフェースを実装するクラスです。アクティベーターを使用するには、OSGi バンドルのバンドル・マニフェスト内で **`Bundle-Activator`** ヘッダーを使用して、アクティベーターを宣言する必要があります。**`BundleActivator`** インターフェースには、作業を初期化または終了するために使用できる `start` メソッドおよび `stop` メソッドがあります。一般的なパターンは、アプリケーション内で使用するサービス依存関係を検索することです。ただし、コンポーネントとそのサービス依存関係をアクティブにするには、宣言型サービスなどのコンポーネント・モデルを使用することのほうが推奨されます。

シングルトン・バンドル

シングルトン・バンドルは、他のバージョンのバンドルがメモリーにロードされないようにするために使用されます。これにより、どの時点においてもランタイムに

存在できる解決済みバージョンは 1 つのみになります。一連のアプリケーションから単一のシステム・リソースにアクセスしなければならない場合は、シングルトン・バンドルを使用することが望ましいです。

OSGi バンドルのフラグメント

フラグメントとは、OSGi フレームワークによって動的にホスト・バンドルに接続される OSGi バンドルのことです。フラグメントはホスト・バンドルのクラス・ローダーを共用し、バンドルのライフサイクルには参加しません。そのため、フラグメントではバンドル・アクティベーターをサポートしていません。フラグメントの一般的なユース・ケースは、バンドルのパッチとして使用することです。**Bundle-ClassPath** で **.** の前にフラグメントを指定すると、クラスをホストからではなくフラグメントから優先的にロードできます。

OSGi サービス・レジストリー

OSGi サービス・レジストリーにより、バンドルがオブジェクトを共用レジストリーに公開することが可能になります。サービスは、Java インターフェースで公示され、OSGi 環境にインストールされている他のバンドルで使用可能になります。

マイクロサービス (µServices)

マイクロサービスは、小さな独立したコンポーネントを組み合わせて複合アプリケーションを構成し、これらのコンポーネントが特定の言語に依存しない API を使用して互いに通信するソフトウェア・アーキテクチャー・スタイルです。サイズが小さく、非常に細かく分離され、小さいタスクを処理することに重点を置くマイクロサービスを使用することで、モジュラー手法のシステム構築が容易になります。OSGi コンポーネントの間で µService を使用すると、バンドルのワイヤリングのみでは達成できない柔軟性および動的更新機能が実現します。 このことから、バンドルのワイヤリングよりも µService を優先して使用することが推奨されます。

バンドルおよびパッケージのバージョン管理

OSGi でのパッケージ・バージョン管理に望ましい手法は、セマンティック・バージョン管理モデルです。バージョン番号が MAJOR.MINOR.PATCH だとすると、以下のように増分します。

1. 互換性のない API 変更を行った場合は、バージョンのメジャー部分 (MAJOR) を増分します。
2. 前のバージョンと互換性のある機能を追加した場合は、バージョンのマイナー部分 (MINOR) を増分します。
3. 前のバージョンと互換性のあるバグ修正を行った場合は、バージョンのパッチ部分 (PATCH) を増分します。

実行環境

実行環境 (EE) は、JRE のシンボル表現です。次に例を示します。

```
Bundle-RequiredExecutionEnvironment: JavaSE-1.7
```

必要なすべての機能を提供する最小バージョンの EE を使用する必要があります。新しい OSGi バンドルを作成する際は、通常はアクティブに保守されている最新の

Java 実行環境で作成するのが適切です。特殊なアプリケーションでそれよりも低いバージョンが必要となる場合に限り、低いレベルに設定します。特定の EE を選択したら、バージョン・アップに明らかな利点がない限り、バージョンを変更してはなりません。EE のバージョンを高くしても、コードが新しい警告にさらされたり、非推奨が生じたりするなど、必要な作業が増えるのみで実質的な価値はありません。

Liberty JVM サーバーで実行する Java アプリケーションの開発

WebSphere Application Server Liberty を使用する Java EE アプリケーションをデプロイする場合には、Web コンテナーを実行するように Liberty JVM サーバーを構成します。

開発環境のセットアップ

JCICS API を使用する Java EE アプリケーションを開発するには、CICS Explorer および IBM CICS SDK for Java EE and Liberty をインストールする必要があります。

このタスクについて

CICS Liberty JVM サーバーでホストされている Java EE アプリケーションは、以下の一連のツールを使用して開発、パッケージ、およびデプロイできます。

- Eclipse および Eclipse Web Tools Platform は、Java EE アプリケーションを開発するためのツールを提供します。
- IBM CICS SDK for Java EE and Liberty は、JCICS、Java EE、および Liberty の各 API を、Java ビルド・パス・ライブラリーまたは OSGi ターゲット・プラットフォームの形式で提供します。
- CICS Explorer。CICS バンドル内で Java EE アプリケーションをパッケージ、デプロイ、および管理するためのツールを提供します。
- Explorer for z/OS。JVM サーバー・ログ・ファイルの表示など、z/OS 上でファイル、データ・セット、およびジョブを処理するためのツールを提供します。

これらのツールは、CICS Explorer のインストールの一部として一緒にインストールできます。

手順

1. CICS Explorer のインストールについて詳しくは、CICS Explorer 製品資料内の『CICS Explorer のダウンロードおよび開始』、および CICS Explorer 製品資料内の『CICS SDK for Java EE and Liberty のインストール』を参照してください。
2. 開発環境を再始動します。

タスクの結果

Liberty JVM サーバー内の Java EE アプリケーションを開発環境で開発、デプロイ、およびテストする準備ができました。

次のタスク

Java EE アプリケーションの開発を開始したり、JCICS、Java EE、および Liberty API を使用したりすることができます。IBM CICS SDK for Java に用意されているサンプルを使用して、作業を始めることができます。詳しくは、Java サンプル: サーブレットの例を参照してください。

Java EE アプリケーションと Liberty アプリケーション

CICS アプリケーションに対する最新のインターフェースを提供するために、Web アプリケーション・テクノロジーを使用するプレゼンテーション層を開発できます。Eclipse ベースの Web 開発ツールは、これらのアプリケーションを作成するための開発プラットフォームを提供します。オプションで CICS Explorer にインストールされる IBM CICS SDK for Java EE and Liberty は、CICS での実行用にこれらをビルドし、パッケージ化し、デプロイするためのサポートを提供します。

このタスクについて

Liberty サーバーにデプロイできる Web アプリケーション・プロジェクトには、次の 3 つのタイプがあります。

- 動的 Web プロジェクト (WAR)
- OSGi アプリケーション・プロジェクト (EBA)
- エンタープライズ・アプリケーション・プロジェクト (EAR)

WAR には、イメージや HTML ファイルのような静的リソースの他に、CICS での Liberty、フィルター、関連メタデータなどの動的 Java EE リソースを含めることができます。

EBA は、WAB と OSGi バンドルを含めることができる Java アーカイブ・ファイルです。WAB は Web 使用可能 OSGi バンドルで、イメージや HTML ファイルのような静的リソースの他に、JSP サーブレットとファイル、フィルター、関連メタデータなどが含まれます。

EAR は、EBA が WAB と OSGi バンドルを編成するのと同じ方法で、WAR と EJB モジュールを 1 つのコンテナーに編成する方法です。

動的 Web プロジェクトの作成

Java アプリケーション用の Web プレゼンテーション層を開発する場合、動的 Web プロジェクトを作成できます。

始める前に

Eclipse IDE に Web 開発ツールがインストールされていることを確認します。詳しくは、 79 ページの『開発環境のセットアップ』を参照してください。

Liberty によって追加された制限により、WAR ファイルにデプロイされたサーブレットから OSGi バンドルにアクセスすることはできません。この制限には、CICS バンドルによって直接インストールされた OSGi バンドルへのアクセスも含まれます。この制限を回避するには、アプリケーションを EBA (OSGi アプリケーション・プロジェクト) に含まれる WAB としてデプロイする必要があります。EBA は、Web と OSGi コンポーネントが対話できるコンテナーです。

このタスクについて

CICS Explorer および IBM CICS SDK for Java のヘルプには、Web アプリケーションを開発してパッケージ化する以下のステップの実行方法が詳細に説明されています。

手順

1. アプリケーション用に 動的 Web プロジェクトを作成します。ビルド・パスを更新して Liberty のライブラリーを追加する必要があります。
   a. 動的 Web プロジェクトを右クリックし、「ビルド・パス」 > 「ビルド・パスの構成」をクリックします。プロジェクトの「プロパティー」ダイアログが開きます。
   b. 「Java のビルド・パス」で、「ライブラリー」タブをクリックします。
   c. 「ライブラリーの追加」をクリックし、「**Liberty JVM** サーバー・ライブラリー」を選択します。
   d. 「次へ」をクリックし、CICS バージョンを選択し、「終了」をクリックしてライブラリーの追加を完了します。
   e. 「**OK**」をクリックして、変更内容を保管します。
2. Web アプリケーションを開発します。JCICS API を使用して CICS サービスに、JDBC を使用して DB2 に、また、JMS を使用して IBM MQ にアクセスできます。 IBM CICS SDK for Java EE and Liberty には、JCICS と JDBC を使用した Web コンポーネントの例が含まれています。
3. オプション: CICS セキュリティーでアプリケーションを保護する場合は、動的 Web プロジェクトに `web.xml` ファイルを作成して、CICS セキュリティー制約を組み込みます。IBM CICS SDK for Java EE and Liberty には、CICS 用の正しい情報が入った、このファイル用のテンプレートが含まれます。詳しくは、Liberty JVM サーバーでのユーザー認証 を参照してください。
4. アプリケーションをパッケージするために、1 つ以上の CICS バンドル・プロジェクトを作成します。CICS リソースの定義とインポートを追加します。各 CICS バンドルには、ID とバージョンが含まれており、変更を細かく管理できます。
5. オプション: URI からのインバウンド Web 要求をマップして特定のトランザクションの下で実行する場合は、URIMAP および TRANSACTION リソースを CICS バンドルに追加します。これらのリソースを定義しない場合、CJSA という名前の提供されたトランザクションの下ですべての処理が実行されます。これらのリソースは、動的にインストールされ、CICS の中のバンドルの一部として管理されます。

タスクの結果

開発環境のセットアップ、動的 Web プロジェクトからの Web アプリケーションの作成、そのアプリケーションをデプロイするためのパッケージ化が終わりました。

次のタスク

アプリケーションをデプロイする用意ができたら、CICS バンドル・プロジェクトを zFS にエクスポートします。参照されたプロジェクトは、ビルドされ、zFS への転

送に組み込まれます。または、Liberty デプロイメント・モデルに従って、アプリケーションを WAR としてエクスポートし、稼働中の Liberty JVM サーバーの dropins ディレクトリーにそれをデプロイすることもできます。

OSGi アプリケーション・プロジェクトの作成

OSGi アプリケーション・プロジェクト (EBA) とは、一連のバンドルをまとめたものです。アプリケーションは、さまざまな種類の OSGi バンドルで構成できます。

始める前に

Eclipse IDE に WebSphere Application Server Developer Toolsがインストールされていることを確認します。詳しくは、 79 ページの『開発環境のセットアップ』を参照してください。

このタスクについて

CICS Explorer および IBM CICS SDK for Java のヘルプには、Web アプリケーションを開発してパッケージ化する以下のステップの実行方法が詳細に説明されています。

手順

1. 「CICS TS 5.5 **with Java EE and Liberty**」テンプレートを使用して、 Java 開発用のターゲット・プラットフォームをセットアップします。ターゲットが、Eclipse の現行インストールよりも新しいバージョンであるという警告が表示されることがありますが、この警告メッセージは無視してかまいません。
2. アプリケーション用に OSGi バンドル・プロジェクトを作成します。ターゲット・プラットフォームでパッケージが実質的に使用可能になるため、適切な Import ステートメントをバンドル・マニフェストに含める必要があります。Web 使用可能 OSGi バンドル・プロジェクトは、動的 Web プロジェクトと同等のバンドルです。Web 使用可能 OSGi バンドル・プロジェクトを使用して、アプリケーションを OSGi アプリケーション・プロジェクト (エンタープライズ・バンドル・アーカイブまたは EBA ファイル) 内にデプロイできます。Web 使用可能 OSGi バンドル・プロジェクト (WAB ファイル) および Web 使用可能でない OSGi バンドル・プロジェクトを OSGi アプリケーション・プロジェクト内で混合できます。通常、Web 使用可能 OSGi バンドル・プロジェクトはアプリケーションのフロントエンドを実装し、(ビジネス・ロジックを含んでいる) 非 Web OSGi バンドルと対話します。
3. Web アプリケーションを開発します。JCICS API を使用して CICS サービスおよび JDBC にアクセスし、DB2 に接続することができます。IBM CICS SDK for Java には、JCICS と DB2 を使用する、Web コンポーネントと OSGi バンドルの例が含まれます。ビジネスとプレゼンテーションのロジックを分離するために、JCICS を使用する OSGi バンドルを作成してください。また、OSGi バンドルの中でセマンティック・バージョン管理を使用して、アプリケーションのビジネス・ロジックの更新を管理できます。JDBC DriverManager インターフェースを介して Db2 を使用する WAB または OSGi バンドルごとに、`com.ibm.db2.jcc` の Import-Package ヘッダーをバンドル・マニフェストに含めます。このインポートを省略すると、「`java.sql.SQLException: jdbc:default:connection` に適切なドライバーが見つかりません (`java.sql.SQLException: No suitable driver found for`

jdbc:default:connection)」というエラー・メッセージが出されます。JDBC DataSource インターフェースを使用する場合、このインポートは不要です。

4. オプション: Web アプリケーションのユーザーを認証したい場合は、Web プロジェクトに web.xml ファイルを作成して、セキュリティー制約を含めます。IBM CICS SDK for Java には、CICS 用の正しい情報が入った、このファイル用のテンプレートが含まれます。詳しくは、Liberty JVM サーバーでのユーザー認証を参照してください。
5. OSGi バンドルを参照する OSGi アプリケーション・プロジェクトを作成します。
6. OSGi アプリケーション・プロジェクトを参照する CICS バンドル・プロジェクトを作成します。CICS リソースの定義とインポートを追加することもできます。各 CICS バンドルには、ID とバージョンが含まれており、変更を細かく管理できます。
7. オプション: URI からのインバウンド Web 要求をマップして特定のトランザクションの下で実行する場合は、URIMAP および TRANSACTION リソースを CICS バンドルに追加します。これらのリソースを定義しない場合、CJSA という名前の提供されたトランザクションの下ですべての処理が実行されます。これらのリソースは、動的にインストールされ、CICS の中のバンドルの一部として管理されます。

タスクの結果

これで、開発環境をセットアップし、OSGi Web アプリケーションを作成し、それをデプロイメント用にパッケージしました。

次のタスク

アプリケーションをデプロイする用意ができたら、CICS バンドル・プロジェクトを zFS にエクスポートします。参照されたプロジェクトは、ビルドされ、zFS への転送に組み込まれます。または、開発デプロイメント・モデルに従って、アプリケーションを EBA ファイルとしてエクスポートし、稼働中の Liberty JVM サーバーの dropins ディレクトリーにそれをデプロイすることもできます。dropins を使用する場合は、セキュリティーおよび他のサービス品質は構成できないことに注意してください。

エンタープライズ・アプリケーション・プロジェクトの作成

Enterprise Java Bean モジュール (EJB モジュール) などのコンポーネントを開発する場合や、Web プロジェクトまたは EJB、あるいはその両方をグループ化する場合には、エンタープライズ・アプリケーション・プロジェクトを使用できます。

始める前に

Eclipse IDE に Web 開発ツールがインストールされていることを確認します。詳しくは、79 ページの『開発環境のセットアップ』を参照してください。

このタスクについて

CICS Explorer および IBM CICS SDK for Java のヘルプには、エンタープライズ・アプリケーションを開発してパッケージ化するための次の各ステップの実行方法が詳細に説明されています。

手順

1. エンタープライズ・アプリケーション・プロジェクトを作成します。
2. アプリケーションのコンポーネントを開発します。これらのコンポーネントは、通常、EJB モジュールと動的 Web プロジェクトです。開発したコンポーネントをエンタープライズ・アプリケーション・プロジェクトに追加します。 詳しくは、Enterprise JavaBeans (EJB) プロジェクトの作成を参照してください。
3. エンタープライズ・アプリケーションをパッケージ化するために、1 つ以上の CICS バンドル・プロジェクトを作成します。CICS リソースの定義とインポートを追加します。すべての CICS バンドルには、ID とバージョンが含まれており、変更を細かく管理できます。
4. オプション: URI からのインバウンド Web 要求をマップして特定のトランザクションの下で実行する場合は、URIMAP および TRANSACTION リソースを CICS バンドルに追加します。これらのリソースを定義しない場合、CJSA という名前の提供されたトランザクションの下ですべての処理が実行されます。これらのリソースは、動的にインストールされ、CICS の中のバンドルの一部として管理されます。

タスクの結果

これで、開発環境をセットアップし、エンタープライズ・アプリケーション・プロジェクトを作成し、それをデプロイメント用にパッケージ化しました。

次のタスク

アプリケーションをデプロイする用意ができたら、CICS バンドル・プロジェクトを zFS にエクスポートします。参照されたプロジェクトは、ビルドされ、zFS への転送に組み込まれます。あるいは、Liberty デプロイメント・モデルに従って、アプリケーションを EAR としてエクスポートして、その EAR を `<application>` 要素と一緒にデプロイするか、または稼働中の Liberty JVM サーバーの drop-ins ディレクトリーに格納することもできます。

URI マップとトランザクションの作成

CSD や BAS などの従来の方法でアプリケーション・リソースをインストールすることも、アプリケーション・リソースを CICS バンドルに追加することもできます。 CICS バンドルは、アプリケーション・コードと CICS リソースをグループ化して同じ場所に配置する便利な手法です。この手法は、例えば Java EE アプリケーションを CICS バンドルにデプロイする場合に役立ちます。インバウンド Web 要求をマップする URI マップを指定し、特定のアプリケーション・トランザクションでそれらの要求を実行できます。

始める前に

アプリケーション・リソースを作成するには、プロジェクト・エクスプローラーに CICS バンドル・プロジェクトが存在していなければなりません。詳細については、CICS Explorer 製品資料内の『CICS バンドル・プロジェクトの作成』を参照してください。この CICS バンドル・プロジェクトを使用して、デプロイするアプリケーションをパッケージします。

このタスクについて

デフォルトでは、すべての Java EE アプリケーション要求で、CICS によって提供される CJSA というトランザクションが使用されます。ただし、インバウンド要求のアプリケーション URI を別のトランザクションにマップすることもできます。このフィーチャーは、アプリケーションへのアクセスを安全な方法で制御する上で役立つ場合があります。セキュリティー管理者は、ユーザーがアクセスできるトランザクションを制御するように CICS を構成できるからです。

手順

1. 以下のようにして、アプリケーション・トランザクションの定義を作成します。
   a. Eclipse の「リソース」パースペクティブに切り替えます。CICS バンドル・プロジェクトを右クリックして、「新規」 > 「トランザクション定義」をクリックします。 「トランザクション定義」ウィザードが開きます。
   b. トランザクションの 4 文字の名前を入力します。 トランザクション名の先頭を C にしないでください。この文字は CICS で予約されているためです。
   c. プログラム名 `DFHSJTHP` を入力します。 この CICS プログラムを使用する必要があるのは、Liberty サーバーへのインバウンド Java EE 要求のセキュリティー検査がこのプログラムで扱われるためです。
   d. 「終了」をクリックして、定義を CICS バンドル・プロジェクトに作成します。

   アプリケーション・トランザクションは Java EE アプリケーションが実行されている CICS 領域で常に実行される必要があるので、リモート・トランザクションを作成するための属性を設定しないでください。
2. 以下のようにして、URI マップの定義を作成します。
   a. CICS バンドル・プロジェクトを右クリックして、「新規」 > 「**URI** マップ定義」をクリックします。
   b. URI マップの 8 文字の名前を入力します。 URI マップ名の先頭を DFH にしないでください。この接頭部は CICS で予約されているためです。
   c. ホスト名を入力します。 `*` を使用すると任意のホスト名と一致させることができます。あるいは、アプリケーションが実行されるマシンのホスト名を指定することもできます。
   d. アプリケーション URI のパスを入力します。 CICS は、インバウンド要求内の URI を、URI マップ内の値と突き合わせ、アプリケーション・トランザクションを実行します。

e. 「使用法」セクションで、「**JVM** サーバー」を選択し、オプションでポート番号を入力します。

f. 「終了」をクリックして、URI マップを作成します。

3. 以下のようにして、URI マップ定義を編集します。

a. 「スキーム」フィールドを編集して、URI マップのスキームを入力します。HTTP がデフォルトですが、要求を暗号化するために SSL セキュリティーを使用する場合は、HTTPS を設定することができます。 HTTP 要求と HTTPS 要求の両方で、HTTP ヘッダーにユーザー ID およびパスワードが指定される基本認証を使用することができます。

b. 「トランザクション」フィールドを編集して、アプリケーション・トランザクションの名前を入力します。

c. オプション: 「ユーザー ID」フィールドを編集して、アプリケーション要求を実行するユーザー ID を入力します。 この値は、基本認証が使用可能である場合には無視されます。値を指定せず、HTTP 要求にユーザー ID とパスワードが含まれない場合、CICS は CICS 領域のデフォルト・ユーザー ID の下で要求を実行します。

タスクの結果

URI マップとトランザクションが CICS バンドル・プロジェクトに作成されました。バンドルがデプロイおよびインストールされると、これらのリソースが CICS 領域に動的に作成されます。

次のタスク

複数の異なるトランザクションを使ってさまざまなアプリケーション操作を実行する場合、または HTTP スキームと HTTPS スキームの両方をサポートする場合は、さらにリソースを作成することができます。アプリケーションを配置する準備ができたら、CICS Explorer 製品資料内の『CICS バンドルのデプロイ』を参照してください。

Liberty JVM サーバー内で実行するよう Java EE アプリケーションをマイグレーションする

ネットワークを介して CICS にアクセスする Liberty インスタンスの中で実行する Java EE アプリケーションがある場合、そのアプリケーションを Liberty JVM サーバーの中で実行すると、パフォーマンスを最適化できます。

このタスクについて

CICS は、Liberty で使用可能な機能のサブセットをサポートしています。CICS 統合モードの Liberty でサポートされる機能のリストについては、 134 ページの『Liberty フィーチャー』を参照してください。

アプリケーションでセキュリティーを使用する場合は、Liberty セキュリティー機能を引き続き使用できますが、追加アクションなしで、CICS タスクがトランザクション CJSA 下で実行され、CICS での URIMAP マッチングが使用不可になり、すべてのリソース・アクセスが CICS のデフォルト・ユーザー ID で実行される可能性があります。Liberty で決定された同じユーザー ID で CICS タスクを実行できる

ようにして、セキュリティー・ソリューションをより効率的に CICS と統合する場合は、Liberty JVM サーバーでのユーザー認証を参照してください。

手順

1. アプリケーションが CICS とデータを受け渡しするときに正しい JCICS エンコードが使用されることを確認して、JCICS API を使用して CICS サービスに直接アクセスするようにアプリケーションを更新します。 エンコードの詳細については、 47 ページの『データ・エンコード』を参照してください。このステップが適用されるのは、`runAsCICS()` API を使用して CICS 統合モードの Liberty または CICS 標準モードの Liberty を使用する場合のみです。
2. 基本認証に CICS セキュリティーを使用する場合は、動的 Web プロジェクトの `web.xml` ファイルのセキュリティー制約を更新して、認証に CICS ロールを使用するようにします。このステップが適用されるのは、CICS 統合モードの Liberty または CICS 標準モードの Liberty を使用して、`runAsCICS()` メソッドによって作業を `CICSExecutorService` に実行依頼する場合のみです。

```
<auth-constraint>
    <description>All authenticated users of my application</description>
    <role-name>cicsAllAuthenticated</role-name>
</auth-constraint>
```

3. アプリケーションを WAR (動的 Web プロジェクト)、EBA (OSGi アプリケーション・プロジェクト) ファイル、または EAR (エンタープライズ・アーカイブ) ファイルとして CICS バンドルにパッケージ化します。 CICS バンドルは、アプリケーションのデプロイメントの単位です。バンドルの中のすべての CICS リソースは、動的にインストールされ、一緒に管理されます。一緒に管理するアプリケーション・コンポーネントの CICS バンドル・プロジェクトを作成します。
4. CICS バンドル・プロジェクトを zFS にデプロイし、CICS バンドルを Liberty JVM サーバーにインストールします。

タスクの結果

アプリケーションは、JVM サーバーで実行中です。

CICS プログラムからの Java EE アプリケーションへのリンク

Liberty JVM サーバーで実行される Java EE アプリケーションへ、CICS トランザクションの初期プログラムをリンクしたり、CICS プログラムから LINK、START、または START CHANNEL コマンドを実行して呼び出したりできます。

CICS プログラムからリンクする Java EE アプリケーションは、Web アーカイブ (WAR) またはエンタープライズ・アプリケーション・アーカイブ (EAR) としてパッケージ化された Plain Java Object (POJO) でなければなりません。EJB、CDI Bean、または OSGi のアプリケーションにリンクすることはできません。`@EJB` を使用した EJB の注入を含め、POJO での依存性注入はサポートされません。EJB などのリソースの参照を取得するには、代わりに JNDI 検索を使用できます。この情報は、CICS 統合モードの Liberty にのみ適用されます。

CICS プログラムから Java EE アプリケーションにリンクする主な理由は、以下の 3 つです。

- Java コードが既存の Web アプリケーションの一部であり、それを CICS アプリケーションにリンクする必要がある。そうすれば、ロジックを 1 つ保守するだけで JCICS API を使用してコードから CICS リソースにアクセスできるようになる。
- CICS アプリケーションの一部として新機能を Java で作成する必要がある。例えば、既に Java の形で存在するサード・パーティーのライブラリーや API を使用する必要がある場合など。
- Java で既存の COBOL アプリケーションを再実装する必要がある。例えば、保守コストを削減したり、Java のスキルを最大限に活用したり、汎用的なプロセッサーではなく特殊なエンジンでアプリケーションを実行できるようにしたりすることが必要な場合があります。

CICS プログラムから Java EE アプリケーションへリンクすると、CICS は、Liberty 内で実行されている JCA リソース・アダプターにメッセージを送信します。JCA リソース・アダプターは、呼び出し側プログラムと同じ CICS タスクで、ターゲットの Java EE アプリケーションにリンクします。Java EE アプリケーションは呼び出し側プログラムと同じ作業単位 (UOW) で実行されるので、リカバリー可能な CICS リソースに対する更新は、トランザクションの終了時にコミット/バックアウトされます。ただし、Java EE アプリケーションが呼び出される場合、JTA トランザクション・コンテキストは存在しません。アプリケーションで JTA トランザクションが開始される場合、CICS UOW をコミットするために同期点が実行され、新しい同期点が作成されます。これは、アプリケーションが REQUIRED トランザクション属性を使用して EJB を呼び出す場合など、アプリケーションの代わりにコンテナーによって JTA トランザクションが開始された場合にも発生します。

ベスト・プラクティスとして、CICS プログラムによってリンクされたコードは、アプリケーションのプレゼンテーション・ロジックではなくビジネス・ロジックに組み込むようにしてください。例えば、HTTP 要求が行われないのに CICS プログラムからサーブレットをリンクしても意味がありません。

CICS プログラムへ Java EE アプリケーションをリンクするための Liberty JVM サーバーの構成

Java EE アプリケーションのリンクに対応できるように Liberty JVM サーバーを構成するには、`server.xml` に `cicsts:link-1.0` フィーチャーを追加します。Java EE アプリケーションをデプロイする前に、このフィーチャーを追加してください。

セキュリティー

CICS プログラムへ Java EE アプリケーションをリンクすると、CICS タスクのユーザー ID が Java EE アプリケーションに渡されます。Liberty はユーザー認証を行わずに、CICS から渡された ID を信頼します。ただし、Liberty はそのユーザー ID が構成済みのユーザー・レジストリーに存在するかどうかを検査します。可能であれば、Liberty で SAF レジストリーを使用してください。このレジストリーが、CICS から渡されたユーザー ID を検査するからです。SAF 以外のタイプのユーザー・レジストリーを使用する場合は、レジストリーに同じユーザー ID が存在すれば、そのユーザー ID が Java EE アプリケーションに渡されます。そのユーザー

ID がユーザー・レジストリーに存在しなければ、認証されないそのユーザー ID で Java EE アプリケーションがリンクされます。

Java EE アプリケーションをリンクするときにセキュリティーを構成するには、`server.xml` に `cicsts:security-1.0` フィーチャーを組み込みます。このフィーチャーを組み込まなければ、認証なしで Java EE アプリケーションがリンクされます。その結果、Java EE アプリケーションの許可検査がまったく行われない可能性があります。それでも、JCICS API による CICS リソースへのアクセスは、CICS タスクのユーザー ID で実行されます。

CICS プログラムに Java EE アプリケーションをリンクする際に、以下の Web セキュリティー・メカニズムは適用されません。

- Java EE の Web セキュリティー・メカニズム。`web.xml` の `<auth-constraint>` やサーブレットの `@HttpConstraint` など。
- トラスト・アソシエーション・インターセプター (TAI)。
- JVM サーバー・プロファイルの `com.ibm.cics.jvmserver.unclassified.userid` プロパティー。
- URIMAP。

Java EE アプリケーションでは、EJB を呼び出したり EJB セキュリティーを適用したりして追加の許可検査を実行できます。例えば、**`@RolesAllowed`** 注釈を使用してセッション Bean メソッドに対する許可検査を実行できます。EJB セキュリティーの詳細については、 Java EE Tutorialを参照してください。

Liberty のセキュリティーについて詳しくは、Liberty JVM サーバーのセキュリティーの構成 (Configuring security for a Liberty JVM server)を参照してください。

CICS プログラムで呼び出すための Java EE アプリケーションの準備

注釈を使用すると、Java メソッドを CICS アプリケーションで呼び出すことができます。CICS がユーザーに代わって PROGRAM リソースを作成します。Java EE アプリケーションは Liberty JVM サーバーで実行され、WAR または EAR 内にデプロイできます。

始める前に

呼び出す Java クラスおよびメソッドを特定し、サイトの標準と CICS 命名規則に従って、適切な CICS プログラム名を決定します。

Liberty JVM サーバーが、Java EE アプリケーションへのリンクが有効になるよう構成されていることを確認します。詳しくは、CICS プログラムから Java EE アプリケーションへのリンクを参照してください。

手順

1. Web プロジェクトのクラスパスに、`@CICSProgram` 注釈クラスを追加します。CICS Explorer を使用している場合、`@CICSProgram` 注釈は Liberty JVM サー

バー・ライブラリーの一部として提供されます。CICS Explorer を使用していない場合は、com.ibm.cics.server.invocation.annotations.jar JAR ファイルを使用する必要があります。

2. CICS が呼び出すメソッドを含めるクラスを作成します。 CICS 固有のコードをアプリケーションの他の部分から切り離しておくことができるため、クラスを作成することをお勧めします。
3. 作成予定の CICS PROGRAM リソースごとにメソッドを作成します。
4. 各メソッドに @CICSProgram で注釈を付け、PROGRAM 名でパラメーターを指定します (例えば、@CICSProgram("PROGNAME"))。

   CICS PROGRAM 名は、以下のようになります。

   - 1 文字から 8 文字であること。
   - A-Z a-z 0-9 $ @ # パターンと一致していること。

   @CICSProgram で注釈を付けた、単一のメソッドが含まれる単純なクラスの例を示します。

```
public class CustomerLinkTarget
{
    @CICSProgram("CUSTGET")
    public void getCustomer()
    {
        // do work here
    }
}
```

5. CICS Explorer を使用している場合は、Web プロジェクトの注釈処理を有効にします。注釈処理を有効にするには、次のいずれかの操作を行います。
   - 警告の下線が表示されている @CICSProgram 注釈上にカーソルを置き、クイック・フィックスを使用して注釈処理を有効にします。
   - Web プロジェクトを右クリックして「プロパティー **(Properties)**」を選択します。「注釈処理 **(Annotation Processing)**」ページを検索します。「プロジェクト固有の設定を有効にする」と「注釈処理を使用可能にする」の両方をチェックします。
6. CICS Explorer を使用している場合、自動的に検証が行われ、注釈が正しく配置されていること、および注釈が付けられているメソッドとそのメソッドが含まれるクラスが以下の要件を満たしていることが確認されます。 注釈の要件は以下のとおりです。
   - メソッドに配置されていること。
   - PROGRAM 名の属性値が指定されていること。

   メソッドの要件は以下のとおりです。

   - 具象メソッドであること (抽象メソッドではないこと)。
   - public メソッドであること。
   - 引数がないこと。

   クラスの要件は以下のとおりです。

   - すべての注釈付きメソッドが静的でない限り、引数 (暗黙的または明示的を問わず) を持たないコンストラクターを使用すること。

- 最上位レベルであること (ネストされていたり、匿名であったりしないこと)。
- 同じ PROGRAM 名で注釈が付けられた複数のメソッドが含まれていないこと。

7. 注釈付きメソッドの内容を書き込みます。多くの場合、内容には以下のステージが伴います。
   a. チャネルからコンテナーを取得します。
   b. チャネル内のコンテナーから入力データを取得します。
   c. データ・マッピング・コードを使用して、入力データを Java オブジェクトに変換します。
   d. アプリケーション・ビジネス・ロジックを呼び出します。
   e. データ・マッピング・コードを使用して、生成された Java オブジェクトを出力データに変換します。
   f. 出力データをチャネル内のコンテナーに格納します。

   `@CICSProgram` で注釈を付けた単一のメソッドが含まれるクラスの例と、コンテナーから入力データを取得し、出力データをコンテナーに格納するコードの例を次に示します。

```
public class CustomerLinkTarget
{
    @CICSProgram("CUSTGET")
    public void getCustomer()
    {
        Channel currentChannel = Task.getTask().getCurrentChannel();
        Container dataContainer = currentChannel.getContainer("DATA");

        // do work here

        Container resultContainer = currentChannel.createContainer("RESULT");
        byte[] results = null; // change this to be the result of the work
        resultContainer.put(results);
    }
}
```

8. アプリケーションをビルドします。
   - CICS Explorer を使用している場合は、Web プロジェクトを右クリックして「**エクスポート**」->「**WAR ファイル (WAR file)**」を選択するか、アプリケーションが含まれる CICS バンドル・プロジェクトを右クリックして「バンドルを **z/OS UNIX** ファイル・システムにエクスポート **(Export Bundle to z/OS UNIX file system)**」を選択します。
   - CICS Build Toolkit を使用している場合は、注釈プロセッサーが自動的に起動されます。
   - その他のツールを使用して Java コードをビルドする場合は、`com.ibm.cics.server.invocation.annotations.jar` JAR ファイル (`@CICSProgram` 注釈を定義しているファイル) が Java コンパイラーのクラスパスにあることを確認してください。また、`com.ibm.cics.server.invocation.jar` JAR ファイル (注釈プロセッサーが含まれているファイル) が Java コンパイラーのクラスパスにあるか、あるいは **-processorpath** オプションで指定されていることも確認する必要があり

ます。両方の JAR ファイルは、z/OS UNIX の *usshome* /lib ディレクトリーにあります。ここで、*usshome* は **USSHOME** システム初期設定パラメーターの値です。

- WAR ファイルの WEB-INF/lib ディレクトリー内にあるライブラリー JAR にクラスがパッケージ化される場合、JAR をビルドする際に生成されたメタデータをエクスポートします。CICS Explorer では、ライブラリー・プロジェクトを動的 Web プロジェクトのデプロイメント・アセンブリーに追加することでメタデータをエクスポートできます。動的 Web プロジェクトのプロパティー・ダイアログで、「デプロイメント・アセンブリー (Deployment Assembly)」ページを選択し、「追加 **(Add)**」ボタンをクリックして、ライブラリー・プロジェクトを選択します。CICS は、EAR ファイル内のユーティリティー JAR にパッケージ化されたクラスでは、@CICSProgram 注釈をサポートしません。

9. アプリケーションをデプロイします。

タスクの結果

アプリケーションが CICS バンドルによってインストールされる場合、CICS バンドルが有効になると、PROGRAM リソースが作成されます。アプリケーションが server.xml から直接インストールされるか、<application> エレメントを使用してファイルからインストールされる場合、アプリケーションがインストールされる時点で PROGRAM リソースが作成されます。

これで、以下のコマンドを使用して、別の CICS プログラムから Java プログラムにリンクできるようになりました。

```
EXEC CICS LINK PROGRAM("CUSTGET") CHANNEL()
```

プログラムのライフサイクル

Java EE アプリケーションが Liberty にインストールされると、cicsts:link-1.0 フィーチャーは @CICSProgram で注釈が付けられたメソッドを検索します。該当する各メソッドに対して、動的に PROGRAM リソースをインストールします。CICS バンドルを使用して Java EE アプリケーションがインストールされる場合、バンドルが有効になった時点で PROGRAM リソースが作成されます。それ以外の場合は、Liberty がアプリケーションをインストールする際に PROGRAM リソースが作成されます。

アプリケーションが削除されると、CICS は、そのアプリケーションに関連付けられ、動的にインストールされたすべての PROGRAM リソースを削除します。アプリケーションが CICS バンドルを使用してインストールされた場合、バンドルが無効にされると、CICS がそのプログラムを削除します。アプリケーションを呼び出したタスクがまだ進行している間にアプリケーションが削除されると、エラーが発生する可能性があります。したがって、Java EE アプリケーションを削除する前に、そのアプリケーションに関連付けられているすべての PROGRAM リソースを無効にし、作業をドレーンさせる必要があります。そうしなければ、Liberty がアプリケーションをアンインストールすると、プログラムが削除されます。

ほとんどの場合、独自の PROGRAM 定義を作成する必要はありません。ただし、CICS に自動的に作成させたくない場合、あるいは特定の属性を指定しなければならない場合は、独自の PROGRAM 定義を作成できます。プラットフォームにデプロ

イされる CICS アプリケーションの一部として専用プログラムを作成するには、そのアプリケーションの一部としてインストールされる CICS バンドルに、その専用プログラムを定義する必要があります。CSD、BAS、または CICS バンドル内にプログラム定義を作成し、自分でインストールできます。CICS が `@CICSProgram` で注釈が付けられたメソッドを検出し、そのメソッドに一致する PROGRAM リソースがすでにインストールされている場合、CICS は既存のリソースを置き換えません。

プログラム定義を作成する際は、`@CICSProgram` で注釈が付けられたメソッドを含むクラスと同じクラス名を指定する必要があります。オプションで、メソッド名も指定できます。CICS はこの情報を、プログラムの呼び出し時に検証します。JVMCLASS 属性には、クラス名およびオプションのメソッド名を `wlp:classname#methodname` の形式で含める必要があります。以下に例を示します。

```
wlp:com.example.CustomerLinkTarget#getCustomer
```

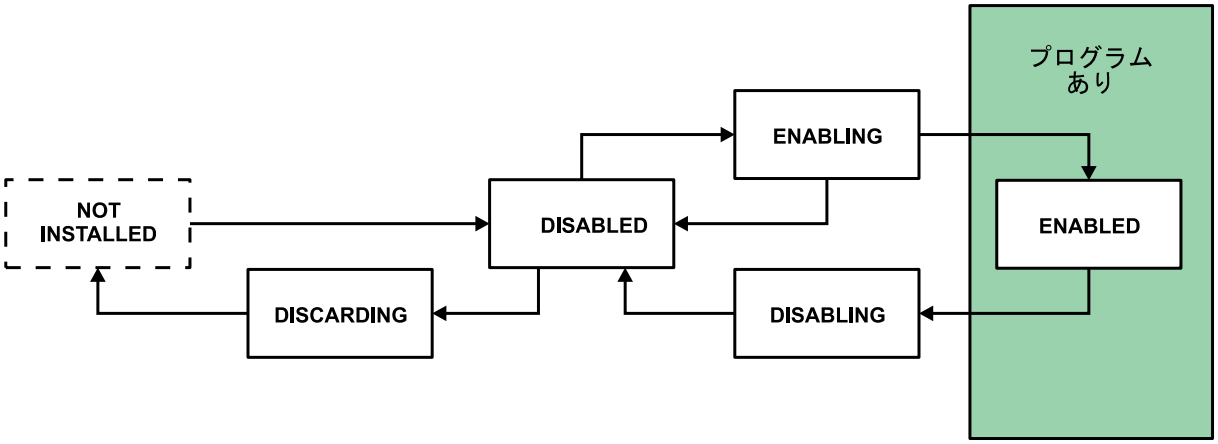


図 2. *CICS* バンドルのライフサイクル (*`@CICSProgram`* 注釈の *PROGRAM* リソースが存在する場合)

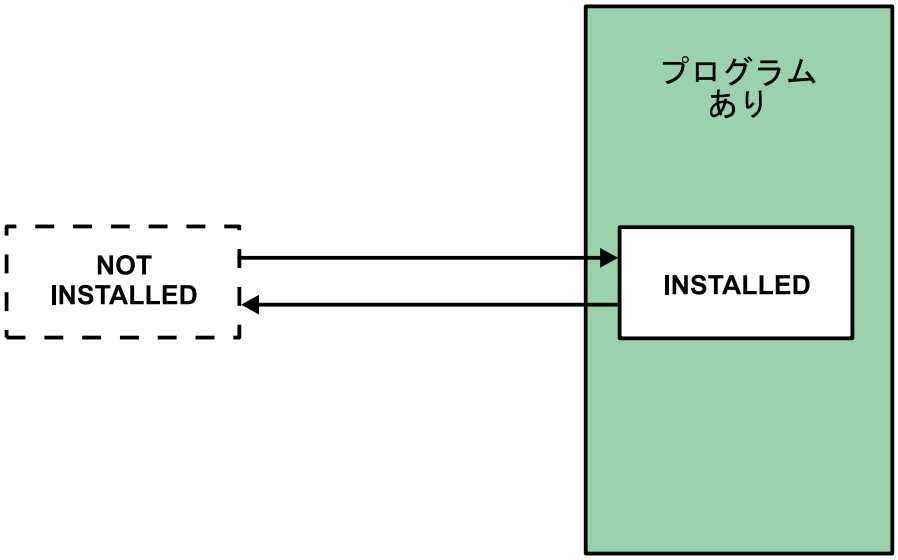


図 3. スタンドアロン *Web* アプリケーションのライフサイクル (*@CICSProgram* 注釈の *PROGRAM* リソースが存在する場合)

Java Transaction API (JTA)

Java Transaction API (JTA) を使用すると、複数のリソース・マネージャーに対するトランザクション更新を調整することができます。

Java Transaction API (JTA) を使用すると、CICS リソースおよび他のサード・パーティー製リソース・マネージャー (Liberty JVM サーバー内のタイプ 4 データベース・ドライバー接続など) に対するトランザクション更新を調整できます。このシナリオでは Liberty トランザクション・マネージャーがトランザクション・コーディネーターです。 CICS 作業単位は、CICS システム外部でトランザクションが開始したかのように従属しています。

注: JVM プロファイル・オプション `com.ibm.cics.jvmserver.wlp.jta.integration=true` を設定して自動構成を使用する場合、または手動で `server.xml` を構成して `<cicsts_jta Integration="true"/>` エレメントを含める場合、CICS 作業単位は JTA トランザクションに参加せず、個別にコミットまたはロールバックされることになります。

CICS データ・ソースを使用したローカル Db2 データベースへのタイプ 2 ドライバー接続では、CICS Db2 接続を使用してアクセスします。 JTA を使用して、他の CICS リソースに対する更新に合わせて調整する必要はありません。

JTA の中で、複数のリソース・マネージャーに対する更新をカプセル化して調整するための UserTransaction オブジェクトを作成します。以下のコード・フラグメントは、UserTransaction を作成して使用する方法を示しています。

```
InitialContext ctx = new InitialContext();
UserTransaction tran = (UserTransaction)ctx.lookup("java:comp/UserTransaction");
```

```
DataSource ds = (DataSource)ctx.lookup("jdbc/SomeDB");
Connection con = ds.getConnection();

// Start the User Transaction
tran.begin();

// Perform updates to CICS resources via JCICS API and
// to database resources via JDBC/SQLJ APIs

if (allOk) {
    // Commit updates on both systems
    tran.commit();
} else {
    // Backout updates on both systems
    tran.rollback();
}
```

OSGi アプリケーションを使用している場合は、必ず次のエントリーを MANIFEST.MF に含めます。

```
Import-Package: javax.transaction;version="[1.1,2)"
```

開発環境によっては、この依存関係がデフォルトで強調表示されない場合もあります。明示的に調べて、最小バージョンの 1.1 が指定されていることを確認するようお勧めします。ランタイム環境自体に依存関係を解決させるようにしている場合、基礎となる JRE により、より低いバージョンのパッケージに解決されて、Liberty ランタイムとの競合が発生する可能性があります。

CICS 作業単位の場合とは異なり、`begin()` メソッドを使って明示的に UserTransaction を開始する必要があります。 `begin()` を呼び出すと、CICS は UserTransaction の開始前に行われたすべての更新をコミットします。UserTransaction は `commit()` または `rollback()` のいずれかのメソッド呼び出しにより終了するか、Web アプリケーション終了時に Web コンテナーによって終了されます。 UserTransaction がアクティブ状態である間、プログラムは JCICS `Task` `commit()` メソッドや `rollback()` メソッドを呼び出すことができません。

JCICS の `Task.commit()` および `Task.rollback()` メソッドは JTA トランザクション・コンテキスト内では無効になります。どちらを試行した場合も、InvalidRequestException がスローされます。

Liberty のデフォルト動作では、未確定 JTA トランザクションのリカバリーを試みる前に最初の UserTransaction が作成されるのを待機します。しかし、CICS は、Liberty JVM サーバーの初期化が完了するとすぐにトランザクション・リカバリーを開始します。JVM サーバーが無効な状態でインストールされていると、JVM サーバーが有効になった時点でリカバリーが実行されます。

EJB を使用している場合は、 EJB での JTA トランザクションの使用を参照してください。

Java Persistence API (JPA)

開発者がアプリケーションで利用する、リレーショナル・データベース・エンティティーのオブジェクト指向バージョンを作成するには、JPA を使用できます。

データ型、キー、およびテーブル間の関係を含め、データベース内のテーブルとそのコンテンツを記述するために使用できる注釈および XML 拡張は、JPA を使用して提供することができます。開発者は SQL を使用する代わりに、API を使用してデータベース操作を実行できます。

CICS は jpa-2.0 および jpa-2.1 をサポートしています。これらのバージョンの違いについては、Java Persistence API (JPA) フィーチャーの概要を参照してください。

- Entity オブジェクトは単純な Java クラスであり、具象クラスにも抽象クラスにもすることができます。それぞれがデータベース表内の行を表し、プロパティーとフィールドを使用して状態を保守します。各フィールドはテーブル内の列にマップされ、その特定のフィールドに関する重要な情報が追加されます。例えば、1 次キー (つまり、ヌルにできないフィールド) を指定できます。

```
@Entity
@Table(name = "JPA")
public class Employee implements Serializable
{
    @Id
    @Column(name = "EMPNO")
    private Long EMPNO;

    @Column(name = "NAME", length = 8)
    private String NAME;

    private static final long serialVersionUID = 1L;

    public Employee()
    {
        super();
    }

    public Long getEMPNO()
    {
        return this.EMPNO;
    }

    public void setEMPNO(Long EMPNO)
    {
        this.EMPNO = EMPNO;
    }

    public String getNAME()
    {
        return this.NAME;
    }

    public void setNAME(String NAME)
    {
        this.NAME = NAME;
    }
}
```

- EntityManagerFactory は、パーシスタンス・ユニットの EntityManager を生成するために使用されます。EntityManager は、アプリケーションで使用されているアクティブな entity オブジェクトのコレクションを維持します。EntityManager クラスを使用して、クラスを初期化し、データ保全性を管理するためのトランザクションを作成できます。次に、Entity クラスの get メソッドと set メソッドを使用してデータと対話してから、Entity トランザクションを使用してデータをコミットします。

以下の例に、レコードを挿入する場合のサンプル・コードを記載します。

```
@WebServlet("/Create")
public class Create extends HttpServlet
{
    private static final long serialVersionUID = 1L;

    @PersistenceUnit(unitName = "com.ibm.cics.test.wlp.jpa.annotation.cics.datasource")
    EntityManagerFactory emf;

    InitialContext ctx;

    /**
    * @throws NamingException
    * @see HttpServlet#HttpServlet()
    */
    public Create() throws NamingException
    {
        super();
        ctx = new InitialContext();
    }

    /**
    * @see HttpServlet#doGet(HttpServletRequest request,
    HttpServletResponse response)
    */
    protected void doGet(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException
    {
        // Get the servlet parms
        String id = request.getParameter("id");
        String name = request.getParameter("name");

        // Create a new employee object
        Employee newEmp = new Employee();
        newEmp.setEMPNO(Long.valueOf(id));
        newEmp.setNAME(name);

        // Get the entity manager factory
        EntityManager em = emf.createEntityManager();

        // Get a user transaction
        UserTransaction utx;

        try
        {
            // Start a user transaction and join the entity manager to it
            utx = (UserTransaction) ctx.lookup("java:comp/UserTransaction");
            utx.begin();
            em.joinTransaction();

            // Persist the new employee
            em.persist(newEmp);

            // End the transaction
            utx.commit();
        }
        catch(Exception e)
        {
            throw new ServletException(e);
        }

        response.getOutputStream().println("CREATE operation completed");
    }
}
```

- `@PersistenceUnit` は、`EntityManagerFactory` とこれに関連付けられたパーシスタンス・ユニットへの依存関係を表します。パーシスタンス・ユニットの名前は、`persistence.xml` ファイルで定義されます。次に、エンティティーとテーブルをデータベースに接続するために、バンドル内に `persistence.xml` ファイルを作成します。`persistence.xml` ファイルで、これらのエンティティーの接続先データベースを記述します。このファイルには、プロバイダーの名前、エンティティー自体、データベース接続 URL、およびドライバーなどの重要な情報を含めます。

  以下の例には、サンプル `persistence.xml` が含まれています。

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
    xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">

    <persistence-unit name="com.ibm.cics.test.wlp.jpa.annotation.cics.datasource">
        <jta-data-source>jdbc/jpaDataSource</jta-data-source>

        <class>com.ibm.cics.test.wlp.jpa.annotation.cics.datasource.entities.Employee</class>

        <properties>
            <property name="openjpa.LockTimeout" value="30000" />
            <property name="openjpa.Log" value="none" />
            <property name="openjpa.jdbc.UpdateManager" value="operation-order" />
        </properties>
    </persistence-unit>
</persistence>
```

Enterprise JavaBeans (EJB)

Enterprise JavaBeans (EJB) は Java API であり、Java EE 仕様のサブセットです。 EJB には、アプリケーションのビジネス・ロジックが含まれています。 Lite サブセットを含めて、EJB は CICS Liberty によって完全にサポートされています。

EJB のサポートを提供する Liberty フィーチャーは、次のとおりです。

表 *9.* サポートを提供する *Liberty* フィーチャー

フィーチャー	サポート	**Java EE** バージョン
ejbLite-3.1	このフィーチャーは、EJB 仕様で定義されているとおりに EJB テクノロジーの Lite サブセットを有効にします。このサブセットには、EJB 3.x API に書き込まれているローカル・セッション Bean のサポートが含まれています。	Java EE 6
mdb-3.1	このフィーチャーは、EJB テクノロジーのメッセージ駆動型 Bean サブセットを有効にします。これは、ejbLite フィーチャーがセッション Bean のサポートを有効にするのに似ています。	Java EE 6
ejbLite-3.2	このフィーチャーは、EJB 仕様で定義されているとおりに EJB テクノロジーの Lite サブセットを有効にします。このサブセットには、EJB 3.x API に書き込まれているローカル・セッション Bean、非永続 EJB タイマー、および非同期ローカル・インターフェース・メソッドのサポートが含まれています。	Java EE 7

表 9. サポートを提供する *Liberty* フィーチャー (続き)

フィーチャー	サポート	**Java EE** バージョン
mdb-3.2	このフィーチャーは、EJB テクノロジーのメッセージ駆動型 Bean サブセットを有効にします。これは、ejbLite フィーチャーがセッション Bean のサポートを有効にするのに似ています。	Java EE 7
ejbHome-3.2	EJB 2.x API のサポート、特に `javax.ejb.EJBLocalHome` インターフェースのサポートを有効にします。ejbRemote フィーチャーと組み合わせると、`javax.ejb.EJBHome` インターフェースもサポートされます。	Java EE 7
ejbRemote-3.2	リモート EJB インターフェースのサポートを有効にします。	Java EE 7
ejbPersistentTimers-3.2	パーシスタント EJB タイマーのサポートを有効にします。	Java EE 7
ejb-3.2	EJB 3.2 の完全サポートを有効にします。リモート EJB テクノロジーを含む、すべての EJB 3.2 テクノロジーをカバーします。	Java EE 7

プロシージャー

フィーチャーは、`server.xml` ファイルで有効にします。以下に例を示します。

```
<featureManager>
    <feature>ejb-3.2</feature>
</featureManager>
```

詳しくは、以下の情報を参照してください。

- Developing EJB 3.x applications : WebSphere Developer Tools を使用した EJB アプリケーションの開発について説明しています。
- エンタープライズ Bean (EJB) パーシスタント・タイマー・アプリケーションの開発: EJB 永続タイマー・アプリケーションの開発について説明しています。
- 別のアプリケーションにあるローカル EJB コンポーネントを呼び出す Enterprise JavaBeans アプリケーションの使用: 別のアプリケーションにあるローカル EJB コンポーネントを呼び出す Enterprise JavaBeans アプリケーションの使用について説明しています。

Enterprise JavaBeans (EJB)プロジェクトの作成

Java アプリケーションの EJB を開発する場合は、EJB プロジェクトを作成できます。

始める前に

Eclipse IDE に Web 開発ツールがインストールされていることを確認します。詳しくは、 79 ページの『開発環境のセットアップ』を参照してください。

このタスクについて

CICS Explorer および IBM CICS SDK for Java のヘルプには、EJB アプリケーションを開発してパッケージ化する以下のステップの実行方法が詳細に説明されています。

手順

1. アプリケーション用に EJB プロジェクトを作成します。
2. EJB アプリケーションを開発します。JCICS API を使用して CICS サービスに、JDBC を使用して DB2 に、また、JMS を使用して IBM MQ にアクセスできます。
3. オプション: アプリケーションを保護するには、セキュリティー注釈を使用するか、`ejb-jar.xml` ファイル内にセキュリティー制約を指定できます。詳しくは、エンタープライズ・アプリケーション・セキュリティーを参照してください。
4. EJB プロジェクトをエンタープライズ・アプリケーション・プロジェクト (EAR) に追加します。

タスクの結果

これで、開発環境のセットアップ、EJB プロジェクトの作成、およびデプロイメント用のパッケージ化が終了しました。

EJB での JTA トランザクションの使用

Liberty の Enterprise JavaBeans (EJB) で JTA トランザクションを使用する方法。

このタスクについて

EJB は、Liberty JVM サーバーによって管理される Java オブジェクトで、Java アプリケーションのモジュラー・アーキテクチャーを可能にします。Liberty JVM サーバーは、EJB Lite 3.1、EJB Lite 3.2、および EJB 3.2 をサポートしています。EJB は、エンタープライズ・アプリケーション・プロジェクトで作成されたエンタープライズ・アプリケーション・アーカイブ (EAR) ファイルを使用して Liberty サーバーにデプロイされます。エンタープライズ・アプリケーション・プロジェクト には、EJB プロジェクトと Web プロジェクトの両方を含めることができます。

EJB Lite を有効にするには、該当する `ejbLite-3.1` または `ejbLite-3.2` フィーチャーを `server.xml` 構成ファイルに追加します。EJB は、`ejb-3.2` を `server.xml` 構成ファイルに追加することによって有効になります。EJB はコンテナーにデプロイされます。このコンテナーはバックグラウンドで動作し、セッション管理、トランザクション、およびセキュリティーなどの面で一定の規準に準拠するようにします。

EJB は、コンテナー管理と Bean 管理の 2 種類のトランザクション管理をサポートします。コンテナー管理トランザクションは、Bean メソッドの呼び出しにトランザクション・コンテキストを提供し、 Java アノテーションまたはデプロイメント記述子ファイル `ejb-jar.xml` を使用して定義されます。 Bean 管理トランザクションは、Java Transaction API (JTA) を使用して直接制御されます。CICS JTA の統合を無効にしていなければ、いずれの場合も、CICS UOW は JTA トランザクションの結果に従属したままの状態になります。コンテナー管理トランザクションには、次の 6 つの異なるトランザクション属性を指定することができます。

- Mandatory
- 必須
- RequiresNew
- Supports
- NotSupported
- Never

JTA トランザクションは、JEE 仕様に定義されている分散作業単位です。メソッドのトランザクション属性の設定によって、メソッドを実行する CICS タスクが独自の作業単位として実行されるのか、それともより広範囲の分散 JTA トランザクションの一部として実行されるのかが決まります。トランザクション属性および呼び出し側のアプリケーションが既に JTA トランザクション・コンテキストを持っているかどうかに応じて、呼び出された EJBメソッドのトランザクション・コンテキストがどのようになるかを、下の表に示します。

重要: Liberty では、アウトバウンドおよびインバウンドのトランザクションの伝搬はサポートされません。詳しくは、Liberty でのリモート・インターフェースによる Enterprise JavaBeans の使用を参照してください。

表 *10. EJB* トランザクション・サポート

トランザクション属性	**JTA** トランザクションが存在しない	**JTA** トランザクションが既に存在する	既存の **JTA** トランザクションでリモート **EJB** にアクセスする	例外の動作
Mandatory	例外 `EJBTransactionRequiredException` をスローする。	既存の JTA トランザクションを継承する。	例外 `com.ibm.websphere.csi.CSITransactionMandatoryException` をスローする。	Rollback
必須	EJB コンテナーは新しい JTA トランザクションを作成する。	既存の JTA トランザクションを継承する。	例外 `com.ibm.websphere.csi.CSITransactionRequiredException` をスローする。	Rollback
RequiresNew	EJB コンテナーは新しい JTA トランザクションを作成する。	例外 `javax.ejb.EJBException` をスローする。	EJB コンテナーは、リモート・サーバーで管理される新しい JTA トランザクションを作成する。	Rollback
Supports	JTA トランザクションなしで続ける。	既存の JTA トランザクションを継承する。	例外 `com.ibm.websphere.csi.CSITransactionSupportedException` をスローする。	JTA から呼び出された場合、ロールバックする。
NotSupported	JTA トランザクションなしで続ける。	JTA トランザクションを停止するが、CICS UOW は停止しない。	リモート・サーバーは JTA トランザクションなしで続ける。	ロールバックなし
Never	JTA トランザクションなしで続ける。	例外 `javax.ejb.EJBException` をスローする。	リモート・サーバーは JTA トランザクションなしで続ける。	ロールバックなし

重要: 「`NotSupported`」のマークが付けられているメソッドを呼び出すと、JTA トランザクションは停止されますが、CICS UOW は停止されません。このメソッドの呼び出し中に行われた CICS リソースへの変更は、元に戻すことができます。

注: トランザクション属性 `RequiresNew` は、CICS Liberty JVM サーバーでサポートされていますが、CICS UOW をネストできないという制限があります。既に

JTA トランザクションが開始された状態で「RequiresNew」のマークが付けられているメソッドを呼び出そうとすると、例外がスローされます。

さらなる Enterprise JavaBeans (EJB) フィーチャーの制限については、『Liberty: ランタイム環境での既知の問題および制約事項』を参照してください。

リモート・インターフェースを備えた Enterprise Java Bean (EJB) メソッド

リモート・インターフェースを備えた EJB メソッドには、RMI-IIOP テクノロジーを使用して、 CICS Liberty からリモート・アクセスするか、CICS Liberty でホストすることができます。 ejbRemote-3.2 フィーチャーを使用してリモート EJB サポートを有効にすることができます。

リモート EJB インターフェースを使用するときに思いに留めておく必要がある考慮事項があります。詳しくは、Liberty でのリモート・インターフェースによる Enterprise JavaBeans の使用を参照してください。

リモート・インターフェースを持つ EJB メソッドへのアクセス

1. リモート・インターフェースを使用して EJB メソッドにアクセスするアプリケーションを実行するように CICS Liberty を構成するには、以下のように、ejbRemote-3.2 フィーチャーを server.xml ファイルに追加して有効にする必要があります。

   ```
   <featureManager>
       <feature>ejbRemote-3.2</feature>
   </featureManager>
   ```

2. デプロイメント記述子 <ejb-ref> またはソース・コード・アノテーション (例えば、@EJB) で定義されたリモート EJB 参照に対して、アプリケーション・バインディング・ファイル (例えば、ibm-*-bnd.xml) を構成します。アノテーションまたはデプロイメント記述子でルックアップ名を指定する EJB 参照にはバインディングは必要ありません。以下のように、バインディング・ファイル内で、EJB の java: 名のいずれかを使用するか、corbaname:names のいずれかを使用して、EJB 参照をバインドできます。

   ```
   @EJB(name="TestBean")
       TestRemoteInterface testBean;
   ```

   バインディングは次のように定義されます:

   ```
   <ejb-ref name="TestBean" binding-name=
   "corbaname:rir:#ejb/global/TestApp/TestModule/TestBean!test.TestRemoteInterface"/>
   ```

3. スタブ・クラスを組み込むようにアプリケーション・クライアントを構成します。

リモート・インターフェースを持つ EJB メソッドのホスト

1. 他の JVM によって EJB を呼び出すことができるように、CICS Liberty でこの EJB をホストするには、以下のように、ejbRemote-3.2 フィーチャーを server.xml ファイルに追加して有効にする必要があります。

   ```
   <featureManager>
       <feature>ejbRemote-3.2</feature>
   </featureManager>
   ```

2. IIOP サーバーの構成でポートとセキュリティーの設定をカスタマイズします。詳細については、リモート EJB の IIOP-RMI トランスポートの構成を参照してください。
3. EJB アプリケーションを作成します。詳しくは、Enterprise JavaBeans (EJB) プロジェクトの作成を参照してください。
4. スタブ・クラスを生成します。 Eclipse で、EJB プロジェクトを右クリックし、「**Java EE** ツール」 > 「**EJB** クライアント **Jar** を作成します」を選択します。
5. EJB アプリケーションを EAR の一部として CICS Liberty にデプロイします。詳しくは、エンタープライズ・アプリケーション・プロジェクトの作成を参照してください。
6. Liberty `messages.log` ファイルを確認し、EJB が有効であること、および名前空間にバインドされていることを確認します。次のメッセージが表示されます。

   ```
   CNTR0167I: サーバーが、ejb.remote アプリケーションの ejb.remote.ejb.jar
   モジュールにある MyBean エンタープライズ Bean の
   ejb.remote.ejb.view.MyBeanRemote インターフェースをバインディングしています。
   バインディング・ロケーションは、
   java:global/ejb.remote/ejb.remote.ejb/MyBean!remote.ejb.view.MyBeanRemote です。
   ```

IIOP-RMI トランスポートでのリモート **EJB** の構成**:**

Internet Inter-ORB Protocol リモート・メソッド呼び出し (IIOP-RMI) トランスポートは、リモート・インターフェースを持つ EJB メソッドと通信するために CICS Liberty によって使用されます。この通信は、Common Secure Interoperability プロトコル・バージョン 2 (CSIv2) を使用して保護することができます。

IIOP-RMI は、リモート・インターフェースを持つ EJB メソッドを呼び出すためのテクノロジーとして CICS Liberty によって使用されます。 `ejbRemote-3.2` 機能を使用すると、インバウンド IIOP-RMI 呼び出しとアウトバウンド IIOP-RMI 呼び出しの両方がサポートされます。

インバウンド呼び出しでは、CICS Liberty がオブジェクト・リクエスト・ブローカー (ORB) として TCP/IP ポートで IIOP-RMI 要求を listen し、ターゲット EJB メソッドを呼び出すことができます。詳しくは、 104 ページの『インバウンド IIOP 通信の構成』を参照してください。

アウトバウンド呼び出しでは、CICS Liberty は EJB メソッドを開始するために ORB に要求を出します。アウトバウンド呼び出しは、呼び出しが行われた JVM サーバーと同じ JVM サーバーに対して行うことも、ORB として動作できる他の Java 仮想マシン (JVM) に対して行うこともできます。詳しくは、 104 ページの『アウトバウンド IIOP 通信の構成』を参照してください。

この通信は、CSIv2 を使用して保護できます。CSIv2 は、認証、委任、および特権に関する CORBA (共通オブジェクト・リクエスト・ブローカー・アーキテクチャー) の基準を満たしているテクノロジーです。 CSIv2 は、トランスポート層セキュリティー (TLS) の使用もサポートしています。詳しくは、CSIv2 での IIOP 通信保護の構成を参照してください。

詳しくは、Common Secure Interoperability バージョン 2 (CSIv2)を参照してください。

インバウンド **IIOP** 通信の構成

ejbRemote-3.2 機能を使用可能にするには、server.xml ファイルにこの機能を追加します。

```
<featureManager>
    <feature>ejbRemote-3.2</feature>
</featureManager>
```

オプションで、server.xml ファイルで IIOP エンドポイントを構成することもできます。

```
<iiopEndpoint id="defaultIiopEndpoint" host="host.example.com" iiopPort="2809" />
```

重要**:** デフォルトで、IIOP エンドポイントは localhost:2809 で listen します。デフォルトの ORB は、IIOP エンドポイント defaultIiopEndpoint を参照します。ORB でのインバウンド・セキュリティーの構成について詳しくは、CSIv2 での IIOP 通信保護の構成を参照してください。

アウトバウンド **IIOP** 通信の構成

ejbRemote-3.2 機能を使用可能にするには、 server.xml ファイルにこの機能を追加します。

```
<featureManager>
    <feature>ejbRemote-3.2</feature>
</featureManager>
```

オプションで、リモート・サーバーのネーム・サービスで ORB を構成することもできます。

```
<orb id="defaultOrb" nameService="corbaname::host.example.com:2809" />
```

重要**:** デフォルトで、ORB はローカル IIOP エンドポイント defaultIiopEndpoint を参照します。 ORB でのアウトバウンド・セキュリティーの構成について詳しくは、CSIv2 での IIOP 通信保護の構成を参照してください。

CSIv2 での **IIOP** 通信保護の構成**:**

ここでは、IIOP 通信のためにインバウンドとアウトバウンドの両方の CSIv2 セキュリティーを構成することに関連したいくつかの一般的なケースを取り上げます。

インバウンド呼び出しでは、CICS Liberty がオブジェクト・リクエスト・ブローカー (ORB) として TCP/IP ポートで IIOP-RMI 要求を listen し、ターゲット EJB メソッドを呼び出すことができます。

アウトバウンド呼び出しでは、CICS Liberty は EJB メソッドを開始するために ORB に要求を出します。アウトバウンド呼び出しは、呼び出しが行われた JVM サーバーと同じ JVM サーバーに対して行うことも、ORB として動作できる他の Java 仮想マシン (JVM) に対して行うこともできます。

以下の例では、*client* はアウトバウンド要求を出す側の JVM であり、*server* はインバウンド要求を受け取る側の JVM です。これらのいずれかまたは両方として CICS Liberty JVM サーバーを使用することができます。詳しくは、Liberty での Common Secure Interoperability Version 2 (CSIv2) の構成を参照してください。

TLS を使用するための **CSIv2** の構成

インバウンド

- サーバーの証明書を含む鍵ストアを作成します。

```
<keyStore id="iiopKeyStore" ... />
```

- 鍵ストアを参照する SSL レパートリー (SSL エレメント) を作成します。

```
<ssl id="iiopSSL" keyStoreRef="iiopKeyStore" />
```

- IIOPS ポートを使用する IIOP エンドポイントを作成します。

```
<iiopEndpoint id="defaultIiopEndpoint" host="host.example.com" iiopPort="2809">
    <iiopsOptions iiopsPort="9402" sslRef="iiopSSL" />
</iiopEndpoint>
```

重要: デフォルトで、IIOP オプション sslRef は、defaultSSLConfig SSL レパートリーを参照します。

アウトバウンド

- 鍵ストアを作成します。鍵ストアでルート証明書が信頼されるようにするための鍵を組み込むことができます。これにより、そのルート証明書によって署名されるすべての証明書が信頼されます。

```
<keystore id="iiopTrustStore" ... />
```

- 鍵ストアを参照する SSL レパートリー (SSL エレメント) を作成します。

```
<ssl id="iiopSSL" trustStoreRef="iiopTrustStore" ... />
```

- CSIv2 クライアント・ポリシーを使用する ORB を作成します。

```
<orb id="defaultOrb" nameService="corbaname::host.example.com">
    <clientPolicy.csiv2>
        <layers>
            <transportLayer sslRef="iiopSSL" />
        </layers>
    </clientPolicy.csiv2>
</orb>
```

クライアントからサーバーにユーザー **ID** を伝搬できるようにするための **CSIv2** の構成

インバウンド

- CSIv2 サーバー・ポリシーを使用する ORB を作成します。

```
<orb id="defaultOrb">
    <serverPolicy.csiv2>
        <layers>
            <attributeLayer identityAssertionEnabled="true" />
        </layers>
    </serverPolicy.csiv2>
</orb>
```

- オプションで、サーバーによって信頼されるようにする 1 つ以上の ID を指定できます。

```
<attributeLayer identityAssertionEnabled="true" trustedIdentities="MYUSER" />
```

アウトバウンド

- CSIv2 クライアント・ポリシーを使用する ORB を作成します。

```
<orb id="defaultOrb" nameService="corbaname::host.example.com:2809">
    <clientPolicy.csiv2>
        <layers>
            <attributeLayer identityAssertionEnabled="true" />
        </layers>
    </clientPolicy.csiv2>
</orb>
```

- オプションで、サーバーで許可する信頼できる ID を指定できます。

```
<attributeLayer identityAssertionEnabled="true" trustedIdentity="MYUSER"
     trustedPassword="MYPASSWD" />
```

重要: 信頼できるユーザーが、サーバーのユーザー・レジストリーに存在している必要があります。 trustedPassword は、Liberty securityUtility ツールを使用してエンコードできます。

TLS クライアント認証を使用するための **CSIv2** の構成

インバウンド

- 鍵ストアを作成します。鍵ストアでルート証明書が信頼されるようにするための鍵を組み込むことができます。これにより、そのルート証明書によって署名されるすべての証明書が信頼されます。

```
<keyStore id="iiopTrustStore" ... />
```

- 鍵ストアを参照する SSL レパートリー (SSL エレメント) を作成します。

```
<ssl id="iiopSSL" trustStoreRef="iiopTrustStore" ... />
```

- IIOPS エンドポイントを使用する IIOP エンドポイントを作成します。

```
<iiopEndpoint id="defaultIiopEndpoint" host="host.example.com" port="2809">
    <iiopsOptions iiopsPort="9402" sslRef="iiopSSL" />
</iiopEndpoint>
```

- CSIv2 サーバー・ポリシーを使用する ORB を作成します。

```
<orb id="defaultOrb">
    <serverPolicy.csiv2>
        <layers>
            <attributeLayer identityAssertionEnabled="true" ... />
            <transportLayer sslRef="iiopSSL" />
        </layers>
    </serverPolicy.csiv2>
</orb>
```

アウトバウンド

- クライアントの証明書を含む鍵ストアを作成します。

```
<keyStore id="iiopKeyStore" ... />
```

- 鍵ストアを参照する SSL レパートリー (SSL エレメント) を作成します。

```
<ssl id="iiopSSL" keyStoreRef="iiopKeyStore" />
```

- CSIv2 クライアント・ポリシーを使用する ORB を作成します。

```
<orb id="defaultOrb">
    <clientPolicy.csiv2>
        <layers>
            <attributeLayer identityAssertionEnabled="true" />
            <transportLayer sslRef="iiopSSL" />
        </layers>
    </clientPolicy.csiv2>
</orb>
```

Java Message Service (JMS)

Java Message Service (JMS) は、Java EE をベースにしたアプリケーション・コンポーネントでメッセージの作成、送信、受信、および読み取りを可能にする API です。Liberty での JMS サポートは、JMS リソース・アダプターのデプロイメントをサポートする一群の関連フィーチャーとして提供されます。

JMS は、キュー、トピック、接続、および他のリソースがサーバー構成によって作成および管理される管理モードで実行できます。これには、JMS 接続ファクトリー、キュー、トピック、およびアクティベーション仕様の構成が含まれます。代わりに、すべてのリソースをアプリケーションの一部として手動で構成する非管理モードで実行することもできます。Liberty 組み込み JMS メッセージング・プロバイダーは管理対象であるため、すべてのリソースは server.xml 構成の一部としてセットアップされます。

JMS 仕様

Liberty JVM サーバーでサポートされる JMS 仕様レベルは、JMS 2.0 サポートです。JMS 2.0 サポート (`jms-2.0`) では、2.0 仕様レベルでの Java Message Service API を使用してメッセージング・システムにアクセスするリソース・アダプターを構成できます。

JMS クライアント

以下の Liberty 機能によって、Liberty JVM サーバーでは各種 JMS クライアント・プロバイダーがサポートされています。

- WebSphere MQ JMS 2.0 クライアント (`wmqJmsClient-2.0`)。この WebSphere MQ JMS クライアント機能によって、JMS 2.0 または 1.1 クライアント・アプリケーションは、リモート MQ サーバーとの間でメッセージを送受信できるようになります。
- WebSphere Application Server JMS 2.0 クライアント (`wasJmsClient-2.0`)。この WebSphere Application Server クライアント機能によって、JMS 2.0 または 1.1 クライアント・アプリケーションは、`wasJmsServer` フィーチャーによって有効にされたメッセージング・エンジンとの間でメッセージを送受信できるようになります。
- JCA 1.6 仕様に準拠するそれ以外の JMS リソース・アダプターも、汎用 JCA リソース・アダプター・リンクを使用することによって Liberty で使用できます (JCA 構成エレメントの概要を参照)。

JMS プロバイダー

CICS TS 内の Liberty は、以下の使用をサポートします。

- Liberty 組み込み JMS メッセージング・プロバイダー.
  - WebSphere メッセージング・サーバー (`wasJmsServer-1.0`)。JMS サーバー機能によって、組み込み JMS メッセージング・プロバイダーを Liberty 内でホストできるようになります。これにより、JMS サーバーを別個にインストールして構成する必要がなくなります (単一 Liberty サーバーでの JMS メッセージングの有効化を参照)。このサーバーは、CICS 内部のまたは z/OS や他の分散プラットフォームでホストされる Liberty サーバーの、別個の Liberty インスタンスでホストすることもできます (2 つの Liberty サーバー

間の JMS メッセージングの有効化を参照)。WebSphere JMS メッセージング・クライアント・コンポーネントを、WebSphere Application Server で実行される SIBUS 経由で JMS と対話するように構成することもできます (Liberty と WebSphere Application Server traditional との間のインターオペラビリティーの使用可能化を参照)。

  - WebSphere メッセージング・セキュリティー (`wasJmsSecurity-1.0`)。この JMS セキュリティー機能は、組み込み JMS メッセージング・プロバイダー・クライアントおよびサーバー・コンポーネントにセキュリティー・サポートを提供します。この JMS セキュリティー機能を `cicsts:security-1.0` フィーチャーと併用することで、組み込みJMS メッセージング・サーバーに対して要求を認証する際に、セキュリティー・レジストリーからどのユーザーを選択して接続ファクトリーで使用するかを指定できます。許可について詳しくは、メッセージング・エンジンに接続するユーザーの許可を参照してください。

- JMS アプリケーションがバインディング・モードまたはクライアント・モードのいずれかの転送を使用して接続する場合、CICS 標準モードの Liberty JVM サーバー内での IBM MQ への JMS アクセス。
- JMS アプリケーションがクライアント・モードの転送を使用して接続する場合、CICS 統合モードの Liberty JVM サーバー内での IBM MQ への JMS アクセス。
- JCA 1.6 仕様に準拠したサード・パーティー JMS リソース・アダプター。

Java Management Extensions API (JMX)

Java Management Extensions API (JMX) は、リソースのモニターと管理に使用します。

JMX は、広く受け入れられている実装を使用してアプリケーションの情報を公開する手段を提供する、Java フレームワークおよび API です。公開された情報を読み取るように、JConsole などの各種のツールを構成できます。情報を公開するための手段としては、管理 Bean (MBean) が使用されます。MBean は、public コンストラクターを備えた非静的 Java クラスです。Bean の get および set メソッドは属性として公開されますが、それ以外のすべてのメソッドは操作として公開されます。

ローカルまたはリモート・マシンから Liberty JVM サーバーの JMX に接続して、これらの MBean の属性と操作を表示できます。ローカルに接続するには、`server.xml` に `localConnector-1.0` 機能を追加して、同じ JVM サーバー内から接続できるようにする必要があります。`restConnector-1.0` フィーチャーを `server.xml` に追加すると、RESTful インターフェースを接続手段として使用できるようになるため、JMX へのリモート・アクセスが可能になります。

WebSphere MBean を使用したアプリケーションのモニター

1. まず、MBeanServer の参照を取得する必要があります。この例では、**`JvmStats`** MBean を検索し、**`findMBeanServer`** メソッドを使用して、この MBean の登録先サーバーを確認します。正しい MBeanServer オブジェクトの参照から、MBean の参照を取得し、その MBean が公開する属性からデータを取得できます。この例では、**`JvmStats`** MBean の **`UpTime`** 属性を検索します。

```
// Create an ObjectName object for the MBean that we're looking for.
ObjectName beanObjName = null;
beanObjName = new ObjectName("WebSphere:type=JvmStats");

// Obtain the full list of MBeanServers.
java.util.List servers = MBeanServerFactory.findMBeanServer(null);
MBeanServer mbs = null;

// Iterate through our list of MBeanServers and attempt to find the one we want.
for (int i = 0; i < servers.size(); i++)
{
    // Check if the MBean domain matches what we're looking for.
    mbs = (MBeanServer)servers.get(i);

    if (mbs.isRegistered(beanObjName))
    {
        Object attributeObj = mbs.getAttribute(beanObjName, "UpTime");
        System.out.println("UpTime of JVM is: " + attributeObj + ".");
    }
}
```

Liberty での JMX へのリモート接続

Liberty JVM サーバーで JMX にリモート接続するには、 SSL 接続と Java Platform, Enterprise Edition (JEE) ロール許可を使用する必要があります。それによって、クライアント・コードは JMXServiceURL を使用してリモート MBean の参照を取得します。

1. REST コネクターでアクセスされるすべての JMX MBeans は、単一の JEE ロール「管理者」によって保護されています。このロールへのアクセスを提供するには、server.xml を編集して、管理者ロールに認証済みユーザーを追加します。

   ```
   <administrator-role>
   <user>myuserid</user>
   <group>group1</group>
   </administrator-role>
   ```

   JEE ロールの使用について詳しくは、SAF ロール・マッピングを使用した許可を参照してください。

2. リモートの RESTful JMX クライアントは SSL を使用して、Liberty JVM サーバーにアクセスする必要があります。Liberty JVM サーバー用の SSL サポートを構成する方法については、RACF を使用した Liberty JVM サーバー用の SSL (TLS) の構成のトピックを参照してください。さらに、JMX クライアントは、restConnector クライアント・サイド JAR ファイルと、サーバーの署名証明書が含まれる SSL クライアント鍵ストアにもアクセスする必要があります。CICS WLP インストールの一部として提供される restConnector.jar は、&USSHOME;/wlp/clients で使用可能になっています。

3. クライアント・サイドのコードで、JMXServiceURL オブジェクトを作成する必要があります。これによって、リモート MBeanServerConnection オブジェクトの参照を取得できます。以下の例を見ると、<host> と <httpsPort> が、サーバーのものと一致しています。

```
JMXServiceURL url = new JMXServiceURL("service:jmx:rest://<host>:<httpsPort>/IBMJMXConnectorREST");
JMXConnector jmxConnector = JMXConnectorFactory.connect(url, environment);
MBeanServerConnection mbsc = jmxConnector.getMBeanServerConnection();
```

4. 接続の取得に成功すると、MBeanServerConnection オブジェクトは MBeanServer オブジェクトからローカル接続する場合と同じ機能およびメソッド一式を提供します。

WebSphere で提供される MBeans について詳しくは、Liberty プロファイル: 提供されている MBean のリストを参照してください。

Java Authorization Contract for Containers (JACC)

Liberty は、デフォルトの許可に加えて、Java Authorization Contract for Containers (JACC) 仕様を基にした許可をサポートしています。Liberty でセキュリティーが有効になっていると、JACC プロバイダーが指定されない限り、デフォルトの許可が使用されます。

このタスクについて

JACC により、アプリケーション・サーバーでの許可の管理に、サード・パーティーのセキュリティー・プロバイダーを使用できます。デフォルトの許可では特別な設定は必要なく、デフォルトの許可エンジンがすべての許可の決定を行います。ただし、JACC プロバイダーが Liberty で使用されるように構成され、設定された場合、すべてのエンタープライズ Bean および Web の許可の決定は、JACC プロバイダーが代行します。JACC は、アプリケーション・サーバーと許可ポリシー・モジュールとの間のセキュリティー契約を定義します。これらの契約で、許可プロバイダーのインストール方法、構成方法、およびアクセス判断での使用方法が指定されます。`jacc-1.5` フィーチャーを Liberty サーバーに追加するには、サード・パーティーの JACC プロバイダーを追加しますが、このプロバイダーは Liberty には含まれていません。

Liberty サーバーで提供されている `com.ibm.wsspi.security.authorization.jacc.ProviderService` インターフェースを実装することによって、Java EE アプリケーションのカスタム許可決定を行う JACC プロバイダーを開発できます。JACC 仕様である JSR 115 は、許可プロバイダー用のインターフェースを定義しています。Liberty サーバーで、JACC プロバイダーをユーザー・フィーチャーとしてパッケージする必要があります。そのフィーチャーは `com.ibm.wsspi.security.authorization.jacc.ProviderService` インターフェースを実装する必要があります。

手順

1. OSGi バンドル・プロジェクトを作成して Java クラスを開発します。 プロジェクトにコンパイル・エラーがある可能性があります。これらのエラーを修正するには、`javax.security.jacc` および `com.ibm.wsspi.security.authorization.jacc` の 2 つのパッケージをインポートする必要があります。

   ファイル `MANIFEST.MF` を編集して、欠落しているパッケージをインポートします。

```
Manifest-Version: 1.0
Service-Component: OSGI-INF/myjaccExampleComponent.xml,
Bundle-ManifestVersion: 2
Bundle-Name: com.example.myjaac.osgiBundle
Bundle-SymbolicName: com.example.myjaac.osgiBundle
```

```
Bundle-Version: 1.0.0
Bundle-RequiredExecutionEnvironment: JavaSE-1.7
Import-Package: com.ibm.wsspi.security.authorization.jacc;version="1.0.0",
javax.security.jacc;version="1.5.0"
```

サービス・コンポーネント XML の例 (myjaccExampleComponent.xml) を以下に示します。

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0" immediate="true"
 name="TestPolicyServiceProvider">
    <implementation class="com.example.myjaac.osgiBundle.TestPolicyServiceProvider"/>
    <property name="javax.security.jacc.policy.provider" type="String" value=""/>
    <property name="javax.security.jacc.PolicyConfigurationFactory.provider" type="String" value=""/>
    <service>
        <provide interface="com.ibm.wsspi.security.authorization.jacc.ProviderService"/>
    </service>
</scr:component>
```

2. Liberty フィーチャー・プロジェクトを作成し、以前の OSGi バンドルをユーザーの Liberty フィーチャー (フィーチャー・マニフェスト・ファイルの Subsystem-Content の下) に追加します。
3. フィーチャー・マニフェストを変更して、必要な OSGi サブシステム・コンテンツ com.ibm.ws.javaee.jacc.1.5; version="[1,1.0.200)"; location:="dev/api/spec/" を追加します。

```
Subsystem-ManifestVersion: 1.0
IBM-Feature-Version: 2
IBM-ShortName: jacc15CICSLiberty-1.0
Subsystem-SymbolicName: com.example.myjaac.libertyFeature;visibility:=public
Subsystem-Version: 1.0.0
Subsystem-Type: osgi.subsystem.feature
Subsystem-Content: com.example.myjaac.osgiBundle;version="1.0.0",
 com.ibm.ws.javaee.jacc.1.5;version="[1,1.0.200)";location:="dev/api/spec/"
Manifest-Version: 1.0
```

サブシステム・コンテンツをもう 1 つ追加する必要がある場合は、コンテンツを入力する前に**スペース**を少なくとも 1 つ追加する必要があります。 **スペース**を追加しないと、CICS は java.lang.IllegalArgumentException を返します。

4. Liberty フィーチャー・プロジェクトを Liberty フィーチャー (ESA) ファイルとしてエクスポートします。
5. ESA ファイルを zFS に FTP でファイル転送します。
6. installUtility コマンドを使用して ESA ファイルをインストールします。

   ```
   ./wlpenv installUtility install myFeature.esa
   ```

7. jacc-1.5 フィーチャー、および JACC プロバイダーが含まれた ESA ファイルをユーザー・フィーチャーとして server.xml に追加します。

   ```
   <feature>jacc-1.5</feature>
   <feature>usr:jacc15CICSLiberty-1.0</feature>
   ```

Java Authentication Service Provider Interface for Containers (JASPIC)

Java Authentication Service Provider Interface for Containers (JASPIC) 仕様では、サービス・プロバイダー・インターフェース (SPI) を定義しています。メッセージ認証メカニズムを実装する認証プロバイダーを、クライアントまたはサーバーのメッセージ処理コンテナーまたはランタイムに統合できます。

このタスクについて

JASPIC インターフェースを介して統合された認証プロバイダーは、呼び出し側コンテナーによって提供されたネットワーク・メッセージを操作します。プロバイダーがメッセージを変換することにより、受信側コンテナーによるメッセージ送信元の認証と、メッセージ送信側によるメッセージ受信側の認証が可能になります。着信メッセージが認証されてその呼び出し側コンテナー (メッセージ認証の結果として確立された ID) に返されます。

JSR 196 は、標準 SPI を定義し、認証モジュールを Java EE コンテナーに組み込む方法を標準化しています。メッセージ処理モデル、およびクライアントとサーバー上の多数の対話ポイントの詳細が提供されます。互換性のある Web コンテナーは、これらのポイントで SPI を使用して、対応するメッセージ・セキュリティー処理をサーバー認証モジュール (SAM) に委任します。

Liberty は、`jaspic-1.1` に指定されたサーブレット・コンテナーに対応するサード・パーティー認証プロバイダーの使用をサポートします。サーブレット・コンテナーにはインターフェースが定義されており、セキュリティー・ランタイム環境は Web コンテナーと連携して、それらのインターフェースを使用します。それらのインターフェースによって、アプリケーションが Web 要求を処理する前後に認証モジュールが始動されます。JASPIC モジュールを使用する認証は、セキュリティー構成で JASPIC が有効にされている場合のみ使用されます。

手順

1. OSGi バンドル・プロジェクトを作成して Java クラスを開発します。 プロジェクトにコンパイル・エラーがある可能性があります。これらのエラーを修正するには、`javax.security.auth.message` および `com.ibm.wsspi.security.jaspi` の 2 つのパッケージをインポートする必要があります。ターゲット・プラットフォームを編集して、欠落している JAR を、`com.ibm.ws.security.jaspic` リスト (`<cics_install>/wlp/lib` ディレクトリー内) および `com.ibm.ws.javaee.jaspic.<version_number>` リスト (`<cics_install>/wlp/dev/api/spec` ディレクトリー内) に追加する必要があります。これらを FTP で開発システムに転送し、ビルド・パスに追加します。

   ファイル `MANIFEST.MF` を編集して、欠落しているパッケージをインポートします。

   ```
   Manifest-Version: 1.0
   Bundle-ManifestVersion: 2
   Bundle-Name: com.example.myjaspic.osgiBundle
   Bundle-SymbolicName: com.example.myjaspic.osgiBundle
   Bundle-Version: 1.0.0
   Bundle-RequiredExecutionEnvironment: JavaSE-1.7
   Import-Package: com.ibm.wsspi.security.jaspi;version="1.0.13",
   javax.security.auth.message;version="1.0.0",
   javax.security.auth.message.callback;version="1.0.0",
   javax.security.auth.message.config;version="1.0.0",
   javax.security.auth.message.module;version="1.0.0",
   javax.servlet;version="2.7.0",
   javax.servlet.http;version="2.7.0"
   Service-Component: myjaspicExampleComponent.xml
   ```

   サービス・コンポーネント XML の例 (`myjaspicExampleComponent.xml`) を以下に示します。

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0" name="com.example.myjaspic.osgiBundle">
    <implementation class="com.example.myjaspic.osgiBundle.TestJASPICProviderService"/>
    <service>
        <provide interface="com.ibm.wsspi.security.jaspi.ProviderService"/>
    </service>
</scr:component>
```

2. Liberty フィーチャー・プロジェクトを作成し、以前の OSGi バンドルをユーザーの Liberty フィーチャー (フィーチャー・マニフェスト・ファイルの `Subsystem-Content` の下) に追加します。
3. フィーチャー・マニフェストを編集して、必要な OSGi サブシステム・コンテンツ `com.ibm.websphere.appserver.jaspic-1.1; type="osgi.subsystem.feature"` を追加します。

```
Subsystem-ManifestVersion: 1.0
IBM-Feature-Version: 2
IBM-ShortName: jaspic11CICSLiberty-1.0
Subsystem-SymbolicName: com.example.myjaspic.libertyFeature;visibility:=public
Subsystem-Version: 1.0.0.201611081617
Subsystem-Type: osgi.subsystem.feature
Subsystem-Content: com.example.myjaspic.osgiBundle;version="1.0.0",
  com.ibm.websphere.appserver.jaspic-1.1;type="osgi.subsystem.feature",
  com.ibm.websphere.appserver.servlet-3.0;ibm.tolerates:="3.1";type="osgi.subsystem.feature"
Manifest-Version: 1.0
```

サブシステム・コンテンツをもう 1 つ追加する必要がある場合は、コンテンツを入力する前に**スペース**を少なくとも 1 つ追加する必要があります。 **スペース**を追加しないと、CICS は `java.lang.IllegalArgumentException` を返します。

4. Liberty フィーチャー・プロジェクトを Liberty フィーチャー (ESA) ファイルとしてエクスポートします。
5. ESA ファイルを zFS に FTP でファイル転送します。
6. `installUtility` を使用して ESA ファイルをインストールします。

   ```
   ./wlpenv installUtility install myFeature.esa
   ```

7. `jaspic-1.1` フィーチャー、および JASPIC プロバイダーが含まれた ESA ファイルをユーザー・フィーチャーとして `server.xml` に追加します。

   ```
   <feature>jaspic-1.1</feature>
   <feature>usr:jaspic11CICSLiberty-1.0</feature>
   ```

Java EE コネクター・アーキテクチャー (JCA)

JCA は、CICS などのエンタープライズ情報システムを JEE プラットフォームに接続します。

JCA は、JEE アプリケーション・サーバーによって提供されるセキュリティー資格情報管理、接続プーリング、およびトランザクション管理のサービスの品質をサポートします。JCA を使用すると、これらのサービスの品質がアプリケーションではなく JEE アプリケーション・サーバーによって管理されるようになります。このことは、プログラマーがビジネス・コードの作成に専念でき、サービスの品質を意識する必要がないことを意味します。サービス品質の提供および構成のガイダンスについては、ご使用の JEE アプリケーション・サーバーの資料を参照してください。JCA は、共通クライアント・インターフェース (CCI) と呼ばれるプログラミング・インターフェースを定義します。このインターフェースにわずかな変更を加えるだけで、どのエンタープライズ情報システムとも通信できます。

プログラミング・インターフェース・モデル

CCI を使用するアプリケーションは、あらゆるエンタープライズ情報システムのための共通の構造を持ちます。 JCA は、CICS などのエンタープライズ情報システム (EIS) を JEE プラットフォームに接続します。これらの接続オブジェクトによって、JEE アプリケーション・サーバーは、リソース・アダプターのセキュリティー、トランザクション・コンテキスト、および接続プールを管理することができます。アプリケーションを開始するには、まず接続ファクトリーにアクセスする必要があります。そこから接続を取得することができます。この接続のプロパティーは、ConnectionSpec オブジェクトによりオーバーライドすることができます。接続が取得された後で、特定の要求を出すために接続から対話を作成することができます。接続の場合と同様に、対話も InteractionSpec クラスによって設定されるカスタム・プロパティーを持つことができます。対話を行うためには、execute() メソッドを呼び出し、record オブジェクトを使用してデータを保持します。例:

```
          ConnectionFactory cf = <Lookup from JNDI namespace>
Connection c = cf.getConnection(ConnectionSpec);
Interaction i = c.createInteraction();
InteractionSpec is = newInteractionSpec();
i.execute(spec, input, output);
i.close();
c.close();
```

この例は、次のシーケンスを示しています。

1. ConnectionFactory オブジェクトを使用して、接続オブジェクトを作成する。
2. 接続オブジェクトを使用して、対話オブジェクトを作成する。
3. Interaction オブジェクトを使用して、エンタープライズ情報システムに対するコマンドを実行する。
4. 対話と接続を閉じる。

JEE アプリケーション・サーバーを使用する場合、接続ファクトリーは、サーバーの管理インターフェースを使用して構成することで作成します。Liberty サーバーでは、これは `server.xml` 構成で定義します。接続ファクトリーを作成したら、エンタープライズ・アプリケーションは、JNDI (Java Naming Directory Interface) でそれを検索して、アクセスできます。このタイプの環境は管理対象環境と呼ばれ、JEE アプリケーション・サーバーが接続のサービス品質を管理することを可能にします。管理対象環境について詳しくは、JEE アプリケーション・サーバーの資料を参照してください。

レコード・オブジェクト

レコード・オブジェクトは、EIS との間で受け渡しされるデータを表わすために使用されます。これらのレコードを生成するには、アプリケーション開発ツールを使用することをお勧めします。Rational Application Developer に備わっている J2C ツールを使用すると、COBOL コピーブックなどの特定のネイティブ言語構造から Record インターフェースの実装を作成でき、Java と Java 以外のデータ型の間のデータ・マーシャルも標準でサポートされます。

リソース・アダプターのサンプル

リソース・アダプターの基本的なサンプルをインストールし、そのリソース・アダプターが提供するリソースのインスタンスを構成することができます。基本的な

JCA リソース・アダプターの構成およびデプロイを参照してください。

Common Client Interface

CCI が提供する標準インターフェースでは、開発者が汎用プログラミング・スタイルを使用して、各 EIS に対応するリソース・アダプターを介して任意の数の EIS と通信できます。CCI は、Java Database Connectivity (JDBC) で使用されるクライアント・インターフェースをモデルとして非常によく似た形に設計されており、Connection (接続) と Interaction (対話) についての考え方は JDBC と同様です。

JCA ローカル ECI リソース・アダプターの使用

CICS TS で提供される JCA ローカル ECI リソース・アダプターは、ローカル CICS プログラムを呼び出します。これは、CICS Transaction Gateway ECI リソース・アダプターを使用するアプリケーションを CICS Liberty にマイグレーションするために最適化されたパスです。このセクションは、統合モードの Liberty にのみ適用されます。

JCA ローカル ECI リソース・アダプターを使用することで、CICS プログラムに接続し、COMMAREA またはチャネルとコンテナーのいずれかでデータを渡します。リソース・アダプターは、CICS Liberty フィーチャーで提供されます。

注: JCA ローカル ECI リソース・アダプターと CICS Transaction Gateway ECI リソース・アダプターを同じ Liberty JVM サーバー内で使用することはできません。

表 1 に、CICS 用語に対応する JCA オブジェクトを示します。

表 *11. CICS* 用語と対応する *JCA* オブジェクト

CICS 用語	**JCA** オブジェクト: プロパティー
異常終了コード	CICSTxnAbendException
COMMAREA	レコード
チャネル	ECIChannelRecord
データ・タイプ BIT のコンテナー	byte[]
データ・タイプ CHAR のコンテナー	String
プログラム名	ECIInteractionSpec:FunctionName
トランザクション	ECIInteractionSpec:TPNName

詳細については、 167 ページの『JCA ローカル ECI サポート』を参照してください。

JCA ローカル ECI リソース・アダプターの構成:

JCA 仕様の定義にしたがって、接続ファクトリーを使用して JCA ローカル ECI リソース・アダプターを構成できます。

JCA ローカル ECI の使用を開始するには、`server.xml` の featureManager エレメントにフィーチャー `cicsts:jcaLocalEci-1.0` を追加します。

```
<featureManager>
<feature>cicsts:jcaLocalEci-1.0</feature>
</featureManager>
```

JCA ローカル ECI は、JNDI 名 **eis/defaultCICSConnectionFactory** にバインドされたデフォルトの接続ファクトリー **defaultCICSConnectionFactory** を提供します。オプションで、異なる JNDI 名が必要な場合は、次のようなプロパティー・サブエレメントを使用して追加の接続ファクトリーを構成することもできます。

```
<connectionFactory id="localEci" jndiName="eis/ECI">
<properties.com.ibm.cics.wlp.jca.local.eci/>
</connectionFactory>
```

ヒント: プロパティー・エレメントには属性は必要ありません。

JCA ECI アプリケーションを **Liberty JVM** サーバーに移植する:

JCA ローカル ECI リソース・アダプター・サポートを使用することで、JCA アプリケーションを簡単に Liberty JVM サーバーに移植できます。

移植

CICS Transaction Gateway ECI リソース・アダプターを使用する既存の JCA アプリケーションを、スタンドアロン JEE アプリケーション・サーバーから CICS Liberty JVM サーバーに移植するには、以下の手順を使用します。

1. `cicsts:jcaLocalEci-1.0` フィーチャーおよび `webProfile-6.0` フィーチャーを `server.xml` ファイルに追加します。

   以下に例を示します。

   ```
   <featureManager>
   ...
   <feature>cicsts:jcaLocalEci-1.0</feature>
   <feature>webProfile-6.0</feature>
   ...
   </featureManager>
   ```

2. リソースを更新して接続ファクトリーの JNDI 名を **eis/defaultCICSConnectionFactory** に変更するか、**connectionFactory** および **properties.com.ibm.cics.wlp.jca.local.eci** を `server.xml` に追加するかのいずれかの方法を使用できます。
3. アプリケーションを CICS にデプロイします (CICS バンドル内の Java EE アプリケーションの Liberty JVM サーバーへのデプロイを参照)。

ECI リソース・アダプターの制限されたフィーチャーを使用するアプリケーションの場合は、アプリケーションのコードを変更してそれらのサポートされていないフィーチャーを削除する必要があります。詳しくは、JCA ローカル ECI リソース・アダプターの制限を参照してください。

ローカル **ECI** リソース・アダプターを使用した **CICS** 内のプログラムへのリンク:

JCA ローカル ECI リソース・アダプターを使用して CICS 内のプログラムを実行するには、ECIInteraction クラスの execute() メソッドを使用します。

このタスクについて

このタスクでは、JCA ローカル ECI リソース・アダプターを使用して 、CICS プログラムを COMMAREA に渡し、JCA レコードを使用して実行する方法をアプリ

ケーション開発者に紹介します。Record インターフェースを拡張して CICS COMMAREA を表現する方法について詳しくは、を参照してください。チャネルとコンテナーを使用する CICS プログラムにリンクする方法について詳しくは、JCA ローカル ECI リソース・アダプターをチャネルおよびコンテナーと共に使用するを参照してください。

手順

1. JNDI を使用して、`eis/defaultCICSConnectionFactory` という名前の ConnectionFactory オブジェクトを検索します。
2. ConnectionFactory から Connection オブジェクトを取得します。
3. Connection から Interaction オブジェクトを取得します。
4. ECIInteractionSpec オブジェクトを新規作成します。
5. ECIInteractionSpec で set メソッドを使用して、プログラム名や COMMAREA の長さなど、実行に関わるプロパティーを設定します。
6. 入力データを入れるレコード・オブジェクトを作成して (COMMAREA/チャネルのトピックを参照)、データを取り込みます。
7. 出力データを入れるレコード・オブジェクトを作成します。
8. Interaction で execute メソッドを呼び出して、ECIInteractionSpec と 2 つの Record オブジェクトを渡します。
9. 出力したレコードからデータを読み取ります。

```
package com.ibm.cics.server.examples.wlp;

      import java.io.ByteArrayInputStream;
      import java.io.ByteArrayOutputStream;
      import java.io.IOException;
      import java.io.InputStream;
      import java.io.OutputStream;

      import javax.annotation.Resource;
      import javax.resource.cci.Connection;
      import javax.resource.cci.ConnectionFactory;
      import javax.resource.cci.Interaction;
      import javax.resource.cci.Record;
      import javax.resource.cci.Streamable;
      import javax.servlet.ServletException;
      import javax.servlet.annotation.WebServlet;
      import javax.servlet.http.HttpServlet;
      import javax.servlet.http.HttpServletRequest;
      import javax.servlet.http.HttpServletResponse;

      import com.ibm.connector2.cics.ECIInteractionSpec;

      /**
      * Servlet implementation class JCAServlet
      */
      @WebServlet("/JCAServlet")
      public class JCAServlet extends HttpServlet
      {
      private static final long serialVersionUID = 4283052088313275418L;

      // 1. Use JNDI to look up the connection factory
      @Resource(lookup = "eis/defaultCICSConnectionFactory")
      private ConnectionFactory cf;

      protected void doGet(HttpServletRequest request, HttpServletResponse
      response)
      throws ServletException, IOException
```

```
{
try
{
// 2. Get the connection object from the connection factory
Connection conn = cf.getConnection();

// 3. Get an interaction object from the connection
Interaction interaction = conn.createInteraction();

// 4. Create a new ECIInteractionSpec
ECIInteractionSpec is = new ECIInteractionSpec();

// 5. Use the set methods on ECIInteractionSpec
// to set the properties of execution.
// Change these properties to suit the target program
is.setCommareaLength(20);
is.setFunctionName("PROGNAME");
is.setInteractionVerb(ECIInteractionSpec.SYNC_SEND_RECEIVE);

// 6. Create a record object to contain the input data and populate
データ
// Change the contents to suit the data required by the program
RecordImpl in = new RecordImpl();
byte[] commarea = "COMMAREA contents".getBytes();
ByteArrayInputStream inStream = new ByteArrayInputStream(commarea);
in.read(inStream);

// 7. Create a record object to contain the output data
RecordImpl out = new RecordImpl();

// 8. Call the execute method on the interaction
interaction.execute(is, in, out);

// 9. Read the data from the output record
ByteArrayOutputStream outStream = new ByteArrayOutputStream();
out.write(outStream);
commarea = outStream.toByteArray();
}
catch (Exception e)
{
// Handle any exceptions by wrapping them into an IOException
throw new IOException(e);
}
}

// A simple class which extends Record and Streamable representing a
commarea.
public class RecordImpl implements Streamable, Record
{
private static final long serialVersionUID = -947604396867020977L;

private String contents = new String("");

@Override
public void read(InputStream is)
{
try
{
int total = is.available();
byte[] bytes = null;
if (total > 0)
{
bytes = new byte[total];
is.read(bytes);
}
// Convert the bytes to a string.
contents = new String(bytes);
```

```
}
catch (Exception e)
{
// Log the exception
e.printStackTrace();
}
}

@Override
public void write(OutputStream os)
{
try
{
// Output the string as bytes
os.write(contents.getBytes());
}
catch (Exception e)
{
// Log the exception
e.printStackTrace();
}
}

@Override
public String getRecordName()
{
// Required by Record, unused in this sample
return "";
}
@Override
public void setRecordName(String newName)
{
// Required by Record, unused in this sample
}
@Override
public void setRecordShortDescription(String newDesc)
{
// Required by Record, unused in this sample
}
@Override
public String getRecordShortDescription()
{
// Required by Record, unused in this sample
return "";
}
@Override
public Object clone() throws CloneNotSupportedException
{
// Required by Record, unused in this sample
return super.clone();
}
}
}
```

タスクの結果

ECI リソース・アダプターを使用した CICS 内のプログラムへのリンクの作成が成功しました。

***JCA** ローカル **ECI** リソース・アダプターをチャネルおよびコンテナーと共に使用する:*

JCA ローカル ECI リソース・アダプターと共にチャネルおよびコンテナーを使用するには、入出力レコードが ECIChannelRecord のインスタンスである必要があります。

ECIChannelRecord が ECIInteraction の execute() メソッドに渡されると、そのメソッドは ECIChannelRecord そのものを使用してチャネルを作成し、ECIChannelRecord 内の項目をコンテナーに変換してから、それを CICS に渡します。

この例は、ECI ChannelRecord で put() メソッドおよび get() メソッドを使用して JCA ローカル・リソース・アダプター用の入出力レコードを作成する方法を示しています。

```
ECIChannelRecord in = new
     ECIChannelRecord("CHANNELNAME");
     byte[] bitData = "Container with BIT data".getBytes();
     String charData = "Container with CHAR data";
     in.put("BITCONTAINER", bitData);
     in.put("CHARCONTAINER", charData);
     ECIChannelRecord out = new ECIChannelRecord("CHANNELNAME");

     interaction.execute(is, in, out);

     bitData = (byte[]) out.get("BITCONTAINER");
     charData = (String) out.get("CHARCONTAINER");
```

入力の種類に応じて、BIT コンテナーおよび CHAR コンテナーが次のように作成されます。

- 入力データが byte[] 型である場合、または Streamable インターフェースを実装するオブジェクトである場合には、BIT コンテナーが作成されます。コード・ページ変換は行われません。
- 入力データが String 型である場合には、CHAR コンテナーが作成されます。ストリング・データは Unicode でエンコードされ、コンテナーのエンコード方式に変換されます。 **EXEC CICS GET CONTAINER** によってこのコンテナーから読み取られるデータは、コンテナーを使用したコード・ページ変換に従って変換されます。

ECIChannelRecord を作成するときに、名前は有効な CICS チャネル名でなければなりません。作成された後、getRecordName() メソッドがチャネルの名前を取得します。 ECIChannelRecord にコンテナーを追加するとき、コンテナー名は有効な CICS コンテナー名でなければなりません。作成された後、KeySet() メソッドがすべてのコンテナーの名前を取得します。

***JCA** ローカル **ECI** リソース・アダプターを **COMMAREA** と共に使用する:*

JCA ローカル ECI リソース・アダプターと共に COMMAREA を使用するには、入出力レコードが、`javax.resource.cci.Record` および `javax.resource.cci.Streamable` を実装するクラスのインスタンスである必要があります。

この例は、Streamable インターフェースで read() メソッドおよび write() メソッドを使用してローカル ECI リソース・アダプター用の入出力レコードを作成する方法を示しています。

```
RecordImpl in = new RecordImpl();
     byte[] commarea = "COMMAREA contents".getBytes();
     ByteArrayInputStream inStream = new ByteArrayInputStream(commarea);
     in.read(inStream);
     RecordImpl out = new RecordImpl();

     interaction.execute(is, in, out);

     ByteArrayOutputStream outStream = new ByteArrayOutputStream();
     out.write(outStream);
     commarea = outStream.toByteArray();
```

出力レコードからバイト配列を取得するには、**java.io.ByteArrayOutputStream** オブジェクトを使って Streamable インターフェースで write メソッドを使用します。 **ByteArrayOutputStream** 上の toByteArray() メソッドは、COMMAREA からの出力データをバイト配列形式で提供します。

特定の JEE コンポーネントの機能を増強するために、コンストラクターを使用してレコード内容を設定できるようにする Record インターフェースの実装を作成することもできます。この方法により、例で使用した **java.io.ByteArrayInputStream** の使用を避けることができます。

Rational Application Developer に備わっている J2C ツールを使用すると、COBOL コピーブックなどの特定のネイティブ言語構造から Record インターフェースの実装を作成でき、Java と Java 以外のデータ型の間のデータ・マーシャルも標準でサポートされます。

JCA による作業単位の管理**:**

CICS ローカル ECI リソース・アダプターを使用する際には、CICS Liberty JVM サーバーによってトランザクション管理機能が提供されます。

CICS ローカル ECI リソース・アダプターを使用する他の CICS プログラムの呼び出しは、CICS 作業単位 (UOW) 管理に統合されます。これにより、syncpoint コマンドまたは JTA トランザクションを介して UOW を制御することができます。

リモート CICS 領域でのプログラムの呼び出しを実行すると、ミラー・トランザクションを使用した DPL 呼び出しになります。 Java トランザクション・コンテキストが使用されている場合、このミラー・タスク UOW は呼び出し側 UOW によって調整されます。この場合、呼び出される側のプログラムは DPL コマンド・サブセットに制限されるため、syncpoint 呼び出しを発行できません。呼び出し側プログラムに JTA トランザクション・コンテキストが含まれない場合は、SYNCONRETURN オプションを使用してミラー・タスク UOW が呼び出されます。このシナリオでは、呼び出されるプログラムの UOW が呼び出し側プログラムによって調整されないため、呼び出されるプログラムは syncpoint コマンドを発行できます。

詳しくは、分散プログラム・リンクのプログラミングに関する考慮事項、 73 ページの『作業単位 (UOW) サービス』、および 94 ページの『Java Transaction API (JTA)』を参照してください。

JCA ローカル ECI リソース・アダプターのトレースの有効化:

JCA ローカル ECI リソース・アダプターのトレースの仕組みを詳しく説明します。リソース・アダプターを使用するアプリケーションの問題を解決する際には、トレースを有効にすると便利な場合があります。

- JCA ローカル ECI リソース・アダプターのトレースは、SJ ドメイン・トレース・レベル 4 (SJ = 4 または SJ = ALL) によって有効になります。
- リソース・アダプターからのトレースは、コンポーネント ID `com.ibm.cics.wlp.jca.local.eci.adapter` を使って zFS で JVM サーバー・トレース出力に含まれます。

JCA ローカル ECI リソース・アダプターの制限:

CICS Transaction Gateway ECI リソース・アダプターで使用可能な一部の API 呼び出しは、CICS TS JCA ローカル ECI リソース・アダプターによってサポートされていません。

制限されたメソッド

以下の API 呼び出しは、JCA ローカル ECI リソース・アダプターによってサポートされていません。

- ECIInteractionSpec クラスの setExecuteTimeout()、getExecuteTimeout()、setReplyLength()、getReplyLength()、setTranName()、getTranName() メソッド
- CICSConnectionSpec クラスの setPassword()、getPassword()、setUserid()、getUserid()、addPropertyChangeListener()、removePropertyChangeListener()、firePropertyChange メソッド
- ECIConnection クラスの getLocalTransaction() メソッド
- ECIChannelRecord クラスの values() メソッド
- CICSUserInputException
- コンストラクター ECIConnectionSpec(String username, String password)。ECIConnectionSpec() が代わりに追加されました。
- コンストラクター ECIInteractionSpec(int verb, int timeout, String prog, int commLen, int repLen)。ECIInteractionSpec(int verb, String prog, int commLen) が代わりに追加されました。

これらの呼び出しは、Web アプリケーションの開発に IBM CICS SDK for Java を使用している場合はサポートされません。既存の ECI JCA アプリケーションを Liberty JVM サーバーに移植可能にするために、これらのメソッドは引き続き機能しますが、トランザクション、タイムアウト、応答の長さ、およびトランザクション名の設定は影響を及ぼしません。ECIInteractionSpec.setTPNName() でトランザクション ID を設定すると、リモート・プログラム (DPL) にリンクするときに、指定されたトランザクションのみが使用されます。ローカル・プログラムへのリンクは、引き続き現在のトランザクションを使用します。

非管理対象環境

JCA ローカル ECI は、(server.xml の構成により作成された) 管理接続ファクトリーのみをサポートします。 ManagedConnectionFactory のインスタンスを使用して作成された非管理対象接続はサポートされません。

例外処理

CICS Transaction Gateway ECI リソース・アダプターと CICS TS JCA ローカル ECI リソース・アダプターでは、例外処理が少し異なる可能性があります。CICS のエラーは、JCICS API の場合と同様に、ECI ローカル・リソース・アダプターに CICSException として伝搬されます。リソース・アダプターは、これらの例外を ResourceException でラップします。CICS の障害を特定できるように、CICSException が例外の原因として設定され、java.lang.Throwable の getCause() メソッドを使用してアクセスできます。

非同期呼び出し

非同期呼び出しは、JCA ローカル ECI リソース・アダプターでは完全な非同期ではありません。SYNC_SEND 対話 verb を使用した呼び出しは、プログラムが完了するまでブロックし、その後 SYNC_RECEIVE 対話 verb を使用して、同じ ECIInteraction を使い、後続の呼び出しによって結果を収集できます。

サポートされない **CICS Transaction Gateway** の機能

次の CICS Transaction Gateway の機能は、CICS TS ローカル ECI リソース・アダプターではサポートされません。

- CICS Transaction Gateway サーバーへのリモート接続
- ID 伝搬
- クロス・コンポーネント・トレース (XCT)
- ユーザー出口のモニタリング要求
- リソース・アダプターからのトレースは CICS TS によって制御され、CICSLogTraceLevels の使用はサポートされていません。

Java 向け CICS リモート開発機能

Java 向け CICS リモート開発機能は、開発者のワークステーションで稼働する Liberty で使用できる ECI リソース・アダプターを提供します。この機能により、開発者は JCA API を使用して CICS TS 内のプログラムを呼び出す Java アプリケーションを迅速にテストおよびデバッグできます。準備ができたら、アプリケーションにさらに変更を加えることなく、アプリケーションを CICS 内で稼働する Liberty にデプロイできます。

この機能は CICS 領域に接続するために、TCPIPSERVICE リソースを使用して定義された IP 相互接続性 (IPIC) 接続を使用します。CICS TS 内のプログラムとの間で送受信されるデータの問題を識別するために、トレース機能を使用できるようになっています。

IPIC 接続の構成:

CICS 領域で Java アプリケーションをテストする前に、IPIC 接続が使用可能になっている必要があります。CICS システム・プログラマーに連絡して、以下の詳細を使用して Liberty プロファイルからの IPIC 要求を受け入れる TCP/IP サービスを要請してください。

このタスクについて

以下の手順では、CICS システム・プログラマーが CICS 内に TCP/IP サービスを定義して、IPIC 接続用のサンプル・ユーザー・プログラムをインストールするためのステップを説明します。

手順

1. **TCPIPSERVICE** リソースに次の属性を定義して、CICS に IPIC サポートをインストールします。

表 *12. TCPIPSERVICE* リソースの属性

TCPIPSERVICE リソース属性	必要な値
URM	DFHISAIP
ポート番号	n
状況	OPEN
Protocol	IPIC
トランザクション	CISS
Backlog	0
Socketclose	いいえ

2. **CEMT INQUIRE TCPIPSERVICE(JCA)** コマンドを実行して、**TCPIPSERVICE** がサービス中であることを確認します。
3. オプション: IPIC 接続をテストするためのサンプル・プログラムをインストールします。
   a. CICSTransaction Gateway Software Development Kit (SDK) をダウンロードし (まだコピーを持っていない場合)、アーカイブ・ファイルを展開します。
   b. `cicsprograms/ec01.cpp` メンバーを見つけて、z/OS 上の COBOL ソース・データにコピーします。
   c. `EC01` サンプル・プログラムをコンパイルし、生成されたモジュールを、CICS がアクセスできるロード・ライブラリーにコピーします。
   d. `autoinstall` プログラムが有効になっていない場合は、`EC01` のプログラム定義を作成してインストールします。
   e. **CECI LINK PROG(EC01) COMMAREA(' ')** を実行して `EC01` プログラムをテストします。 **RESPONSE** が `NORMAL` であることを確認します。

タスクの結果

これで、IPIC サポートを CICS 領域で利用できるようになりました。

ローカル **Java** テスト環境のセットアップ:

CICS 領域で Java アプリケーションをテストする前に、必要なツールがインストールされていることを確認するともに、ローカル作業環境を構成する必要があります。

このタスクについて

CICS 領域で Java アプリケーションをテストできるようにローカル作業環境を作成するには、次の手順を実行します。

手順

1. WebSphere Developer Tools (WDT) が含まれる Eclipse IDE for Java EE Developers をダウンロードしてインストールします。次に、ローカル Liberty プロファイル・サーバー・インスタンスをインストールし、Hello World JavaServer Pages (JSP) を作成します。インストールしたサーバー上に Hello World Web アプリケーションをデプロイすることによってテストします。 詳しくは、「Getting started with WebSphere Developer Tools for Eclipse and Liberty」を参照してください。
2. Liberty リポジトリーから JCA リモート ECI リソース・アダプターをインストールします。以下のように **installUtility** コマンドを使用することで、リポジトリーからフィーチャーをインストールできます。

   ```
   <liberty_install>/bin/installUtility install
   --acceptLicense jcaRemoteEci-1.0
   ```

3. usr:jcaRemoteEci-1.0、localConnector-1.0、および webProfile-6.0 フィーチャーを server.xml ファイルに追加します。 例えば、Eclipse で「**WebSphere Application Server Liberty Profile**」プロジェクトを展開してから、「サーバー **(servers)**」を展開します。server.xml を編集するために、「**defaultServer**」をダブルクリックします。「ソース」タブをクリックし、以下のフィーチャーを追加します。

   ```
   <featureManager>
   ...
   <feature>usr:jcaRemoteEci-1.0</feature>
   <feature>localConnector-1.0</feature>
   <feature>webProfile-6.0</feature>
   ...
   </featureManager>
   ```

4. **connectionFactory** および **properties.com.ibm.cics.wlp.jca.remote.eci** を server.xml に追加します。

   connectionFactory jndiName は、アプリケーションが接続を作成するために使用します。**properties.com.ibm.cics.wlp.jca.remote.eci** は、JCA リモート ECI リソース・アダプターを構成する際に使用するため、少なくとも **serverName** で、TCPIPSERVICE リソースで定義されている IPIC 接続のホスト名とポート番号を指定する必要があります。

   注: 追加のパラメーターを指定しなければならない場合があります。例えば、Secure Sockets Layer (SSL) を使用するには、ユーザー ID とパスワードを指定する必要があります。表 1 に、使用可能なパラメーターをリストします。表 2 に、JCA リモート ECI リソース・アダプターでサポートされていない ECI

リソース・アダプター・デプロイメント・パラメーターをリストします。詳しくは、ECIリソース・アダプター・デプロイメント・パラメーターを参照してください。

```
<server>
...
<connectionFactory id="com.ibm.cics.wlp.jca.local.eci" jndiName="eis/ECI">
<properties.com.ibm.cics.wlp.jca.remote.eci serverName="tcp://
hostname
:
port"/>
</connectionFactory>
...
</server>
```

表 1 に、サポートされている JCA リモート ECI リソース・アダプター・プロパティーを示します。

表 *13.* サポートされている *JCA* リモート *ECI* リソース・アダプター・プロパティー

JCA オブジェクト**:** プロパティー	注
applid	
applidQualifier	このプロパティーは必須です。
cipherSuites	
ipicHeartbeatInterva l	
ipicSendSessions	このプロパティーはデフォルトで 5 に設定されます。
keyRingClass	
keyRingPassword	
password	
socketConnectTimeout	
serverName	このプロパティーは必須です。
traceLevel	
traceRequest	
userName	

表 2 に、JCA リモート ECI リソース・アダプターでサポートされていない CICS Transaction Gateway ECI リソース・アダプター・デプロイメント・パラメーターを示します。

表 *14.* サポートされていない *JCA* リモート *ECI* リソース・アダプター・プロパティー

JCA オブジェクト**:** プロパティー
interceptPlugin
portNumber
tPNName
tranName

サンプル **Java EE JCAServlet** アプリケーションのテスト**:**

サンプル Java EE JCAServlet アプリケーションを追加した後、この Java EE アプリケーションが CICS 内のサンプル・プログラムを呼び出せることを確認します。

このタスクについて

次の手順を実行して、Java EE JCAServlet アプリケーションを追加します。その後、Java EE JCAServlet アプリケーションが CICS 内の EC01 サンプル・プログラムを呼び出せることを確認します。

手順

1. Hello World Web アプリケーション内に JCAServlet クラスを作成します。「**Hello World**」プロジェクトを展開した後、「**Java** リソース **(Java Resource)**」を展開します。「新規」を右クリックし、「サーブレット」を選択します。
   - **Java package** には、`com.ibm.ctg.samples.liberty` と入力します。
   - **Class name** には、`JCAServlet` と入力します。

   その後、「終了」をクリックします。
2. `JCAServlet.java` を編集し、すべてのコードを GitHub の CICS サンプル `JCAServlet.java` に置き換えます。 詳しくは、JCAServlet.java を参照してください。
3. 「**Hello World**」プロジェクトを展開した後、「**Java** リソース **(Java Resources)**」 > 「**src**」 > 「**com.ibm.ctg.samples.liberty**」と展開します。「**JCAServlet.java**」アプリケーションを右クリックし、「実行 **(Run As)**」 > 「サーバーで実行 **(Run on server)**」と選択します。
4. Liberty サーバーが始動され、Liberty サーバー・コンソールに URL を示すメッセージが表示されます。この URL をクリックすると Java EE アプリケーションを実行できます。以下に、表示されるメッセージの一例を示します。

   ```
   [AUDIT ] CWWKT0016I: Web application available
   (default_host):
   http://localhost:9080/GenappCustomerSearchWeb/
   ```

タスクの結果

これで、ローカル Liberty サーバーで Java EE アプリケーションをテストし、デバッグできるようになりました。

ローカル **Liberty** プロファイル内でのトレース機能の構成**:**

ローカル Liberty プロファイル内で Java Web アプリケーションをトレースする前に、ローカル作業環境を構成する必要があります。

このタスクについて

次の手順を実行して、ローカル Liberty プロファイル内でトレース機能を構成します。

手順

トレースを有効にするには、`server.xml` ファイル内の接続ファクトリーに `traceRequests="ON"` パラメーターを追加します。
`traceRequests="ON"` を指定した状態でアプリケーション要求を送信すると、Eclipse コンソールにアプリケーションが送信する要求と CICS から受信する応答が表示さ

れます。
以下の例に、CICS に送信された要求と、CICS から受信した応答を示します。

```
Starting DataFlowsMonitor log stream at
Thu Apr 07 14:21:54 BST 2016[00000000001]:
com.ibm.ctg.monitoring.DataFlowsMonitor:eventFired called with
event = RequestEntry
FlowType = EciSynconreturn Fully qualified APPLID = No APPLID
CtgCorrelator =
1Program = EC01Server =
TCP://WINMVS2C.HURSLEY.IBM.COM:27723PayLoad = COMMAREA is 20
bytes00000000 00000000 00000000 00000000 00000000
'??????????????????????'

[00000000001]:
com.ibm.ctg.monitoring.DataFlowsMonitor:eventFired called with
event = ResponseExit
FlowType = EciSynconreturn Fully qualified APPLID = No APPLID
CtgCorrelator =
1OriginData - Transaction Group ID = 1B114040
40404040 40402EF0 F0F0F0F0 F0F0F2D0 8F53F115 220100Program = EC01Server =
TCP://WINMVS2C.HURSLEY.IBM.COM:27723PayLoad = COMMAREA is 20
bytesF0F761F0 F461F1F6 40F1F47A F2F17AF5 F4000000
'??a??a??@??z??z?????'CtgReturnCode = 0CicsReturnCode = 0
```

タスクの結果

これで、問題を識別できるよう、アプリケーションをトレースすることが可能になります。

セキュア **SSL** 接続の構成:

SSL を使用して、JCA リモート ECI リソース・アダプターから CICS への IPIC 接続を保護できます。

このタスクについて

セキュア SSL 接続を構成するには、次の手順を実行します。

このセットアップを完了すると、MVS とローカル・クライアントの両方からエクスポートされた信頼できる証明書が SSL に提供されます。認証には MVS ユーザー ID とパスワードも必要になります。

手順

1. CICS RACF® 環境をセットアップします。 詳しくは、 Configuring SSL server authentication on the CICS server を参照してください。
2. クライアント・セキュリティーをセットアップします。 詳しくは、Configuring SSL server authentication on the clientを参照してください。
3. クライアント認証を構成します。 詳しくは、Configuring SSL client authenticationを参照してください。
4. CICS 上で IPIC 接続を構成します。 詳しくは、 Configuring the IPIC connection on CICS を参照してください。
5. `sever.xml` を変更して、ステップ 2 で作成したローカル **KeyRingClass** を使用し、ユーザー ID とパスワードを送信するようにします。

```
<connectionFactory id="com.ibm.cics.wlp.jca.local.eci"
jndiName="eis/ECI">
<properties.com.ibm.cics.wlp.jca.remote.eci
```

```
serverName="ssl://hostname:port"
keyRingClass="C:¥Users¥IBM_ADMIN¥Documents¥CICS¥JCA¥ctgclientkeyring.jks"
keyRingPassword="password"
userName="user_ID"
password="*******"
applid="JCASSL"
applidQualifier="ABCDEFGH"
/>
</connectionFactory>
```

タスクの結果

JCA リモート ECI リソース・アダプターが、SSL と、`server.xml` に指定された鍵リング、ユーザー ID、およびパスワードを使用して、CICS への要求を保護するようになります。

MicroProfile によるマイクロサービスの開発

Eclipse MicroProfile では、Enterprise Java 環境でマイクロサービス・アプリケーションを開発するためのプログラミング・モデルを定義します。これは、Eclipse Foundation の下にあるオープン・ソース・プロジェクトであり、マイクロサービスを Enterprise Java コミュニティーに提供します。MicroProfile は Liberty でサポートされます。

MicroProfile では、弾力性がありセキュアで、モニターが容易なマイクロサービスを作成するための多数の仕様が定義されます。

表 *15. Eclipse MicroProfile 1.2* に含まれる内容

仕様	説明
JSR 346: Contexts and Dependency Injection for Java EE 1.1	CDI では、Enterprise Java ランタイムでのオブジェクトの注入とライフサイクルを管理する一連のサービスが定義されます。
JSR 339: JAX-RS 2.0: The Java API for RESTful Web Services	JAX-RS は、Java API for RESTful Web Services です。
JSR 353: Java API for JSON Processing	JSON-P は、JSON を処理するための Java API です。
Eclipse MicroProfile Config 1.1	Config は、アプリケーション構成を管理するための Java API および SPI です。
Eclipse MicroProfile Fault Tolerance 1.0	Fault Tolerance では、外部サービスの呼び出し時に障害を処理するための戦略を提供します。
Eclipse MicroProfile Health Check 1.0	Health Check を使用すると、コンポーネントはその稼働状況を広範囲のシステムに報告できます。
Eclipse MicroProfile Health Metrics 1.0	Health Metrics では、アプリケーションでモニター・データを公開するための統一された方法を提供します。
Eclipse MicroProfile JWT Propagation 1.0	JWT Propagation では、Java EE の役割ベースのアクセス制御 (RBAC) による認証および許可に JSON Web Token (JWT) を使用できます。

既知の制約事項

- CDI は MicroProfile API で幅広く使用されていますが、Liberty では、エンタープライズ・バンドル・アーカイブ (EBA) にパッケージ化された OSGi Web アプリケーションでの CDI はサポートされません。代わりに、MicroProfile を使用するアプリケーションを Web アプリケーション・アーカイブ (WAR) またはエンタープライズ・アプリケーション・アーカイブ (EAR) にパッケージ化します。
- MicroProfile Fault Tolerance 1.0 は、他のサービスに対する呼び出しを管理するように設計されています。トランザクション・コンテキスト内のリソースに対する更新を管理するようには設計されていません。CICS リソースは、`@Bulkhead`、`@CircuitBreaker`、`@Fallback`、`@Timeout`、または `@Retry` の注釈が付けられたメソッドでは更新できません。CICS では、JTA が使用されている場合でも、例外の発生時にこれらの更新がリカバリーされる保証はありません。
- `mpJwt-1.0` フィーチャーが Liberty JVM サーバーの `server.xml` で有効になっている場合、すべての認証を JWT ベアラー・トークンを使用して行う必要があります。他の形式の認証を使用するには、別個の Liberty JVM サーバーを使用する必要があります。

CICS Liberty JVM サーバーのサービス・アーキテクチャー

モノリシック・アーキテクチャー

モノリシック・アーキテクチャーでは、単一の単位でアプリケーションを実装します。内部的にはロジックをモジュール形式にすることができますが、外部的には、アプリケーションは全体的に使用可能であるか、まったく使用不可であるかのいずれかになります。モノリスはマイクロサービスと比較して良好に機能し、セキュリティーおよびトランザクション・コンテキストを管理する場合、それほど複雑ではありません。モノリスのスケーリングでは、アプリケーション全体のインスタンスを追加する必要があり、各パーツを個別にスケーリングすることはできません。

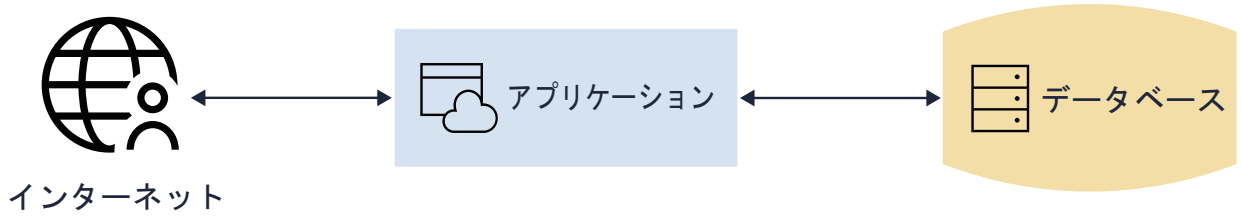


図 4. モノリシック・アーキテクチャー

バッキング・サービス

バッキング・サービスを使用すると、バックエンド・データおよびプログラムをメイン・アプリケーションから分離できます。データとプログラムは、別々のアプリケーションにすることによって、プラットフォーム非依存の通信方式 (HTTP、ソケット、メッセージ・キューなど) を使用して呼び出されます。これらのソースと通信するためのすべてのロジックを保持するメイン・アプリケーションの代わりに、これらのサービスで一部の作業が行われます。

CICS では、z/OS Connect を使用し、CICS プログラムを REST API を介してバッキング・サービスとして公開します。この例では、アプリケーションは JDBC を使用してデータベースと通信します。E メールの送信には SMTP が使用され、z/OS Connect を介した CICS プログラムの呼び出しには HTTP が使用されます。

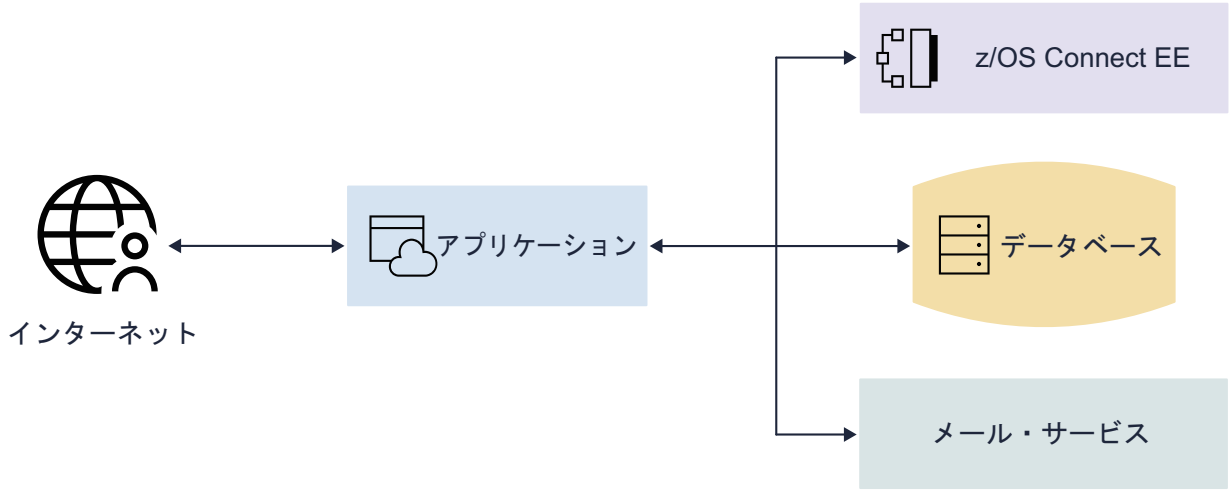


図 5. バッキング・サービス

ホスト・サービス

サービスは、メイン・アプリケーションをさまざまなコンポーネントからさらに分離するために、CICS でホストされます。バッキング・サービスと同様に、追加機能は CICS Web サービスを介して CICS で公開されるか、またはサーブレット、JAX-RS、JAX-WS などのテクノロジーを使用して、アプリケーションにより CICS Liberty で公開されます。

JAX-RS は RESTful Web サービスを作成するための一般的なテクノロジーであり、JAX-WS は、リモート手続き呼び出し (RPC) 指向の Web サービスを作成するために使用されます。

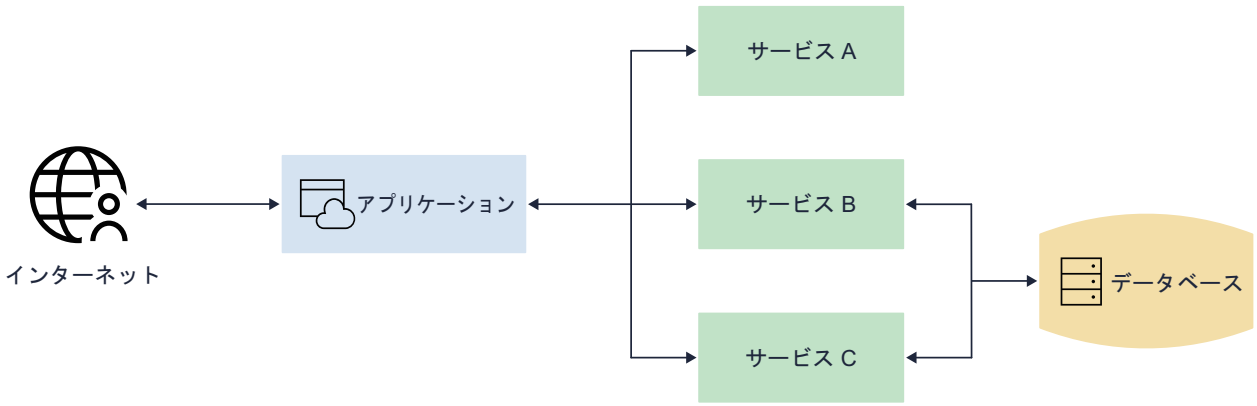


図 6. ホスト・サービス

注: REST と RPC はどちらも同様に、マイクロサービスでの通信に有効なオプションです。REST はリソース管理に重点を置いています。RPC はアクションに重点を置いています。マイクロサービス・アーキテクチャーでは、REST、RPC、またはその他のテクノロジーは必須ではありません。

マイクロサービス

完全なマイクロサービス・アーキテクチャーは、分離されたサービスが相互接続された Web であり、単一の中心点はありませんが、専用のエントリー・ポイントは存在する場合があります。サービスは、必要に応じて相互に通信できます。マイクロサービスのスケーリングでは、スケーリングを必要とするパーツのインスタンスを追加する必要があります。マイクロサービスは、モノリスよりも障害に対して弾力性があります。

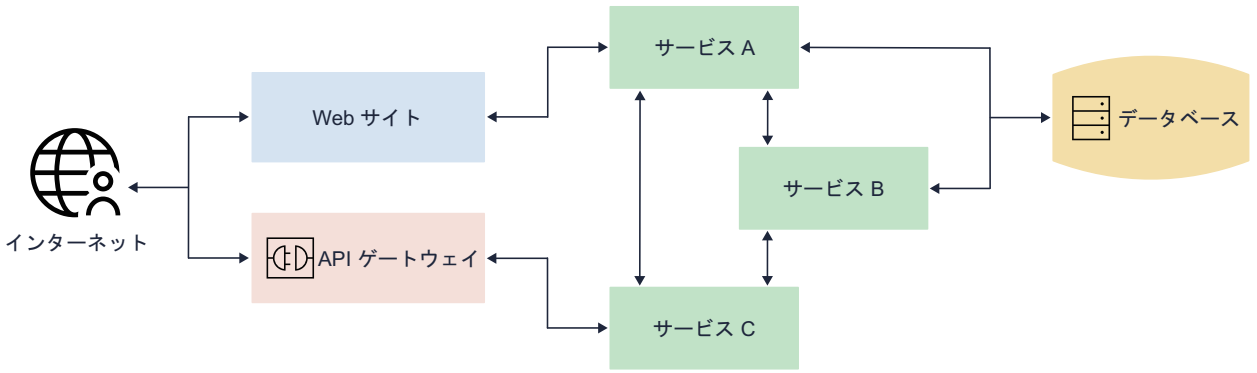


図 7. マイクロサービス

CICS でのサービスのスケーリング

CICS では、領域のトポロジーおよびセットアップに応じて、サービスを複数の方法でスケーリングできます。通常、マイクロサービスは単一のコンテナーに分離されます。CICS では、サービスまたはサービスのセットは、領域または JVM サーバー内で分離できます。スケーリングを行うには、同じサービスまたはサービスのセットをホストする複数の CICS 領域を実行します。また、スレッド数を増加して JVM

サーバーをスケーリングすることもできます。

マイクロサービスの保護

可能な場合、マイクロサービスではパブリック・ネットワークをオフにしておく必要があります。API ゲートウェイを使用して、マイクロサービスへのアクセスを制御できます。MicroProfile では、マイクロサービス・エンドポイントの役割ベースのアクセス制御 (RBAC) に Open ID Connect (OIDC) ベースの JSON Web Token (JWT) を使用するための方式を提供します。セキュリティー・トークンにより、さまざまなサービスにわたるユーザー ID の単純で相互運用可能な伝搬が可能となります。

MicroProfile JWT Authentication 1.0 では、JWT ベアラー・トークンに基づいてユーザーを認証および許可する機能を提供します。トークンをサービス・コードに注入して、ID をマイクロサービス・ネットワーク全体に伝搬するために使用できます。JWT の伝搬は、アウトバウンド要求の許可 HTTP ヘッダーにベアラー・トークンとして JWT を組み込むことにより、手動で行うことができます。または、次の例のように `server.xml` の `webTarget` エレメントに `authnToken` を構成して、Liberty により JWT を自動的に伝搬することもできます。

```
<webTarget uri="http://microservice.example.ibm.com/protected/*" authnToken="mpjwt" />
```

重要: JWT ID は、自動的にはユーザー・レジストリーにマップされず、CICS タスク・ユーザー ID に伝搬されません。ID のマッピングを有効にするには、`mapToUserRegistry="true"` 構成属性を `server.xml` の `<mpJwt>` エレメントに追加します。
Liberty での MicroProfile JWT 認証の構成について詳しくは、MicroProfile JSON Web トークンの構成を参照してください。

マイクロサービスでのデータ整合性

マイクロサービスで分散トランザクションを使用することは容易ではありません。代わりに、saga パターンなどの代替のトランザクション戦略が使用されます。この場合、イベントはサービスで更新された後に公開されます。例えば、サービス A とサービス B にどちらも必須の更新が含まれる場合、以下の順序で行われます。

1. A の更新が保留状態になります。
2. A から B へメッセージが送信されます。
3. B の更新が完了状態になります。
4. B から A へメッセージが送信されます。
5. A の更新が完了状態になります。

マイクロサービスを使用する場合

マイクロサービスを適用するのに最適なシチュエーションは、アプリケーションをより小さな分離されたサービスに分解可能な場合です。マイクロサービスを使用すると、制御されたスケーリング、独立したデプロイメント、およびより自律型の開発が可能となります。マイクロサービスのアーキテクチャーにより、デプロイメントおよびデータ整合性においては特に、複雑性が増す可能性があります。HTTP などのプロトコルを介した通信では、メモリー内の呼び出しと比較して、パフォーマンス・コストが増加します。コンポーネントを個別にスケーリングできるようにす

ることで、コンポーネントの障害に対する弾力性が向上します。マイクロサービス・アーキテクチャーを管理する際には、正常でないサービスの診断を支援するためにモニター・ソリューションがさらに重要となります。

Liberty Web サーバー・プラグイン

Web サーバー・プラグインを使用することにより、サポートされる Web サーバーから 1 つ以上のアプリケーション・サーバーに HTTP 要求を転送することが可能になります。

Web サーバー・プラグインを使用する主な理由が 3 つあります。

- 静的コンテンツを提供する Web サーバーの統合を可能にします。
- HTTPS 使用時に Web サーバーの SSL エンドポイントの終了を可能にします。
- 一群の Liberty サーバー間で HTTP 要求のロード・バランシングとフェイルオーバーを可能にします。

Liberty サーバーで `plugin-cfg.xml` ファイルを生成し、そのファイルを Web サーバーのホスト・マシンにコピーすることによって、Web サーバー・プラグインは構成されます。プラグインはインバウンド要求を受け取り、その要求をこのファイルに含まれている構成データと照らしてチェックし、着信 HTTP 要求を構成済み Liberty サーバーの URI とホストに転送します。

Liberty プロファイル・サーバーで `plugin-cfg.xml` を生成する手順で使用する `generatePluginConfig` 操作は、Libertyが提供する `com.ibm.ws.jmx.mbeans.generatePluginConfig` MBean で公開されます。この JMX MBean をリモートから呼び出すには、 IBM Java SDK で提供されている JConsole ユーティリティーと Liberty サーバーの `restConnector-1.0` 機能を組み合わせて使用するか、または、MBean の必要な操作を呼び出すカスタム JMX アプリケーションを作成します。CICS Liberty サーバーでの JMX の使用について詳しくは、 108 ページの『Java Management Extensions API (JMX)』を参照してください。

Web サーバー・プラグインのセットアップの詳細な情報は、WAS Knowledge Center にあります。Web サーバーへのプラグイン構成の追加を参照してください。

Liberty フィーチャー

CICS では、WebSphere Application Server Liberty からのフィーチャーがサポートされます。これにより、Java EE アプリケーションを Liberty JVM サーバーにデプロイできます。

表 2 から 10 のすべてのフィーチャーは、CICS 統合モードの Liberty に関連しています。これらのフィーチャーは、特に断りのない限り、CICS 標準モードの Liberty でも制限なくサポートされます。表 11 では、Liberty フィーチャーを CICS のサービスの品質に統合するための一連の CICS フィーチャーを示します。

Java EE 6 および Java EE 7 からのフィーチャーは、同時に使用しないでください。`server.xml` の編集については、サーバー構成を参照してください。

注: CICS 標準モードの Liberty は Java EE 8 もサポートします。どのフィーチャーが含まれるかについて詳しくは、Liberty での Java EE 8を参照してください。

表 16. *Liberty* フィーチャー (アルファベット順)

フィーチャー **A-D**	フィーチャー **E-Jd**	フィーチャー **Jm-MpJ**	フィーチャー **MpM-Z**
appClientSupport-1.0	ejb-3.2	jms-1.1	mpMetrics-1.0
appSecurity-1.0	ejbHome-3.2	jmsMdb-3.1	mongodb-2.0
appSecurity-2.0	ejbLite-3.1	jndi-1.0	monitor-1.0
batch-1.0	ejbLite-3.2	jpa-2.0	oauth-2.0
batchManagement-1.0	ejbPersistentTimer-3.2	jpa-2.1	openidConnectClient-1.0
beanValidation-1.0 for JEE7	ejbRemote-3.2	jsf-2.0	openidConnectServer-1.0
beanValidation-1.0 for JEE6	el-3.0	jsf-2.2	osgiConsole-1.0
beanValidation-1.1 for JEE7	j2eeManagement-1.1	json-1.0	osgi.jpa-1.0
beanValidation-1.1 for JEE6	jacc-1.5	jsonp-1.0	restConnector-1.0
blueprint-1.0	jaspic-1.1	jsp-2.2	servlet-3.0
cdi-1.0	javaMail-1.5	jsp-2.3	servlet-3.1
cdi-1.2	javaee-7.0	jta-1.1	sessionDatabase-1.0
cicsts:core-1.0	jaxb-2.2 (JEE7 が対象)	jta-1.2	ssl-1.0
cicsts:defaultApp-1.0	jaxb-2.2 (JEE6 が対象)	jwt-1.0	wab-1.0
cicsts:distributedIdentity-1.0	jaxrs-1.1	ldapRegistry-3.0	wasJmsClient-1.1
cicsts:jcaLocalEci-1.0	jaxrs-2.0	localConnector-1.0	wasJmsClient-2.0
cicsts:jdbc-1.0	jaxrsClient-2.0	managedBeans-1.0	wasJmsSecurity-1.0
cicsts:link-1.0	jaxws-2.2 (JEE7 が対象)	mdb-3.1	wasJmsServer-1.0
cicsts:security-1.0	jaxws-2.2 (JEE6 が対象)	mdb-3.2	webCache-1.0
cicsts:standard-1.0	jca-1.6	microProfile-1.0	webProfile-6.0
cicsts:zosConnect-1.0	jca-1.7	microProfile-1.2	webProfile-7.0
cicsts:zosConnect-2.0	jcaInboundSecurity-1.0 for JEE7	mpConfig-1.1	websocket-1.0
concurrent-1.0	jcaInboundSecurity-1.0 for JEE6	mpFaultTolerance-1.0	websocket-1.1
distributedMap-1.0	jdbc-4.0	mpHealth-1.0	wmqJmsClient-2.0
	jdbc-4.1	mpJwt-1.0	zosTransaction-1.0

CICS Liberty フィーチャー

表 17. *Java EE 7 Web Profile* 用にサポートされる *Liberty* フィーチャー

Liberty フィーチャー	**Liberty** フィーチャーの説明	**CICS** でのフィーチャーの使用
beanValidation-1.0	JavaBeans を検証するための、アノテーション・ベースのモデルを提供します。	
beanValidation-1.1	JavaBeans を検証するための、アノテーション・ベースのモデルを提供します。	

表 17. *Java EE 7 Web Profile* 用にサポートされる *Liberty* フィーチャー (続き)

Liberty フィーチャー	**Liberty** フィーチャーの説明	**CICS** でのフィーチャーの使用
cdi-1.2	EJB や Managed Bean などのコンポーネントを、JSP や EJB といった他のコンポーネントに注入するためのメカニズムを提供します。	
ejbLite-3.2	EJB 仕様の EJB Lite サブセットに記載されている Enterprise JavaBeans のサポートを有効にします。	**制約事項:** トランザクション属性「`NotSupported`」は、Liberty JVM サーバーではサポートされません。
el-3.0	Enterprise Language (EL) 3.0 仕様のサポートを可能にします。	
jaxrs-2.0	Liberty での Java API for RESTful Web Services (JAX-RS) のサポートを提供します。	
jaxrsClient-2.0	Java Client API for JAX-RS 2.0 のサポートを有効にします。	jaxrsClient-2.0 フィーチャーは、jaxrs-2.0 によって有効になります。JAX-RS 2.0 クライアントの構成.
jdbc-4.1	アプリケーションからデータベースにアクセスするためのデータ・ソースの構成を使用可能にします。	**注:** DB2 JCC ドライバー内の jdbc-4.0 実装と jdbc-4.1 実装を同時に使用することはできません。
jndi-1.0	Liberty のサーバー構成における、単一の Java Naming and Directory Interface (JNDI) エントリー定義のサポートを提供します。	
jpa-2.1	アプリケーション管理およびコンテナー管理の JPA を使用したアプリケーションのサポートを有効にします。	
jsf-2.2	JavaServer Faces (JSF) フレームワークを使用する Web アプリケーションのサポートを提供します。	
jsonp-1.0	JavaScript Object Notation を処理する Java API の定義をサポートします。これには、JSON の構文解析、生成、変換、および照会機能のサポートが含まれます。	
jsp-2.3	サーブレットおよび JavaServer Pages (JSP) アプリケーションのサポートを有効にします。	80 ページの『Java EE アプリケーションと Liberty アプリケーション』
managedBeans-1.0	コンテナーによって管理される、さまざまな種類の Java EE コンポーネントのための共通基盤を提供します。Managed Bean で利用できる共通サービスには、リソース・インジェクション、ライフサイクル管理、インターセプターの使用などがあります。	

表 17. *Java EE 7 Web Profile* 用にサポートされる *Liberty* フィーチャー (続き)

Liberty フィーチャー	**Liberty** フィーチャーの説明	**CICS** でのフィーチャーの使用
servlet-3.1	Java サーブレット仕様に書き込まれる HTTP サーブレットのサポートを提供します。	80 ページの『Java EE アプリケーションと Liberty アプリケーション』 **制約事項:** サーブレット・ログインおよびログアウト API を使用すると、ユーザー ID が CICS タスクに伝搬されません。
webProfile-7.0	Java EE 7 Web プロファイルをサポートするのに必要な Liberty フィーチャーの便利な組み合わせを提供します。	
websocket-1.0	Web ブラウザーまたはクライアント・アプリケーションと Web サーバー・アプリケーションが単一の全二重接続を使用して通信できるようにします。	
websocket-1.1	Web ブラウザーまたはクライアント・アプリケーションと Web サーバー・アプリケーションが単一の全二重接続を使用して通信できるようにします。	

表 18. *Java EE 7 Full Platform* 用にサポートされる *Liberty* フィーチャー

Liberty フィーチャー	**Liberty** フィーチャーの説明	**CICS** でのフィーチャーの使用
appClientSupport-1.0	Liberty サーバーによるクライアント・モジュールの処理およびリモート・クライアント・コンテナーのサポートを可能にします。	**ヒント:** アプリケーション・クライアント・モジュールは、クライアントとサーバーの両方で実行されます。クライアントは、アプリケーションのクライアント固有のロジックを実行します。コードの他の部分はサーバー上のクライアント・コンテナー内で実行され、サーバーで実行されているビジネス・ロジックからクライアントにデータを送信します。詳細については、アプリケーション・クライアントの準備と実行を参照してください。
batch-1.0	JSR-352 で定義されている Java Batch 1.0 API のサポートを可能にします。このフィーチャーは、エンタープライズ・バンドル・アーカイブ (EBA) にパッケージ化された Java バッチ・アプリケーションはサポートしていません。	**制約事項:** DB2 JDBC タイプ 2 の接続は、バッチ・パーシスタンスではサポートされていません。

表 18. *Java EE 7 Full Platform* 用にサポートされる *Liberty* フィーチャー (続き)

Liberty フィーチャー	**Liberty** フィーチャーの説明	**CICS** でのフィーチャーの使用
concurrent-1.0	管理対象の実行プログラムの作成を可能にします。これにより、同時に実行可能な複数のタスクをアプリケーションが実行依頼できるようになります。	**制約事項:** トランザクション・プロパティー「`ManagedTask.SUSPEND`」は、Liberty JVM サーバーでサポートされていません。 **制約事項:** 新しいスレッドのトランザクションに関連付けられるユーザー ID は、常に親トランザクションに関連付けられているユーザー ID です。 **制約事項:** `ManagedThreadFactory` を使用すると、CICS 対応 Java スレッドではなく標準 Java スレッドが作成されます。
ejb-3.2	EJB 3.2 仕様に従って作成された Enterprise JavaBeans のサポートを可能にします。	
ejbHome-3.2	EJB 2.x API のサポートを提供します。	
ejbPersistentTimer-3.2	永続 EJB タイマーのサポートを提供します。	**制約事項:** DB2 JDBC タイプ 2 の接続は、永続 EJB タイマーではサポートされていません。
ejbRemote-3.2	リモート EJB インターフェースのサポートを提供します。	
jacc-1.5	Java Authorization Contract for Containers (JACC) バージョン 1.5 のサポートを可能にします。	Java Authorization Contract for Containers (JACC) 許可プロバイダーの開発
jaspic-1.1	Java Authentication SPI for Containers (JASPIC) により、Java EE アプリケーション・サーバーがカスタム認証を使用できるようになります。JASPIC プロバイダーは JSR-196 で定義されています。JASPIC プロバイダーと TAI が同じサーバーに構成されている場合、TAI は無効になります。JASPIC は標準 Java EE テクノロジーであるため、Java EE アプリケーションには TAI よりも移植可能性に優れたソリューションです。	
j2eeManagement-1.1	JEE アプリケーション・サーバー内のアプリケーションを管理およびモニターするための一連のインターフェースを提供します。	

表 18. *Java EE 7 Full Platform* 用にサポートされる *Liberty* フィーチャー (続き)

Liberty フィーチャー	**Liberty** フィーチャーの説明	**CICS** でのフィーチャーの使用
javaMail-1.5	アプリケーションで JavaMail 1.5 API を使用できるようにします。	
javaee-7.0	Java EE 7.0 Full Platform をサポートする Liberty フィーチャーを結合します。	
jaxb-2.2	Java クラスと XML 表現をマップするためのサポートを提供します。	
jaxws-2.2	SOAP Web サービスのサポートを提供します。	
jca-1.7	アプリケーションからエンタープライズ情報システム (EIS) にアクセスするためのリソース・アダプターの構成を有効にします。	113 ページの『Java EE コネクター・アーキテクチャー (JCA)』 **制約事項:** JCA API `javax.resource.spi.BootstrapContext.createTimer()` によって作成されたスレッド内では、JCICS API および DB2 JDBC タイプ 2 の接続機能の使用はサポートされません。同じ効果を持つ機能として、同時 API (`javax.enterprise.concurrent.ManagedScheduledExecutorService`) を使用します。
jcaInboundSecurity-1.0	`javax.resource.spi.work.SecurityContext` 抽象クラスを拡張することで、JCA インバウンド・リソース・アダプターがセキュリティー・コンテキストを渡せるようにします。	
mdb-3.2	EJB 3.2 仕様に従って作成されたメッセージ駆動型 Enterprise JavaBeans の使用を可能にします。MDB によって、Java EE コンポーネント内のメッセージの非同期処理が可能になります。	
wasJmsClient-2.0	アプリケーションが JMS API を使用して、Liberty でホストされるメッセージ・キューにアクセスできるようにします。	107 ページの『Java Message Service (JMS)』
wasJmsSecurity-1.0	組み込みメッセージング・サーバーがクライアントからのアクセスを認証して許可できるようにします。	107 ページの『Java Message Service (JMS)』

表 18. *Java EE 7 Full Platform* 用にサポートされる *Liberty* フィーチャー (続き)

Liberty フィーチャー	**Liberty** フィーチャーの説明	**CICS** でのフィーチャーの使用
wasJmsServer-1.0	サーバー内で組み込みメッセージング・サーバーを使用できるようにします。アプリケーションは、`wasJmsClient` フィーチャーを使用して、メッセージを操作できます。	107 ページの『Java Message Service (JMS)』

表 19. 拡張型プログラミング・モデルでサポートされている *Liberty* フィーチャー

Liberty フィーチャー	**Liberty** フィーチャーの説明	**CICS** でのフィーチャーの使用
json-1.0	Java 環境用の一連の JSON ハンドリング・クラスを提供する、JavaScript Object Notation (JSON4J) ライブラリーを利用できるようにします。	
jta-1.1	Java Transaction API (JTA) をサポートします。 **注:** Java Transaction API は、保護 Liberty フィーチャーです。	94 ページの『Java Transaction API (JTA)』
jta-1.2	Java Transaction API (JTA) をサポートします。 **注:** Java Transaction API は、保護 Liberty フィーチャーです。	94 ページの『Java Transaction API (JTA)』
	MongoDB Java Driver のサポートを提供し、サーバー構成でリモート・データベース・インスタンスを構成できるようにします。アプリケーションは、`MongoDB` API を介してこれらのデータベースとやり取りします。	

表 20. *Enterprise OSGi* でサポートされている *Liberty* フィーチャー

Liberty フィーチャー	**Liberty** フィーチャーの説明	**CICS** でのフィーチャーの使用
blueprint-1.0	OSGi blueprint container 仕様を使用した OSGi アプリケーションをデプロイするためのサポートを有効にします。	**制約事項:** トランザクション属性「`NotSupported`」は、Liberty JVM サーバーではサポートされません。
osgi.jpa-1.0	このフィーチャーは、OSGi 機能が組み込まれた、`blueprint-1.0` フィーチャーおよび `jpa-2.0` フィーチャーに置き換えられました。これらのフィーチャーが両方ともサーバーに追加されると、このフィーチャーが自動的に追加されます。	

表 20. *Enterprise OSGi* でサポートされている *Liberty* フィーチャー (続き)

Liberty フィーチャー	**Liberty** フィーチャーの説明	**CICS** でのフィーチャーの使用
wab-1.0	エンタープライズ・バンドル (EBA) に含まれる Web アプリケーション・バンドル (WAB) のサポートを提供します。	82 ページの『OSGi アプリケーション・プロジェクトの作成』 **注:** このフィーチャーは、JVM システム・プロパティー `com.ibm.cics.jvmserver.wlp.wab=true` が設定されていれば、CICS によって自動的に追加されます。

表 21. *MicroProfile* 用にサポートされる *Liberty* フィーチャー

Liberty フィーチャー	**Liberty** フィーチャーの説明	**CICS** でのフィーチャーの使用
microProfile-1.0	Enterprise Java の Micro Profile をサポートする Liberty フィーチャーを結合します。	
microProfile-1.2	Enterprise Java のための Micro Profile 1.2 をサポートする Liberty フィーチャーを結合します。	このフィーチャーには Java 8 が必要です。
mpConfig-1.1	構成にアクセスするための統一メカニズムが規定されており、複数ソースの単一ビューが提供されます。	このフィーチャーには Java 8 が必要です。
mpFaultTolerance-1.0	Enterprise Java のための MicroProfile Fault Tolerance API のサポートを提供します。	このフィーチャーには Java 8 が必要です。 **制約事項:** MicroProfile Fault Tolerance 1.0 は、トランザクション (UOW や JTA など) を処理するように設計されていません。CICS リソースの更新は、@Bulkhead、@CircuitBreaker、@Fallback、@Retry、または @Timeout の注釈が付けられたメソッドでは実行できません。
mpHealth-1.0	Enterprise Java のための MicroProfile Health API のサポートを提供します。	このフィーチャーには Java 8 が必要です。
mpJwt-1.0	Web アプリケーションまたはマイクロサービスは、構成されているユーザー・レジストリーの代わりに、またはそれに加えて、JSON Web Token (JWT) を使用してユーザーを認証することが可能になります。	このフィーチャーには Java 8 が必要です。 属性 `ignoreApplicationAuthMethod` のデフォルト値は false です。これは、Liberty によって受信されるすべての要求の HTTP ヘッダーに JWT トークンが含まれている必要があることを示します。 属性 `mapToUserRegistry` のデフォルト値は false です。CICS セキュリティーと統合する場合は、この値を true に設定します。

表 21. *MicroProfile* 用にサポートされる *Liberty* フィーチャー (続き)

Liberty フィーチャー	**Liberty** フィーチャーの説明	**CICS** でのフィーチャーの使用
mpMetrics-1.0	Enterprise Java のための MicroProfile Metrics API のサポートを提供します。	このフィーチャーには Java 8 が必要です。

表 22. 操作に関してサポートされている *Liberty* フィーチャー

Liberty フィーチャー	**Liberty** フィーチャーの説明	**CICS** でのフィーチャーの使用
batchManagement-1.0	Java バッチ・コンテナーの管理バッチ・サポートを提供します。これには、バッチ REST 管理インターフェース、ジョブ・ロギングのサポート、および外部スケジューラーの統合のためのコマンド・ライン・ユーティリティーが含まれます。	
distributedMap-1.0	`DistributedMap` API を介してアクセスできるローカル・キャッシュ・サービスを提供します。	
localConnector-1.0	JVM にビルドされたローカル JMX コネクターを使用して、サーバー内の JMX リソースにアクセスできるようにします。	108 ページの『Java Management Extensions API (JMX)』
monitor-1.0	JMX クライアントを使用した Liberty ランタイム・コンポーネントのパフォーマンス・モニターを可能にします。	108 ページの『Java Management Extensions API (JMX)』
osgiConsole-1.0	ランタイムのデバッグを支援する OSGi コンソールを使用できるようにします。	Troubleshooting Java applications
restConnector-1.0	REST ベースのコネクターを介した JMX クライアントによるリモート・アクセスを可能にします。SSL およびユーザー・セキュリティー構成が必要です。	108 ページの『Java Management Extensions API (JMX)』
sessionDatabase-1.0	JDBC を使用したデータ・ソースへの HTTP セッションのパーシスタンスを有効にします。	
webCache-1.0	Web 応答のローカル・キャッシングを可能にします。これには `distributedMap` フィーチャーが含まれ、応答時間とスループットを改善するために、Web アプリケーション応答の自動キャッシングを行います。	

表 22. 操作に関してサポートされている *Liberty* フィーチャー (続き)

Liberty フィーチャー	**Liberty** フィーチャーの説明	**CICS** でのフィーチャーの使用
	IBM MQ でホストされるメッセージ・キューへの `JMS 2.0` API を介したアクセスをアプリケーションに提供します。	**制約事項:** JMS アプリケーションがクライアント・モードの転送を使用して IBM MQ に接続する場合にのみサポートされます。 Liberty 用の IBM MQ Resource Adapter V9.0.1 が必要です。 **重要:** この制約事項は、CICS 標準モードの Liberty にも適用されます。

表 23. セキュリティーに関してサポートされている *Liberty* フィーチャー

Liberty フィーチャー	**Liberty** フィーチャーの説明	**CICS** でのフィーチャーの使用
appSecurity-1.0	サーバー・ランタイム環境とアプリケーションを保護するためのサポートを提供します。`appSecurity-2.0` は `appSecurity-1.0` を置き換えるものです。	Liberty JVM サーバーのセキュリティーの構成 (Configuring security for a Liberty JVM server)
appSecurity-2.0	サーバー・ランタイム環境とアプリケーションを保護するためのサポートを提供します。`appSecurity-2.0` は `appSecurity-1.0` を置き換えるものです。	Liberty JVM サーバーのセキュリティーの構成 (Configuring security for a Liberty JVM server)
jwt-1.0	ランタイムで JWT トークンを作成できます。	
ldapRegistry-3.0	LDAP サーバーをユーザー・レジストリーとして使用するためのサポートが有効になります。LDAP バージョン 3.0 をサポートする任意のサーバーを使用できます。複数の LDAP レジストリーを構成してから統合して、単一の論理レジストリー・ビューを実現できます。	分散 ID マッピングを使用した Liberty JVM サーバーのセキュリティーの構成
oauth-2.0	Web アプリケーションが、ユーザーの認証および許可のために OAuth 2.0 を統合できるようになります。	OAuth 2.0 を使用した許可 パーシスタント OAuth 2.0 サービスの構成
openidConnectClient-1.0	Web アプリケーションは、構成されているユーザー・レジストリーの代わりまたはユーザー・レジストリーに追加して、ユーザーを認証するために OpenID Connect クライアント 1.0 を統合できます。	
openidConnectServer-1.0	Web アプリケーションは、構成されているユーザー・レジストリーの代わりまたはユーザー・レジストリーに追加して、ユーザーを認証するために OpenID Connect サーバー 1.0 を統合できます。	

表 23. セキュリティーに関してサポートされている *Liberty* フィーチャー (続き)

Liberty フィーチャー	**Liberty** フィーチャーの説明	**CICS** でのフィーチャーの使用
ssl-1.0	Secure Sockets Layer (SSL) 接続および SAF 鍵リングのサポートを提供します。	RACF を使用した Liberty JVM サーバー用の SSL (TLS) の構成 Liberty JVM サーバーでの SSL (TLS) クライアント証明書認証のセットアップ Java 鍵ストアを使用した Liberty JVM サーバー用の SSL (TLS) の構成

表 24. *z/OS* でサポートされている *Liberty* フィーチャー

Liberty フィーチャー	**Liberty** フィーチャーの説明	**CICS** でのフィーチャーの使用
	Liberty が z/OS リソース・リカバリー・サービス (RRS)、アプリケーション・サーバーのトランザクション・マネージャー、およびリソース・マネージャー間でトランザクション・アクティビティーを同期化し、管理することを可能にします。	**制約事項:** `zosTransaction-1.0` がサポートされるのは、CICS 標準モードの Liberty でバインディング・モードの転送を使用して IBM MQ に接続する JMS アプリケーションのみです。

表 25. *Java EE 6 Web Profile* 用にサポートされる *Liberty* フィーチャー

Liberty フィーチャー	**Liberty** フィーチャーの説明	**CICS** でのフィーチャーの使用
beanValidation-1.0	JavaBeans を検証するための、アノテーション・ベースのモデルを提供します。	
beanValidation-1.1	JavaBeans を検証するための、アノテーション・ベースのモデルを提供します。	
cdi-1.0	EJB や Managed Bean などのコンポーネントを、JSP や EJB といった他のコンポーネントに注入するためのメカニズムを提供します。	
ejbLite-3.1	EJB 仕様の EJB Lite サブセットに記載されている Enterprise JavaBeans のサポートを有効にします。	**制約事項:** トランザクション属性「`NotSupported`」は、Liberty JVM サーバーではサポートされません。
jndi-1.0	Liberty のサーバー構成における、単一の Java Naming and Directory Interface (JNDI) エントリー定義のサポートを提供します。	
jpa-2.0	アプリケーション管理およびコンテナー管理の JPA を使用したアプリケーションのサポートを有効にします。	
jsf-2.0	JavaServer Faces (JSF) フレームワークを使用する Web アプリケーションのサポートを提供します。	

表 25. *Java EE 6 Web Profile* 用にサポートされる *Liberty* フィーチャー (続き)

Liberty フィーチャー	**Liberty** フィーチャーの説明	**CICS** でのフィーチャーの使用
jsp-2.2	サーブレットおよび JavaServer Pages (JSP) アプリケーションのサポートを有効にします。	80 ページの『Java EE アプリケーションと Liberty アプリケーション』
servlet-3.0	Java サーブレット仕様に書き込まれる HTTP サーブレットのサポートを提供します。	80 ページの『Java EE アプリケーションと Liberty アプリケーション』 **制約事項:** サーブレット・ログインおよびログアウト API を使用すると、ユーザー ID が CICS タスクに伝搬されません。
webProfile-6.0	Java EE 6 Web プロファイルをサポートするのに必要な Liberty フィーチャーの便利な組み合わせを提供します。	

表 26. *Java EE 6 Technologies* 用にサポートされる *Liberty* フィーチャー

Liberty フィーチャー	**Liberty** フィーチャーの説明	**CICS** でのフィーチャーの使用
jaxb-2.2	Java クラスと XML 表現をマップするためのサポートを提供します。	
jaxrs-1.1	Liberty での Java API for RESTful Web Services (JAX-RS) のサポートを提供します。	
jaxws-2.2	SOAP Web サービスのサポートを提供します。	
jca-1.6	アプリケーションからエンタープライズ情報システム (EIS) にアクセスするためのリソース・アダプターの構成を有効にします。	113 ページの『Java EE コネクター・アーキテクチャー (JCA)』 **制約事項:** JCA API `javax.resource.spi.BootstrapContext.createTimer()` によって作成されたスレッド内では、JCICS API および DB2 JDBC タイプ 2 の接続機能の使用はサポートされません。代わりに、同じ効果を持つ機能として、同時 API (`javax.enterprise.concurrent.ManagedScheduledExecutorService`) を使用します。
jcaInboundSecurity-1.0	`javax.resource.spi.work.SecurityContext` 抽象クラスを拡張することで、JCA インバウンド・リソース・アダプターがセキュリティー・コンテキストを渡せるようにします。	
jdbc-4.0	アプリケーションからデータベースにアクセスするためのデータ・ソースの構成を使用可能にします。	**注:** DB2 JCC ドライバー内の `jdbc-4.0` 実装と `jdbc-4.1` 実装を同時に使用することはできません。

表 26. *Java EE 6 Technologies* 用にサポートされる *Liberty* フィーチャー (続き)

Liberty フィーチャー	**Liberty** フィーチャーの説明	**CICS** でのフィーチャーの使用
jms-1.1	Java Message Service API を使用してメッセージング・システムにアクセスするためのリソース・アダプターの構成を使用可能にします。	107 ページの『Java Message Service (JMS)』
jmsMdb-3.1	JMS メッセージ駆動型 Enterprise JavaBeans の使用を可能にします。MDB によって、Java EE コンポーネント内のメッセージの非同期処理が可能になります。	
mdb-3.1	メッセージ駆動型 Enterprise JavaBeans の使用を可能にします。MDB によって、Java EE コンポーネント内のメッセージの非同期処理が可能になります。	
wasJmsClient-1.1	アプリケーションが JMS API を使用して、Liberty でホストされるメッセージ・キューにアクセスできるようにします。	Java Message Service (JMS)
wasJmsSecurity-1.0	組み込みメッセージング・サーバーがクライアントからのアクセスを認証して許可できるようにします。	Java Message Service (JMS)
wasJmsServer-1.0	サーバー内で組み込みメッセージング・サーバーを使用できるようにします。アプリケーションは、`wasJmsClient` フィーチャーを使用して、メッセージを操作できます。	Java Message Service (JMS)

これらのフィーチャーの機能について詳しくは、 Liberty の資料 (Liberty の概要) を参照してください。Liberty の制約事項について詳しくは、ランタイム環境での既知の制約事項を参照してください。

以下の表では、Liberty フィーチャーを CICS のサービスの品質に統合するための一連の CICS フィーチャーを示します。Liberty JVM サーバー・モードは、JVM プロファイルに CICS_WLP_MODE を指定することによって設定できます。

表 27. *CICS Liberty* フィーチャー

CICS フィーチャー	**CICS Liberty** のモード	説明	**CICS** フィーチャーの使用
cicsts:core-1.0	統合モード	コア CICS フィーチャー、および Java Transaction API (JTA) 1.0 を提供します。	このフィーチャーは、統合モードの CICS Liberty を使用する場合に必要です。 **制約事項:** このフィーチャーを追加または削除する前に、JVM サーバーを無効にする必要があります。
cicsts:defaultApp-1.0	統合モード・モードおよび標準モード	Liberty サーバーが実行中であり、サーバー構成に関する情報を提供することを確認します。FileViewer サーブレットを使用して、JVM プロファイル、JVM サーバー・ログ、Liberty の server.xml、およびメッセージ・ログを参照します。	CICS デフォルト Web アプリケーションの構成
cicsts:distributedIdentity-1.0	統合モード・モードおよび標準モード	配布 ID のマッピングのサポートを提供します。	分散 ID マッピングを使用した Liberty JVM サーバーのセキュリティーの構成
cicsts:jcaLocalEci-1.0	統合モード	CICS プログラムを呼ぶための、ローカル用に最適化された JCA ECI リソース・アダプターを提供します。	115 ページの『JCA ローカル ECI リソース・アダプターの使用』 **制約事項:** このフィーチャーを追加または削除する前に、JVM サーバーを無効にする必要があります。
cicsts:jdbc-1.0	統合モード・モードおよび標準モード	アプリケーションが JDBC を使用してローカル CICS DB2 データベースにアクセスするためのサポートを提供します。このフィーチャーは、jdbc-4.0 および jdbc-4.1 によって置き換えられています。ただし、DriverManager で直接使用する場合を除きます。	データベースへの接続の取得 **制約事項:** このフィーチャーを追加または削除する前に、JVM サーバーを無効にする必要があります。
cicsts:link-1.0	統合モード	Liberty JVM サーバーで実行される Java EE アプリケーションを、CICS トランザクションの初期プログラムとして、または CICS プログラムから **LINK**、**START**、または **START CHANNEL** コマンドを使用して開始するためのサポートを提供します。	87 ページの『CICS プログラムからの Java EE アプリケーションへのリンク』

表 27. *CICS Liberty* フィーチャー (続き)

CICS フィーチャー	**CICS Liberty** のモード	説明	**CICS** フィーチャーの使用
cicsts:security-1.0	統合モード・モードおよび標準モード	スレッド ID の伝搬を含め、Liberty セキュリティーと CICS セキュリティーの統合を提供します。	Liberty JVM サーバーのセキュリティーの構成 (Configuring security for a Liberty JVM server) **制約事項:** このフィーチャーを追加または削除する前に、JVM サーバーを無効にする必要があります。
cicsts:standard-1.0	標準モード	アプリケーションを変更せずに、他のプラットフォームから CICS にアプリケーションを移植およびデプロイできるようにします。標準モードは、Java EE Full Platform を対象に作成され、このプラットフォームに依存するアプリケーションをホスティングするのに理想的ですが、CICS との完全な統合は必要ありません。	CICS 標準モードの Liberty: CICS と完全に統合せずに Java EE 7 Full Platform をサポート
cicsts:zosConnect-1.0	統合モード	z/OS Connect を CICS Liberty JVM サーバーと統合します。	z/OS Connect EE の構成 **制約事項:** このフィーチャーを追加または削除する前に、JVM サーバーを無効にする必要があります。
cicsts:zosConnect-2.0	統合モード	z/OS Connect を CICS Liberty JVM サーバーと統合します。	z/OS Connect EE の構成 **制約事項:** このフィーチャーを追加または削除する前に、JVM サーバーを無効にする必要があります。

Java アプリケーションからのデータへのアクセス

Db2 および VSAM のデータのアクセスと更新を行うことができる Java アプリケーションを作成できます。または、他の言語のプログラムにリンクして、Db2、VSAM、および IMS にアクセスすることができます。

CICS のデータにアクセスするための Java アプリケーションを作成する際に、次のいずれかの手法を使用できます。CICS リカバリー・マネージャーがデータ保全性を維持します。

リレーショナル・データへのアクセス

次のいずれかの方法を使用して、Db2 のリレーショナル・データにアクセスするための Java アプリケーションを作成できます。

- 構造化照会言語 (SQL) コマンドを使用してデータにアクセスするプログラムにリンクする JCICS **LINK** コマンド。
- 適切なドライバーが使用可能な場合は、Java Data Base Connectivity (JDBC) または Structured Query Language for Java (SQLJ) 呼び出しを使用して、データに直接アクセスします。Db2 に適切な JDBC ドライバーが使用可能です。JDBC および SQLJ アプリケーション・プログラミング・インターフェースの使用について詳しくは、Java プログラムから Db2 データにアクセスするための JDBC および SQLJ の使用を参照してください。
- 基礎のアクセス機構として JDBC または SQLJ を使用する JavaBeans。このような JavaBeans を開発するには、適切な Java 統合開発環境 (IDE) を使用できます。

DL/I データへのアクセス

IMS の DL/I データにアクセスするには、Java アプリケーションで JCICS **LINK** コマンドを使用して、EXEC DLI コマンドを発行してデータにアクセスする中間プログラムにリンクする必要があります。

VSAM データへのアクセス

VSAM データにアクセスするには、Java アプリケーションで次のいずれかの方法を使用できます。

- VSAM に直接アクセスする場合は、JCICS ファイル制御クラス。
- CICS ファイル制御コマンドを出してデータにアクセスするプログラムにリンクする JCICS **LINK** コマンド。

Java からの構造化データとの対話

多くの場合、CICS Java プログラムは、当初は他のプログラミング言語用に設計されたデータと対話します。例えば、Java プログラムは、COBOL コピーブックで定義された COMMAREA を使用して COBOL プログラムにリンクしたり、アセンブラー言語 DSECT を使用してデータが定義される VSAM ファイルからレコードを読み取ったりすることができます。

構造化データの Java へのインポート

インポーターを使用して、他の言語からの構造化レコード・データとの対話を容易にする Java クラスを生成できます。インポーターは、言語構造ソースに含まれるデータ型をマップし、Java アプリケーションで基礎となるレコード構造内の個々のフィールドを容易に設定および取得できるようにします。

IBM Record Generator for Java または Rational の Java EE コネクター (J2C) ツールを使用すると、データと対話して Java クラスを生成できます。これにより、CICS で Java とその他のプログラム間でデータを受け渡すことができます。

IBM Record Generator for Java V3.0.0

IBM Record Generator for Java はスタンドアロン・ユーティリティーであり、COBOL コピーブックまたはアセンブラー DSECT のコンパイルから生成される関連データ (ADATA) ファイルに基づいて、Java ヘルパー・クラスを生成します。次

に、Java アプリケーションで、これらの Java ヘルパー・クラスを COBOL 固有またはアセンブラー言語固有のレコード構造間でのデータ・マーシャルに使用できます。

詳しくは、IBM Record Generator for Java V3.0.0を参照してください。

Rational J2C ツール

Rational J2C ツール、リソース・アダプター、およびファイル・インポーターを使用して、J2C 成果物を作成できます。この成果物を使用すると、CICS などのエンタープライズ情報システムに接続するエンタープライズ・アプリケーションを作成できます。Rational J2C ツールを使用するには、Rational Application Developer for WebSphere Software または IBM Developer for z Systems が必要です。

J2C ツールの CICS/IMS データ・バインディング・ウィザードでは、カスタマイズ可能な Eclipse ベースのウィザードを使用して COBOL、PL/I、または C の各アプリケーション・プログラムのデータ構造にマップする Java クラスを生成します。次に、Java アプリケーションで、これらのヘルパー・クラスを言語固有のレコード構造間でのデータ・マーシャルに使用できます。

詳しくは、エンタープライズ情報システムへの接続 (Rational Application Developer for WebSphere ソフトウェア製品資料)を参照してください。

関連情報:

- IBM Redbooks: IBM CICS と JVM サーバー: Java アプリケーションの開発とデプロイ
- IBM Record Generator for Java を使用して COBOL から Java レコードを構築する
- COBOL Importer の概要 (Rational Application Developer for WebSphere ソフトウェア製品資料)
- Rational J2C Tools を使用して COBOL から Java レコードを生成する

OSGi JVM サーバーで JZOS Toolkit API を使用する Java アプリケーションの開発

IBM JZOS Toolkit は、パッケージ `com.ibm.jzos` 内のクラスで構成され、このパッケージは、IBM Java SDK for z/OS に付属して単一の JAR ファイル `ibmjzos.jar` で配布されます。

これらのクラスにより、z/OS 上の Java アプリケーションは、バイト配列フィールドを Java データ型にマッピングする際に、従来の z/OS データ・セットとデータ・ファイルに直接アクセスでき、z/OS システム・サービスとコンバーター・クラスにアクセスできます。

始める前に

JZOS Toolkit API をワークステーションにダウンロードしていない場合は、ibmjzos.jar ファイルを z/OS 上の関連するバージョンの IBM Java SDK からワークステーションに転送します。

手順

1. ターゲット・プラットフォームを設定します。開発環境で JCICS API を使用する準備を行うには、ローカルで解決できるように Eclipse ターゲット・プラットフォームを設定します。OSGi 開発環境のターゲット・プラットフォーム定義は、ワークスペース内のアプリケーションが作成されるプラグインを定義するために使用されます。CICS Explorer の場合、Eclipse メニューの「ウィンドウ」 > 「設定」 > 「プラグイン開発」 > 「ターゲット・プラットフォーム」を使用します。「追加」をクリックし、提供されているテンプレートから、ご使用のランタイム環境用の CICS TS リリースを選択します。ワークスペースにターゲット・プラットフォームを適用することを忘れないでください。
2. JZOS Toolkit 用の OSGi ラッパー・バンドルを作成します。IBM CICS SDK for Java EE and Liberty プラグインを使用している場合は、「ファイル」 > 「インポート」 > 「**OSGi** バンドルへの **Java** アーカイブ」を選択して、新しい OSGi バンドル・プロジェクトを作成します。新しく作成されたバンドルにより、Java アプリケーションに必要な使用可能なすべての JZOS Toolkit パッケージ (com.ibm.jzos、com.ibm.jzos.fields、com.ibm.jzos.wlm など) がエクスポートされることを確認します。これにより、これらのパッケージを Eclipse ワークスペース内の他の OSGi プロジェクトによってインポートできるようになります。 次に例を示します。

   ```
   Export-Package: com.ibm.jzos,
     com.ibm.jzos.fields
   ```

3. CICS Java アプリケーションを作成します。
   a. ウィザードの「ファイル」 > 「新規」 > 「その他のプラグイン・プロジェクト **(Other Plug-in Project)**」を使用して、Eclipse 内で OSGi バンドル・プロジェクトを作成します。
   b. Java パッケージ com.ibm.cicsdev.jzos.sample を作成し、クラス ZFilePrint を追加します。
   c. 次のコード例をコピーします。これにより、//INPUT DD が指す MVS データ・セットが開き、出力が CICS 一時記憶域キューに書き込まれます。

   ```
   package com.ibm.cicsdev.jzos.sample;

   import com.ibm.jzos.ZFile;
   import com.ibm.jzos.ZUtil;
   import com.ibm.cics.server.TSQ;

   public class ZFilePrint
   {
       public static void main(String[] args) throws Exception
       {
           ZFile zFile = new ZFile("//DD:INPUT", "rb,type=record,noseek");
           TSQ tsqQ = new TSQ();
           tsqQ.setName("JZOSTSQ");

           try
           {
               byte[] recBuf = new byte[zFile.getLrecl()];
   ```

```
                int nRead;
                String encoding = ZUtil.getDefaultPlatformEncoding();

                while ((nRead = zFile.read(recBuf)) >= 0)
                {
                    String line = new String(recBuf, 0, nRead, encoding);
                    tsqQ.writeString(line);
                }
            }
            finally
            {
                zFile.close();
            }
        }
    }
```

4. JCICS および JZOS パッケージのバンドル・マニフェストに、以下の Import-Package ステートメントを追加します。JCICS インポートでは、アプリケーションが動作するバージョンの範囲を指定するというベスト・プラクティスに従う必要があります。通常、この範囲は、次の API の互換性を破る変更の直前までになります。JCICS の場合、これはバージョン 2.0.0 であるため、この例では範囲 `com.ibm.cics.server;version="[1.401.0,2.0.0)"` を使用します。これは、JCICS `TSQ.writeString()` メソッドをサポートするために必要な最小レベルです。JZOS パッケージは、バージョン管理されたバンドルから取得されません。バージョンなしで基盤の JAR ファイルからのものがランタイムに表示されるため、バージョンを参照せずに `com.ibm.jzos` をリストできます。これにより、任意の使用可能なバージョン (0.0.0 を含む) が選択可能になります。

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: com.ibm.cicsdev.jzos.sample
Bundle-SymbolicName: com.ibm.cicsdev.jzos.sample
Bundle-Version: 1.0.0
Bundle-RequiredExecutionEnvironment: JavaSE-1.7
Import-Package: com.ibm.cics.server;version="[1.401.0,2.0.0)",
 com.ibm.jzos
```

5. CICS-MainClass: 定義をバンドル・マニフェストに追加して、`com.ibm.cicsdev.jzos.sample.ZFilePrint` クラスの **MainClass** サービスを登録します。 これにより、Java クラスを CICS プログラム定義を使用してリンクできます。現時点で、マニフェストの内容は次の例のようになっているはずです。

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: com.ibm.cicsdev.jzos.sample
Bundle-SymbolicName: com.ibm.cicsdev.jzos.sample
Bundle-Version: 1.0.0
Bundle-RequiredExecutionEnvironment: JavaSE-1.7
Import-Package: com.ibm.cics.server;version="[1.401.0,2.0.0)",
 com.ibm.jzos
CICS-MainClass: com.ibm.cicsdev.jzos.sample.ZFilePrint
```

タスクの結果

これで、アプリケーションをテストする準備ができました。以下のように、CICS バンドル・プロジェクトを使用して、CICS OSGi JVM サーバーにアプリケーションをデプロイできます。

1. Eclipse 内に CICS バンドル・プロジェクトを作成し、「新規 **OSGi** バンドル・プロジェクトを含める **(New OSGi Bundle Project Include)**」メニューを使用して、OSGi バンドル・プロジェクトを追加します。
2. 「**z/OS UNIX** ファイル・システムへのバンドル・プロジェクトのエクスポート」メニューを使用して、バンドル・プロジェクトを zFS にデプロイします。
3. この zFS ロケーションを参照する CICS バンドル定義を作成してインストールします。
4. `JVMClass` 属性内の CICS-MainClass: `com.ibm.cicsdev.jzos.sample.ZFilePrint` を指定する CICS プログラム定義を作成してインストールします。
5. アプリケーションを実行する前に、有効な MVS データ・セットを参照する MVS DD を CICS JCL で定義してから、CICS 領域を再始動する必要があります。次に例を示します。

   ```
   //INPUT    DD DISP=SHR,DSN=CICS.USER.INPUT
   ```

6. 3270 コンソールからアプリケーションを実行する必要がある場合は、ステップ 4 で定義したプログラムを参照するトランザクション定義を作成します。

開始時に、Java クラス `ZFilePrint` は JZOS Toolkit API を使用して定義された MVS データ・セットを読み取り、その内容を JCICS API を使用して CICS 一時記憶域キューに書き込みます。

Java プログラムから IBM MQ へのアクセス

CICS で実行される Java プログラムでは、IBM MQ classes for Java または IBM MQ classes for JMS を使用して IBM MQ にアクセスできます。IBM MQ classes for JMS は、CICS で実行される Java アプリケーションから IBM MQ にアクセスする場合の優先インターフェースです。IBM MQ classes for Java は引き続きサポートされますが、新しいアプリケーションでは IBM classes for JMS を使用する必要があります。

CICS が IBM MQ で動作する仕組みの概要については、CICS と IBM MQを参照してください。

IBM MQ classes for Java は、ネイティブ IBM MQ API である Message Queue Interface (MQI) をカプセル化します。これらのクラスでは、IBM MQ の C++ および .NET インターフェースと類似のオブジェクト・モデルが使用されています。また、JMS で使用可能な機能にとどまらず、IBM MQ の完全な機能一式を活用できます。IBM MQ classes for JMS は、メッセージング・システムとして IBM MQ 用の JMS インターフェースを実装しています。

CICS では、以下の 3 つの異なる JVM サーバー環境で IBM MQ クラスへのアクセスがサポートされます。

- CICS 統合モードの Liberty JVM サーバー。この JVM サーバーでは、IBM MQ classes for JMS がサポートされます。また、管理対象 JMS 接続ファクトリーと MDB サポート、および統合 CICS のトランザクションとセキュリティーが提供されます。IBM MQ classes for Java はサポートされません。
- CICS 標準モードの Liberty JVM サーバー。この JVM サーバーでは、IBM MQ classes for JMS がサポートされます。また、管理対象 JMS 接続ファクト

リーと MDB サポートが提供されますが、統合 CICS トランザクションは含まれません。IBM MQ classes for Java はサポートされません。

- OSGi JVM サーバー。この JVM サーバーでは、IBM MQ classes for JMS がサポートされます。また、非管理対象 JMS 接続ファクトリーおよび統合 CICS のトランザクションとセキュリティーがサポートされます。IBM MQ classes for Java もバインディング・モードでのみサポートされます。

さらに、CICS から IBM MQ に接続するには、以下の 3 つの方法があります。

- MQ クライアント・モード: IBM MQ キュー・マネージャーへの TCP/IP ネットワーク接続
- MQ バインディング・モード: IBM MQ RRS アダプターを使用した、キュー・マネージャーへのローカル・クロスメモリー・インターフェース
- CICS-MQ アダプターおよび MQCONN: CICS-MQ アダプターを使用した、キュー・マネージャーへのローカル・クロスメモリー・インターフェース

表 28 は、どの JVM サーバーでどの IBM MQ クラスをサポートしているか、および使用される接続オプションを示しています。

表 *28. Java* アプリケーションから *IBM MQ* にアクセスするための *CICS* サポートの要約

MQ 接続	**CICS** 標準モードの **Liberty JVM** サーバー	**CICS** 統合モードの **Liberty JVM** サーバー	**OSGi JVM** サーバー
クライアント・モード	• IBM MQ classes for JMS: JMS 1.1 および JMS 2.0 • IBM MQ classes for Java: サポートされません 詳しくは、 155 ページの『CICS Liberty JVM サーバーでの IBM MQ classes for JMS の使用』を参照してください。	• IBM MQ classes for JMS: JMS 1.1 および JMS 2.0 • IBM MQ classes for Java: サポートされません 詳しくは、 155 ページの『CICS Liberty JVM サーバーでの IBM MQ classes for JMS の使用』を参照してください。	サポート対象外
バインディング・モード	• IBM MQ classes for JMS: JMS 1.1 および JMS 2.0 • IBM MQ classes for Java: サポートされません 詳しくは、 155 ページの『CICS Liberty JVM サーバーでの IBM MQ classes for JMS の使用』を参照してください。	サポート対象外	• IBM MQ classes for JMS: サポートされません • IBM MQ classes for Java: サポートされます 詳しくは、 165 ページの『OSGi JVM サーバーでの IBM MQ classes for Java の使用』を参照してください。

表 28. *Java* アプリケーションから *IBM MQ* にアクセスするための *CICS* サポートの要約 *(続き)*

MQ 接続	**CICS** 標準モードの **Liberty JVM** サーバー	**CICS** 統合モードの **Liberty JVM** サーバー	**OSGi JVM** サーバー
CICS-MQ アダプターおよび MQCONN	適用外	適用外	• IBM MQ classes for JMS: JMS 1.1 および JMS 2.0 • IBM MQ classes for Java: サポートされません 詳しくは、 160 ページの『OSGi JVM サーバーでの IBM MQ classes for JMS の使用』を参照してください。

CICS Liberty JVM サーバーでの IBM MQ classes for JMS の使用

CICS Liberty JVM サーバーで実行される Java プログラムでは、JMS を使用して IBM MQ にアクセスできます。IBM MQ JMS 機能が CICS Liberty JVM サーバーにインストールされている場合、JMS 要求は MQ メッセージング・プロバイダーによって処理されます。JMS 2.0 機能のサポートにより、従来の (JMS 1.1) インターフェースおよび単純化された (JMS 2.0) インターフェースへのアクセスが提供されます。CICS は、適切なレベルの JMS をサポートし、適切なバージョンの IBM MQ classes for JMS を使用しているレベルの IBM MQ キュー・マネージャーに接続する必要があります。

概要については、仕組み: IBM MQ classes for JMSを参照してください。IBM MQ での JMS の実装方法については、IBM MQ 資料のIBM MQ classes for JMS の使用を参照してください (IBM MQ classes for JMS JavaDocや、メッセージ、アプリケーション機能、および MQ 機能についてはIBM MQ classes for JMS アプリケーションの作成など)。JMS 仕様のレベルを比較するには、Java Message Service Specificationを参照してください。

CICS Liberty 環境では、IBM MQ メッセージング・プロバイダーにより、IBM MQ キュー・マネージャーに対して作成された JMS 接続が以下のようにサポートされます。

- CICS 統合モードの Liberty JVM サーバーでは、JMS アプリケーションが MQ クライアント・モードの転送を使用してキュー・マネージャーに接続できます。MQ バインディング・モードの使用はサポートされません。このタイプの CICS Liberty JVM サーバーでは、統合 CICS のトランザクションとセキュリティーとともに JMS サポートが提供されます。
- CICS 標準モードの Liberty JVM サーバーでは、JMS アプリケーションは、MQ バインディング・モードまたはクライアント・モードのいずれかの転送を使用してキュー・マネージャーに接続できます。このタイプの CICS Liberty JVM サーバーでは、統合 CICS トランザクションを含まない JMS サポートが提供されます。

Java アプリケーションでは、以下の 2 つの方法のいずれかを使用して IBM MQ と通信します。

- メッセージ駆動型 Bean (MDB)
- JMS 接続ファクトリーを使用するサーブレット

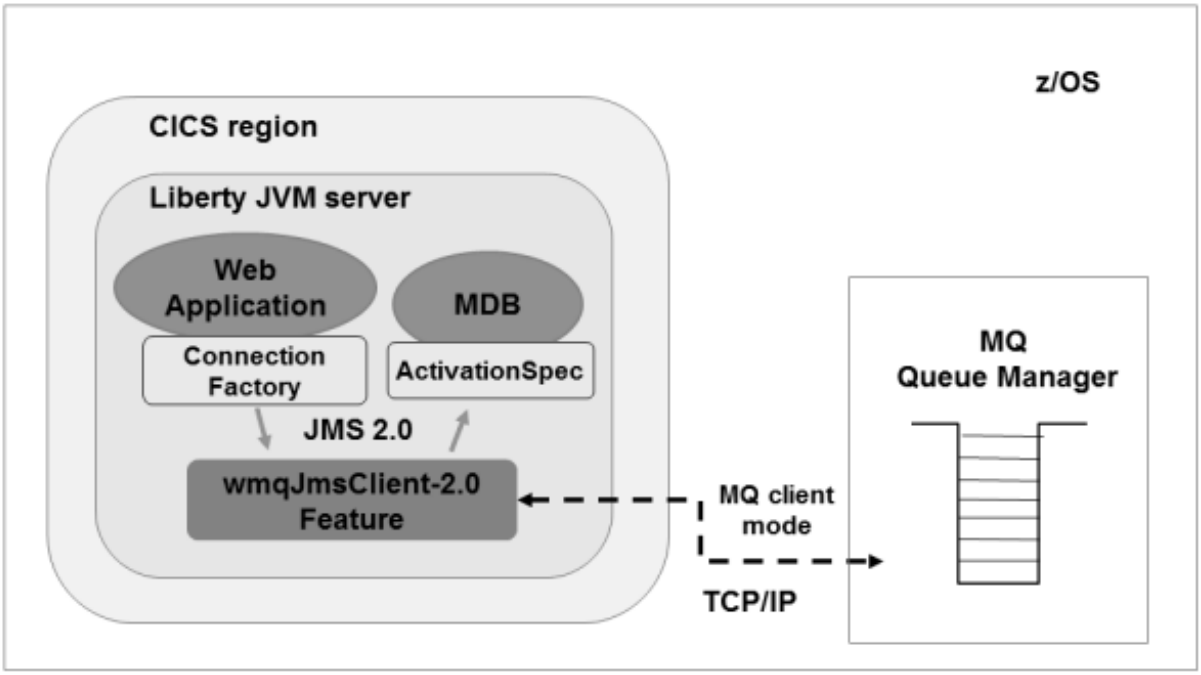


図 *8. JMS* を使用して *IBM MQ* に接続し、*CICS Liberty JVM* サーバーで実行するアプリケーション： この図は、CICS Liberty JVM サーバーから IBM MQ と対話するために使用可能な 2 つのタイプのプログラミングを示しています。示されている CICS Liberty JVM サーバーは、CICS 標準モードまたは CICS 統合モードのいずれかです。示されている MQ クライアント・モード接続に加えて、CICS 標準モードの Liberty JVM サーバーではバインディング・モードでも接続できます。

確認事項

- MQ への意図している接続が CICS Liberty JVM サーバーのご使用のバージョンでサポートされている。 153 ページの『Java プログラムから IBM MQ へのアクセス』の 154 ページの表 28を参照してください。
- CICS の接続先である IBM キュー・マネージャーによってサポートされる JMS のレベル。IBM MQ for z/OS バージョン 7.1 では、JMS 1.1 のみがサポートされます。IBM MQ バージョン 8.0 以上では、JMS 1.1 と JMS 2.0 の両方がサポートされます。
- バインディング・モードの転送 (CICS 標準モードの Liberty でのみサポートされます) を使用する場合は、以下のようになります。
  - バインディング・モードを使用して MQ キュー・マネージャーに接続する JMS アプリケーションでは、同じ CICS 領域にインストールされたすべての CICS MQCONN リソースで指定されているキュー・マネージャーとは異なるキュー・マネージャーを指定する必要があります。
  - Liberty と IBM MQ の両方が同じサーバー上にデプロイされます。

- CICSExecutorService を使用して開始されたすべての CICS タスクは、クライアント・モードの転送を使用してキュー・マネージャーに接続する必要があります。
- プログラミングに関していくつかの制約事項があり、これについては 159 ページの『JMS プログラミングに関する考慮事項 (Liberty JVM サーバー)』で説明されています。

次に行うこと

アプリケーションで JMS を使用するには、以下を実行する必要があります。

- 開発環境で、IBM MQ 製品のコンポーネントまたは FixCentral からの JAR ファイルとして、IBM MQ classes for JMS にアクセスできることを確認します (これを行う方法については、IBM MQ classes for JMS の使用を参照してください)。
- 管理対象 JMS 接続ファクトリーまたはメッセージ駆動型 Bean (MDB) のいずれかを使用するアプリケーションを開発します。詳しくは、『Liberty JVM サーバーでの IBM MQ classes for JMS を使用したプログラミング』を参照してください。
- アプリケーションを CICS バンドル・プロジェクトに追加し、zFS にエクスポートして、Liberty JVM サーバーにインストールします。
- CICS Liberty JVM サーバー環境を構成します。`server.xml` の構成に加えて、Liberty から IBM MQ に接続するために必要な IBM MQ リソース・アダプターをセットアップする必要があります。詳しくは、JMS をサポートするための Liberty JVM サーバーの構成を参照してください。

Liberty JVM サーバーでの IBM MQ classes for JMS を使用したプログラミング

JMS を CICS Java アプリケーションで使用して IBM MQ とメッセージを交換する場合、2 つのオプションがあります。IBM MQ キュー・マネージャーからの着信メッセージを受信するメッセージ駆動型 Bean (MDB)、または JMS 接続ファクトリーを使用して JMS メッセージを送受信するサーブレットのいずれかを使用できます。

JMS 接続ファクトリーの使用

このタイプのアプリケーションの開発方法を示すチュートリアルについては、CICS Developer Center: CICS Liberty 用の MQ JMS アプリケーションの開発を参照してください。このチュートリアルには、ダウンロード可能なサンプル・サーブレットとサポート・コードへのリンクが含まれています。

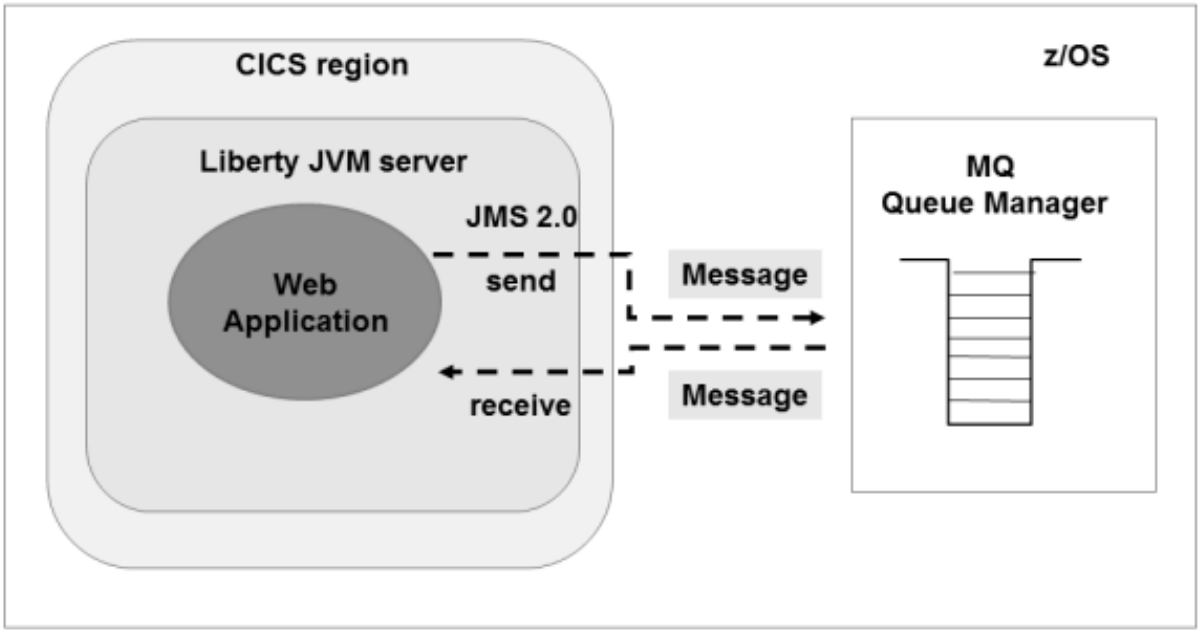


図 *9. JMS* 接続ファクトリーを使用する *Java* アプリケーションを介した *IBM MQ* へのアクセス

メッセージ駆動型 Bean (MDB) の使用

この形式のプログラミングでは、MDB に関連付けられているキューにメッセージが着信すると、MDB の onMessage() メソッドが呼び出されます。次に、`javax.jms.Message` オブジェクトが、さらに処理を行うために MDB への入力として渡されます。MDB は EJB のタイプであるため、コンテナー管理または Bean 管理のいずれかの Java トランザクションを使用できます。

CICS 開発者センターのCICS Developer Center: CICS Liberty 用の MQ JMS アプリケーションの開発は、この種のアプリケーションの開発方法のチュートリアルです。このチュートリアルには、ダウンロード可能なサンプル・サーブレットとサポート・コードへのリンクが含まれています。

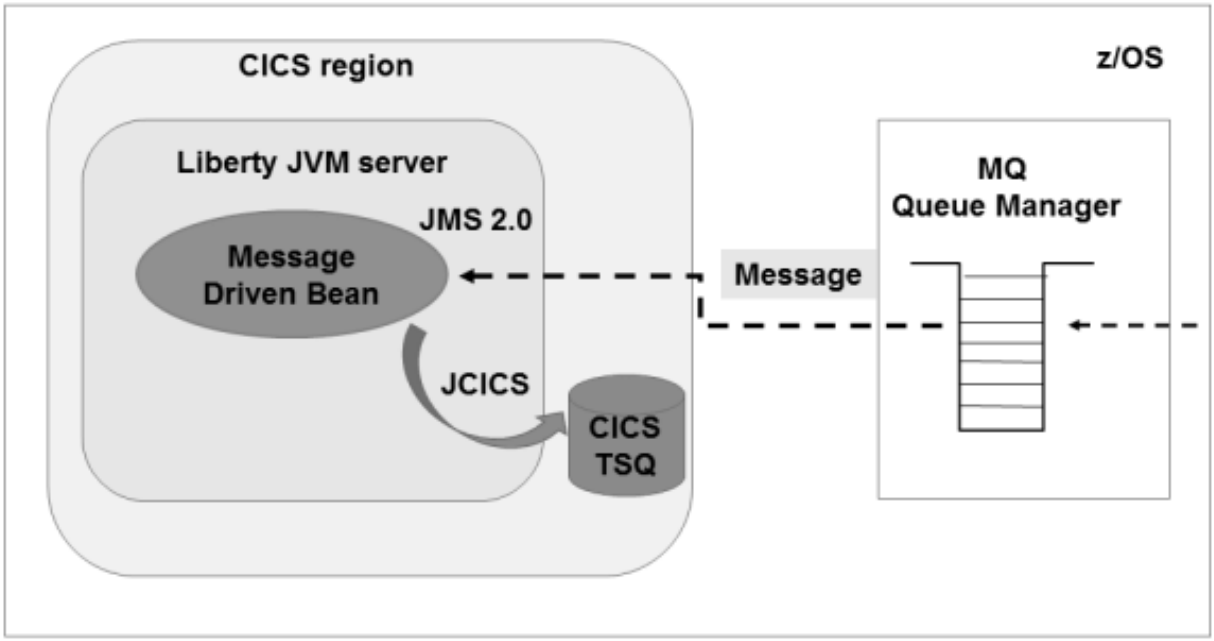


図 *10. MDB* を使用する *Java* アプリケーションを介した *IBM MQ* へのアクセス

JMS プログラミングに関する考慮事項 (Liberty JVM サーバー)

- runAsCICS() メソッドを使用して CICSExecutorService に実行依頼される処理には、JMS 要求が含まれていないこと。
- MDB 要求が実行される CICS トランザクション ID のデフォルトは CJSU (JVM サーバーの分類されていない要求プロセッサー) です。これは、システム・プロパティー `com.ibm.cics.jvmserver.unclassified.tranid` を使用して、JVM サーバーごとに変更できます。
- Liberty JVM サーバーで JMS を使用する場合、IBM MQ classes for JMS で送受信されるメッセージは、Liberty トランザクション・マネージャーを使用して調整されます。CICS によって管理されるリカバリー可能リソースへの更新を同じ作業単位で調整するには、アプリケーションで、Java Transaction API (JTA) を UserTransaction.begin() メソッドによって明示的に、または EJB コンテナー管理トランザクションによって暗黙的に使用する必要があります。UOW を完了するには、UserTransaction commit() メソッドまたは rollback() メソッドを使用します。UOW をコミットまたはロールバックするために、以下のオブジェクトで EXEC CICS SYNCPOINT コマンド (混合言語アプリケーション) または commit() メソッドと rollback() メソッドを使用することは、サポートされません。
  - javax.jms.Session (JMS 1.1 API)
  - javax.jms.JmsContext (JMS 2.0 API)
  - com.ibm.cics.server.Task

  JTA について詳しくは、Java Transaction API (JTA)を参照してください。

OSGi JVM サーバーでの IBM MQ classes for JMS の使用

OSGi JVM サーバーで実行される Java プログラムは、JMS を使用して IBM MQ にアクセスできます。CICS Java アプリケーションが JMS 要求を行うと、その要求は MQ メッセージング・プロバイダーによって処理されます。クラシック (JMS 1.1) インターフェースおよび単純化 (JMS 2.0) インターフェースの使用がサポートされています。ただし、CICS が、適切なレベルの JMS をサポートできるレベルの IBM MQ キュー・マネージャーに接続されており、適切なバージョンの IBM MQ classes for JMS を使用していることが条件です。

概要については、仕組み: IBM MQ classes for JMSを参照してください。IBM MQ での JMS の実装方法については、IBM MQ 資料のIBM MQ classes for JMS の使用を参照してください (IBM MQ classes for JMS JavaDocや、メッセージ、アプリケーション機能、および MQ 機能についてはIBM MQ classes for JMS アプリケーションの作成など)。JMS 仕様のレベルを比較するには、Java Message Service Specificationを参照してください。

CICS 環境では、IBM MQ classes for JMS を使用すると、OSGi JVM サーバーを介して接続を作成できます。これにより、非管理対象 JMS 接続ファクトリーおよび統合 CICS のトランザクションとセキュリティーがサポートされます。管理対象の JMS 接続ファクトリーまたは MDB をアプリケーションで使用する場合は、代わりに CICS Liberty JVM サーバーを使用します。

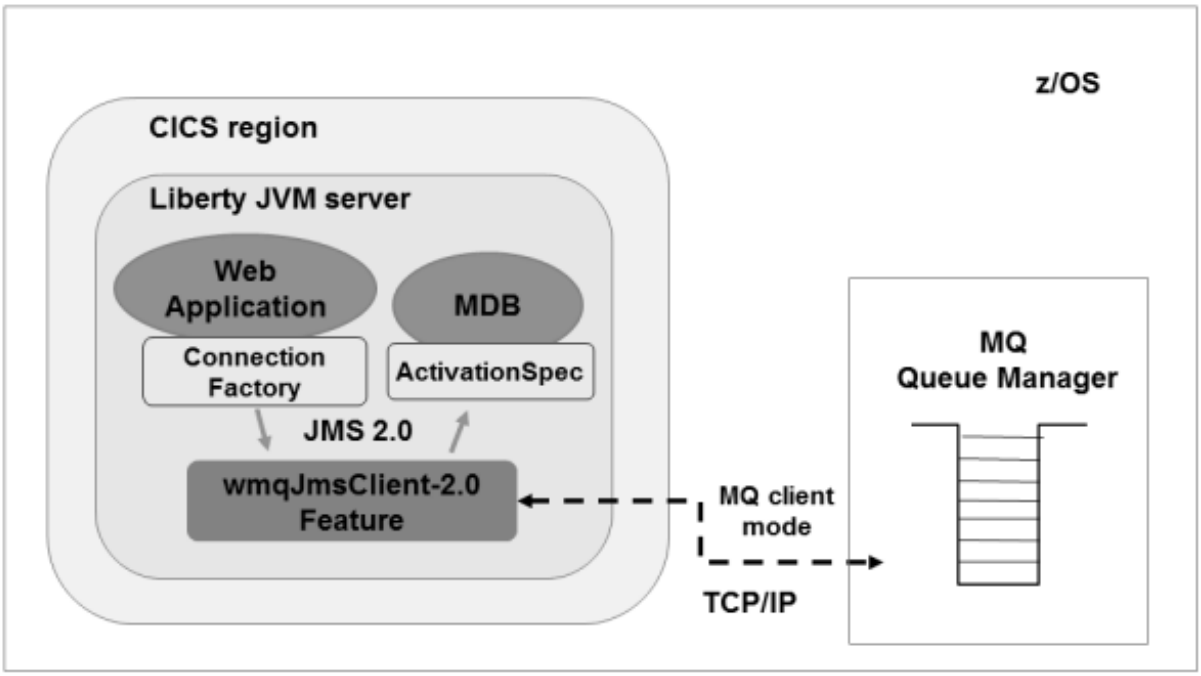


図 *11. JMS* を使用して *IBM MQ* に接続し、*CICS OSGi* サーバーで実行するアプリケーション

確認事項

- IBM MQ への接続。OSGi JVM サーバーでは、ローカル・キュー・マネージャーへのバインディング・モードの接続のみがサポートされます。

- CICS の接続先である IBM キュー・マネージャーによってサポートされる JMS のレベル。IBM MQ for z/OS バージョン 7.1 では、JMS 1.1 のみがサポートされます。IBM MQ バージョン 8.0 以上では、JMS 1.1 と JMS 2.0 の両方がサポートされます。
- CICS MQCONN リソースを定義した。
- プログラミングに関していくつかの制約事項があり、これについては『OSGi JVM サーバーでの IBM MQ classes for JMS を使用したプログラミング』で説明されています。

次に行うこと

アプリケーションで JMS を使用するには、以下を実行する必要があります。

- 開発環境で、IBM MQ 製品のコンポーネントまたは FixCentral からの JAR ファイルとして、IBM MQ classes for JMS にアクセスできることを確認します (これを行う方法については、IBM MQ 資料のIBM MQ classes for JMS の使用を参照してください)。
- 非管理対象 JMS 接続ファクトリーを使用するアプリケーションを開発します。詳しくは、『OSGi JVM サーバーでの IBM MQ classes for JMS を使用したプログラミング』を参照してください。
- アプリケーションを CICS バンドル・プロジェクトに追加し、zFS にエクスポートして、OSGi JVM サーバーにインストールします。
- IBM MQ に接続する CICS-MQ アダプターを構成します。詳しくは、CICS-MQ アダプターのセットアップを参照してください。
- CICS OSGi サーバー環境を構成します。詳しくは、JMS をサポートするための OSGi JVM サーバーの構成を参照してください。

OSGi JVM サーバーでの IBM MQ classes for JMS を使用したプログラミング

JMS を CICS Java アプリケーションで使用して IBM MQ とメッセージを交換するには、JMS 接続ファクトリーを使用します。

このタイプのアプリケーションの開発方法を示すチュートリアルについては、CICS Developer Center: CICS OSGi JVM サーバーでの MQ JMS の使用を参照してください。

JMS プログラミングに関する考慮事項 (OSGi JVM サーバー)

- XA 接続ファクトリー (例えば com.ibm.mq.jms.MQXAConnectionFactory) の使用はサポートされていません。
- JVM サーバー環境で IBM® MQ classes for JMS によって送受信されるメッセージは常に、現行スレッドでアクティブな CICS® の作業単位 (UOW) と関連付けられます。その UOW は、com.ibm.cics.server.Task オブジェクトでコミット・メソッドまたはロールバック・メソッドを呼び出すことによってのみ完了できます。また、CICS が正常に終了する場合にも完了しますが、その場合 UOW は暗黙的にコミットされます。(これには 1 つの例外があり、このリストの次の項目で説明されています。)Connection.createSession または ConnectionFactory.createContext のいずれかのメソッドを呼び出す際に、

transacted 引数と acknowledgeMode 引数の値は無視されます。また、以下のメソッドはサポートされておらず、これらを呼び出すと、セッション・ケースで IllegalStateException になります。

- javax.jms.Session.commit()
- javax.jms.Session.recover()
- javax.jms.Session.rollback()

以下は、JMS コンテキスト・ケースで IllegalStateRuntimeSession になります。

- javax.jms.JMSContext.commit()
- javax.jms.JMSContext.recover()
- javax.jms.JMSContext.rollback()

- 上記のトランザクション性に対する例外は、セッションまたは JMS コンテキストが以下に示すいずれかのメカニズムを使用して作成された場合、以下のようになります。
  - Connection.createSession(false, Session.AUTO_ACKNOWLEDGE)
  - Connection.createSession(Session.AUTO_ACKNOWLEDGE)
  - ConnectionFactory.createContext(JMSContext.AUTO_ACKNOWLEDGE)

  このセッションの動作、または JMS コンテキストは、以下のようになります。
  - 送信されるメッセージはすべて、CICS UOW の外で転送されます。つまり、ターゲット宛先で即時に、または指定された送達遅延間隔が完了したときに使用可能になります。
  - セッションまたは JMS コンテキストを作成した接続ファクトリーで `syncPointAllGets` プロパティーが指定されていない場合、非永続メッセージは CICS UOW の外で受信されます。
  - 永続メッセージは、常に CICS UOW 内で受信されます。

  混合言語アプリケーションの場合、非 Java™ プログラムから発行された EXEC CICS SYNCPOINT コマンドは、Java プログラムによる IBM MQ の更新を含め、作業単位全体をコミットします。
- JMS では、javax.jms.MessageListener、javax.jms.ExceptionListener、JMS 2 を使用する場合には javax.jms.CompletionListener など、多数のさまざまなリスナー・インターフェースがサポートされます。これらのどのインターフェースを使用する場合も、MQ JMS では、CICS 環境でサポートされない複数のスレッドが使用されます。これらのうちいずれかのリスナーを登録しようとすると、JMSException または JMSRuntimeException が発生します。
- MQ JMS は、CICS での IBM MQ に対するネイティブ・サポートを基盤としているため、IBM MQ セキュリティー・サポートが利用されます。このセキュリティー・サポートについては、ここを参照してください。その結果、ユーザー ID またはパスワードを指定しているときに接続または JMS コンテキスト・オブジェクトのいずれかを作成しようとすると、JMSException または JMSRuntimeException が発生します。
- CICS MQCONN リソースを定義する必要があります。MQ JMS が接続するキュー・マネージャーまたはキュー共用グループの名前は、この MQCONN 定義から取得されます。キュー・マネージャーまたはキュー共用グループをプログラムで指定しようとしても、無効になります。

- 以下の方式で、接続ファクトリーおよび宛先の IBM MQ 実装を作成して構成することができます。
  - JNDI を使用した管理対象オブジェクトの取得
  - IBM JMS 拡張機能の使用
  - IBM MQ JMS 拡張機能の使用

  たいていの JMS ユーザーは、JNDI リポジトリーを使用して、事前構成された一式の接続ファクトリーおよび宛先を見つけます。CICS では、JNDI 実装は提供されていません。また、LDAP は OSGi 環境では使用できません。

  使用可能なオプション、およびバンドル・アダプターの start メソッドを使用して OSGi に初期コンテキスト・ファクトリーと IBM MQ オブジェクト・ファクトリーを登録する方法の例について詳しくは、接続ファクトリーおよび宛先の作成および構成を参照してください。

  CICS と IBM MQ キュー・マネージャーの間の接続は、CICS アドレス・スペースのユーザー ID を使用して管理されます。キューへのリソース・アクセスは、トランザクション・ユーザー ID によって許可されます。そのため、接続ファクトリーでユーザー ID およびパスワードを指定することはサポートされていません。
- CICS で MQ JMS を使用するすべてのアプリケーションでは、アプリケーションが実行されるたびに必ず、すべての JMS リソースを MQConnectionFactory から再作成する必要があります。つまり、セッション・インスタンス、メッセージ・コンシューマー、またはその他すべての MQ JMS オブジェクトを、アプリケーションの実行間で共用できるようアプリケーション・コードの静的変数に格納することはできません。この制約事項が存在するのは、CICS-MQ アダプターにより、キュー入力ハンドルなどのすべてのリソースが、これらを作成したトランザクションの完了時にタイディアップされるためです。これらのリソースのいずれかを同じトランザクションの別の実行または別のトランザクションで使用しようとすると、JMS 例外が発生します。
- JMS 仕様の観点から、IBM MQ classes for JMS では、JVM サーバーを、進行中の JTA トランザクションが常に存在する Java™ EE 準拠のアプリケーション・サーバーとして扱います。例えば、CICS では javax.jms.Session.commit() を呼び出すことができません。JMS 仕様では、JTA トランザクションが進行中の間は、JEE EJB または Web コンテナーでこれを呼び出せないことになっているためです。その結果、CICS での JMS API には制約事項が存在します。

従来の JMS API (JMS 1.1) には、以下の制約事項が適用されます。

- javax.jms.Connection.createConnectionConsumer(javax.jms.Destination, String, javax.jms.ServerSessionPool, int) は常に JMSException をスローします。
- javax.jms.Connection.createDurableConnectionConsumer(javax.jms.Topic, String, String, javax.jms.ServerSessionPool, int) は常に JMSException をスローします。
- 接続の既存のセッションがアクティブの場合、javax.jms.Connection.createSession の 3 つのバリアントすべては常に JMSException をスローします。

- javax.jms.Connection.createSharedConnectionConsumer(javax.jms.Topic, String, String, javax.jms.ServerSessionPool, int) は常に JMSException をスローします。
- javax.jms.Connection.createSharedDurableConnectionConsumer(javax.jms.Topic, String, String, javax.jms.ServerSessionPool, int) は常に JMSException をスローします。
- javax.jms.Connection.setClientID() は常に JMSException をスローします。
- javax.jms.Connection.setExceptionListener(javax.jms.ExceptionListener) は常に JMSException をスローします。
- javax.jms.Connection.stop() は常に JMSException をスローします。
- javax.jms.MessageConsumer.setMessageListener(javax.jms.MessageListener) は常に JMSException をスローします。
- javax.jms.MessageConsumer.getMessageListener() は常に JMSException をスローします。
- javax.jms.MessageProducer.send(javax.jms.Destination, javax.jms.Message,javax.jms.CompletionListener) は常に JMSException をスローします。
- javax.jms.MessageProducer.send(javax.jms.Destination, javax.jms.Message, int, int, long, javax.jms.CompletionListener) は常に JMSException をスローします。
- javax.jms.MessageProducer.send(javax.jms.Message, int, int, long, javax.jms.CompletionListener) は常に JMSException をスローします。
- javax.jms.MessageProducer.send (javax.jms.Message,javax.jms.CompletionListener) は常に JMSException をスローします。
- javax.jms.Session.run() は常に JMSRuntimeException をスローします。
- javax.jms.Session.setMessageListener(javax.jms.MessageListener) は常に JMSException をスローします。
- javax.jms.Session.getMessageListener() は常に JMSException をスローします。

単純化された JMS API (JMS 2.0) には、以下の制約事項が適用されます。

-
  - javax.jms.JMSContext.createContext(int) は常に JMSRuntimeException をスローします。
  - javax.jms.JMSContext.setClientID(String) は常に JMSRuntimeException をスローします。
  - javax.jms.JMSContext.setExceptionListener(javax.jms.ExceptionListener) は常に JMSRuntimeException をスローします。
  - javax.jms.JMSContext.stop() は常に JMSRuntimeException をスローします。
  - javax.jms.JMSProducer.setAsync(javax.jms.CompletionListener) は常に JMSRuntimeException をスローします。
  - javax.jms.JMSConsumer.getMessageListener() は常に JMSRuntimeException をスローします。

– javax.jms.JMSConsumer.setMessageListener(javax.jms.MessageListener) は常に JMSRuntimeException をスローします。

JMS 要求を処理中の CICS の異常終了

IBM MQ classes for JMS とバインディング・モードの転送を使用すると、IBM MQ MQI コマンドが発行されます。MQI コマンドの処理中に発生した CICS の異常終了は、Java 例外に変換されないため、CICS Java アプリケーションによってキャッチされません。

この状況では、CICS トランザクションが異常終了し、最後の同期点までロールバックされます。

OSGi JVM サーバーでの IBM MQ classes for Java の使用

OSGi JVM サーバーで実行される Java プログラムでは、IBM MQ によって提供される IBM MQ classes for Java を使用して、IBM MQ にアクセスできます。IBM MQ classes for Java では、Message Queue Interface (MQI) の Java バリアントが提供されます。このバリアントを使用すると、CICS アプリケーションで CICS によって保持される MQ 接続を使用して、キューに対してメッセージの書き込みと取得を行うことができます。CICS アプリケーションでの IBM MQ classes for Java に対するサポートは、IBM MQ for z/OS 7.1 から提供されます。

CICS 環境では、IBM MQ が提供するクラスは、バインディング・モードでの MQ への接続のみを許可します。クライアント・モードでリモート・キュー・マネージャーへの接続を使用しようとすると、例外が発生します。バインディング・モードでは、呼び出し要求が IBM MQ の MQI 呼び出しに変換され、既存の CICS-MQ アダプターによって通常どおりに処理されます。変換された要求は、他のプログラム (COBOL プログラムなど) からの MQI 要求とまったく同じ方式で CICS-MQ アダプターに到着します。そのため、IBM MQ にアクセスする Java プログラムと他のプログラムの間に、動作の違いはありません。

ご使用のアプリケーションで IBM MQ classes for Java を使用するには、以下を実行する必要があります。

- 開発環境で、MQ classes for Java にアクセスできることを確認します。ご使用のワークステーションに IBM MQ がインストールされていない場合は、IBM MQSupportPacs から入手してください。ここでは、IBM MQ クライアントを無料でダウンロードする資格があります。
- アプリケーションを CICS バンドル・プロジェクトに追加し、zFS にエクスポートして、JVM サーバーにインストールします。
- 適切なレベルの IBM MQ Java およびネイティブ・ライブラリーを使用して、CICS JVM サーバー環境を構成します。これらは、CICS STEPLIB に指定されている IBM MQ ライブラリーのレベルと一致している必要があります。詳細については、IBM MQ classes for Java をサポートするための OSGi JVM サーバーの構成を参照してください。

IBM MQ classes for Java については、IBM MQ 資料のIBM MQ classes for Java の使用を参照してください。クラスのリストは、IBM MQ の資料の IBM MQ

classes for JMS JavaDoc にあります。チュートリアルについては、CICS Developer Center: CICS OSGi JVM サーバーでの MQ JMS の使用を参照してください。

WebSphere MQ 要求に関わる作業単位のコミット

CICS JVM サーバー環境で IBM MQ classes for Java によって送受信されるメッセージには、常に CICS 作業単位 (UOW) が関連付けられます。

その UOW を完了するには、`com.ibm.cics.server.Task` オブジェクトのコミット・メソッドまたはロールバック・メソッドを呼び出すしかありません。あるいは、CICS タスクが正常に終了すれば、UOW は暗黙的にコミットされます。MQQueueManager に対するトランザクション制御メソッドの使用は、サポートされません。

混合言語アプリケーションの場合、非 Java プログラムから発行された **EXEC CICS SYNCPOINT** コマンドは、Java プログラムによる IBM MQ の更新を含め、作業単位全体をコミットします。

IBM MQ 要求を処理中の CICS の異常終了

IBM MQ classes for Java を使用すると、IBM MQ MQI コマンドが発行されます。MQI コマンドの処理中に発生した CICS の異常終了は、Java 例外に変換されないため、CICS Java アプリケーションによってキャッチされません。

この状況では、CICS トランザクションが異常終了し、最後の同期点までロールバックされます。

CICS 内の Java アプリケーションからの接続

CICS 環境内の Java プログラムは、TCP/IP ソケットをオープンし、外部プロセスと通信することができます。Java プログラムをゲートウェイとして使用すると、他の言語の CICS プログラムからは使用できない可能性がある他のエンタープライズ・アプリケーションに接続することができます。例えば、リモート・サーブレットまたはデータベースと通信する Java プログラムを作成できます。

この接続が CICS に統合されて、分散トランザクションや ID 伝搬などのエンタープライズ・サービス品質を提供する場合があります。また、CICS によって提供される分散トランザクションやその他のサービスなしに接続を使用できる場合もあります。必要な接続のタイプによっては、CICS で本来はサポートされないエンタープライズ・アプリケーションとの接続を可能にするサード・パーティー・ベンダー製品が使用できる場合があります。

一般に、CICS 環境における JVM の機能は、バッチ・モード JVM とほぼ同じです。バッチ・モード JVM は、CICS 環境の外部ではスタンドアロン・プロセスとして実行され、通常は、UNIX システム・サービスのコマンド行から、または JCL ジョブで開始されます。バッチ・モード JVM で作動可能な大部分のアプリケーションは、同じ範囲で CICS における JVM でも実行できます。例えば、サード・パーティーの JDBC ドライバーを使用して IBM 以外のデータベースと通信するバッチ・モード Java アプリケーションを作成する場合、同じアプリケーションがおそらく、CICS における JVM で作動します。ベンダー提供のコード (IBM 以外の

JDBC ドライバーなど) を CICS における JVM で使用したい場合は、ベンダーに問い合わせて、そのコードが CICS における JVM で実行されることをサポートするかどうかを判別してください。

CICS での Java アプリケーションの動作について詳しくは、 31 ページの『CICS での Java ランタイム環境』を参照してください。

CICS 環境における JVM で実行されるバッチ・モード・アプリケーションは、通常、CICS の機能を利用しません。例えば、CICS の Java プログラムが、サード・パーティーの JDBC ドライバーを使用して IBM 以外のデータベース内のレコードを更新する場合、CICS はこのアクティビティーを認識せず、現行の CICS トランザクションに更新を組み込もうとしません。

JCA ローカル ECI サポート

JCA ローカル ECI リソース・アダプターを使用するよう構成されている Liberty JVM サーバーに、JCA ECI アプリケーションをデプロイすることができます。このトピックは、CICS 統合モードの Liberty にのみ適用されます。

アプリケーションの開発について詳しくは、 113 ページの『Java EE コネクター・アーキテクチャー (JCA)』を参照してください。既存の CICS Transaction Gateway アプリケーションの移植の詳細については、 116 ページの『JCA ECI アプリケーションを Liberty JVM サーバーに移植する』を参照してください。JCA の構成については、 115 ページの『JCA ローカル ECI リソース・アダプターの構成』を参照してください。

CICS TS JCA ローカル ECI リソース・アダプターに備わっている JCA ECI プログラミング・インターフェースは、クラス定義から生成される Javadoc で記述されています。Javadoc は、JCA ローカル ECI Javadoc 情報から入手できます。

アプリケーション開発に必要なライブラリーと OSGi バンドルは、IBM CICS SDK for Java に備わっています。

JVM サーバーで実行する既存のアプリケーションのパッケージ化

プールされた JVM で Java アプリケーションを実行している場合、JVM サーバーで実行するようにそれらのアプリケーションを移動することができます。JVM サーバーは、同じ JVM 内で Java アプリケーションに対する複数の要求を処理できるので、同じワークロードの実行に必要な JVM 数を減らすことができます。

Java アプリケーションを 1 つ以上の OSGi バンドルとしてパッケージ化する必要があります。アプリケーションのパッケージ化に、3 つの方法の 1 つを使用できます。

既存の Java プロジェクトのプラグイン・プロジェクトへの変換

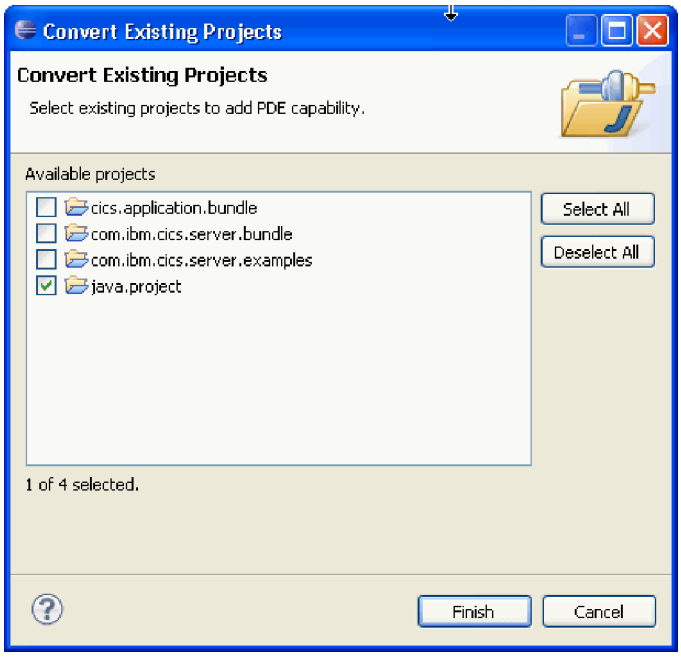
既存の Java プロジェクトがある場合は、OSGi プラグイン・プロジェクトに変換できます。OSGi バンドルは、プールされた JVM 環境および JVM サーバーで実行できます。

このタスクについて

このタスクは、ワークスペースに既存の Java プロジェクトがあり、OSGi プラグイン・プロジェクトに変換することを想定しています。

手順

1. 「パッケージ・エクスプローラー」ビューで、プラグイン・プロジェクトに変換する Java プロジェクトを右クリックして、「構成 **(Configure)**」 > 「プラグイン・プロジェクトに変換 **(Convert to Plug-in Projects)**」をクリックします。 「既存のプロジェクトを変換 (Convert Existing Projects)」ダイアログが表示されます。



   ダイアログには、ワークスペースのすべての Java プロジェクトのリストが含まれます。変換に選んだものが選択されます。選択内容を変更することも、複数の Java プロジェクトを選択してプラグイン・プロジェクトに変換することもできます。

2. 「終了」をクリックします。 Java プロジェクトがプラグイン・プロジェクトに変換されます。プロジェクト名は変更されませんが、これでプロジェクトにはマニフェスト・ファイルとビルド・プロパティー・ファイルが組み込まれました。

3. 必須: ここでプラグイン・マニフェスト・ファイルを編集して、JCICS API の依存関係を追加する必要があります。これらのステップを実行しない場合、バンドルをエクスポートしてインストールできますが、動作しません。

   注: CICS TS バージョン 4.2 より前の CICS では、Java クラス・ライブラリー `dfjcics.jar` を Java ビルド・パスに追加する必要がありました。 CICS TS バージョン 4.2 では、OSGi がユーザーの代わりにビルド・パスを管理します。次のステップを実行する前に、現在のビルド・パスを編集して `dfhjcics.jar` の参照をすべて削除する必要があります。 `dfhjcics.jar` へ参照をすべて削除しない場合、実行時に `NoSuchMethodException` エラーが発生します。

a. 「パッケージ・エクスプローラー」ビューで、プロジェクト名を右クリックして、「プラグイン・ツール」 > 「マニフェストを開く」をクリックします。 マニフェスト・ファイルがマニフェスト・エディターで開きます。

b. 重要: CICS TS バージョン 4.2 より前の CICS のバージョンでは、JCICS という Java クラス・ライブラリーが `dfjcics.jar` JAR ファイルの中に提供されています。 CICS TS バージョン 4.2 では、このライブラリーは `com.ibm.cics.server.jar` ファイル内に提供されています。プロジェクトのマニフェストに宣言 **`Import-Package: dfhjcics.jar;`** が含まれている場合は、この宣言を削除してから残りのステップを続行する必要があります。

c. 「依存関係 **(Dependencies)**」タブを選択して、「インポートされたパッケージ (Imported Packages)」セクションで、「追加」をクリックします。 「パッケージの選択 (Package Selection)」ダイアログが開きます。

d. パッケージ `com.ibm.cics.server` を選択して、「**OK**」をクリックします。 パッケージが「インポートされたパッケージ (Imported Packages)」リストに表示されます。

e. オプション: アプリケーションで必要な場合には、前述のステップを繰り返して、以下のパッケージをインストールします。

   `com.ibm.record`
   VisualAge に付属の Java レコード・フレームワークから IByteBuffer を使用するレガシー・プログラム用の Java API。以前は `dfjcics.jar` ファイル内にありました。

f. 「ファイル」 > 「保管」を選択してマニフェスト・ファイルを保管します。

タスクの結果

既存の Java プロジェクトのプラグイン・プロジェクトへの変換が正常に完了しました。

次のタスク

この時点で、マニフェスト・ファイルを更新して CICS-MainClass 宣言を追加する必要があります。詳しくは、関連リンクを参照してください。

JAR ファイルの内容の OSGi プラグイン・プロジェクトへのインポート

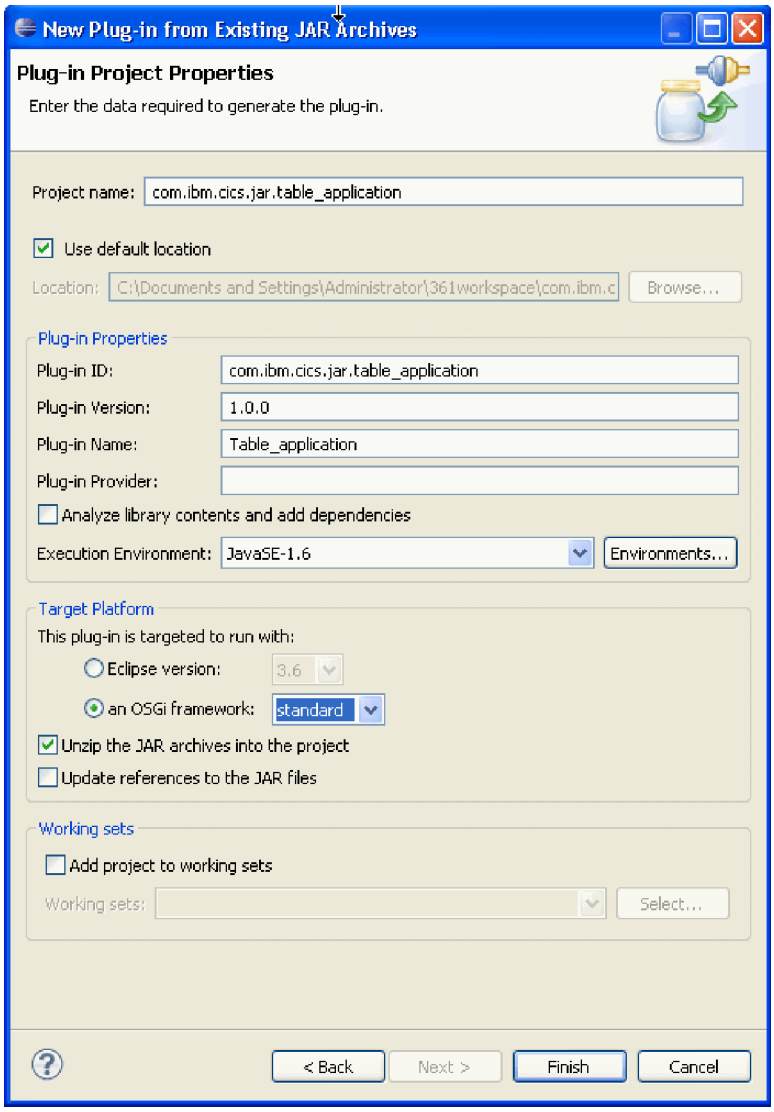
既存の JAR ファイルからプラグイン・プロジェクトを作成できます。この方法は、アプリケーションが既にスレッド・セーフであり、リファクタリングや再コンパイルが必要ない場合に役立ちます。OSGi バンドルは、プールされた JVM 環境および JVM サーバーで実行できます。

このタスクについて

このタスクでは、既存の JAR ファイルから新しい OSGi プラグイン・プロジェクトを作成します。 JAR ファイルは、ローカル・ファイル・システム上に存在している必要があります。

手順

1. Eclipse メニュー・バーで、「ファイル」 > 「新規」 > 「プロジェクト **(Project)**」をクリックして、新規ウィザードを開きます。
2. 「プラグイン開発」フォルダーを展開して、「既存の **JAR** アーカイブからのプラグイン **(Plug-in from Existing JAR Archives)**」をクリックします。「次へ」をクリックします。 「JAR の選択 (JAR selection)」ダイアログが開きます。
3. 変換する JAR ファイルを見つけます。ファイルが Eclipse ワークスペース内にある場合は、「追加」をクリックします。ファイルがコンピューター上のフォルダーにある場合は、「外部を追加 **(Add External)**」をクリックして、JAR ファイルを参照します。必要なファイルを選択して「オープン」をクリックし、「JAR の選択 (Jar selection)」ダイアログで追加します。「次へ」をクリックします。 「プラグイン・プロジェクトのプロパティー (Plug-in Project Properties)」ダイアログが開きます。



4. 「プロジェクト名」フィールドで、作成するプロジェクトの名前を入力します。プロジェクト名は必須です。

5. 必要に応じて、「プラグインのプロパティー (Plug-in Properties)」セクションの次のフィールドを入力します。

   プラグイン **ID**
   : プラグイン ID はプロジェクト名から自動的に生成されます。ただし、必要に応じて ID を変更できます。

   プラグイン名
   : プラグイン名はプロジェクト名から自動的に生成されます。ただし、必要に応じて名前を変更できます。

   実行環境
   : このフィールドは、プラグインの実行に必要な JRE の最小レベルを指定します。 CICS ランタイム・ターゲット・プラットフォームの実行環境に一致する Java のレベルを選択します。

6. 「ターゲット・プラットフォーム」セクションで、「**OSGI** フレームワーク **(an OSGI framework)**」を選択して、メニューから「標準」を選択します。
7. 「**JAR** アーカイブをプロジェクトに **unzip** する **(Unzip the JAR archives into the project)**」が選択されていることを確認して、「終了」をクリックします。 Eclipse がプラグイン・プロジェクトをワークスペースに作成します。
8. 必須: ここでプラグイン・マニフェスト・ファイルを編集して、JCICS API の依存関係を追加する必要があります。これらのステップを実行しない場合、バンドルをエクスポートしてインストールできますが、動作しません。
   a. 「パッケージ・エクスプローラー」ビューで、プロジェクト名を右クリックして、「プラグイン・ツール」 > 「マニフェストを開く」をクリックします。 マニフェスト・ファイルがマニフェスト・エディターで開きます。
   b. 「依存関係 **(Dependencies)**」タブを選択して、「インポートされたパッケージ (Imported Packages)」セクションで、「追加」をクリックします。 「パッケージの選択 (Package Selection)」ダイアログが開きます。
   c. パッケージ `com.ibm.cics.server` を選択して、「**OK**」をクリックします。 パッケージが「インポートされたパッケージ (Imported Packages)」リストに表示されます。
   d. オプション: アプリケーションで必要な場合には、前述のステップを繰り返して、以下のパッケージをインストールします。

      `com.ibm.record`
      : VisualAge に付属の Java レコード・フレームワークから IByteBuffer を使用するレガシー・プログラム用の Java API。以前は `dfjcics.jar` ファイル内にありました。

   e. 「ファイル」 > 「保管」を選択してマニフェスト・ファイルを保管します。

タスクの結果

既存の JAR ファイルから OSGi プラグイン・プロジェクトを作成しました。

次のタスク

この時点で、マニフェスト・ファイルを更新して CICS-MainClass 宣言を追加する必要があります。詳しくは、関連リンクを参照してください。

バイナリー JAR ファイルの OSGi プラグイン・プロジェクトへのインポート

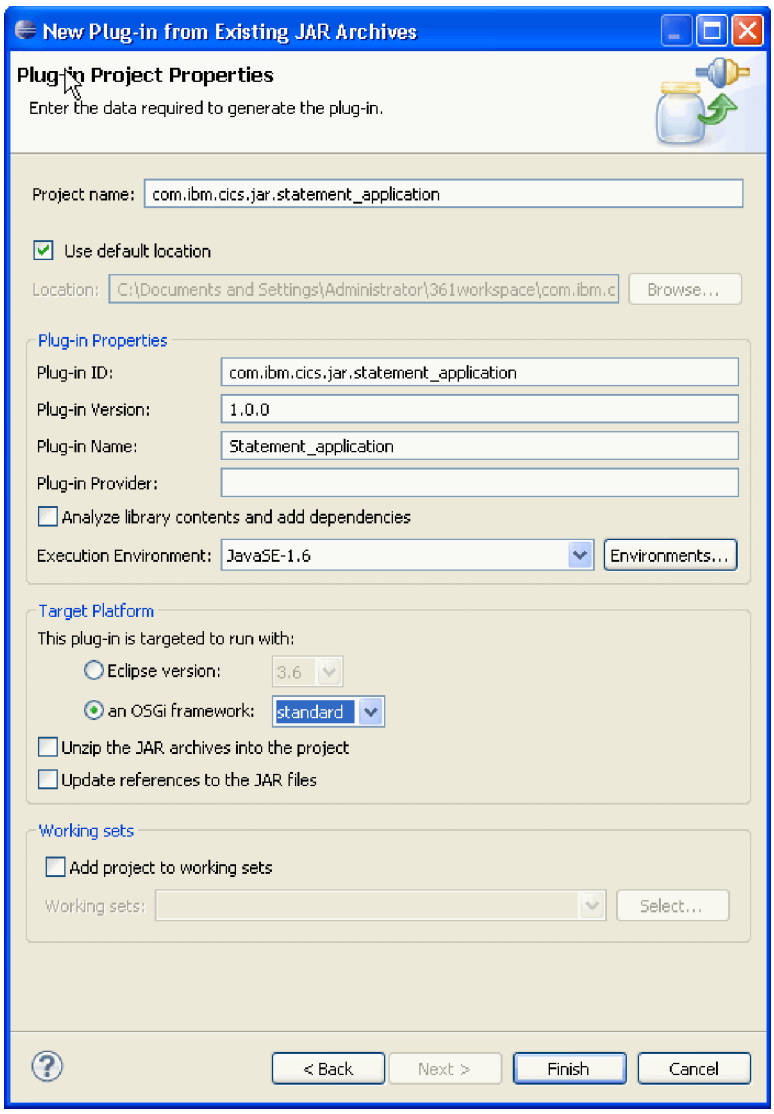
既存のバイナリー JAR ファイルからプラグイン・プロジェクトを作成できます。この方式は、ライセンスの制限がある状態、またはバイナリー・ファイルを抽出できない状態で便利です。ただし、JAR ファイルを含む OSGi バンドルは、プールされた JVM 環境ではサポートされません。

このタスクについて

このタスクでは、既存のバイナリー JAR ファイルから新しい OSGi プラグイン・プロジェクトを作成します。 JAR ファイルは、ローカル・ファイル・システム上に存在している必要があります。

手順

1. Eclipse メニュー・バーで、「ファイル」 > 「新規」 > 「プロジェクト **(Project)**」をクリックして、新規ウィザードを開きます。
2. 「プラグイン開発」フォルダーを展開して、「既存の **JAR** アーカイブからのプラグイン **(Plug-in from Existing JAR Archives)**」をクリックします。「次へ」をクリックします。 「JAR の選択 (JAR selection)」ダイアログが開きます。
3. 変換する JAR ファイルを見つけます。ファイルが Eclipse ワークスペース内にある場合は、「追加」をクリックします。ファイルがコンピューター上のフォルダーにある場合は、「外部を追加 **(Add External)**」をクリックして、JAR ファイルを参照します。必要なファイルを選択して「オープン」をクリックし、「JAR の選択 (Jar selection)」ダイアログで追加します。「次へ」をクリックします。 「プラグイン・プロジェクトのプロパティー (Plug-in Project Properties)」ダイアログが開きます。



4. 「プロジェクト名」フィールドで、作成するプロジェクトの名前を入力します。プロジェクト名は必須です。
5. 必要に応じて、「プラグインのプロパティー (Plug-in Properties)」セクションの次のフィールドを入力します。

   プラグイン **ID**
   : プラグイン ID はプロジェクト名から自動的に生成されます。ただし、必要に応じて ID を変更できます。

   プラグイン名
   : プラグイン名はプロジェクト名から自動的に生成されます。ただし、必要に応じて名前を変更できます。

   実行環境
   : このフィールドは、プラグインの実行に必要な JRE の最小レベルを指定します。 CICS ランタイム・ターゲット・プラットフォームの実行環境に一致する Java のレベルを選択します。

6. 「ターゲット・プラットフォーム」セクションで、「**OSGI** フレームワーク **(an OSGI framework)**」を選択して、メニューから「標準」を選択します。
7. 「**JAR** アーカイブをプロジェクトに **unzip** する **(Unzip the JAR archives into the project)**」が選択されていないことを確認して、「終了」をクリックします。 Eclipse がプラグイン・プロジェクトをワークスペースに作成します。プロジェクトにはバイナリー JAR ファイルが含まれますが、プロジェクトはプールされた JVM 環境ではサポートされません。
8. 必須: ここでプラグイン・マニフェスト・ファイルを編集して、JCICS API の依存関係を追加する必要があります。これらのステップを実行しない場合、バンドルをエクスポートしてインストールできますが、動作しません。
   a. 「パッケージ・エクスプローラー」ビューで、プロジェクト名を右クリックして、「プラグイン・ツール」 > 「マニフェストを開く」をクリックします。 マニフェスト・ファイルがマニフェスト・エディターで開きます。
   b. 「依存関係 **(Dependencies)**」タブを選択して、「インポートされたパッケージ (Imported Packages)」セクションで、「追加」をクリックします。「パッケージの選択 (Package Selection)」ダイアログが開きます。
   c. パッケージ `com.ibm.cics.server` を選択して、「**OK**」をクリックします。 パッケージが「インポートされたパッケージ (Imported Packages)」リストに表示されます。
   d. オプション: アプリケーションで必要な場合には、前述のステップを繰り返して、以下のパッケージをインストールします。

      `com.ibm.record`
      VisualAge に付属の Java レコード・フレームワークから IByteBuffer を使用するレガシー・プログラム用の Java API。以前は `dfjcics.jar` ファイル内にありました。
   e. 「ファイル」 > 「保管」を選択してマニフェスト・ファイルを保管します。

タスクの結果

ワークスペースでのプラグイン・プロジェクトの作成が正常に完了しました。

次のタスク

この時点で、マニフェスト・ファイルを更新して CICS-MainClass 宣言を追加する必要があります。詳しくは、関連リンクを参照してください。

第 3 章 JVM サーバーへのアプリケーションのデプロイ

Java アプリケーションを JVM サーバーにデプロイするには、そのアプリケーションを正常にインストールして実行するために適切にパッケージ化する必要があります。アプリケーションのパッケージ化とデプロイには、IBM CICS SDK for Java を使用できます。

Java アプリケーションをデプロイするには、以下のようないくつかのオプションがあります。

- OSGi フレームワークを実行する JVM サーバーの中に、アプリケーションの OSGi バンドルを含む 1 つ以上の CICS バンドルをデプロイする。
- Liberty JVM サーバーの中に、1 つ以上の WAR ファイルを含む 1 つ以上の CICS バンドルをデプロイする。
- Liberty JVM サーバーの中に、Enterprise Bundle Archive (EBA) ファイルを含む 1 つ以上の CICS バンドルをデプロイする。
- EAR ファイルを含む 1 つ以上の CICS バンドルを Liberty JVM サーバーにデプロイする。
- プラットフォームに、CICS バンドルと OSGi バンドルで構成されるアプリケーション・バンドルをデプロイする。

JVM サーバーにおける OSGi バンドルのデプロイ

JVM サーバーに Java アプリケーションを配置するには、ターゲット JVM サーバーの OSGi フレームワークにそのアプリケーションの OSGi バンドルをインストールする必要があります。

始める前に

アプリケーションの OSGi バンドルを含む CICS バンドルは、zFS にデプロイされる必要があります。ターゲット JVM サーバーが CICS 領域で有効になっている必要があります。

このタスクについて

CICS バンドルには、1 つ以上の OSGi バンドルを含むことができます。 CICS バンドルは配置の単位であるため、すべての OSGi バンドルは、BUNDLE リソースの一部として一緒に管理されます。また、OSGi フレームワークは、依存関係とバージョン管理方式の管理を含めて、OSGi バンドルのライフサイクルを管理します。

1 つの Java アプリケーション・コンポーネントを構成するすべての OSGi バンドルを、必ず同じ CICS バンドル内にデプロイしてください。 OSGi バンドル相互間に依存関係がある場合は、それらを同じ CICS バンドルに配置してください。CICS BUNDLE リソースをインストールする際に、CICS によって、OSGi バンドル間のすべての依存関係が解決されていることが確認されます。

© Copyright IBM Corp. 1999, 2018

共通コードのライブラリーを含む OSGi バンドルへの依存関係がある場合、そのライブラリー用に 1 つの別個の CICS バンドルを作成してください。この場合、そのライブラリーを含む CICS BUNDLE リソースを最初にインストールすることが重要です。 Java アプリケーションをインストールしてから、その Java アプリケーションが依存する CICS バンドルをインストールすると、OSGi フレームワークは、その Java アプリケーションの依存関係を解決できません。

OSGi バンドルを含む CICS バンドルを Liberty JVM サーバーの中にインストールしようと試みないでください。このような構成はサポートされていません。その代わりに、OSGi バンドルを Web アプリケーションと一緒にエンタープライズ・バンドル・アーカイブ (EBA) にパッケージ化できます。または WebSphere Liberty プロファイル・バンドル・リポジトリーを使用して、Liberty JVM サーバー内のすべての Web アプリケーションに対して OSGi バンドルを使用可能にすることもできます。

手順

1. zFS 内のバンドルのディレクトリーを指定する BUNDLE リソースを作成します。
   a. CICS SM パースペクティブで、CICS Explorer メニュー・バーの「定義」 > 「バンドル定義」をクリックして、「バンドル定義」ビューを開きます。
   b. そのビュー内の任意の場所を右クリックし、「**New**」をクリックして「New Bundle Definition」ウィザードを開きます。 そのウィザードのフィールドに、BUNDLE リソースの詳細を入力してください。
   c. BUNDLE リソースをインストールします。 リソースを Enabled 状態または Disabled 状態のどちらかでインストールできます。
      - DISABLED 状態でリソースをインストールすると、CICS は OSGi バンドルをフレームワークにインストールし、依存関係を解決しますが、バンドルを開始しようとしません。
      - ENABLED 状態でリソースをインストールすると、CICS は OSGi バンドルをインストールし、依存関係を解決し、OSGi バンドルを開始します。遅延的なバンドル・アクティベーターが OSGi バンドルに含まれる場合、別の OSGi バンドルによって呼び出されるまでは、OSGi フレームワークはバンドルを開始しようとしません。
2. オプション: BUNDLE リソースがまだ ENABLED 状態でない場合、そのリソースを使用可能にして、フレームワークで OSGi バンドルを開始します。
3. CICS Explorer メニュー・バーの「**Operations**」 > 「**Bundles**」をクリックして、「Bundles」ビューを開きます。BUNDLE リソースの状態を確認します。
   - BUNDLE リソースが ENABLED 状態である場合、CICS はバンドル内のすべてのリソースを正常にインストールできました。
   - BUNDLE リソースが DISABLED 状態である場合、CICS はバンドル内の 1 つ以上のリソースをインストールできませんでした。

   BUNDLE リソースが ENABLED 状態でインストールできなかった場合、BUNDLE リソースのバンドル・パーツを確認してください。いずれかのバンドル・パーツが UNUSABLE 状態である場合、CICS は OSGi バンドルを作成で

きませんでした。通常、この状態は、zFS で CICS バンドルに問題があることを示します。 その BUNDLE リソースを破棄し、問題を修正してから、BUNDLE リソースを再度インストールする必要があります。

4. CICS Explorer メニュー・バーで「**Operations (操作)**」 > 「**Java**」 > 「**OSGi Bundles (OSGi バンドル)**」をクリックして、「OSGi Bundles (OSGi バンドル)」ビューを開きます。 OSGi フレームワークにインストールされた OSGi バンドルおよびサービスの状態を確認します。
   - OSGi バンドルが STARTING 状態である場合、バンドル・アクティベーターが呼び出されましたが、まだ戻っていません。OSGi バンドルに遅延活動化ポリシーがある場合、OSGi フレームワークで呼び出されるまで、そのバンドルはこの状態のままです。
   - OSGi バンドルと OSGi サービスがアクティブである場合、Java アプリケーションは作動可能です。
   - OSGi サービスが非アクティブである場合、CICS は、その名前を持つ OSGi サービスが OSGi フレームワークに既に存在することを検出した可能性があります。
   - BUNDLE リソースを使用不可にすると、OSGi バンドルは RESOLVED 状態に移ります。
   - OSGi バンドルが INSTALLED 状態である場合、OSGi バンドル内の依存関係を解決できなかったために、このバンドルは開始されなかったか開始に失敗しました。
5. 182 ページの『JVM サーバー内の Java アプリケーションの呼び出し』

タスクの結果

BUNDLE が使用可能になり、OSGi バンドルが OSGi フレームワークに正常にインストールされ、すべての OSGi サービスがアクティブです。OSGi バンドルは、フレームワーク内の他のバンドルから使用可能です。

次のタスク

OSGi フレームワークの外部にある他の CICS アプリケーションから Java アプリケーションを使用可能にすることができます。それには、 182 ページの『JVM サーバー内の Java アプリケーションの呼び出し』の説明に従ってください。

CICS バンドル内の Java EE アプリケーションの Liberty JVM サーバーへのデプロイ

Liberty JVM サーバーに、CICS バンドルとしてパッケージされている Java EE アプリケーションをデプロイすることができます。

始める前に

WAR ファイル、EAR ファイルまたは EBA ファイルのいずれの形式の Java EE アプリケーションも、zFS に CICS バンドルとしてデプロイする必要があります。ターゲット JVM サーバーが CICS 領域で有効になっている必要があります。

Java アプリケーションの作成および再パッケージ化に関する一般情報については、32 ページの『IBM CICS SDK for Java を使用したアプリケーションの開発』を参照してください。

共通コードのライブラリーを含んでいる OSGi バンドルに依存している場合は、そのバンドルを Liberty バンドル・リポジトリーにインストールします。 175 ページの『JVM サーバーにおける OSGi バンドルのデプロイ』を参照してください。

このタスクについて

CICS アプリケーション・モデルでは、Java アプリケーション・コンポーネントが CICS バンドルにパッケージされて zFS にデプロイされます。 CICS バンドルをインストールすることによって、アプリケーション・コンポーネントのライフサイクルを管理できます。

Java EE アプリケーションには、次のものを含めることができます。

- アプリケーションのプレゼンテーション層およびビジネス・ロジックを提供する 1 つ以上の WAR ファイル
- EBA ファイルにエクスポートされた 1 つの OSGi アプリケーション・プロジェクト (その中にはプレゼンテーション層を提供する 1 つの Web 対応 OSGi バンドル・プロジェクトと、ビジネス・ロジックを提供する追加の OSGi バンドル・セットが含まれる)
- プレゼンテーション層およびビジネス・ロジックを提供する 1 つ以上の WAR ファイルが含まれたエンタープライズ・アプリケーション・アーカイブ (EAR) ファイル

手順

1. zFS 内のバンドルのディレクトリーを指定する BUNDLE リソースを作成します。
   a. CICS Explorer の CICS SM パースペクティブで、CICS Explorer メニュー・バーの「定義」 > 「バンドル定義」をクリックして、「バンドル定義」ビューを開きます。
   b. そのビュー内の任意の場所を右クリックし、「**New**」をクリックして「New Bundle Definition」ウィザードを開きます。 そのウィザードのフィールドに、BUNDLE リソースの詳細を入力してください。
   c. BUNDLE リソースをインストールします。 以下に示すように、使用可能または使用不可の状態でリソースをインストールできます。
      - DISABLED 状態でリソースをインストールすると、CICS は Java EE アプリケーションを Liberty サーバーにインストールしようとしません。
      - ENABLED 状態でリソースをインストールすると、CICS は、Java EE アプリケーション (WAR ファイル、EAR ファイル、EBA ファイル) を `${server.output.dir}/installedApps` ディレクトリーにインストールし、`<application>` 項目を `${server.output.dir}/installedApps.xml` に追加します。

2. オプション: BUNDLE リソースがまだ ENABLED 状態でない場合は、そのリソースを有効にして、Liberty サーバーで Java EE アプリケーションを開始します。
3. CICS Explorer メニュー・バーの「**Operations**」 > 「**Bundles**」をクリックして、「Bundles」ビューを開きます。BUNDLE リソースの状態を確認します。
   - BUNDLE リソースが ENABLED 状態である場合、CICS はバンドル内のすべてのリソースを正常にインストールし、バンドルに含まれるすべての Liberty アプリケーションが開始されました。
   - BUNDLE リソースが ENABLING 状態である場合、バンドル内のすべてのリソースを CICS が現在インストール中であるか、バンドルに含まれる 1 つ以上の Liberty アプリケーションがまだインストール中または開始中です。
   - BUNDLE リソースが DISABLED 状態である場合、CICS はバンドル内の 1 つ以上のリソースをインストールできませんでした。この状況は、バンドルに含まれる Liberty アプリケーションが開始に失敗したか、タイムアウト前にアプリケーションが Liberty をインストールしなかった場合に発生する可能性があります。タイムアウトは JVM システム・プロパティー `com.ibm.cics.jvmserver.wlp.bundlepart.timeout` によって構成されます。

   BUNDLE リソースが ENABLED 状態でインストールできなかった場合、BUNDLE リソースのバンドル・パーツを確認してください。いずれかのバンドル・パーツが UNUSABLE 状態である場合、問題の原因を説明するメッセージが発行されます。例えば、この状態は、zFS で CICS バンドルに問題がある、あるいは関連付けられている JVMSERVER リソースが利用不可であることを示します。その BUNDLE リソースを破棄し、報告された問題を解決してから、BUNDLE リソースを再度インストールする必要があります。
4. オプション: アプリケーション・トランザクションで Java EE アプリケーション要求を実行するために、URIMAP および TRANSACTION リソースを作成できます。 アプリケーションに対するセキュリティーを制御する場合は、URI マップの定義が役に立ちます。 URI を特定のトランザクションにマップして、トランザクション・セキュリティーを使用できるからです。通常、これらのリソースは CICS バンドルの一部として作成され、アプリケーションによって管理されます。しかし、それらのリソースを別に定義することが望ましい場合は、そうすることもできます。
   a. PROGRAM 属性を DFHSJTHP に設定するアプリケーション用に TRANSACTION リソースを作成します。 この CICS プログラムは、Liberty JVM サーバーに対するインバウンド Java EE 要求のセキュリティー検査を処理します。何らかのリモート属性を設定しても、それらは CICS によって無視されます。トランザクションは常にローカル CICS 領域と接続されている必要があるからです。
   b. JVMSERVER のタイプが USAGE である URIMAP リソースを作成します。 TRANSACTION 属性をアプリケーション・トランザクションの名前に設定し、SCHEME 属性を HTTP または HTTPS に設定します。USERID 属性を使用してユーザー ID を設定することもできます。アプリケーションのセキュリティー認証メカニズムを使用する場合、この値は無視

されます。認証が行われず、URI マップにユーザー ID が設定されていない場合、デフォルトの CICS のユーザー ID で処理は実行されます。

タスクの結果

CICS リソースが有効になり、Java EE アプリケーションが Liberty JVM サーバーに正常にインストールされます。

次のタスク

Web クライアントを介して Java アプリケーションの使用可能性をテストすることができます。アプリケーションを更新または削除する場合は、Java アプリケーションの管理を参照してください。

Liberty JVM サーバーへの Java EE アプリケーションの直接デプロイ

Java EE アプリケーションをデプロイするには、`server.xml` 内の `application` エレメントを定義するか、アプリケーションを以前に定義した `dropins` ディレクトリーにコピーします。

始める前に

JVM サーバーは、Liberty テクノロジーを使用するように構成する必要があります。

このタスクについて

Java EE アプリケーションは、Web アーカイブ (WAR)、エンタープライズ・バンドル・アーカイブ (EBA)、またはエンタープライズ・アプリケーション・アーカイブ (EAR) としてパッケージ化できます。

Liberty には、Java EE アプリケーションをインストールするための次のような 2 つの方法が用意されています。

- `application` エレメントを `server.xml` に追加できます。
- あるいは、アプリケーションを Liberty JVM サーバーの `dropins` ディレクトリーにコピーすることもできます。 `dropins` を使用すると、CICS は常にトランザクション CJSA の下で実行され、CICS セキュリティーなどの追加のサービス品質による利点は得られません。

注:

- 両方の手法を使用して、同じアプリケーションを同じ JVM サーバーにデプロイしないようにしてください。
- CICS 自動構成で提供されるデフォルトを受け入れた場合、`dropins` ディレクトリーは自動的に作成されません。

手順

- サーバー構成ファイルに追加することでアプリケーションをデプロイするには、次のようにします。

  `server.xml` にアプリケーションの以下の属性を構成する必要があります。

- id - 固有でなければなりません。サーバーによって内部で使用されます。
- name - 固有でなければなりません。
- type - アプリケーションのタイプを指定します。サポートされるタイプは、WAR、EBA、および EAR です。
- location - アプリケーションの場所を指定します。この場所は、絶対パスまたは URL になります。

以下に例を示します。

```
<application
    id="com.ibm.cics.server.examples.wlp.tsq.app"
    name="com.ibm.cics.server.examples.wlp.tsq.app"
    type="eba"
    location="${server.output.dir}/path_to_app"/>
```

- **dropins** ディレクトリーを作成して、そこにアプリケーションをデプロイするには、次のようにします。
  1. dropins を使用可能にするには、以下の例と同様な構成を server.xml に追加する必要があります。

     ```
     <applicationMonitor dropins="dropins" dropinsEnabled="true" pollingRate="5s"
     updateTrigger="disabled"/>
     ```

     詳しくは、Controlling dynamic updatesを参照してください。
  2. FTP を使用して、エクスポートしたファイルをバイナリー・モードで dropins ディレクトリーに転送します。ディレクトリー・パスは WLP_USER_DIR/servers/server_name/dropins です。 server_name は com.ibm.cics.jvmserver.wlp.server.name プロパティーの値です。プロパティーが設定されていない場合、このプロパティーは defaultServer になります。

タスクの結果

Liberty JVM サーバーによってアプリケーションがインストールされます。

次のタスク

Web ブラウザーから Java EE アプリケーションにアクセスして、Java EE アプリケーションが正常に実行されていることを確認します。アプリケーション・ファイルを削除するには、dropins ディレクトリーから WAR、EBAまたは EAR ファイルを削除します。アプリケーション・エレメントと一緒にデプロイされている場合は、そのエレメントを server.xml から削除します。

Liberty JVM サーバーへの共通ライブラリーのデプロイ

DLL ファイル、JAR ファイル、または OSGi バンドルのどちらとして提供されているかに応じて、共通ライブラリーをデプロイします。

手順

- DLL ファイルとして提供される共通ライブラリーの場合は、JVM プロファイルの LIBPATH_SUFFIX オプションによって参照されているディレクトリーにファイルをコピーします。

LIBPATH_PREFIX および LIBPATH_SUFFIX について詳しくは、JVM プロファイルに使用するシンボルを参照してください。

- OSGi バンドルの JAR ファイルとして提供される共通ライブラリーの場合は、`server.xml` ファイルの bundleRepository 定義で参照されているディレクトリーに JAR ファイルをコピーします。

  詳しくは、server.xml の手動調整の『バンドル・リポジトリー』を参照してください。

- JAR ファイルとして提供されるが、OSGi バンドルではない共通ライブラリーの場合は、`server.xml` ファイルのグローバル・ライブラリー定義で参照されているディレクトリーに JAR ファイルをコピーします。

  詳しくは、server.xml の手動調整の『グローバル/共用ライブラリー』を参照してください。

JVM サーバー内の Java アプリケーションの呼び出し

JVM サーバーで稼働している Java アプリケーションを呼び出す方法は多数あります。使用される方式は JVM サーバーの特性によって異なります。

このタスクについて

特定の URL を指定した HTTP 要求を使用することにより、Liberty JVM サーバーで実行される Web アプリケーションを呼び出すことができます。 **EXEC CICS LINK** または **EXEC CICS START** から直接 Web アプリケーションを起動することはできません。Plain Old Java Object (POJO) として実装され、WAR や EAR にパッケージ化された Java EE アプリケーションがある場合は、**EXEC CICS LINK** または **EXEC CICS START** を使用してそれらのアプリケーションのビジネス・ロジック・コンポーネントを呼び出すことができます。

OSGi JVM サーバーで実行されている Java アプリケーションを呼び出す場合は、Java で定義されている `PROGRAM` に対して **EXEC CICS LINK** を実行するか、ターゲット `PROGRAM` が Java で定義されている `TRANSACTION` に対して **EXEC CICS START** を実行できます。 `PROGRAM` 定義では、JVMSERVER と、呼び出す CICS 生成 OSGi サービスの名前を指定します。このようなリンク可能な OSGi サービスは、マニフェストに `CICS-MainClass` ヘッダーが含まれている OSGi バンドルのインストール時に、CICS によって作成されます。 `CICS-MainClass` ヘッダーには、アプリケーションへのエントリー・ポイントとして機能させる、OSGi バンドル内の Java クラスの main メソッドが指定されています。

OSGi サービスは、OSGi フレームワークに登録されている明確に定義されたインターフェースです。 OSGi バンドルやリモート・アプリケーションは OSGi サービスを使用して、OSGi バンドルにパッケージされているアプリケーション・コードを呼び出します。 OSGi バンドルは複数の OSGi サービスをエクスポートできます。詳しくは、 OSGi JVM サーバーの OSGi バンドルの更新を参照してください。

クラスパス・ベースの JVM サーバーでの Java 関数の呼び出しは、通常、バッチ、Axis2 および SAML などの JVM サーバー固有の機能の一部として実行されます。これらの機能用に、DFHSJJI ベンダー・インターフェースが提供されています。

手順

- Web アーカイブ (WAR) ファイル、エンタープライズ・アーカイブ (EAR) ファイル、または、Web アプリケーション・バンドル (WAB) を含み Liberty JVM サーバーで実行されるエンタープライズ・バンドル・アーカイブ (EBA) ファイルとして開発された Web アプリケーションの場合は、URL を使用してクライアントのブラウザーからアプリケーションを呼び出します。 Java EE アプリケーションのビジネス・ロジック・コンポーネントの呼び出しについて詳しくは、 89 ページの『CICS プログラムで呼び出すための Java EE アプリケーションの準備』を参照してください。
- OSGi JVM サーバーにデプロイされている OSGi バンドルの場合は、以下のステップに従ってください。
  1. OSGi フレームワークで使用したい、アクティブな OSGi サービスのシンボル名を判別します。 CICS Explorer で「**Operations**」 > 「**Java**」 > 「**OSGi Services**」をクリックして、アクティブな OSGi サービスをリストします。
  2. 他の CICS アプリケーションに対して OSGi サービスを表す PROGRAM リソースを作成します。
     - JVM 属性で、`YES` を指定して、プログラムが Java プログラムであることを示します。
     - JVMCLASS 属性で、OSGi サービスのシンボル名を指定します。この値は大/小文字の区別があります。
     - JVMSERVER 属性で、OSGi サービスが実行される JVMSERVER リソースの名前を指定します。
  3. 以下の 2 つの方法で Java アプリケーションを呼び出すことができます。
     - トランザクション ID を指定する 3270 または **`EXEC CICS START`** 要求を使用します。OSGi サービスの PROGRAM リソースを定義する TRANSACTION リソースを作成します。
     - **`EXEC CICS LINK`** 要求、ECI 呼び出し、または EXCI 呼び出しを使用します。要求をコーディングする際に、OSGi サービスの `PROGRAM` リソースの名前を指定します。
- Axis2 または SAML 機能については、Axis2 用の JVM サーバーの構成および SAML 用の CICS の構成を参照してください。

タスクの結果

他のコンポーネントが Java アプリケーションを使用できるようにするための定義を作成しました。 CICS は、ターゲット JVM サーバーで要求を受け取ると、指定された Java クラスまたは Web アプリケーションを新しい CICS Java スレッドで呼び出します。関連付けられた OSGi サービスまたは Web アプリケーションが登録されていないか、非アクティブである場合、呼び出し側プログラムにエラーが戻されます。

CICS の非 OSGi Java アプリケーションの配置

Java アプリケーションは、CICS バンドルに組み込むことができ、z/OS UNIX System Services (z/OS UNIX) ファイル・システムに CICS Explorer から直接配置できます。エクスポートされたバンドルには、CICS で使われるアプリケーションの JAR ファイルが含まれます。

このタスクについて

このタスクでは、非 OSGi Java アプリケーションを配置する手順の概要を示します。 OSGi アプリケーションの場合と同じ手順ですが、唯一の違いは、CICS がバンドルの代わりにアプリケーション JAR ファイルを使用するという点です。 z/OS ファイル・システムに直接バンドルを配置する権限がない場合、バンドルを圧縮ファイルとしてエクスポートできます。詳しくは、CICS Explorer 製品資料内の『ローカル・ファイル・システムへの CICS バンドル・プロジェクトのエクスポート』を参照してください。

手順

1. Java アプリケーションをプラグイン・プロジェクトに変換します。 167 ページの『既存の Java プロジェクトのプラグイン・プロジェクトへの変換』の説明に従ってください。
2. プラグイン・プロジェクトを CICS バンドルに追加します。 39 ページの『CICS バンドル・プロジェクトへのプロジェクトの追加』の説明に従ってください。
3. バンドル・プロジェクトを z/OS UNIX ファイル・システムにデプロイします。 CICS Explorer 製品資料内の『CICS バンドルのデプロイ』の説明に従ってください。

タスクの結果

Java アプリケーションが z/OS UNIX にエクスポートされます。エクスポートされたバンドルには、アプリケーションの JAR ファイルが含まれます。

第 4 章 Java サポートのセットアップ

基本的なセットアップ・タスクを実行して、CICS 領域で Java をサポートし、Java アプリケーションを実行するように JVM サーバーを構成します。

始める前に

CICS に必要な Java コンポーネントは、製品のインストール時にセットアップされます。Java コンポーネントが正しくインストールされていることを確認する必要があります。

このタスクについて

CICS は z/OS UNIX のファイルを使用して JVM を開始します。 CICS 領域が正しい zFS ディレクトリーを使用するように構成されており、それらのディレクトリーに正しい権限があることを確認する必要があります。 CICS を構成して zFS をセットアップした後、Java アプリケーションを実行するように JVM サーバーを構成できます。

手順

1. JVMPROFILEDIR システム初期設定パラメーターを、CICS 領域で使用される JVM プロファイルを保管する先の z/OS UNIX 内の適切なディレクトリーに設定します。 詳しくは、 186 ページの『JVM プロファイルのロケーションの設定』を参照してください。
2. CICS 領域には、必ず Java アプリケーションを実行できる十分なメモリーを持たせるようにします。 詳しくは、 187 ページの『Java のメモリー制限の設定』を参照してください。
3. JVM の作成に必要な JVM プロファイル、ディレクトリー、およびファイルを含めて、z/OS UNIX に保持されているリソースにアクセスする許可を CICS 領域に付与します。 詳しくは、 188 ページの『z/OS UNIX ディレクトリーおよびファイルに対するアクセス権の CICS 領域への付与』を参照してください。
4. JVM サーバーをセットアップします。 さまざまなワークロードを実行するように JVM サーバーを構成できます。詳細については、 191 ページの『JVM サーバーのセットアップ』を参照してください。
5. オプション: 安全でない可能性のあるアクションを Java アプリケーションで実行しないように保護するには、Java セキュリティー・マネージャーを有効にします。 詳しくは、Java セキュリティー・マネージャーの有効化を参照してください。
6. 未フォーマットのストレージ・ダンプ・パラメーター `JAVA_DUMP_TDUMP_PATTERN` を設定します。 ダンプは、順次 MVS データ・セットに書き込まれます。これは、環境変数 `JAVA_DUMP_TDUMP_PATTERN` に値を指定して変更できます。CICS 領域ユーザー ID にこのパターンと一致するデータ・セットに対する UPDATE

© Copyright IBM Corp. 1999, 2018

アクセス権限があることを確認してください。ない場合、診断データは失われます。詳細については、 z/OS でのダンプ・エージェントの使用を参照してください。

タスクの結果

Java をサポートする CICS 領域をセットアップし、Java アプリケーションを実行する JVM サーバーを作成しました。

次のタスク

既存の Java アプリケーションをアップグレードする場合は、アップグレード のガイダンスに従ってください。 JVM サーバーで Java アプリケーションの実行を開始するには、JVM サーバーへのアプリケーションのデプロイを参照してください。

JVM プロファイルのロケーションの設定

CICS は、**JVMPROFILEDIR** システム初期設定パラメーターで指定された z/OS UNIX ディレクトリーから、JVM プロファイルをロードします。**JVMPROFILEDIR** パラメーターの値を新規ロケーションに変更し、提供されたサンプル JVM プロファイルをこのディレクトリーにコピーする必要があります。その結果、それらのプロファイルを使用してインストールを検証できます。

始める前に

USSHOME システム初期設定パラメーターは、z/OS UNIX に CICS ファイル用のルート・ディレクトリーを指定する必要があります。

このタスクについて

CICS 提供のサンプル JVM プロファイルは、CICS インストール処理時にシステムに合わせてカスタマイズされるので、これらのプロファイルを即時に使用してインストールを検証できます。独自の Java アプリケーション用にこれらのファイルのコピーをカスタマイズすることができます。

JVM プロファイルでの使用に適している設定は、CICS のリリースごとに異なる可能性があるので、問題判別を容易にするために、すべてのプロファイルのベースとして CICS 提供のサンプルを使用してください。アップグレード情報を確認して、JVM プロファイルの新規オプションまたは変更されたオプションを見つけてください。

手順

1. JVMPROFILEDIR システム初期設定パラメーターを、CICS 領域で使用される JVM プロファイルを保管する先の z/OS UNIX のロケーションに設定します。指定する値の長さは 240 文字までにすることができます。

   JVMPROFILEDIR システム初期設定パラメーターに指定された設定は、`/usr/lpp/cicsts/cicsts55/JVMProfiles` です。これは、サンプル JVM プロファイルのインストール場所です。このディレクトリーは、カスタマイズされた JVM プロファイルを安全に保管できる場所ではありません。プログラムの保守時にサンプル JVM プロファイルが上書きされると、変更内容が失われる危険

があるためです。したがって、必ず **JVMPROFILEDIR** を変更して、JVM プロファイルを保管できる別の z/OS UNIX ディレクトリーを指定する必要があります。JVM プロファイルをカスタマイズする必要があるユーザーに該当する許可を付与できるディレクトリーを選択してください。

2. 提供されるサンプル JVM プロファイルを、そのインストール場所から z/OS UNIX ディレクトリーにコピーします。

   CICS をインストールすると、サンプル JVM プロファイルが zFS ディレクトリーに置かれます。このディレクトリーは、DFHISTAR インストール・ジョブの **USSDIR** パラメーターで指定されます。デフォルトのインストール・ディレクトリーは `/usr/lpp/cicsts/cicsts55/JVMProfiles` です。

タスクの結果

サンプル JVM プロファイルが zFS ディレクトリーにコピーされ、そのディレクトリーを使用するように CICS が構成されました。サンプル JVM プロファイルにはデフォルト値が含まれているため、それをすぐに使用して JVM サーバーをセットアップできます。

次のタスク

『Java のメモリー制限の設定』で説明されているように、Java アプリケーションを実行するために十分なメモリーが CICS および Java にあることを確認してください。また、Java のインストール先であり、Java アプリケーションのデプロイ先である z/OS UNIX ディレクトリーへのアクセス権限が CICS 領域にあることを確認する必要もあります。詳しくは、 188 ページの『z/OS UNIX ディレクトリーおよびファイルに対するアクセス権の CICS 領域への付与』を参照してください。

Java のメモリー制限の設定

Java アプリケーションには、他の言語で作成されたプログラムよりも多くのメモリーが必要です。Java アプリケーションを実行するために使用できる、十分なストレージとメモリーが CICS および Java にあることを確認する必要があります。

このタスクについて

Java は、16 MB 境界より下のストレージ、31 ビット・ストレージ、および 64 ビット・ストレージを使用します。JVM ヒープに必要なストレージは、CICS DSA ではなく、MVS 内の CICS 領域ストレージから取得されます。

手順

1. z/OS **MEMLIMIT** パラメーターが適切な値に設定されていることを確認します。このパラメーターは、CICS アドレス・スペースが使用できる 64 ビット・ストレージの量を制限します。CICS は 64 ビット・バージョンの Java を使用するので、**MEMLIMIT** が、CICS 領域で 64 ビット・ストレージを使用するこの機能やその他の機能の両方に十分な大きい値に設定されていることを確認する必要があります。

   次のトピックを参照してください。

   - JVM サーバーのストレージ要件の計算

- 「パフォーマンスの改善」の『MEMLIMIT の見積もり、確認、および設定』

2. 開始ジョブ・ストリームの **REGION** パラメーターに、Java の実行に十分な大きさの値が指定されていることを確認します。 それぞれの JVM に、アプリケーション (ジャストインタイム・コンパイル・コードを含む) を実行するための 16 MB 境界より下のストレージ、およびパラメーターを CICS に渡すための作業用ストレージが必要です。

z/OS UNIX ディレクトリーおよびファイルに対するアクセス権の CICS 領域への付与

CICS では、z/OS UNIX 内のディレクトリーとファイルへのアクセスが必要です。各 CICS 領域には、インストール時に z/OS UNIX ユーザー ID (UID) が割り当てられます。これらの領域は、z/OS UNIX グループ ID (GID) を割り当てられた ESM グループに接続されます。この UID および GID を使用して、CICS 領域が z/OS UNIX 内のディレクトリーおよびファイルにアクセスする権限を付与します。

始める前に

自分が z/OS UNIX のスーパーユーザーであるか、ディレクトリーおよびファイルの所有者であることを確認してください。ディレクトリーおよびファイルの所有者は、最初は、製品をインストールしたシステム・プログラマーの UID として設定されます。ディレクトリーおよびファイルの所有者は、インストール時に GID が割り当てられた ESM グループに接続されなければなりません。所有者は、この ESM グループをデフォルト・グループ (DFLTGRP) として持つか、補足グループの 1 つとして ESM グループに接続することができます。

このタスクについて

z/OS UNIX システム・サービスでは、個々の CICS 領域が単一の UNIX ユーザーとして扱われます。 z/OS UNIX ディレクトリーおよびファイルにアクセスするユーザー権限をさまざまな方法で付与することができます。例えば、CICS 領域の接続先の ESM グループに対して、ディレクトリーまたはファイルに関する適切なグループ権限を付与できます。このオプションは、実稼働環境に最適な場合があり、以下の手順で説明しています。

手順

1. CICS 領域からのアクセスが必要となる z/OS UNIX 内のディレクトリーおよびファイルを識別します。

JVM サーバー・オプション	デフォルト・ディレクトリー	権限	説明
JAVA_HOME	`/usr/lpp/java/J8.0_64`	読み取りおよび実行	IBM 64-bit SDK for z/OS, Java テクノロジー・エディション ディレクトリー

JVM サーバー・オプション	デフォルト・ディレクトリー	権限	説明
USSHOME	`/usr/lpp/cicsts/cicsts55`	読み取りおよび実行	z/OS UNIX 上の CICS ファイルのインストール・ディレクトリー。このディレクトリー内のファイルには、サンプル・プロファイルと CICS 提供の JAR ファイルが含まれます。
WORK_DIR	`/u/`*`CICS region userid`*	読み取り、書き込み、および実行	CICS 領域の作業ディレクトリー。このディレクトリーには、JVM からの入力、出力、およびメッセージが入っています。
JVMPROFILEDIR	*`USSHOME`*`/JVMProfiles/`	読み取りおよび実行	**`JVMPROFILEDIR`** システム初期設定パラメーターで指定された、CICS 領域の JVM プロファイルが入っているディレクトリー。
WLP_USER_DIR	`WORK_DIR/APPLID/JVMSERVER/wlp/usr/`	読み取り、書き込み、および実行	Liberty JVM サーバーの構成ファイルを格納するディレクトリーを指定します。Liberty JVM サーバーの自動構成を使用する場合は、CICS が server.xml に書き込みを実行できる必要があるため、WLP_USER_DIR に追加の x 許可 (読み取り、書き込み、実行) が必要になります。
WLP_OUTPUT_DIR	`WLP_USER_DIR/servers`	読み取り、書き込み、および実行	Liberty JVM サーバーの出力ディレクトリーを指定します。

2. ディレクトリーおよびファイルをリストして、権限を表示します。 開始したいディレクトリーに移動し、次の UNIX コマンドを発行します。

   ```
   ls -la
   ```

   現行ディレクトリーが `CICSHT##` のホーム・ディレクトリーであるときに、このコマンドが z/OS UNIX システム・サービスのシェル環境で発行されると、次の例のようなリストが表示される場合があります。

   ```
   /u/cicsht##:>ls -la
   total 256
   drwxr-xr-x   2 CICSHT## CICSTS55     8192 Mar 15  2008 .
   drwx------   4 CICSHT## CICSTS55     8192 Jul  4 16:14 ..
   -rw-------   1 CICSHT## CICSTS55     2976 Dec  5  2010 Snap0001.trc
   -rw-r--r--   1 CICSHT## CICSTS55     1626 Jul 16 11:15 dfhjvmerr
   -rw-r--r--   1 CICSHT## CICSTS55        0 Mar 15  2010 dfhjvmin
   -rw-r--r--   1 CICSHT## CICSTS55      458 Oct  9 14:28 dfhjvmout
   /u/cicsht##:>
   ```

3. グループ権限を使用してアクセス権限を付与する場合は、各ディレクトリーおよびファイルのグループ権限が、CICS がリソースに対して必要とするアクセス・レベルを認可するものであることを確認します。 `r`、`w`、`x`、および `-` という文字を使用して 3 組の権限が指定されます。これらの文字は、「読み取り」、「書き込み」、「実行」、および「なし」を表し、コマンド行の左側の列の 2 文字目から表示されます。最初の組は所有者権限、2 番目の組はグループ権限、3 番目の組はその他の権限です。 前述の例では、所有者には `dfhjvmerr`、

dfhjvmin、および dfhjvmout に対する読み取りおよび書き込み権限がありますが、グループおよびその他すべてには、読み取り権限しかありません。

4. 特定のリソースのグループ権限を変更したい場合は、 UNIX コマンド chmod を使用します。 次の例では、指定されたディレクトリーおよびそのサブディレクトリーとファイルのグループ権限を、読み取り、書き込み、および実行に設定します。 -R は、すべてのサブディレクトリーおよびファイルに権限を再帰的に適用します。

   ```
   chmod -R g=rwx directory
   ```

   次の例では、指定されたファイルのグループ権限を読み取りおよび実行に設定します。

   ```
   chmod g+rx filename
   ```

   次の例では、指定された 2 つのファイルでグループに対する書き込み権限をオフにします。

   ```
   chmod g-w filename filename
   ```

   上記のすべての例で、g はグループ権限を指定します。その他に訂正したい権限がある場合、 u はユーザー (所有者) 権限を指定し、o はその他の権限を指定します。

5. CICS 領域から z/OS UNIX にアクセスするために選択された ESM グループに対して、リソースごとにグループ権限を割り当てます。個々のディレクトリーとそのサブディレクトリー、およびそこに含まれるファイルに対して、グループ権限を割り当てる必要があります。 次の UNIX コマンドを入力します。

   ```
   chgrp -R GID directory
   ```

   GID は ESM グループの GID の数値であり、*directory* は、CICS 領域の権限を付与する対象となるディレクトリーの絶対パスです。 例えば、/usr/lpp/cicsts/cicsts55 ディレクトリーにグループ権限を割り当てるには、次のコマンドを使用します。

   ```
   chgrp -R GID /usr/lpp/cicsts/cicsts55
   ```

   CICS 領域のユーザー ID は ESM グループに接続されるため、CICS 領域は、これらすべてのディレクトリーおよびファイルに関する適切な権限を持つことになります。

タスクの結果

これで、CICS には、Java アプリケーションを実行するために z/OS UNIX 内のディレクトリーとファイルにアクセスする適切な権限があることが確実になりました。

ファイルの移動または新規ファイルの作成などにより、セットアップ中の CICS 機能を変更した場合は、新規のファイルまたは移動したファイルに関する CICS 領域のアクセス権限が保持されるように、この手順を忘れずに繰り返してください。

次のタスク

サンプルのプログラムおよびプロファイルを使用して、Java サポートが正しくセットアップされたことを確認します。

JVM サーバーのセットアップ

JVM サーバー内で Java アプリケーション、Web アプリケーション、Axis2、または CICS セキュリティー・トークン・サービスを実行するには、CICS リソースをセットアップして、JVM にオプションを渡す JVM プロファイルを作成する必要があります。

このタスクについて

JVM サーバーは、単一の JVM でさまざまな Java アプリケーションの複数の並行要求を処理できます。 JVMSERVER リソースは、CICS における JVM サーバーを表します。このリソースによって定義される JVM プロファイルは、JVM の構成オプション、Language Environment エンクレーブに値を提供するプログラム、およびスレッドの限度を指定します。 JVM サーバーは、異なるタイプのワークロードを実行できます。JVM サーバーの用途ごとに、次のように異なる JVM プロファイルが用意されています。

- OSGi バンドルとしてパッケージされているアプリケーションを実行するには、`DFHOSGI.jvmprofile` を使用して JVM サーバーを構成します。このプロファイルには、JVM サーバーで OSGi フレームワークを実行するためのオプションが含まれています。
- Liberty を含むアプリケーションを CICS で実行するためには、`DFHWLP.jvmprofile` を使用して JVM サーバーを構成します。このプロファイルには、Liberty テクノロジーに基づく Web コンテナーを実行するためのオプションが含まれています。Web コンテナーには OSGi フレームワークも含まれるため、OSGi バンドルとしてパッケージされているアプリケーションを実行することができます。
- Axis2 SOAP エンジンを使用して Web サービスの SOAP 処理を実行するには、`DFHJVMAX.jvmprofile` を使用して JVM サーバーを構成します。このプロファイルには、JVM サーバーで Axis2 を実行するためのオプションが含まれています。
- CICS セキュリティー・トークン・サービス (STS) を実行するには、`DFHJVMST.jvmprofile` を使用して JVM サーバーを構成します。このプロファイルには STS を実行するためのオプションが含まれています。

プロファイルに加える変更はすべて、そのプロファイルを使用するすべての JVM サーバーに適用されます。各プロファイルをカスタマイズする際に、変更内容が、JVM サーバーを使用するすべての Java アプリケーションに適切であることを確認してください。

CICS オンライン・リソース定義を使用して JVM サーバーや JVM プロファイルを構成することができます。あるいは、CICS Explorer を使用して CICS バンドル内に JVMSERVER リソースと JVM プロファイルを定義し、パッケージ化することもできます。詳しくは、CICS Explorer 製品資料内の『Working with bundles』を参照してください。

タスクの結果

JVM サーバーが構成され、Java ワークロードを実行する準備ができました。

次のタスク

Java 環境のためのセキュリティーを構成します。 Java アプリケーションのデプロイやインストールのための適切なアクセス権をアプリケーション開発者に付与し、アプリケーション・ユーザーが CICS で Java プログラムやトランザクションを実行するための権限を与えます。

OSGi JVM サーバーの構成

OSGi バンドルにパッケージされている Java アプリケーションをデプロイする場合、OSGi フレームワークを実行するように JVM サーバーを構成します。

このタスクについて

JVM サーバーには、自動的にクラス・ロードを処理する OSGi フレームワークが含まれているため、標準クラスパス・オプションを JVM プロファイルに追加することはできません。提供されているサンプルの DFHOSGI.jvmprofile は OSGi JVM サーバーに適しています。このタスクでは、このサンプル・プロファイルから OSGi アプリケーション用の JVM サーバーを定義する方法を示します。

JVM サーバーは、CICS オンライン・リソース定義を使用して定義することも、CICS Explorer の CICS バンドル内で定義することもできます。

手順

1. JVM サーバー用の JVMSERVER リソースを作成します。
   a. JVM サーバー用の JVM プロファイルの名前を指定します。 JVMSERVER の JVMPROFILE 属性では、1 文字から 8 文字の名前を指定します。この名前は、JVM サーバーの構成オプションを保持するファイルである JVM プロファイルの接頭部に使用されます。ここでは接尾部として .jvmprofile を指定する必要はありません。
   b. JVM サーバーのスレッド限度を指定します。 JVMSERVER の THREADLIMIT 属性では、JVM サーバーの Language Environment エンクレーブで許可されているスレッドの最大数を指定します。スレッド数は、JVM サーバーで実行するワークロードによって異なります。始めにデフォルト値を受け入れ、後で環境を調整できます。 1 つの JVM サーバーで最大 256 個のスレッドを設定できます。
2. JVM プロファイルを作成して、JVM サーバーの構成オプションを定義します。ベースとして、サンプル・プロファイル DFHOSGI.jvmprofile を使用することができます。このプロファイルには、JVM サーバーを開始するために適したオプションのサブセットが含まれています。 JVM プロファイルのすべてのオプションと値については、 216 ページの『JVM プロファイルの検証およびプロパティー』で説明されています。 217 ページの『プロファイルのコーディング規則』のコーディング規則 (プロファイル名のコーディング規則を含む) に従ってください。
   a. JVM プロファイルの場所を設定します。 JVM プロファイルは、システム初期設定パラメーター JVMPROFILEDIR で指定するディレクトリー内になければなりません。詳しくは、 186 ページの『JVM プロファイルのロケーションの設定』を参照してください。
   b. サンプル・プロファイルを次のように変更します。

- JAVA_HOME を IBM Java SDK のインストール先に設定します。
- WORK_DIR を、JVM サーバーからのメッセージ、トレース、および出力の適当な宛先ディレクトリーに設定します。
- TZ を、JVM サーバーからのメッセージに対するタイム・スタンプのタイム・ゾーンを指定するように設定します。英国の例は TZ=GMT0BST,M3.5.0,M10.4.0 です。

c. 変更内容を JVM プロファイルに保管します。 z/OS UNIX システム・サービスのファイル・システムでは、JVM プロファイルを EBCDIC として保管する必要があります。

3. JVMSERVER リソースをインストールして使用可能にします。

タスクの結果

CICS は Language Environment エンクレーブを作成し、JVM プロファイルから JVM サーバーにオプションを渡します。 JVM サーバーが始動し、OSGi フレームワークはすべての OSGi ミドルウェア・バンドルを解決します。 JVM サーバーが始動を正常に完了すると、JVMSERVER リソースが ENABLED 状態でインストールされます。

エラーが発生する場合 (CICS が JVM プロファイルの検出も読み取りもできない場合など)、JVM サーバーは始動できません。 JVMSERVER リソースは DISABLED 状態でインストールされ、CICS はシステム・ログにエラー・メッセージを発行します。

次のタスク

- JVM 出力、ログ、ダンプ、およびトレースの場所の制御の説明に従って、JVM ログのロケーションを構成します。
- JVM サーバーにおける OSGi バンドルのデプロイの説明に従って、JVM サーバーの OSGi フレームワークにアプリケーションの OSGi バンドルをインストールします。
- Db2 や IBM MQ など、ネイティブ C ダイナミック・リンク・ライブラリー (DLL) ファイルを含むディレクトリーを指定します。これらのディレクトリーは、JVM プロファイルの LIBPATH_SUFFIX オプションで指定します。
- OSGi フレームワークで実行するミドルウェア・バンドルを指定します。ミドルウェア・バンドルは、IBM MQ および Db2 への接続などの共用サービスを実行するための Java クラスを含む OSGi バンドルの 1 つのタイプです。これらのバンドルは、JVM プロファイルの OSGI_BUNDLES オプションで指定します。

JVM プロファイルの例

OSGi アプリケーション用の JVM プロファイルの例を示します。

次の例では、DB2 バージョン 11 および JDBC 4.0 OSGi ミドルウェア・バンドルを使用する OSGi フレームワークを開始するように構成された JVM プロファイルを抜粋して示しています。

```
#*********************************************************************
#
#                         Required parameters
#                         -------------------
#
```

```
# When using a JVM server, the set of CICS options that are supported
JAVA_HOME=/usr/lpp/java/J8.0_64
WORK_DIR=.
LIBPATH_SUFFIX=/usr/lpp/db2v11/jdbc/lib
...
#*********************************************************************
#
#                    JVM server specific parameters
#                    ------------------------------
#
OSGI_BUNDLES=/usr/lpp/db2v11/jdbc/classes/db2jcc4.jar,\
             /usr/lpp/db2v11/jdbc/classes/db2jcc_license_cisuz.jar
OSGI_FRAMEWORK_TIMEOUT=60
#
#*********************************************************************
#
#                              JVM options
#                              -----------
# The following option sets the Garbage collection Policy.
#
-Xgcpolicy:gencon
#
#*********************************************************************
#
#                 Setting user JVM system properties
#                 ----------------------------------
#
# -Dcom.ibm.cics.some.property=some_value
#
#*********************************************************************
#
#              Unix System Services Environment Variables
#              ------------------------------------------
#
JAVA_DUMP_OPTS="ONANYSIGNAL(JAVADUMP,SYSDUMP),ONINTERRUPT(NONE)"
#
#
```

JMS をサポートするための OSGi JVM サーバーの構成

JMS を使用したアプリケーションをサポートするように OSGi JVM サーバーを構成することができます。

このタスクについて

このタスクでは、JMS を使用して IBM MQ に接続するアプリケーションをサポートするように、OSGi JVM サーバーの構成をセットアップします。 CICS-MQ アダプターを介して IBM MQ に接続するように CICS を構成します。 JMS バンドルを、JVM サーバー内の OSGi フレームワーク内で実行されるミドルウェア・バンドルのセットに追加する必要があります。また、このフレームワークは、関連付けられた IBM MQ ネイティブ・ライブラリーのセットにアクセスできなければなりません。

作業を開始する前に、CICS Liberty JVM サーバーでの IBM MQ classes for JMS の使用 の考慮事項を検討してください。

手順

1. CICS-MQ アダプターのセットアップの説明に従って、CICS-MQ アダプターをセットアップします。

2. IBM MQ classes for JMS を JVM サーバーに OSGi ミドルウェア・バンドルとして追加します。 これを行うには、JVM サーバーの JVM プロファイルに以下の行を含めます。

```
OSGI_BUNDLES=MQ_ROOT/OSGi/com.ibm.mq.osgi.allclientprereqs_VERSION.jar,\
MQ_ROOT/OSGi/com.ibm.mq.osgi.allclient_VERSION.jar
```

   ここで、 *MQ_ROOT* は IBM MQ for z/OS® UNIX System Services インストール済み環境の `java/lib/` ディレクトリー (例えば `/usr/lpp/V8R0M0/java/lib`)、*VERSION* は使用される IBM MQ classes for JMS のバージョン (例えば `8.0.0.0`) です。

3. JVM サーバーの JVM プロファイルで、IBM MQ classes for JMS ネイティブ・ライブラリーを格納するディレクトリーを `LIBPATH_SUFFIX` に追加します。例えば、`LIBPATH_SUFFIX=MQ_ROOT` のようにします。

   注**:** https://www.ibm.com/support/knowledgecenter/SSFKSJ_8.0.0/com.ibm.mq.dev.doc/q121760_.htm も必要になります。

4. JVM サーバーを停止して、再始動します。

IBM MQ classes for Java をサポートするための OSGi JVM サーバーの構成

JVM サーバーは、Java アプリケーション用のランタイム環境です。IBM MQ classes for Java を使用したアプリケーションをサポートするように OSGi JVM サーバーを構成することができます。

このタスクについて

IBM MQ classes for Java を使用したアプリケーションを OSGi JVM サーバーがサポートできるようにするには、IBM MQ for Java バンドルを、JVM サーバー内の OSGi フレームワークで実行されるミドルウェア・バンドルのセットに追加する必要があります。また、このフレームワークは、関連付けられたネイティブ・ライブラリーのセットにアクセスできなければなりません。

手順

1. IBM MQ classes for Java を JVM サーバーに OSGi ミドルウェア・バンドルとして追加します。 IBM MQ バージョン 8.0 以降でこれらのクラスを追加するには、OSGi JVM サーバーの JVM プロファイルに以下の行を含めます。

```
OSGI_BUNDLES=<MQ_ROOT>/OSGi/com.ibm.mq.osgi.allclientprereqs_<VERSION>.jar,¥
<MQ_ROOT>/OSGi/com.ibm.mq.osgi.allclient_<VERSION>.jar
```

   WebSphere MQ for z/OS バージョン 7.1 の場合は、以下の行を含めます。

```
OSGI_BUNDLES=<MQ_ROOT>/OSGi/com.ibm.mq.osgi.java_<VERSION>.jar
```

   各部の意味は次のとおりです。

   - *MQ_ROOT* は、IBM MQ for z/OS Unix System Services インストール済み環境の `java/lib/` ディレクトリーです (例: `/usr/lpp/V8R0M0/java/lib`)。
   - *VERSION* は、ご使用の IBM MQ classes for Java のバージョンです (例: `8.0.0.0`)。

2. IBM MQ classes for Java ネイティブ・ライブラリーが含まれるディレクトリーを、OSGi JVM サーバーの JVM プロファイル内の LIBPATH_SUFFIX オプションに追加します。 例:

```
LIBPATH_SUFFIX=<MQ_ROOT>
```

ここで、*MQ_ROOT* は IBM MQ for z/OS Unix System Services インストール済み環境の `java/lib/` ディレクトリーです (例: `/usr/lpp/V8R0M0/java/lib`)。

Liberty JVM サーバーの構成

Java EE アプリケーション (EJB、JSP、JSF、サーブレットなど) をデプロイする場合は、Liberty JVM サーバーを構成します。

このタスクについて

Liberty JVM サーバーの構成方法には次の 2 つがあります。

自動構成
: CICS は Liberty 用の構成ファイル `server.xml` を、CICS インストール・ディレクトリーに用意されているテンプレートから自動的に作成および更新します。自動構成を使用すると、Liberty の最小の構成値セットを使用して迅速に開始することができます。自動構成を有効にするには、JVM システム・プロパティーの **-Dcom.ibm.cics.jvmserver.wlp.autoconfigure** プロパティーを `true` に設定します。CICS バンドル内で JVM サーバーを定義する場合は、このオプションを設定します。

手動構成
: これはデフォルト設定です。構成ファイルとすべての値を提供します。手動構成は、Liberty サーバー構成を完全に制御したい場合に適しています。

JVM サーバーを定義するには、Ways of defining CICS resourcesを参照してください。

手順

1. JVMSERVER リソースを作成します。CICS バンドル内に JVMSERVER リソースを作成する場合は、バンドルにデプロイ可能な成果物を参照してください。
   a. JVM サーバー用の JVM プロファイルの名前を指定します。 JVMSERVER の JVMPROFILE 属性では、1 文字から 8 文字の名前を指定します。この名前は、JVM サーバーの構成オプションを保持するファイルである JVM プロファイルのファイル名に使用されます。ここではファイル・タイプとして `.jvmprofile` を指定する必要はありません。
   b. JVM サーバーのスレッド限度を指定します。 JVMSERVER の THREADLIMIT 属性で、割り振るスレッドの最大数を指定します。実際に使用されるスレッド数は、JVM サーバーで実行するワークロードによって異なります。始めにデフォルト値を受け入れ、後で環境を調整できます。 1 つの JVM サーバーで最大 256 個のスレッドを設定できます。

c. JVM プロファイルの場所を設定します。 JVM プロファイルは、システム初期設定パラメーター **JVMPROFILEDIR** で指定するディレクトリー内になければなりません。詳しくは、 186 ページの『JVM プロファイルのロケーションの設定』を参照してください。

2. JVM プロファイルを作成して、JVM サーバーの構成オプションを定義します。ベースとして、サンプル・プロファイル `DFHWLP.jvmprofile` を使用することができます。 このプロファイルには、JVM サーバーを開始するために適したオプションのサブセットが含まれています。 JVM プロファイルのすべてのオプションと値については、 216 ページの『JVM プロファイルの検証およびプロパティー』で説明されています。 217 ページの『プロファイルのコーディング規則』のコーディング規則 (プロファイル名のコーディング規則を含む) に従ってください。

   a. サンプル・プロファイルを次のように変更します。

      - **JAVA_HOME** を IBM Java SDK のインストール先に設定します。
      - **WORK_DIR** を、JVM サーバーからのメッセージ、トレース、および出力の適当な宛先ディレクトリーに設定します。
      - **WLP_INSTALL_DIR** を `&USSHOME;/wlp` に設定します。
      - **TZ** を、JVM サーバーからのメッセージに対するタイム・スタンプのタイム・ゾーンを指定するように設定します。英国の例は `TZ=GMT0BST,M3.5.0,M10.4.0` です。
      - **-Dfile.encoding** を ISO-8859-1 に設定します (例: `-Dfile.encoding=ISO-8859-1`)。
      - (オプション) **CICS_WLP_MODE** を設定して、CICS と Liberty の統合のレベルを選択します。

      JVM サーバー・オプションについて詳しくは、「JVM プロファイルに使用するシンボル」を参照してください。

   b. 変更内容を JVM プロファイルに保管します。 JVM プロファイルは、UNIX System Services 上に EBCDIC ファイル・エンコード方式で保管する必要があり、ファイル・タイプは `.jvmprofile` でなければなりません。

3. Liberty のサーバー構成を作成します。

   手動による JVM サーバーの作成は、構成ファイルを慎重に管理する必要がある場合に適しています。詳しくは、 199 ページの『Liberty サーバーの手動作成』および server.xml の手動調整を参照してください。

   重要: CICS バンドルに JVM サーバーを定義する場合は、`server.xml` 構成ファイルを JVM プロファイルと一緒に CICS バンドルに組み込むことはできないため、自動構成を使用する必要があります。

4. JVMSERVER リソースをインストールして使用可能にします。

タスクの結果

JVMSERVER は、JVM プロファイルを読み取り、指定されている設定に基づいて自身を初期設定します。自動構成が有効な場合、Liberty サーバー構成が存在しなければ、作成されます。自動構成が有効でない場合は、構成が存在しない、または構成が正しくないと、JVMSERVER は DISABLED になり、該当する障害を報告しま

す。2 回目以降の始動では、JVMSERVER は既存の構成を使用し、Liberty サーバー・インスタンスを起動します。JVMSERVER が始動を正常に完了すると、JVMSERVER リソースが ENABLED 状態でインストールされます。

エラーが発生する場合 (CICS が JVM プロファイルの検出も読み取りもできない場合など)、JVM サーバーは初期化できません。 JVM サーバーは DISABLED 状態でインストールされ、CICS はシステム・ログにエラー・メッセージを発行します。詳しくは、Liberty JVM サーバーおよび Java Web アプリケーションのトラブルシューティングを参照してください。 JVM サーバー内で Liberty が正常に開始されたことを確認する場合は、zFS 上の WLP_USER_DIR 出力ディレクトリーにある `messages.log` ファイルを参照してください。

注意**:**
JVM サーバーで実行される **Liberty** サーバーを開始または停止するために **Liberty** の **bin/server** スクリプトを使用しないでください。

注**:** CICS 統合モードの Liberty の場合、実行中のワークロードが存在しないのに、JVM サーバーが示す現在のスレッド数が正の値を返し、変動することがあります。これは、効率性のためにスレッドが Liberty 内にプールされているためです。

次のタスク

- CICS Liberty のデフォルトの Web アプリケーションを実行し、URL `http://server:port/com.ibm.cics.wlp.defaultapp/` を使用して、Liberty JVM サーバーが実行されていることを確認します。詳細については、CICS デフォルト Web アプリケーションの構成を参照してください。
- IBM MQ など、ネイティブ C ダイナミック・リンク・ライブラリー (DLL) ファイルを含むディレクトリーを指定します。IBM またはベンダーによって提供されるミドルウェアおよびツールによっては、DLL ファイルをライブラリー・パスに追加する必要があります。
- セキュリティーのサポートを追加します。 Liberty JVM サーバーのセキュリティーの構成 (Configuring security for a Liberty JVM server)を参照してください。
- CICS バンドル内の Java EE アプリケーションの Liberty JVM サーバーへのデプロイの説明に従って、Java EE アプリケーション (EAR ファイル、WAR ファイルおよび EBA ファイル) をインストールします。
- 一般的な Liberty のセットアップについては、Liberty の概要、Liberty の概要を参照してください。

CICS 標準モードの Liberty: CICS と完全に統合せずに Java EE Full Platform をサポート

アプリケーションを変更せずに他のプラットフォームから CICS に Liberty アプリケーションを移植およびデプロイするには、CICS 組み込み Liberty JVM サーバーを標準モードで使用します。標準モードは、Java Enterprise Edition (Java EE) Full Platform を対象に作成され、このプラットフォームに依存するアプリケーションをホスティングするのに理想的ですが、CICS との完全な統合は必要ありません。CICS 標準モードの Liberty で実行されるアプリケーションは、Liberty のサービス、管理、およびセキュリティーを利用できます。さらに、z/OS 上の Java、z

Systems プラットフォームから、また、Db2 と IBM MQ のデータに近接していることから、パフォーマンス上および機能上の利点も得られます。

CICS 標準モードの Liberty は、Java EE 7 および Java EE 8 認定 IBM WebSphere Application Server Liberty をベースとしています。Java EE は、多層型のスケーラブルでセキュアなネットワーク・アプリケーションを実行するための API と環境を提供することで、コアの Java SE を拡張します。Java EE には、Web Profile、Enterprise JavaBeans や Batch Applications for the Java Platform も組み込まれています。

CICS 標準モードの Liberty の作成、ライフサイクル、構成の管理には、CICS JVM サーバー・テクノロジーが使用されます。 CICS 標準モードの Liberty で稼働するアプリケーションは、デフォルトでは CICS リソースにアクセスできませんが、`runAsCICS()` メソッドを使用して `CICSExecutorService` に作業を実行依頼できます。`CICSExecutorService` に実行依頼された作業には JCICS API に対する全アクセス権限が与えられます。これらの作業は、CICS タスクの下、CICS 作業単位で実行され、スレッドの完了時にコミットされます。 `CICSExecutorService` に実行依頼された作業に、Java EE API にへのアクセス権限は与えられません。

JVM プロファイルの例

Liberty サーバー用の JVM プロファイルの例を示します。

次の例では、必要な構成ファイルとディレクトリー構造を自動的に作成するように構成された JVM プロファイルを抜粋して示しています。 DB2 バージョン 11 が使用されています。

```
#*********************************************************************
# JVM profile: DFHWLP
#
JAVA_HOME=/java/java71_64/J7.1_64
WORK_DIR=.
#*********************************************************************
# JVM server parameters
#
OSGI_FRAMEWORK_TIMEOUT=60
#*********************************************************************
# Liberty JVM server
#
-Dcom.ibm.ws.logging.console.log.level=INFO
-Dcom.ibm.cics.jvmserver.wlp.autoconfigure=true
-Dcom.ibm.cics.jvmserver.wlp.server.http.port=12345
-Dcom.ibm.cics.jvmserver.wlp.server.host=*
-Dcom.ibm.cics.jvmserver.wlp.jdbc.driver.location=/usr/lpp/db2v11/jdbc
-Dfile.encoding=ISO-8859-1
WLP_INSTALL_DIR=&USSHOME;/wlp
WLP_USER_DIR=./&APPLID;/&JVMSERVER;
#*********************************************************************
# JVM options
-Xgcpolicy:gencon
-Xms128M
-Xmx256M
-Xmso128K
#*********************************************************************
# Unix System Services Environment Variables
TZ=CET-1CEST,M3.5.0,M10.5.0
```

Liberty サーバーの手動作成

JVM サーバーの zFS 内に Liberty サーバーを手動で作成します。

手順

1. JVM サーバーの zFS 内に Liberty サーバーのディレクトリー構造を作成します。 JVM サーバーは、構成ファイルが *WLP_USER_DIR*/servers/*server_name* ディレクトリーにあると想定します。ここで、*WLP_USER_DIR* は WLP_USER_DIR オプションの値で、*server_name* は **com.ibm.cics.jvmserver.wlp.server.name** プロパティーの値です。*server_name* プロパティーには、常に先頭に -D が付きます。これらのサーバー・オプションについて詳しくは、JVM サーバー・オプションを参照してください。
2. *server_name* ディレクトリーに Liberty サーバー構成を作成します。 少なくとも、server.xml ファイルを作成する必要があります。Liberty のインストール・ディレクトリーに wlp/templates/servers/defaultServer/server.xml として提供されているテンプレートをベースにできます。このファイルは ISO-8859-1 の ASCII ファイル・エンコード方式で保管し、UNIX コマンドの **chtag -c ISO8859-1 -t <file>** を使用して、このエンコードにタグを付ける必要があります。
3. ご使用のシステムに合わせて、server.xml ファイルを編集します。 <httpEndpoint> のホスト名とポート番号を更新します。JVM サーバーにおける server.xml の構成について詳しくは、『 201 ページの『server.xml の手動調整』』を参照してください。セキュリティーを適用する場合は、Liberty JVM サーバーのセキュリティーの構成 (Configuring security for a Liberty JVM server)を参照してください。

CICS デフォルト Web アプリケーションの構成

CICS Liberty デフォルト Web アプリケーションを使用して、Liberty サーバーが実行中であることを確認し、サーバー構成を表示することができます。これを使用して、JVM プロファイル、サーバー・ログ、Liberty server.xml ファイルおよび messages.log ファイルを表示できます。

始める前に

アプリケーション・セキュリティーを有効にしないと、デフォルト Web アプリケーションのフルアクセスがすべてのユーザーに付与されます。自動構成を有効にし、CICS セキュリティー (sec=yes) で実行している場合、または server.xml を手動で構成して cicsts:security-1.0 機能を追加した場合、アプリケーションを実行するにはユーザー ID に追加のアクセス権が必要です。デフォルト・サーブレットと基本的な情報にアクセスするには、**ユーザー**のロールでなければなりません。FileViewer サーブレットにアクセスするには、**管理者**のロールでなければなりません。

手順

1. SAF 権限を使用し、server.xml に <safAuthorization .../> エレメントが含まれている場合、次のプロファイルを作成する必要があります。
   a. デフォルト・サーブレットにアクセスするには、次の例を使用します。

```
RDEFINE EJBROLE BBGZDFLT.com.ibm.cics.wlp.defaultapp.User UACC(NONE)
PERMIT BBGZDFLT.com.ibm.cics.wlp.defaultapp.User CLASS(EJBROLE) ID(WLPSVRS) ACCESS(READ)
SETROPTS RACLIST(EJBROLE) REFRESH
```

   b. FileViewer サーブレットにアクセスするには、次の例を使用します。

```
RDEFINE EJBROLE BBGZDFLT.com.ibm.cics.wlp.defaultapp.Administrator UACC(NONE)
PERMIT BBGZDFLT.com.ibm.cics.wlp.defaultapp.Administrator CLASS(EJBROLE) ID(WLPSVRS) ACCESS(READ)
SETROPTS RACLIST(EJBROLE) REFRESH
```

2. また、SAF 権限が構成されていない場合は、デフォルトの JEE ロールベースのアクセス権限が使用されます。
   - CICS は、次の構成に示すように、デフォルトの権限定義を提供します。デフォルト・サーブレットへのアクセスは、特殊サブジェクト **ALL_AUTHENTICATED_USERS** への**ユーザー**・ロールを通じて付与されます。つまり、すべてのユーザーが認証されます。デフォルトで、CICS は FileViewer サーブレットへのアクセスを提供しません。

     ```
     <authorization-roles id="com.ibm.cics.wlp.defaultapp">
         <security-role name="User">
             <special-subject type="ALL_AUTHENTICATED_USERS"/>
         </security-role>
     </authorization-roles>
     ```

   - ただし、次の例のように許可エレメントを追加することで、デフォルトの JEE ロールの情報を `server.xml` でカスタマイズできます。この例は、`user2` を**管理者**ロールに追加し、FileViewer サーブレットへのアクセス権限を付与することで、デフォルトの構成を拡張します。

     ```
     <authorization-roles id="com.ibm.cics.wlp.defaultapp">
         <security-role name="User">
             <user name="user1"/>
             <user name="user2"/>
         </security-role>
         <security-role name="Administrator">
             <user name="user2"/>
         </security-role>
     </authorization-roles>
     ```

     この場合、`user1` はデフォルト・サーブレットにアクセスできますが、FileViewer サーブレットにはアクセスできず、`user2` はデフォルト・サーブレットと FileViewer サーブレットにアクセスできます。

タスクの結果

CICS デフォルト Web アプリケーションの構成が正常に実行されました。

server.xml の手動調整

`server.xml` に手動で変更を加える場合、適用できる基本的な構成がいくつかあります。CICS 領域ユーザー ID は、`server.xml` ファイルに対する読み取り権限と書き込み権限の両方を持っている必要があります。

サーバー構成の規則

Liberty では、`server.xml` のカスタマイズが可能です。規則について詳しくは、サーバー構成を参照してください。

HTTP エンドポイントの構成

アプリケーションへの Web アクセスが必要な場合は、適切なホスト名とポート番号を使って `httpEndpoint` 属性を更新します。以下に例を示します。

```
<httpEndpoint host="winmvs2c.example.com" httpPort="28216" httpsPort="28217"
    id="defaultHttpEndpoint"/>
```

CICS 領域で開くことができるポート番号を排他的に使用するか、共用ポートとして使用します。

HTTPS を使用できるのは、SSL が構成されている場合のみです (Java 鍵ストアを使用した Liberty JVM サーバー用の SSL (TLS) の構成 を参照)。

詳しくは、Liberty の概要を参照してください。

機能の追加

以下の機能を `<featureManager>` の機能リストに追加します。

- CICS 機能 `cicsts:core-1.0`。この機能は、CICS システム OSGi バンドルを Liberty フレームワークにインストールします。この機能は、CICS 統合モードの Liberty JVM サーバーを始動する場合に必要です。SAF レジストリーも定義する必要があります。
- CICS 機構 `cicsts:standard-1.0`。CICS 標準モードの Liberty JVM サーバーを始動するには、この機能が必要です。`cicsts:standard-1.0` 機能はデフォルトでは CICS リソースを利用できません。詳しくは、 198 ページの『CICS 標準モードの Liberty: CICS と完全に統合せずに Java EE Full Platform をサポート』を参照してください。

  注: `cicsts:core-1.0` または `cicsts:standard-1.0` 機能を指定してください。両方の機能を `server.xml` に指定することはできません。
- CICS セキュリティー機能 `cicsts:security-1.0`。この機能は、CICS Liberty セキュリティーに必要な CICS システム OSGi バンドルを Liberty フレームワークにインストールします。CICS 外部セキュリティーが有効 (SIT で **SEC**=YES) になっており、Liberty サーバーのセキュリティーが必要な場合には、この機能が必要です。 `cicsts:security-1.0` 機能を使用するには、ユーザー・レジストリーも構成する必要があります。詳しくは、 204 ページの『ユーザー・レジストリー』を参照してください。
- `jsp-2.3`。この機能は、サーブレットおよび JavaServer Pages (JSP) アプリケーションのサポートを有効にします。この機能は、動的 Web プロジェクト (WAR ファイル)、および CICS バンドルとしてインストールされる Web サポート付き OSGi バンドル・プロジェクト を格納する OSGi アプリケーション・プロジェクト で必要とされます。
- `cicsts:jdbc-1.0`。この機能は、JDBC DriverManager または DataSource インターフェースを介してアプリケーションが DB2 にアクセスできるようにします。

例:

```
<featureManager>
    <feature>cicsts:core-1.0</feature>
    <feature>cicsts:security-1.0</feature>
    <feature>jsp-2.3</feature>
    <feature>cicsts:jdbc-1.0</feature>
</featureManager>
```

詳細については、Liberty の機能を参照してください。

CICS バンドル・デプロイ・アプリケーション

CICS バンドルを使用する Liberty アプリケーションをデプロイするには、以下の項目を server.xml ファイルに含める必要があります。

```
<include location="${server.output.dir}/installedApps.xml"/>
```

インクルードされるファイルは、CICS バンドル・デプロイ・アプリケーションを定義するために使用されます。

バンドル・リポジトリー

共通 OSGi バンドルを共用するには、それらを 1 つのディレクトリーに入れて、bundleRepository エレメントでそのディレクトリーを参照します。以下に例を示します。

```
<bundleRepository>
    <fileset dir="directory_path" include="*.jar"/>
</bundleRepository>
```

グローバル/共用ライブラリー

Web アプリケーション間で共通の JAR ファイルを共用するには、それらを 1 つのディレクトリーに入れて、グローバル/共用ライブラリー定義でそのディレクトリーを参照します。

```
<library id="global">
    <fileset dir="directory_path" include="*.jar"/>
</library>
```

グローバル/共用ライブラリーは EBA 内の OSGi アプリケーションでは使用できません。その場合は、バンドル・リポジトリーを使用する必要があります。詳しくは、すべての Java EE アプリケーションのグローバル・ライブラリーの提供、または 共用ライブラリーを参照してください。

Liberty サーバー・アプリケーションおよび構成更新のモニター

Liberty JVM サーバーは server.xml ファイルをスキャンして更新があるかどうか調べます。デフォルトでは 500 ミリ秒ごとにスキャンします。この値を変更するには、次のような項目を追加します。

```
<config monitorInterval="5s" updateTrigger="polled"/>
```

さらに dropins ディレクトリーもスキャンして、アプリケーションの追加、更新、または削除を検出します。 CICS バンドルの中に Web アプリケーションをインストールする場合、次のように dropins ディレクトリーを無効にしてください。

```
<applicationMonitor updateTrigger="disabled" dropins="dropins"
dropinsEnabled="false" pollingRate="5s"/>
```

詳しくは、Controlling dynamic updatesを参照してください。

JTA トランザクション・ログ

Java Transaction API (JTA) を使用する場合、Liberty トランザクション・マネージャーは、リカバリー可能ログ・ファイルを zFS ファイリング・システムに保管します。トランザクション・ログのデフォルトの場所は `${WLP_USER_DIR}/tranlog/` です。この場所は、

`<transaction transactionLogDirectory="/u/cics/CICSPRD/DFHWLP/tranlog/"/>` のように `transaction` エレメントを `server.xml` に追加してオーバーライドできます。

CICS デフォルト Web アプリケーション

CICS デフォルト Web アプリケーション (**CICSDefaultApp**) は、Liberty JVM サーバーが開始したことを確認する構成サービスです。このアプリケーションを使用可能にするには、JVM プロファイル・オプション `com.ibm.cics.jvmserver.wlp.defaultapp=true` を JVM プロファイルに追加するか、自動構成を使用していない場合は、`cicsts:defaultApp-1.0` 機能を `server.xml` に追加します。URL `http://<server>:<port>/com.ibm.cics.wlp.defaultapp/`を使用して、アプリケーションを実行します。

```
<featureManager>
    <feature>cicsts:defaultApp-1.0</feature>
</featureManager>
```

ユーザー・レジストリー

分散 ID マッピングを使用している場合を除き、CICS セキュリティー機能を使用するように SAF レジストリーを定義する必要があります。

```
<safRegistry id="saf"/>
```

詳しくは、分散 ID マッピングを使用した Liberty JVM サーバーのセキュリティーの構成 または Java Database Connectivity 4.1を参照してください。

CICS JTA の統合

XA トランザクションを Liberty で使用する場合、デフォルトでは CICS 作業単位は XA トランザクションで制御されます。CICS と JTA の自動統合からオプトアウトするには、JVM プロファイル・オプション `com.ibm.cics.jvmserver.wlp.jta.integration=true` を設定します。自動構成を使用しない場合は、server.xml に `server.xml: <cicsts_jta Integration="true"/>` 要素を手動で設定します。

jdbc-4.0 または jdbc-4.1 機能により、CICS を介したタイプ 2 接続を行う Db2 DataSource の自動構成

自動構成プロパティーを使用して、タイプ 2 接続の CICS 対応 Db2 DataSource を作成します。これは、Java EE 6 を使用する場合には `jdbc-4.0` 機能、Java EE 7 を使用する場合には `jdbc-4.1` 機能を使用します。

始める前に

Db2 に接続するために CICS 領域を構成します。詳しくは、Defining the CICS Db2 connectionを参照してください。

このタスクについて

JVM プロファイルの自動構成プロパティーを使用して、Liberty の `server.xml` に、CICS DB2CONN を介して機能するタイプ 2 接続の Db2 DataSource を作成できます。 JNDI 名は `jdbc/defaultCICSDataSource` です。

`id="defaultCICSDataSource"` を使用するタイプ 2 接続の Db2 DataSource が既に存在する場合、自動構成を使用して新規に作成することはできません。手動での作成方法について詳しくは、jdbc-4.0 または jdbc-4.1 機能により、CICS を介したタイプ 2 接続を行う Db2 DataSource の手動構成を参照してください。

手順

1. JVM プロファイルに **-Dcom.ibm.cics.jvmserver.wlp.autoconfigure=true** を設定して自動構成を有効にします。
2. JVM プロファイルに **com.ibm.cics.jvmserver.wlp.jdbc.driver.location**、Db2 JDBC ライブラリーの場所を設定します。 例:
   `-Dcom.ibm.cics.jvmserver.wlp.jdbc.driver.location=/usr/lpp/db2v11/jdbc`
3. デフォルト・スキーマが現在のユーザー ID ではない場合は、JVM プロファイルの **Ddb2.jcc.currentSchema** をスキーマの名前に設定します。 例:
   `-Ddb2.jcc.currentSchema=CICSDB2`
4. JVMSERVER リソースをインストールして使用可能にします。

タスクの結果

タイプ 2 接続を使用する Db2 DataSource が、Liberty サーバー構成ファイル `server.xml` に追加されます。この例では、`jdbc-4.1` 機能を使用します。

```
<featureManager>
    ...
    <feature>jdbc-4.1</feature>
</featureManager>

<dataSource id="defaultCICSDataSource" jndiName="jdbc/defaultCICSDataSource" transactional="false">
    <jdbcDriver libraryRef="defaultCICSDb2Library"/>
    <properties.db2.jcc driverType="2"/>
    <connectionManager agedTimeout="0"/>
</dataSource>

<library id="defaultCICSDb2Library">
    <fileset dir="/usr/lpp/db2v11/jdbc/classes" includes="db2jcc4.jar db2jcc_license_cisuz.jar"/>
    <fileset dir="/usr/lpp/db2v11/jdbc/lib" includes="libdb2jcct2zos4_64.so"/>
</library>
```

jdbc-4.0 または jdbc-4.1 機能により、CICS を介したタイプ 2 接続を行う Db2 DataSource の手動構成

CICS Liberty JVM サーバーは、CICS を介してタイプ 2 接続で Java アプリケーションから Db2 データベースにアクセスするときに JDBC DataSource を使用するように構成できます。

始める前に

Db2 に接続するように CICS 領域を構成する必要があります。詳しくは、Defining the CICS Db2 connectionを参照してください。

このタスクについて

このタスクでは、ローカル Db2 データベースへの JDBC タイプ 2 ドライバー接続を有効にするために server.xml に必要なエレメントを定義する方法を説明します。

手順

1. jdbc-4.1 または jdbc-4.0 機能を server.xml ファイルに追加します。

   Java EE 6 を使用する場合には jdbc-4.0 機能、Java EE 7 を使用する場合には jdbc-4.1 機能を使用します。

```
<featureManager>
    <feature>jdbc-4.1</feature>
</featureManager>
```

2. ライブラリー・エレメントを server.xml ファイルに追加して、zFS 上の JDBC ドライバーの場所を指定します。

```
<library id="defaultCICSDb2Library">
    <fileset dir="/usr/lpp/db2v11/jdbc/classes" includes="db2jcc4.jar db2jcc_license_cisuz.jar"/>
    <fileset dir="/usr/lpp/db2v11/jdbc/lib" includes="libdb2jcct2zos4_64.so"/>
</library>
```

3. DataSource 定義を使用して Db2 にアクセスするには、**dataSource** エレメントが必要です。アプリケーションによって参照される JNDI 名を定義するためには、**jndiName** 属性が必要です。 **properties.db2.jcc** エレメントを使用して、**dataSource** の属性を設定できます。次の例は、その方法を示しています。

```
<dataSource id="defaultCICSDataSource" jndiName="jdbc/defaultCICSDataSource" transactional="false">
    <jdbcDriver libraryRef="defaultCICSDb2Library"/>
    <properties.db2.jcc driverType="2"/>
    <connectionManager agedTimeout="0"/>
</dataSource>
```

   CICS にトランザクションの管理を許可するには、transactional="false" が必要です。

   Db2 へのタイプ 2 接続を使用するには、driverType="2" が必要です。

   Liberty 接続プーリングを無効にするには、agedTimeout="0" が必要です。DB2CONN リソースが Db2 接続プーリングを提供しているので、Liberty 接続プーリングは必要ありません。

タスクの結果

Liberty サーバーは、CICS DB2CONN リソースを介して、JDBC タイプ 2 接続を使用した Db2 データベースへのアクセスを許可するように構成されます。

jdbc-4.0 または jdbc-4.1 機能により、Liberty を介したタイプ 4 接続を行う Db2 DataSource の手動構成

CICS Liberty JVM サーバーは、タイプ 4 接続で Java アプリケーションから Db2 データベースにアクセスする JDBC DataSource を使用するように構成できます。

始める前に

タイプ 4 接続を使用する Liberty Db2 DataSource は CICS Db2 接続リソースを使用しません。ただし、APAR PI18798 と PI18799 を適用していない場合は、Db2 SDSNLOAD および SDSNLOD2 ライブラリーを CICS STEPLIB 連結に追加する必要があります。

このタスクについて

このタスクでは、ローカルまたはリモートの Db2 データベースへの JDBC タイプ 4 ドライバー接続を有効にするために `server.xml` 構成ファイルに必要なエレメントを手動で定義する方法を説明します。タイプ 4 接続を使用する Db2 データベースに対して行われる更新では、CICS Db2 接続リソースは使用されません。それらの更新は 2 フェーズ・コミット・トランザクションには含まれません。ただし、DataSource 接続のタイプが `javax.sql.XADataSource` である場合や、それらの更新が JTA ユーザー・トランザクション内で行われた場合は別です。詳細については、データベースへの接続の取得を参照してください。

手順

1. `jdbc-4.0` または `jdbc-4.1` 機能を featureManager エレメントに追加します。これにより、`server.xml` ファイルで後で使用する **dataSource** および **jdbcDriver** エレメントが有効になります。

   Java EE 6 を使用する場合には `jdbc-4.0` 機能、Java EE 7 を使用する場合には `jdbc-4.1` 機能を使用します。

   ```
   <featureManager>
       <feature>jdbc-4.1</feature>
   </featureManager>
   ```

2. **dataSource** および **jdbcDriver** エレメントを追加します。 **dataSource** エレメントは、JDBC ドライバーのコンポーネント (Db2 JDBC jar およびネイティブ DLL ファイル) のロード元ライブラリーを示すライブラリー定義を参照する必要があります。標準的な定義は次のようになります。

   ```
   <dataSource jndiName="jdbc/defaultCICSDataSource">
       <jdbcDriver libraryRef="db2Lib"/>
       <properties.db2.jcc driverType="4"
           serverName="winmvs2c.hursley.ibm.com"
           portNumber="41100"
           databaseName="DSNV11P2"
           user="DBUSER"
           password="{xor}Lz4sLCgwLTs="/>
   </dataSource>

   <library id="db2Lib">
       <fileset dir="/usr/lpp/db2v11/jdbc/classes" includes="db2jcc4.jar
           db2jcc_license_cisuz.jar" />
   </library>
   ```

   APAR PI18798 と PI18799 を適用していない場合は、Db2 ネイティブ・ライブラリーのファイル・セット・エントリーをライブラリー構成に追加する必要があります。例えば、次のようにします。

   ```
   <fileset dir="/usr/lpp/db2v11/jdbc/lib" />
   ```

dataSource は、そのデータ・ソースへの接続を確立する際にアプリケーション・プログラムによって参照される **jndiName** 属性を指定します。必要なプロパティーが以下のように **properties.db2.jcc** エレメントに設定されます。

driverType

説明: データベース・ドライバーのタイプ。ピュア Java ドライバーを使用する場合は、4 に設定する必要があります。

デフォルト値: 4

必須: いいえ

データ型: int

serverName

説明 : データベースが稼働しているサーバーのホスト名。これは、Db2 DISPLAY DDF コマンドの SQL DOMAIN 値です。

デフォルト値: localhost

必須: いいえ

データ型: ストリング

portNumber

説明: データベース接続を取得するポート。これは、Db2 DISPLAY DDF コマンドの TCPPORT 値です。

デフォルト値: 50000

必須: いいえ

データ型: int

databaseName

説明: データ・ソースの名前を指定します。これは、Db2 DISPLAY DDF コマンドの LOCATION 値です。

必須 : はい

データ型: ストリング

ユーザー

説明: データベースへの接続に使用するユーザー ID。

必須 : はい

データ型: ストリング

パスワード **(password)**

説明: データベースへの接続に使用するユーザー ID のパスワード。値は、平文形式またはエンコード形式で保管することができます。パスワードはエンコードすることをお勧めします。エンコードするには、encode オプションを指定して securityUtility ツールを使用します。securityUtility コマンドを参照してください。

必須 : はい

データ型: ストリング

タスクの結果

Liberty サーバーは、始動時に、JDBC タイプ 4 接続を使用した Db2 データベースへのアクセスを許可するように構成されます。詳細については、Java Database Connectivity 4.1を参照してください。

cicsts:jdbc-1.0 機能を使用した **CICS** を介するタイプ **2** 接続を行う **Db2 DataSource** または **DriverManager** インターフェースの手動構成

CICS Liberty JVM サーバーは、CICS を介して JDBC タイプ 2 接続を使用するように構成して、Db2 データベースにアクセスするための **javax.sql.DataSource** または **java.sql.DriverManager** インターフェースを Java アプリケーションに提供することができます。

始める前に

Db2 に接続するために CICS 領域を構成します。詳しくは、Defining the CICS Db2 connectionを参照してください。

このタスクについて

cicsts:jdbc-1.0 機能を使用したタイプ 2 接続で Db2 にアクセスするように CICS Liberty を構成することもできますが、推奨される方法は、Liberty の `jdbc-4.x` 機能を使用する方法です。詳しくは、jdbc-4.0 または jdbc-4.1 機能により、CICS を介したタイプ 2 接続を行う Db2 DataSource の手動構成を参照してください。ただし、DriverManager インターフェースを使用する場合は、以下の手順に記述されているように **cicsts:jdbc-1.0** 機能を使用する必要があります。

手順

1. `cicsts:jdbc-1.0` 機能を featureManager エレメントに追加します。 これにより、**cicsts_jdbcDriver** エレメントと **cicsts_dataSource** エレメントを使用できるようになります (後で `server.xml` ファイルで使用します)。

```
<featureManager>
    <feature>cicsts:jdbc-1.0</feature>
</featureManager>
```

2. **cicsts_jdbcDriver** エレメントを追加します。これにより、**java.sql.DriverManager** または **javax.sql.DataSource** インターフェースで JDBC タイプ 2 接続を使用できるようになります。 **cicsts_jdbcDriver** エレメントは、JDBC ドライバー・コンポーネント (Db2 JDBC jar およびネイティブ dll ファイル) のロード元ライブラリーを指定するライブラリー定義を参照する必要があります。標準的な定義は次のようになります。

```
<cicsts_jdbcDriver libraryRef="defaultCICSDb2Library"/>
<library id="defaultCICSDb2Library">
    <fileset dir="/usr/lpp/db2v11/jdbc/classes" includes="db2jcc4.jar db2jcc_license_cisuz.jar"/>
    <fileset dir="/usr/lpp/db2v11/jdbc/lib" includes="libdb2jcct2zos4_64.so"/>
</library>
```

注: 必要な **cicsts_jdbcDriver** エレメントは 1 つのみです。複数のエレメントを指定した場合は、`server.xml` ファイルの最後の **cicsts_jdbcDriver** エレメントのみが使用され、その他のエレメントは無視されます。

java.sql.DriverManager サポートのみが必要な場合は、前のステップで十分です。

3. DataSource インターフェースを使用して Db2 にアクセスするには、**cicsts_dataSource** エレメントが必要です。推奨される DataSource アクセスの方法は、jdbc-4.0 または jdbc-4.1 機能により、CICS を介したタイプ 2 接続を行う Db2 DataSource の手動構成で説明されているように、Liberty `jdbc-4.x` 機能を使用する方法です。 **cicsts_dataSource** では、アプリケーションによって参照される JNDI 名を定義するための `jndiName` 属性が必要です。定義は次のようになります。

```
<cicsts_dataSource id="defaultCICSDataSource" jndiName="jdbc/defaultCICSDataSource"/>
```

ヒント: 使用される DataSource クラスは **com.ibm.db2.jcc.DB2SimpleDataSource** であり、これは **javax.sql.DataSource** を実装します。

4. オプション: **properties.db2.jcc** エレメントを使用して、**cicsts_dataSource** の属性を設定できます。次の例は、その方法を示しています。

```
<cicsts_dataSource id="defaultCICSDataSource" jndiName="jdbc/defaultCICSDataSource">
    <properties.db2.jcc currentSchema="DB2USER" fullyMaterializeLobData="true" />
</cicsts_dataSource>
```

properties.db2.jcc エレメントで指定できる一部の属性は、タイプ 2 接続を使用する DataSources では無効です。それらの無効な属性を指定した場合、それらは無視され、警告メッセージが発行されます。以下の属性が無効です。

- *driverType*
- *serverName*
- *portNumber*
- *user*
- *password*
- *databaseName*

タスクの結果

Liberty JVM サーバーは、CICS DB2CONN リソースを介して JDBC タイプ 2 接続で Db2 データベースに接続できます。

注: `cicsts_dataSource` およびそのコンポーネントの動的更新はサポートされていません。Liberty サーバーの実行中に構成を更新すると、Db2 アプリケーションに障害が生じる可能性があります。変更をアクティブにするために、サーバーをリサイクルする必要があります。

JMS をサポートするための Liberty JVM サーバーの構成

JMS を使用するアプリケーションをサポートするように CICS Liberty JVM サーバーを構成します。Liberty JVM サーバーは CICS 標準モード Liberty または CICS 統合モード Liberty のいずれかにすることができますが、IBM MQ に対して使用する接続のタイプによって、構成には違いがあります。

このタスクについて

このタスクでは、IBM MQ classes for JMS を介して IBM MQ に接続するアプリケーションをサポートするように、CICS Liberty JVM サーバーの `server.xml` をセットアップします。 Liberty から IBM MQ に接続するには、バージョン 9.0.1

以降の IBM MQ リソース・アダプターが必要です。 Liberty には IBM MQ リソース・アダプターが含まれていないため、Fix Central からそれを取得する必要があります (Liberty へのリソース・アダプターのインストールを参照)。構成ステップで参照される Liberty の機能については、Liberty の機能で詳しく説明しています。

作業を開始する前に、CICS Liberty JVM サーバーでの IBM MQ classes for JMS の使用 の考慮事項を検討してください。

手順

1. wmqJmsClient-2.0 機能を `server.xml` ファイルに追加します。
   wmqJmsClient-2.0 機能を追加すると、Liberty サーバーは必要な IBM MQ バンドルをロードできるので、接続ファクトリーやアクティベーション・スペックのプロパティーなどの JMS リソースの定義が可能になります。

   JNDI 検索を実行する場合は、jndi-1.0 フィーチャーも追加する必要があります。

   ```
   <featureManager>
       <feature>wmqJmsClient-2.0</feature>
       <feature>jndi-1.0</feature>
   </featureManager>
   ```

2. JMS アプリケーションを、バインディング・モードで IBM MQ に接続するように構成する場合 (CICS 標準モード Liberty でのみサポートされます)、zosTransaction-1.0 機能を追加します。

   ```
   <featureManager>
       <feature>zosTransaction-1.0</feature>
   </featureManager>
   ```

3. `server.xml` ファイルの `variable` エレメントで、zFS 内での IBM MQ リソース・アダプターの場所を指定します。

   ```
   <variable name="wmqJmsClient.rar.location" value="/path/to/wmq/rar/wmq.jmsra.rar"/>
   ```

   `value` 属性で、IBM MQ リソース・アダプター・ファイル `wmq.jmsra.rar` の絶対パスを指定します。

4. JMS 接続ファクトリーを使用して IBM MQ キュー・マネージャーに接続する場合は、接続ファクトリー定義を `server.xml` ファイルに追加します。 IBM MQ システムにおける、キュー・マネージャーの名前、そのシステムのホスト名、キュー・マネージャーが listen しているポート、およびキュー・マネージャーへのチャネルに関する情報が必要です。アプリケーションがメッセージ駆動型 Bean を介して IBM MQ と通信する場合、接続ファクトリーは適用されません。

   IBM MQ のプロパティーについて詳しくは、Liberty の資料でJMS 接続ファクトリーの構成を参照してください。

   - IBM MQ へのクライアント・モード接続の場合は、以下のエレメントを追加します。

     ```
     <jmsConnectionFactory jndiName="jms/wmqCF" connectionManagerRef="ConMgr6">
         <properties.wmqJms transportType="CLIENT"
             hostName="localhost" port="1414"
             channel="SYSTEM.DEF.SVRCONN" queueManager="QM1"/>
     </jmsConnectionFactory>

     <connectionManager id="ConMgr6" maxPoolSize="10"/>
     ```

maxPoolSize にある値の 10 は、例として示されているだけです。maxPoolSize は、接続ファクトリーの同時ユーザーの最大数に設定してください。

- IBM MQ にバインディング・モードで接続する場合 (CICS 標準モード Liberty でのみサポートされます)、次のエレメントを追加します。

```
<jmsConnectionFactory jndiName="jms/qm1" connectionManagerRef="ConMgr6">
    <properties.wmqJms transportType="BINDINGS" queueManager="QM1"/>
</jmsConnectionFactory>

<connectionManager id="ConMgr6" maxPoolSize="10"/>
```

maxPoolSize にある値の「10」は、例として示されているだけです。maxPoolSize は、接続ファクトリーの同時ユーザーの最大数に設定してください。

5. jmsConnectionFactory または jmsActivationSpec によって参照されるキュー定義を server.xml に追加します。

```
<jmsQueue id="jms/queue1" jndiName="jms/queue1">
    <properties.wmqJms baseQueueName="QUEUE1" baseQueueManagerName="QM1"/>
</jmsQueue>
```

6. バインディング・モードで接続するように JMS アプリケーションを構成する場合は、server.xml ファイル内の wmqJmsClient エレメントを使用して、IBM MQ ネイティブ・ライブラリーの場所を指定します。

```
<wmqJmsClient nativeLibraryPath="/opt/mqm/java/lib64"/>
```

7. メッセージ駆動型 Bean を使用する場合は、**mdb-3.2** 機能を server.xml に追加します。接続ファクトリーを使用する場合、この機能は適用されません。

```
<featureManager>
    <feature>mdb-3.2</feature>
</featureManager>
```

その後、Liberty server.xml に、jmsQueue 要素、IBM MQ チャネル、キュー・マネージャー、ホスト、ポート、およびトランスポート・タイプを参照する jmsActivationSpec を定義します。 IBM MQ のプロパティーについて詳しくは、Liberty の資料でJMS 接続ファクトリーの構成を参照してください。

- クライアント・モードで接続する場合は、jmsActivationSpec エレメントを次のように追加します。

```
<jmsActivationSpec id="MQ.JMS.mdb.app/MQ.JMS.mdbEJB/MessageDrivenBean">
    <properties.wmqJms transportType="CLIENT"
        destinationRef="jms/queue1" destinationType="javax.jms.Queue"
        hostName="localhost" port="1414"
        channel="SYSTEM.DEF.SVRCONN" queueManager="QM1"/>
</jmsActivationSpec>
```

jmsActivationSpec id 属性は、*application name/module name/bean name* の形式でなければなりません。

- バインディング・モードで接続する場合 (CICS 標準モード Liberty でのみサポートされます)、次のように jmsActivationSpec エレメントを追加します。

```
<jmsActivationSpec id="MQ.JMS.mdb.app/MQ.JMS.mdbEJB/MessageDrivenBean">
    <properties.wmqJms transportType="BINDINGS"
        destinationRef="jms/queue1" destinationType="javax.jms.Queue"
        queueManager="QM1"/>
</jmsActivationSpec>
```

jmsActivationSpec id 属性は、*application name/module name/bean name* の形式でなければなりません。

タスクの結果

これで、JMS を使用するアプリケーションをサポートするように CICS Liberty JVM サーバーが構成されました。

Axis2 用の JVM サーバーの構成

Java Web サービスを実行するかまたは SOAP 要求をパイプラインで処理する場合、Axis2 を実行するように JVM サーバーを構成します。

このタスクについて

Axis2 は、Web サービス要求をプロバイダーおよびリクエスターのパイプラインで処理できる Java SOAP エンジンです。 Axis2 を実行するように JVM サーバーを構成している場合、CICS は要求された JAR ファイルを自動的にクラスパスに追加します。

JVM サーバーは、CICS オンライン・リソース定義を使用するか、CICS バンドルに含めることにより定義できます。

手順

1. JVM サーバー用の JVMSERVER リソースを作成します。
   a. JVM サーバー用の JVM プロファイルの名前を指定します。 JVMSERVER の JVMPROFILE 属性では、1 文字から 8 文字の名前を指定します。この名前は、JVM サーバーの構成オプションを保持するファイルである JVM プロファイルの接頭部に使用されます。ここでは接尾部として .jvmprofile を指定する必要はありません。
   b. JVM サーバーのスレッド限度を指定します。 JVMSERVER の THREADLIMIT 属性では、JVM サーバーの Language Environment エンクレーブで許可されているスレッドの最大数を指定します。必要なスレッド数は、JVM サーバーで実行したいワークロードによって異なります。始めにデフォルト値を受け入れてから、環境を調整できます。1 つの JVM サーバーで最大 256 個のスレッドを設定できます。
2. JVM プロファイルを作成して、JVM サーバーの構成オプションを定義します。ベースとして、サンプル・プロファイル DFHJVMAX.jvmprofile を使用することができます。このプロファイルには、JVM サーバーを開始するために適したオプションのサブセットが含まれています。 JVM プロファイルのすべてのオプションと値については、 216 ページの『JVM プロファイルの検証およびプロパティー』で説明されています。 217 ページの『プロファイルのコーディング規則』のコーディング規則 (プロファイル名のコーディング規則を含む) に従ってください。
   a. JVM プロファイルの場所を設定します。 JVM プロファイルは、システム初期設定パラメーター JVMPROFILEDIR で指定するディレクトリー内になければなりません。詳しくは、 186 ページの『JVM プロファイルのロケーションの設定』を参照してください。
   b. サンプル・プロファイルを次のように変更します。
      - JAVA_HOME を IBM Java SDK のインストール先に設定します。
      - JAVA_PIPELINE を Axis2 を実行するように設定します。

- CLASSPATH_SUFFIX を、Java で記述されている Axis2 アプリケーションおよび SOAP ハンドラーのクラスを指定するように設定します。
- WORK_DIR を、JVM サーバーからのメッセージ、トレース、および出力の適当な宛先ディレクトリーに設定します。
- TZ を、JVM サーバーからのメッセージに対するタイム・スタンプのタイム・ゾーンを指定するように設定します。英国の例は TZ=GMT0BST,M3.5.0,M10.4.0 です。

c. 変更内容を JVM プロファイルに保管します。 z/OS UNIX システム・サービスのファイル・システムでは、JVM プロファイルを EBCDIC として保管する必要があります。

3. JVMSERVER リソースをインストールして使用可能にします。

タスクの結果

CICS は Language Environment エンクレーブを作成し、JVM プロファイルから JVM サーバーにオプションを渡します。 JVM サーバーが始動し、Axis2 JAR ファイルがロードされます。 JVM サーバーが始動を正常に完了すると、JVMSERVER リソースが ENABLED 状態でインストールされます。

エラーが発生する場合 (CICS が JVM プロファイルの検出も読み取りもできない場合など)、JVM サーバーは始動できません。 JVMSERVER リソースは DISABLED 状態でインストールされ、CICS はシステム・ログにエラー・メッセージを発行します。

次のタスク

- Db2 や IBM MQ など、ネイティブ C ダイナミック・リンク・ライブラリー (DLL) ファイルを含むディレクトリーを指定します。これらのディレクトリーは、JVM プロファイルの LIBPATH_SUFFIX オプションで指定します。
- Web サービスでの Java の使用の説明に従って、JVM サーバーで Web サービス要求を実行するように CICS を構成します。

JVM プロファイルの例

Axis2 を開始するように構成された JVM プロファイルの例。

次の例では、Axis2 を開始するように構成された JVM プロファイルを抜粋して示しています。

```
#*********************************************************************
#
#                          Required parameters
#                          -------------------
#
# When using a JVM server, the set of CICS options that are supported
JAVA_HOME=/usr/lpp/java/J8.0_64
WORK_DIR=.
LIBPATH_SUFFIX=/usr/lpp/db2910/lib
...
#*********************************************************************
#
#                     JVM server specific parameters
#                     ------------------------------
#
JAVA_PIPELINE=YES
#
```

```
#*********************************************************************
#
#                              JVM options
#                              -----------
# The following option sets the Garbage collection Policy.
#
-Xgcpolicy:gencon
#
#*********************************************************************
#
#                 Setting user JVM system properties
#                 ----------------------------------
#
# -Dcom.ibm.cics.some.property=some_value
#
#*********************************************************************
#
#              Unix System Services Environment Variables
#              ------------------------------------------
#
JAVA_DUMP_OPTS="ONANYSIGNAL(JAVADUMP,SYSDUMP),ONINTERRUPT(NONE)"
#
#
```

CICS セキュリティー・トークン・サービスのための JVM サーバーの構成

SAML トークンを検証して処理する場合は、CICS セキュリティー・トークン・サービスを実行するように JVM サーバーを構成します。

このタスクについて

提供されるサンプルの DFHJVMST.jvmprofile は、CICS セキュリティー・トークン・サービスを実行する JVM サーバーに適しています。

JVM サーバーは、CICS オンライン・リソース定義を使用するか、CICS バンドルに含めることにより定義できます。 CICS Explorer を使用して CICS バンドル内のリソースを作成および編集する方法について詳しくは、CICS Explorer 製品資料内の『Working with bundles』を参照してください。

手順

JVM サーバー用の JVMSERVER リソースを作成します。

1. JVM サーバー用の JVM プロファイルの名前を指定します。 JVMPROFILE 属性では、1 文字から 8 文字の名前を指定します。この名前は、JVM サーバーの構成オプションを保持するファイルである JVM プロファイルの接頭部に使用されます。接尾部として .jvmprofile を指定する必要はありません。
2. JVM サーバーのスレッド限度を指定します。 スレッド数は、JVM サーバーで実行するワークロードによって異なります。始めにデフォルト値を受け入れてから、後で環境を調整できます。 1 つの JVM サーバーで最大 256 個のスレッドを設定できます。
3. JVM プロファイルを作成して、JVM サーバーの構成オプションを定義します。 JVM プロファイルは、システム初期設定パラメーター JVMPROFILEDIR で指定するディレクトリー内になければなりません。ベースとして、サンプル・プロファイル DFHJVMST.jvmprofile を使用することができます。このプロファイルには、JVM サーバーを開始するために適したオプションのサブセットが含ま

れています。 DFHJVMST.jvmprofile は、インストール・ディレクトリーから JVMPROFILEDIR で指定するディレクトリーにコピーすることも、CICS Explorer で選択してターゲット・ディレクトリーに保管することもできます。

JVM プロファイルのすべてのオプションと値については、『JVM プロファイルの検証およびプロパティー』で説明されています。 217 ページの『プロファイルのコーディング規則』のコーディング規則に従ってください。

サンプル・プロファイルを次のように変更します。

- JAVA_HOME を IBM Java SDK のインストール先に設定します。
- WORK_DIR を、JVM サーバーからのメッセージ、トレース、および出力の適当な宛先ディレクトリーに設定します。
- SECURITY_TOKEN_SERVICE を YES に設定します。
- TZ を、JVM サーバーからのメッセージに対するタイム・スタンプのタイム・ゾーンを指定するように設定します。英国の例は TZ=GMT0BST,M3.5.0,M10.4.0 です。

4. 変更内容を JVM プロファイルに保管します。 USS ファイル・システムでは、JVM プロファイルを EBCDIC として保管する必要があります。

タスクの結果

JVMSERVER リソースをインストールして使用可能にすると、CICS は Language Environment エンクレーブを作成し、JVM プロファイルから JVM サーバーにオプションを渡します。 JVM が始動し、OSGi フレームワークはすべての OSGi ミドルウェア・バンドルを解決します。 JVM サーバーが始動を正常に完了すると、JVMSERVER リソースが ENABLED 状態でインストールされます。

エラーが発生する場合、例えば、CICS が JVM プロファイルの検出も読み取りもできない場合、JVM サーバーは初期化できません。 JVMSERVER リソースは DISABLED 状態でインストールされ、CICS はエラー・メッセージを発行します。

次のタスク

JVM サーバーをさらにカスタマイズできます。例を以下に示します。

- Db2 や IBM MQ など、ネイティブ C ダイナミック・リンク・ライブラリー (DLL) ファイルを含むディレクトリーを指定します。これらのディレクトリーは LIBPATH_SUFFIX オプションで指定します。
- 詳しくは、Configuring the CICS Security Token Serviceを参照してください。

JVM プロファイルの検証およびプロパティー

JVM プロファイルには、開始時に JVM に渡される一連のオプションとシステム・プロパティーが含まれています。一部の JVM プロファイル・オプションは CICS 環境に固有であり、他の環境の JVM には使用されません。 CICS は、JVM サーバーの始動時に JVM プロファイルが正しくコーディングされていることを検証します。

JVM オプションについては、 220 ページの『CICS 環境における JVM のオプション』に説明があります。 CICS には、CICS でサポートされる JVM サーバー構成

ごとのサンプル・プロファイルが用意されています。これらのサンプル・プロファイルには、最も一般的な JVM オプションのデフォルト値があります。サンプル・プロファイルは zFS の /usr/lpp/cicsts/cicsts55/JVMProfiles/に保管されています。

また、JVM プロファイルで z/OS UNIX システム・サービス環境変数を指定することもできます。有効な JVM オプションではない名前と値のペアは、z/OS UNIX システム・サービス環境変数として扱われ、エクスポートされます。 JVM プロファイルで指定される z/OS UNIX システム・サービス環境変数は、そのプロファイルで作成される JVM にのみ適用されます。

環境変数の例として、Liberty プロファイルの WLP_INSTALL_DIR 変数や、JVM のタイム・ゾーンを変更するための TZ 変数があります。

Java クラス・ライブラリーには、JVM プロファイルに設定できる他のシステム・プロパティーが含まれています。例えば、アプリケーションに独自のシステム・プロパティーが含まれている場合があります。 IBM Java 資料は、主要な Java 情報源です。 JVM システム・プロパティーについて詳しくは、IBM SDK, Java Technology Edition バージョン 8 の z/OS ユーザーズ・ガイドを参照してください。

プロファイルのコーディング規則

任意の標準テキスト・エディターを使用して JVM プロファイルを編集することができます。JVM プロファイルをコーディングする際には、以下の規則に従ってください。

ディレクトリー

プロファイルでディレクトリーの値を指定する場合は、引用符を使用しないでください。

継続

オプションの場合、値は、テキスト・ファイル内の行の終わりで区切られます。入力または編集しようとする値が、エディターのウィンドウには長すぎる場合は、スクロールしないですむように改行することができます。次の行に継続するには、次の例のように、現在行の終わりに円記号 (¥) 文字とブランクの継続文字を付けます。

```
STDERR=/example/a/long/path/which/you/would/like\
 /to/break/over/a/line
```

同じ行に複数のオプションを置くことはできません。

コメント

コメントを追加するか、オプションを削除する代わりにコメント化するには、コメントの各行の先頭に # 記号を付けます。ファイルがランチャーによって読み取られるときに、コメント行は無視されます。

ブランク行も無視されます。オプション間、またはオプションのグループ間の分離文字としてブランク行を使用できます。

プロファイル構文解析コードは、インライン・コメントを除去します。インライン・コメントは以下のように定義されます。

- コメントの先頭に # 記号が付いている。

- 前に 1 つ以上のスペース (またはタブ) がある。
- 引用符付きテキストには含まれない。

表 29. インライン・コメントの例

コード	結果
`MYVAR=myValue  # Comment`	MYVAR=myValue
`MYVAR=#myValue  # Comment`	MYVAR=#myValue
`MYVAR=myValue  "# Quoted comment" # Comment`	MYVAR=myValue "# Quoted comment"

ファイルの組み込み

`%INCLUDE=<file_path>` を使用して、プロファイルにファイルを組み込みます。ファイルには、プロファイルとは別個に保守できる、システム全体に対する共通の構成を含めることができます。これにより、複数のプロファイルに共通する構成を共用することが可能になるので、プロファイルを制御しやすくして、保守を容易にすることができます。

以下の規則が適用されます。

- `<file_path>` は、zFS 内の完全修飾ファイルでなければなりません。
  - `<file_path>` の先頭で相対ディレクトリー (`.` や `..` など) を使用しないでください。それらは UNIX System Services では、Language Environment の現行作業ディレクトリーを基準とした相対ディレクトリーとして解釈されます。この位置は処理中に変更されることがあります。
  - `<file_path>` が存在しない場合や CICS 領域のユーザー ID に `<file_path>` に対する読み取り権限がない場合は、メッセージ DFHSJ1308 が発行されます。
- `<file_path>` には、`&USSCONFIG;` などのシンボルを含めることができます。
  - シンボルの `&DATE;` と `&TIME;` は許可されません。それらのフォーマットが、`%INCLUDE` ディレクティブの前にも後にも指定できる時間帯オプション (TZ) によって設定されるためです。
- `<file_path>` の内容は、EBCDIC または ASCII でエンコードする必要があります。
- %INCLUDE ディレクティブは、`<file_path>` の内容に置き換わります。
- プロファイルには、任意の数の %INCLUDE ディレクティブを含めることができます。
- 循環参照の結果として、`Skipping duplicate` というメッセージが表示されます。例えば、Profile-A に Profile-B を含めることや Profile-B に Profile-C を含めることは可能ですが、Profile-B に Profile-A が含まれている場合、そのディレクティブは無視されます。

オプションの複数インスタンス

同じオプションの複数のインスタンスがプロファイルに含まれている場合、最後に検出されるオプションの値が使用され、それより前の値は無視されます。

値の付加

指定された値を変数の既存の値に、コンマ区切りを使用して付加するには、その変数の前に + 文字を使用します。例えば、

```
LIBERTY_INCLUDE_XML=/path/file1
+LIBERTY_INCLUDE_XML=/path/file2
```

これは、次と同等です。

```
LIBERTY_INCLUDE_XML=/path/file1,/path/file2
```

JVM プロファイルに固有の規則

プロファイルの名前

- JVM プロファイルの名前の長さは最大 8 文字にすることができます。
- ファイル・システムでは JVM プロファイルのファイル拡張子を `.jvmprofile` にする必要があります。ファイル拡張子は小文字で設定され、これを変更してはなりません (JVM プロファイルにのみ適用されます)。
- 名前は、z/OS UNIX システム・サービス内のファイルに有効な任意の名前にすることができます。 DFH で始まる名前を使用しないでください。これらの文字は CICS が使用するために予約されているからです。
- プロファイルは UNIX ファイルであるため、大文字小文字が重要です。CICS で名前を指定する場合は、z/OS UNIX ファイル名にあるのと同じ大文字と小文字の組み合わせを使用して名前を入力する必要があります。

大/小文字の区別

パラメーター・キーワードおよびオペランドはすべて、大/小文字の区別があり、 220 ページの『CICS 環境における JVM のオプション』および 234 ページの『JVM システム・プロパティー』に示されているとおり正確に指定する必要があります。

CEDA

CEDA パネルは、端末の UCTRAN 設定にかかわらず、JVMPROFILE フィールドでの大文字小文字混合入力を受け入れます。ただし、コマンド行から CEDA を使用する場合、または別の CICS トランザクションを使用する場合は、大文字小文字混合で JVM プロファイルの名前を入力する必要があります。使用する端末が、大文字変換が抑止された状態で正しく構成されていることを確実にしてください。提供される CEOT トランザクションを使用して、現行セッションに対してのみ、独自の端末の英大文字変換状況 (UCTRAN) を変更することができます。

クラスパス分離文字

`CLASSPATH_SUFFIX` などのクラスパス・オプションで指定するディレクトリー・パスを分離するには、**:** (コロン) 文字を使用してください。

ストレージ・サイズ

JVM プロファイルでストレージ関連のオプションを指定する場合、ストレージ・サイズを 1024 バイトの倍数で指定してください。文字 K は KB を、文字 M は MB を、および文字 G は GB をそれぞれ表します。例えば、ヒープの初期サイズとして 6 291 456 バイトを指定するには、以下のいずれかの方法で **-Xms** をコーディングします。

```
-Xms6144K
-Xms6M
```

CICS 環境における JVM のオプション

JVM プロファイル内のオプションは JVM サーバーを始動するために CICS によって使用されます。一部のオプションは CICS に固有ですが、環境変数と Java システム・プロパティーを指定することもできます。

コーディング規則

JVM オプションを指定する際には、コーディング規則に従っていることを確認してください。詳しくは、 217 ページの『プロファイルのコーディング規則』を参照してください。

フォーマット

オプションの形式は、変わる場合があります。

- JVM プロファイル内のオプションでは、キーワードと値が等号 (=) によって区切られる形式 (例: `JAVA_PIPELINE=TRUE`) またはハイフンで始まる形式 (例: `-Xmx16M`) が使用されます。
- キーワードと値のペアは、`JAVA_PIPELINE=TRUE` などの CICS 変数であるか、あるいは CICS オプションとして認識されない場合には z/OS UNIX System Services の環境変数として扱われ、エクスポートされます。
- JVM プロファイル内の `-D` で始まるオプションは JVM システム・プロパティーです。 `-X` で始まるオプションは JVM コマンド行オプションとして扱われます。 `-` で始まるオプションは、置換シンボルが展開された後に JVM に渡されます。詳しくは、 234 ページの『JVM システム・プロパティー』を参照してください。

JVM プロファイルで使用されるシンボル:

JVM プロファイルに指定する変数または JVM サーバー・プロパティーでは、置換シンボルを使用できます。これらのシンボルの値は JVM サーバーの始動時に決定されるため、多くの JVM サーバーおよび CICS 領域で共通のプロファイルを使用できます。

以下のシンボルがサポートされています。

&APPLID;
このシンボルを使用すると、実行時に CICS 領域のアプリケーション ID に置換されます。この方法ですべての領域に同じプロファイルを使用でき、同時に領域固有の作業ディレクトリーまたは出力宛先を用いることができます。APPLID は常に大文字です。

&BUNDLE;
このシンボルを使用すると、シンボルは、JVM サーバーのインストール元となっている CICS バンドルの名前に置き換えられます。

&BUNDLEID;
このシンボルを使用すると、シンボルは、JVM サーバーのインストール元となっている CICS バンドルの ID に置き換えられます。

&CONFIGROOT;
このシンボルを使用すると、JVM プロファイルが入っているディレクトリーの絶対パスが実行時に置換されます。 CICS バンドル内で定義される JVM サーバーの場合、JVM プロファイルはデフォルトでバンドルのルート・ディレクトリーの中に置かれます。他の方法で定義される JVM サーバーの場合、JVM プロファイルは JVMPROFILEDIR システム初期設定パラメーターで指定されたディレクトリーの中に置かれます。

&DATE;
このシンボルを使用すると、シンボルは実行時に *Dyymmdd* 形式の現在日付で置き換えられます。

&JVMSERVER;
このシンボルを使用する際、JVMSERVER リソースの名前は実行時に置換されます。JVM サーバーごとに固有の出力またはダンプ・ファイルを作成する場合に、このシンボルを使用します。

&TIME;
このシンボルを使用すると、シンボルは実行時に *Thhmmss* 形式の JVM 開始時刻で置き換えられます。

&USSCONFIG;
このシンボルを使用すると、シンボルは、USSCONFIG システム初期設定パラメーターの値に置き換えられます。これは CICS 構成ファイルのディレクトリーです。

&USSHOME;
このシンボルを使用すると、シンボルは USSHOME システム初期設定パラメーターの値に置き換えられます。このシンボルを指定することで、CICS が Java と Liberty プロファイル用にライブラリーを提供する、z/OS UNIX のホーム・ディレクトリーを自動的に取得できます。

JVM サーバー・プロファイルのオプション**:**

JVM サーバー・オプション、それらが JVM サーバーのさまざまな用途に適用される方法、およびそれらの説明をリストに記載します。

JVM サーバーのさまざまな用途に対するオプションの適用方法

次の表に、JVM サーバーの特定の用途に対して、各オプションが必須であるか、任意指定であるか、それともサポート対象外であるかを示します。

表 *30. JVM* サーバー・オプション、およびそれらが *JVM* サーバーのさまざまな用途に適用される方法

オプション	**OSGi**	**Liberty**	**Axis2**	**STS**
_DFH_UMASK	オプション	オプション	オプション	オプション

表 30. *JVM* サーバー・オプション、およびそれらが *JVM* サーバーのさまざまな用途に適用される方法 (続き)

オプション	**OSGi**	**Liberty**	**Axis2**	**STS**
CICS_WLP_MODE	サポート対象外	オプション	サポート対象外	サポート対象外
CLASSPATH_PREFIX	サポート対象外	サポート対象外	オプション	オプション
CLASSPATH_SUFFIX	サポート対象外	サポート対象外	オプション	オプション
DISPLAY_JAVA_VERSION	オプション	オプション	オプション	オプション
IDENTITY_PREFIX	オプション	オプション	オプション	オプション
JAVA_DUMP_TDUMP_PATTERN	オプション	オプション	オプション	オプション
JAVA_HOME	必須	必須	必須	必須
JAVA_PIPELINE	サポート対象外	サポート対象外	必須	サポート対象外
JNDI_REGISTRATION	オプション	サポート対象外	オプション	オプション
JVMLOG	オプション	オプション	オプション	オプション
JVMTRACE	オプション	オプション	オプション	オプション
LIBERTY_INCLUDE_XML	サポート対象外	サポートの有無	サポート対象外	サポート対象外
LIBPATH_PREFIX	オプション - IBM サービス担当員の指示があった場合にのみ使用	オプション - IBM サービス担当員の指示があった場合にのみ使用	オプション - IBM サービス担当員の指示があった場合にのみ使用	オプション - IBM サービス担当員の指示があった場合にのみ使用
LIBPATH_SUFFIX	オプション	オプション	オプション	オプション
LOG_FILES_MAX	オプション	オプション	オプション	オプション
LOG_LEVEL	オプション	オプション	オプション	オプション
LOG_PATH_COMPATIBILITY	オプション	オプション	オプション	オプション
OSGI_BUNDLES	オプション	サポート対象外	サポート対象外	サポート対象外
OSGI_CONSOLE	オプション	サポート対象外	サポート対象外	サポート対象外
OSGI_FRAMEWORK_TIMEOUT	オプション	オプション	サポート対象外	サポート対象外
PRINT_JVM_OPTIONS	オプション	オプション	オプション	オプション
PRINT_PROFILE	オプション	オプション	オプション	オプション
SECURITY_TOKEN_SERVICE	サポート対象外	サポート対象外	サポート対象外	必須
STDERR	オプション	オプション	オプション	オプション
STDIN	オプション	オプション	オプション	オプション
STDOUT	オプション	オプション	オプション	オプション
USEROUTPUTCLASS	オプション	サポート対象外	オプション	オプション
WLP_INSTALL_DIR	サポート対象外	必須	サポート対象外	サポート対象外
WLP_LINK_TIMEOUT	サポート対象外	オプション	サポート対象外	サポート対象外
WLP_OUTPUT_DIR	サポート対象外	オプション	サポート対象外	サポート対象外
WLP_USER_DIR	サポート対象外	オプション	サポート対象外	サポート対象外
WLP_ZOS_PLATFORM	サポート対象外	オプション	サポート対象外	サポート対象外
WORK_DIR	オプション	オプション	オプション	オプション
WSDL_VALIDATOR	オプション	サポート対象外	オプション	オプション
ZCEE_INSTALL_DIR	サポート対象外	オプション	サポート対象外	サポート対象外

JVM サーバーのオプションと説明

デフォルト値 (該当する場合) とは、オプションが指定されていない場合に CICS で使用される値のことです。サンプル JVM プロファイルで、デフォルト値とは異なる値が指定される場合があります。

注: これまでの資料に記載されていた YES|NO もまだ使用可能ですが、推奨される構文は TRUE|FALSE です。

_DFH_UMASK={007|nnn}

JVMSERVER ファイルの作成時に適用される UNIX System Services プロセス UMASK を設定します。この値は 3 桁の 8 進数です。例えば、デフォルト値 007 を使用した場合、ファイルの作成時に所有者およびグループの read/write/execute アクセス権は許可されますが、other (他のユーザー) には read/write/execute は付与されません。指定する値は 000 (最小の制限) から 777 (最大の制限) の範囲内でなければなりません。UMASK は JVM の存続期間中適用されます。

CICS_WLP_MODE={INTEGRATED|STANDARD}

Liberty JVM サーバーの場合に、CICS と Liberty の統合のレベルを選択します。

CICS 統合モードの Liberty を使用するには、INTEGRATED モードを指定します。Liberty JVM サーバーは、CICS 対応スレッドを使用して実行され、CICS セキュリティーに従い、CICS の作業単位と統合され、CICS (JCICS) API の Java クラス・ライブラリーを Java Web アプリケーションに提供します。このオプションを省略した場合、またはこのオプションが有効でない場合は、デフォルトの INTEGRATED が使用されます。

CICS 標準モードの Liberty を使用するには、STANDARD モードを指定します。Liberty JVM サーバーは、Liberty のすべてのサポート対象プラットフォームにとってより標準的なモードで実行されます。このモードを使用する場合は、他のプラットフォームの Liberty アプリケーションを変更せずに CICS に移植してデプロイすることができます。JVM サーバーは、Liberty サーバーの制御を保持し、サーバーの作成、ライフサイクル、および構成を管理します。ただし、スレッドはデフォルトでは CICS 対応ではなく、CICS トランザクション・コンテキスト内で実行されません。CICS 作業単位の統合、CICS セキュリティーの統合、および JCICS API を Java アプリケーションから直接使用することはできません。

CLASSPATH_PREFIX, CLASSPATH_SUFFIX=*class_pathnames*

これらのオプションを使用して、OSGi 非対応の JVM によって検索されるディレクトリー・パス、Java アーカイブ・ファイル、圧縮ファイルを指定します。例えば、Java Web サービスに使用されます。 OSGi フレームワークは自動的にクラス・ロードを処理するので、OSGi フレームワークを使用する場合はクラスパスを設定しないでください。これらのオプションを使用して、Axis2 の標準クラスパスを指定する場合、Axis2 エンジンを開始するために、JAVA_PIPELINE=TRUE を指定する必要もあります。

CLASSPATH_PREFIX は、クラスパス・エントリーを標準クラスパスの先頭に追加し、CLASSPATH_SUFFIX は標準クラスパスの末尾に追加します。複数行にまたがってエントリーを指定するには、継続行の最後に ¥ (バックスラッシュ) を使用します。

CLASSPATH_PREFIX オプションは注意して使用してください。CLASSPATH_PREFIX のクラスは、CICS と Java ランタイムによって提供される、同じ名前のクラスより優先されるため、誤ったクラスがロードされる可能性があります。

CICS は、**USSHOME** システム初期設定パラメーターと JVM プロファイルの JAVA_HOME オプションで指定されたディレクトリーの /lib サブディレクトリーを使用して、JVM の基本クラスパスを作成します。この基本クラスパスには、CICS および JVM によって提供される Java アーカイブ・ファイルが含まれます。それは JVM プロファイルでは見られません。 JVM プロファイルのクラスパスでこれらのファイルを再度指定することはありません。

オプション CLASSPATH_PREFIX または CLASSPATH_SUFFIX を使用して複数の項目を指定する場合は、コンマではなくコロンを使用して項目を区切ってください。

DISPLAY_JAVA_VERSION={TRUE|FALSE}

このオプションを TRUE に設定すると、アプリケーションによって JVM が始動されるたびに、CICS は、使用される IBM Software Developer Kit for z/OS、Java Technology Edition のバージョンとビルドを示すメッセージ DFHSJ0901 を MSGUSER ログに書き込みます。

IDENTITY_PREFIX={TRUE|FALSE}

JVM サーバー出力の発信元を示すために、JES に転送されるすべての STDOUT および STDERR 項目は、接頭辞として JVM サーバー名の文字列を付けて書き込まれます。これは、複数の JVM サーバーで 1 つの JES 宛先を共用する場合に便利です。この動作を無効にするには、IDENTITY_PREFIX=FALSE を設定します。この設定により、接頭辞文字列が使用不可になります。

JAVA_DUMP_TDUMP_PATTERN=

JVM からのトランザクション・ダンプ (TDUMP) に使用するファイル名パターンを指定する z/OS UNIX System Services 環境変数。 Java TDUMP は、JVM 異常終了の場合にデータ・セット宛先に書き込まれます。

JAVA_HOME=/usr/lpp/java/*javadir*/

z/OS UNIX で IBM 64-bit SDK for z/OS, Java テクノロジー・エディションのインストール場所を指定します。この場所には、Java サポートに必要なサブディレクトリーと Java アーカイブ・ファイルが入ります。

提供されたサンプル JVM プロファイルには、DFHISTAR CICS インストール・ジョブで **JAVADIR** パラメーターによって生成されたパスが含まれています。**JAVADIR** パラメーターのデフォルトは、java/J8.0_64/ であり、これは、IBM 64-bit SDK for z/OS, Java テクノロジー・エディション のデフォルトのインストール場所です。この値では、JVM プロファイルの JAVA_HOME 設定値は /usr/lpp/java/J8.0_64/ になります。

JAVA_PIPELINE={TRUE|FALSE}

Java 標準 SOAP パイプラインでの Web サービス処理を JVM サーバーがサポートできるように、必要な Java アーカイブ・ファイルをクラスパスに追加し

ます。デフォルト値は FALSE です。この値を設定すると、JVM サーバーは、OSGi ではなく、Axis2 をサポートするように構成されます。CLASSPATH オプションを使用して、クラスパスに JAR ファイルをさらに追加することができます。

注: オプション JAVA_PIPELINE=TRUE と SECURITY_TOKEN_SERVICE=TRUE を同時に指定することはできません。

JNDI_REGISTRATION={TRUE|FALSE}

Java アプリケーションによる JNDI の使用をサポートするために、JNDI 登録 JAR ファイルを JVM ランタイム環境に自動的に追加することを指定します。Liberty JVM サーバーに関しては、このオプションは無視されます。JNDI_REGISTRATION=FALSE を設定すると、これらのファイルの自動追加をオプトアウトできます。この機能が不要な場合は、オプトアウトすることで、新しい JAR ファイルとの競合の可能性を排除し、JVM のフットプリントを小さく維持して不要なクラス・ロードを回避できます。

JVMLOG={{&APPLID;.&JVMSERVER;.}Dyyyymmdd.Thhmmss.dfhjvmlog|*file_name*|JOBLOG|//DD:data_definition}

JVM サーバーの動作中に JVM サーバーのロギングを書き込む z/OS UNIX ファイルまたは JES DD の名前を指定します。このオプションに値を設定しない場合、CICS によって自動的に、JVM サーバーごとに固有のログ・ファイルが作成されます。

JVMLOG がデフォルトのままであるか、相対ファイル名である場合、出力の場所は LOG_PATH_COMPATIBILITY オプションによって決まります。LOG_PATH_COMPATIBILITY=FALSE を指定すると、ファイルは WORK_DIR/applid/jvmserver ディレクトリーに置かれます。LOG_PATH_COMPATIBILITY=TRUE を指定すると、ファイルは WORK_DIR ディレクトリーに置かれます。

絶対ファイル名を JVMLOG で指定した場合、CICS は存在しないディレクトリーをそのパス内に作成します。

ファイルが存在する場合には、そのファイルの末尾に出力が追加されます。JVM サーバーごとに固有の出力ファイルを作成するには、サンプルの JVM プロファイルに示されているように JVMSERVER シンボルと APPLID シンボルをファイル名に使用します。JVMLOG がデフォルトのままである場合、CICS は APPLID と JVMSERVER シンボル、および JVM サーバーが固有の出力ファイルの作成を開始した日時のタイムスタンプを使用します。

JES の DD に転送するには、構文 //DD:data_definition を使用して、JES のデータ定義名を指定します。

このオプションを JOBLOG に設定すると、JVMLOG は現在の stdout の場所に転送されます。

JVMTRACE={{&APPLID;.&JVMSERVER;.}Dyyyymmdd.Thhmmss.dfhjvmtrc|*file_name*|JOBLOG|//DD:data_definition}

JVM サーバーの稼働中に JVM サーバーのトレースを書き込む z/OS UNIX ファイルまたは JES DD の名前を指定します。このオプションに値を設定しない場合、CICS によって自動的に、JVM サーバーごとに固有のトレース・ファイルが作成されます。

JVMTRACE がデフォルトのままであるか、相対ファイル名である場合、出力の場所は LOG_PATH_COMPATIBILITY オプションによって決まります。
LOG_PATH_COMPATIBILITY=FALSE を指定すると、ファイルは WORK_DIR/applid/jvmserver ディレクトリーに置かれます。
LOG_PATH_COMPATIBILITY=TRUE を指定すると、ファイルは WORK_DIR ディレクトリーに置かれます。

絶対ファイル名を JVMTRACE で指定した場合、CICS は存在しないディレクトリーをそのパス内に作成します。

ファイルが存在する場合には、そのファイルの末尾に出力が追加されます。JVM サーバーごとに固有の出力ファイルを作成するには、サンプルの JVM プロファイルに示されているように JVMSERVER シンボルと APPLID シンボルをファイル名に使用します。JVMTRACE がデフォルトのままである場合、CICS は APPLID と JVMSERVER シンボル、および JVM サーバーが固有の出力ファイルの作成を開始した日時のタイムスタンプを使用します。

JES の DD に転送するには、構文 //DD:data_definition を使用して、JES のデータ定義名を指定します。

このオプションを JOBLOG に設定すると、JVMTRACE は現在の stdout の場所に転送されます。

LIBERTY_INCLUDE_XML=filenames

server.xml に <include> エレメントとして追加されるファイル (またはコンマで区切られたファイルのリスト) を指定します。

変数の前に + 文字を使用すると、コンマおよび指定値が、その変数の既存値に付加されます。以下に例を示します。

```
LIBERTY_INCLUDE_XML=/path/file1
+LIBERTY_INCLUDE_XML=/path/file2
```

これは、次と同等です。

```
LIBERTY_INCLUDE_XML=/path/file1,/path/file2
```

LIBPATH_PREFIX, LIBPATH_SUFFIX=*pathnames*

JVM によって使用される、z/OS UNIX で拡張子 .so を持つネイティブ C ダイナミック・リンク・ライブラリー (DLL) ファイルを検索するときの対象となるディレクトリー・パスを指定します。これには、アプリケーション・コードまたはサービスによってロードされる JVM および追加のネイティブ・ライブラリーを実行するのに必要なファイルが含まれます。

JVM の基本ライブラリー・パスは、**USSHOME** システム初期設定パラメーターと JVM プロファイルの JAVA_HOME オプションで指定されたディレクトリーを使用して自動的に作成されます。この基本ライブラリー・パスは、JVM プロファイルでは表示されません。このライブラリー・パスには、CICS が使用する JVM とネイティブ・ライブラリーを実行するのに必要なすべての DLL ファイルが含まれています。

LIBPATH_SUFFIX オプションを使用すると、このライブラリー・パスを拡張できます。このオプションはディレクトリーを、ライブラリー・パスの最後、基本ライブラリー・パスの後に追加します。このオプションは、アプリケーションで使用する追加のネイティブ・ライブラリーが含まれるディレクトリーを指定するのに使用します。また、このオプションは、CICS の標準 JVM セットアップに含まれていないサービスで使用するディレクトリーを指定するのに使用します。例えば、追加のネイティブ・ライブラリーには、Db2 JDBC ドライバーを使用するのに必要な DLL ファイルを含めることができます。

LIBPATH_PREFIX オプションは、ライブラリー・パスの先頭で、基本ライブラリー・パスの前にディレクトリーを追加します。このオプションは注意して使用してください。 指定されたディレクトリー内の DLL ファイルが、基本ライブラリー・パス上の DLL ファイルと同じ名前である場合は、提供されたファイルの代わりにロードされます。

オプション LIBPATH_PREFIX または LIBPATH_SUFFIX を使って複数の項目を指定するときには、コンマではなくコロンを使ってそれらを区切ってください。

アプリケーションで使用するためにライブラリー・パス上にある DLL ファイルをコンパイルし、XPLink オプションを指定してリンクする必要があります。XPLink オプションを指定してコンパイルおよびリンクすると、最適なパフォーマンスが得られます。基本ライブラリー・パスで提供される DLL ファイルおよび Db2 JDBC ドライバーなどのサービスで使用される DLL ファイルは、XPLink オプションを指定して作成されます。

LOG_FILES_MAX={0|*number*}

システム上に保持する古いログ・ファイルの数を指定します。デフォルト設定の 0 では、すべての古いバージョンのログ・ファイルが保持されます。この値を変更して、ファイル・システム上に保持する古いログ・ファイルの数を指定できます。

LOG_PATH_COMPATIBILITY=TRUE の場合は、**LOG_FILES_MAX** は無視されます。

If **STDOUT**、 **STDERR**、**JVMLOG**、および **JVMTRACE** がデフォルト・スキームを使用している場合、またはカスタマイズされている場合は、&DATE;.&TIME; パターンを含めると、各ログ・タイプの最新の nn のみがシステムに保持されます。カスタマイズに出力を固有にする変数が含まれていない場合は、ファイルが追加され、削除の要求はありません。出力変数がカスタマイズされ、DD:// または JOBLOG に出力するよう経路指定されている場合、クリーンアップは適用されません。特殊値 0 は、削除しないことを意味します。

LOG_LEVEL={INFO|WARNING|ERROR|NONE}

dfhjvmlog ファイルに入れて戻されるログ記録情報についての制御を提供します。値が NONE の場合、すべての出力は抑止され、ファイルは空になります。それ以外の値は、dfhjvmlog ファイルに書き込まれる最低ログ・タイプを示します。例えば、WARNING を選択すると、WARNING レベル以上のログ項目になります。

LOG_PATH_COMPATIBILITY={TRUE|FALSE}

この動作のデフォルト値は LOG_PATH_COMPATIBILITY=FALSE です。この場合は、統合されたログ出力動作になります。この新しい動作では、**JVMSERVER** ログ・ファイルは、**JVMSERVER** の既存のサブコンポーネント (OSGi フレームワ

ークや Liberty サーバーなど) によって使用されるものと同じ出力ディレクトリー構造に配置されます。以前のリリースの動作に戻すには、このパラメーターを LOG_PATH_COMPATIBILITY=TRUE に設定します。すると、**JVMSERVER** ログ・ディレクトリーは元の場所に作成されます。

OSGI_BUNDLES=*pathnames*

OSGi JVM サーバーの OSGi フレームワークで有効なミドルウェア・バンドルのディレクトリー・パスを指定します。これらの OSGi バンドルには、IBM MQ または Db2 への接続などのシステム機能をフレームワークに実装するためのクラスが含まれています。複数の OSGi バンドルを指定する場合は、コンマを使用してそれらのバンドルを分離してください。

OSGI_CONSOLE={TRUE|FALSE}

必要な OSGi バンドルを OSGi フレームワークに追加し、OSGi コンソールを使用可能にします。また、プロパティー `-Dosgi.console=host:port` および `-Dosgi.file.encoding={ISO-8859-1|US-ASCII|ASCII}` を JVM プロファイルに設定する必要もあります。デフォルト値は FALSE です。OSGi バンドルおよびサービスの状態を確認する場合は、Troubleshooting Java applicationsを参照してください。

OSGI_FRAMEWORK_TIMEOUT={60|*number***}**

OSGi フレームワークの初期設定またはシャットダウンがタイムアウトになるまでに CICS が待機する秒数を指定します。1 秒から 60000 秒までの範囲で値を設定できます。デフォルト値は 60 秒です。 OSGi フレームワークが開始するのに、指定された秒数よりも長くかかる場合、JVM サーバーは初期設定できず、DFHSJ0215 メッセージが CICS によって発行されます。 zFS の JVM サーバー・ログ・ファイルにもエラー・メッセージが書き込まれます。OSGi フレームワークがシャットダウンするのに、指定された秒数よりも長くかかる場合、JVM サーバーは正常にシャットダウンできません。

PRINT_JVM_OPTIONS={TRUE|FALSE}

このオプションを TRUE に設定すると、JVM が始動するたびに、始動時の JVM に渡されるオプションも SYSPRINT に出力されます。 JVM がプロファイル内でこのオプションを指定して開始されるたびに、出力が生成されます。このオプションを使用すると、JVM プロファイルでは見えない、特定の JVM プロファイルのクラスパス (CICS によって作成される基本ライブラリー・パスと基本クラスパスを含む) の内容を確認できます。

PRINT_PROFILE={TRUE|FALSE}

このオプションを TRUE に設定すると、JVM サーバーとアプリケーションに渡されるプロファイル内のオプション、システム・プロパティー、および環境変数は、SYSPRINT に出力されます。

SECURITY_TOKEN_SERVICE={TRUE|FALSE}

このオプションを TRUE に設定すると、JVM サーバーはセキュリティー・トークンを使用できます。このオプションを FALSE に設定すると、JVM サーバーのセキュリティー・トークン・サービスのサポートは無効になります。

注**:** オプション SECURITY_TOKEN_SERVICE=TRUE と JAVA_PIPELINE=TRUE を同時に指定することはできません。

STDERR={{&APPLID;.
&JVMSERVER;.}Dyyyymmdd.Thhmmss.dfhjvmerr|*file_name*|JOBLOG|//
DD:data_definition}

stderr ストリームをリダイレクトする z/OS UNIX ファイルまたは JES DD の名前を指定します。このオプションに値を設定しない場合、CICS によって自動的に、JVM サーバーごとに固有のトレース・ファイルが作成されます。

STDERR がデフォルトのままであるか、相対ファイル名である場合、出力の場所は LOG_PATH_COMPATIBILITY オプションによって決まります。
LOG_PATH_COMPATIBILITY=FALSE を指定すると、ファイルは WORK_DIR/applid/jvmserver ディレクトリーに置かれます。
LOG_PATH_COMPATIBILITY=TRUE を指定すると、ファイルは WORK_DIR ディレクトリーに置かれます。

絶対ファイル名を STDERR で指定した場合、CICS は存在しないディレクトリーをそのパス内に作成します。

ファイルが存在する場合には、そのファイルの末尾に出力が追加されます。JVM サーバーごとに固有の出力ファイルを作成するには、サンプルの JVM プロファイルに示されているように JVMSERVER シンボルと APPLID シンボルをファイル名に使用します。STDERR がデフォルトのままである場合、CICS は APPLID と JVMSERVER シンボル、および JVM サーバーが固有の出力ファイルの作成を開始した日時のタイムスタンプを使用します。

JES の DD に転送するには、構文 //DD:data_definition を使用して、JES のデータ定義名を指定します。

このオプションを JOBLOG に設定した場合、STDERR は SYSOUT が定義されていれば SYSOUT に、定義されていなければ動的な SYSnnn に転送されます。

JVM プロファイルで USEROUTPUTCLASS オプションを指定すると、このオプションで指定された Java クラスが、代わりに System.err 要求を処理します。USEROUTPUTCLASS オプションで指定したクラスが目的の宛先にデータを書き込めない場合も、STDERR オプションで指定した z/OS UNIX ファイルが使用されることがあります。例えば、用意されているサンプル・クラス com.ibm.cics.samples.SJMergedStream を使用する場合です。またこのファイルは、USEROUTPUTCLASS オプションで指定されたクラスによって、他の何らかの理由のために出力がそのファイルに送信される場合にも使用できます。

STDIN=*file_name*

stdin ストリームの読み取り元の z/OS UNIX ファイルの名前を指定します。このオプションに値が指定されない限り、CICS はこのファイルを作成しません。

STDOUT={{&APPLID;.
&JVMSERVER;.}Dyyyymmdd.Thhmmss.dfhjvmout|*file_name*|JOBLOG|//
DD:data_definition}

stdout ストリームをリダイレクトする z/OS UNIX ファイルまたは JES DD の名前を指定します。このオプションに値を設定しない場合、CICS によって自動的に、JVM サーバーごとに固有のトレース・ファイルが作成されます。

STDOUT がデフォルトのままであるか、相対ファイル名である場合、出力の場所は LOG_PATH_COMPATIBILITY オプションによって決まります。

LOG_PATH_COMPATIBILITY=FALSE を指定すると、ファイルは WORK_DIR/applid/jvmserver ディレクトリーに置かれます。LOG_PATH_COMPATIBILITY=TRUE を指定すると、ファイルは WORK_DIR ディレクトリーに置かれます。

絶対ファイル名を STDOUT で指定した場合、CICS は存在しないディレクトリーをそのパス内に作成します。

ファイルが存在する場合には、そのファイルの末尾に出力が追加されます。JVM サーバーごとに固有の出力ファイルを作成するには、サンプルの JVM プロファイルに示されているように JVMSERVER シンボルと APPLID シンボルをファイル名に使用します。STDOUT がデフォルトのままである場合、CICS は APPLID と JVMSERVER シンボル、および JVM サーバーが固有の出力ファイルの作成を開始した日時のタイムスタンプを使用します。

JES の DD に転送するには、構文 //DD:data_definition を使用して、JES のデータ定義名を指定します。

このオプションを JOBLOG に設定した場合、STDOUT は SYSPRINT が定義されていれば SYSPRINT に、定義されていなければ動的な SYSnnn に転送されます。

JVM プロファイルで USEROUTPUTCLASS オプションを指定した場合は、このオプションで指定した Java クラスが、代わりに System.out 要求を処理します。USEROUTPUTCLASS オプションで指定したクラスが目的の宛先にデータを書き込めない場合も、STDOUT オプションで指定した z/OS UNIX ファイルが使用されることがあります。例えば、用意されているサンプル・クラス com.ibm.cics.samples.SJMergedStream を使用する場合です。またこのファイルは、USEROUTPUTCLASS オプションで指定されたクラスによって、他の何らかの理由のために出力がそのファイルに送信される場合にも使用できます。

USEROUTPUTCLASS=*classname*

JVM からの出力および JVM 内部からのメッセージを代行受信する Java クラスの完全修飾名を指定します。この Java クラスを使って JVM からの出力とメッセージをリダイレクトし、出力レコードにタイム・スタンプとヘッダーを追加することができます。これは Liberty ではサポートされていません。Java クラスがデータを目的の宛先に書き込めない場合、STDOUT および STDERR オプションで指定されたファイルが引き続き使用される場合があります。

USEROUTPUTCLASS オプションを指定すると、JVM のパフォーマンスに悪影響が出ます。実稼働環境で最高のパフォーマンスを発揮するには、このオプションは使用しないでください。ただし、このオプションは、JVM 出力を識別可能な宛先に送信できるので、同じ CICS 領域を使用するアプリケーション開発者に便利な場合があります。

このクラスおよび提供されたサンプルについて詳しくは、JVM 出力、ログ、ダンプ、およびトレースの場所の制御を参照してください。

WLP_INSTALL_DIR={&USSHOME;/wlp}

Liberty プロファイル・テクノロジーのインストール・ディレクトリーを指定します。 Liberty プロファイルは、CICS の z/OS UNIX ホームにある wlp というサブディレクトリーにインストールされます。デフォルトのインストール・

ディレクトリーは /usr/lpp/cicsts/cicsts55/wlp です。常に &USSHOME; シンボルを使用して、正しいファイル・パスを設定し、wlp ディレクトリーを追加します。

この環境変数は、Liberty JVM サーバーを始動する場合に必要です。この環境変数を設定すると、Liberty JVM サーバーを構成するための他の環境変数およびシステム・プロパティーも指定できます。環境変数の接頭部は WLP です。また、システム・プロパティーについては、 234 ページの『JVM システム・プロパティー』に説明されています。

WLP_LINK_TIMEOUT={30000|*number*}

Liberty JVM サーバーでアプリケーションの起動要求のディスパッチを CICS が待機する、タイムアウトになるまでの時間をミリ秒で指定します。 0 を指定した場合、CICS は無期限に待機します。デフォルト値は 30000 ミリ秒です。指定されたミリ秒数の後に Liberty JVM サーバーにタスクがディスパッチされていない場合、EXEC CICS LINK コマンドは失敗し、 DFHSJ1006 メッセージが CICS によって発行されます。

WLP_OUTPUT_DIR=$WLP_USER_DIR/servers

Liberty プロファイルの出力ファイルを格納するディレクトリーを指定します。デフォルトでは、Liberty プロファイルは、サーバーにちなんだ名前を持つディレクトリー内に、そのサーバーに関するログ、作業域、構成ファイル、アプリケーションを格納します。

この環境変数はオプションです。これを指定しない場合、CICS はデフォルトで *$WORK_DIR/&APPLID;/&JVMSERVER;*/wlp/usr/servers を設定し、シンボルをランタイム値に置き換えます。

この環境変数が設定されている場合、出力ログと作業域は $WLP_OUTPUT_DIR/*server_name* に保管されます。

WLP_USER_DIR={&APPLID;/&JVMSERVER;/wlp/usr/|*directory_path*}

Liberty JVM サーバーの構成ファイルを格納するディレクトリーを指定します。この環境変数はオプションです。これを指定しない場合、CICS は作業ディレクトリー内の *&APPLID;/&JVMSERVER;*/wlp/usr/ を使用し、シンボルをランタイム値に置き換えます。構成ファイルは servers/*server_name* に書き込まれます。

WLP_ZOS_PLATFORM={TRUE|FALSE}

重要: このオプションは、バージョン 5.5 では推奨されません。現在では、同じ領域内に複数の完全構成 Liberty サーバーが可能になったためです。
Liberty JVM サーバーでの z/OS プラットフォーム拡張を無効にします。このモードでは、cicsts:security-1.0 および cicsts:distributedIdentity-1.0 機能の使用は許可されていません。

WORK_DIR={.|/tmp|*directory_name*}

JVMSERVER に関連するアクティビティー用に CICS 領域が使用する z/OS UNIX 上の作業ディレクトリーを指定します。 CICS JVMSERVER は、このディレクトリーを構成および出力の手段として使用します。提供された JVM プロファイルではピリオド (.) が定義されています。これは、CICS 領域ユーザー ID のホーム・ディレクトリーが作業ディレクトリーとして使われることを示し

ます。このディレクトリーは、CICS インストール時に作成されます。このディレクトリーが存在しないか、WORK_DIR が省略されている場合には、z/OS UNIX ディレクトリー名として /tmp が使用されます。

作業ディレクトリーの絶対パスまたは相対パスを指定できます。作業ディレクトリーの相対パスは、CICS 領域ユーザー ID のホーム・ディレクトリーから見た相対パスです。ホーム・ディレクトリーを Java に関連した活動の作業ディレクトリーとして使用しない場合や、CICS 領域が z/OS ユーザー ID (UID) を共用しているために同じホーム・ディレクトリーを持っている場合には、CICS 領域ごとに異なる作業ディレクトリーを作成できます。

&APPLID; シンボル (これは CICS によって実際の CICS 領域 APPLID に置換されます) を使用するディレクトリー名を指定すると、すべての CICS 領域で JVM プロファイルのセットを共用する場合であっても、領域ごとに固有の作業ディレクトリーを作成できます。例えば、次のように指定するとします。

```
WORK_DIR=/u/&APPLID;/javaoutput
```

その JVM プロファイルを使用する各 CICS 領域は、それぞれ独自の作業ディレクトリーを持ちます。該当するディレクトリーが z/OS UNIX 上に作成されていること、およびそれに対する読み取り/書き込み/実行アクセス権限が CICS 領域に与えられていることを確認してください。

また、作業ディレクトリーの固定名を指定することもできます。この場合は、適切なディレクトリーが z/OS UNIX に作成され、アクセス権限が正しい CICS 領域に付与されていることを確認する必要もあります。作業ディレクトリーに固定名を使用する場合は、JVM プロファイルを共用する CICS 領域内のすべての JVM サーバーからの出力ファイルが、そのディレクトリーに作成されます。ご使用の出力ファイルに固定のファイル名を使用する場合には、こうした CICS 領域内のすべての JVM サーバーからの出力は同じ z/OS UNIX ファイルに追加されます。同じファイルに追加されないようにするには、JVMSERVER シンボルと APPLID シンボルを使用して、JVM サーバーごとに固有の出力およびダンプ・ファイルを生成します。

USSHOME システム初期設定パラメーターで定義された CICS ファイルのホーム・ディレクトリーである、z/OS UNIX 上の CICS インストール・ディレクトリーで、作業ディレクトリーを定義しないでください。

WSDL_VALIDATOR={TRUE|FALSE}

SOAP 要求と応答をそれぞれの定義およびスキーマに照らして検証する操作を可能にします。 Liberty JVM サーバーに関しては、このオプションは無視されます。詳しくは、SOAP メッセージの検証を参照してください。WSDL_VALIDATOR=FALSE を設定することで、このオプションをオフにすることができます。オプトアウトすることで、新しい JAR ファイルとの競合の可能性を排除し、ストレージの浪費や、始動速度の低下を回避できます。

ZCEE_INSTALL_DIR={<*installation_directory*>}

z/OS Connect Enterprise Edition 機能をインストールする場所を指定します。z/OS Connect Enterprise Edition V2.0 の場合、デフォルトは /usr/lpp/IBM/zosconnect/v2r0/runtime です。 z/OS Connect Enterprise Edition V3.0 の場合、デフォルトは /usr/lpp/IBM/zosconnect/v3r0/runtime です。

JVM コマンド行オプション**:**

JVM コマンド・ライン・オプション、およびその説明。

コマンド・ライン・オプションのリスト

注**:** このリストは完全なものではありません。IBM® JVM 専用の便利なオプションのリストです。-X を含んだオプションは、IBM JVM に固有のものです。

-agentlib

JVM でデバッグ・サポートを使用可能にするかどうかを指定します。

詳しくは、Java アプリケーションのデバッグを参照してください。Java Platform Debugger Architecture (JPDA) について詳しくは、Oracle Technology Network Java の Web サイトを参照してください。

-Xcompressedrefs

Java 1.7.1 はデフォルトで圧縮参照を設定します。この設定は、オブジェクト、クラス、スレッド、モニターに対するすべての参照を 64 ビット値ではなく 32 ビット値として保管するように仮想マシン (VM) に命令します。圧縮参照を使用すると、多くのアプリケーションのパフォーマンスが向上します。オブジェクトのサイズが小さくなるため、ガーベッジ・コレクションの発生頻度が減少し、メモリー・キャッシュの使用率が向上するからです。ただし、これにより、31 ビット・ストレージの初期割り振りの量が多くなります。

Java 1.7.1 より前は、圧縮参照の使用はオプションでした。JVM サーバーの `-Xcompressedrefs` の使用のバランスを取り、31 ビット・ストレージの大量の初期割り振りを相殺するために、JVM サーバーは `-XXnosuballoc32bitmem` オプションを自動的に設定します。このオプションの効果として、必要に応じた増分割り振りが優先され、大量の初期割り振りが回避されます。多くのアプリケーションにとって、この動作により、パフォーマンスとストレージ使用量のバランスが適切になります。多くの参照を使用して、使用可能な 31 ビット・ストレージを減らすアプリケーション (または 31 ビット・ストレージが制約された環境で動作するアプリケーション) にとっては、`-Xnocompressedrefs` を使用することが望ましい場合があります。31 ビット・ストレージが制約されている場合は、このオプションの使用を検討してください。

-Xnocompressedrefs

多くの参照を使用して、使用可能な 31 ビット・ストレージを減らすアプリケーション (または 31 ビット・ストレージが制約された環境で動作するアプリケーション) にとっては、`-Xnocompressedrefs` を使用することが望ましい場合があります。

-Xms

ヒープの初期サイズを指定します。ストレージ・サイズは 1024 バイトの倍数で指定します。文字 K は KB を、文字 M は MB を、および文字 G は GB をそれぞれ表します。例えば、ヒープの初期サイズとして 6,291,456 バイトを指定するには、以下のいずれかの方法で **-Xms** をコーディングします。

```
-Xms6144K
-Xms6M
```

size を KB 数または MB 数として指定します。詳しくは、IBM SDK の『JVM コマンド行オプション』を参照してください。

-Xmso

オペレーティング・システム・スレッドの初期スタック・サイズを設定します。

-Xmso JVM オプションとデフォルト値については、IBM SDK の『JVM コマンド行オプション』を参照してください。

-Xmx

ヒープの最大サイズを指定します。なお、この固定ストレージ量は、JVM 初期設定時に JVM によって割り振られます。

size を KB 数または MB 数として指定します。

-Xscmx

共用クラス・キャッシュのサイズを指定します。最小サイズは 4 KB です。最大サイズとデフォルト・サイズはプラットフォームによって異なります。

size を KB 数または MB 数として指定します。詳しくは、IBM SDK の『JVM コマンド行オプション』を参照してください。

-Xshareclasses

共用クラス・キャッシュでクラス・データ共用を使用可能にする場合に、このオプションを指定します。JVM は既存のキャッシュに接続するか、キャッシュが存在しない場合は作成します。**-Xshareclasses** オプションにサブオプションを追加すると、複数のキャッシュを持つことができ、正しいキャッシュを指定することができます。詳しくは、IBM SDK の『JVM 間でのクラス・データの共用』を参照してください。

-XX:[+|-]EnableCPUMonitor

JVM サーバーで動作している場合、これはデフォルトで **-XX:-EnableCPUMonitor** に設定されます。ただし、JMX CPU モニタリングの拡張機能を使用する場合は、 **-XX:+EnableCPUMonitor** に設定する必要があります。このオプションを使用可能にすると、CPU の使用量が増加します。

JVM システム・プロパティー**:**

JVM システム・プロパティーは、JVM とそのランタイム環境に固有の構成情報を提供します。 JVM システム・プロパティーを指定するには、それらを JVM プロファイルに追加します。実行時に、CICS はプロパティーを JVM プロファイルから読み取り、JVM に渡します。

プロパティーの接頭部

システム・プロパティーは -D 接頭部を使用して設定する必要があります。例えば、**com.ibm.cics** の場合の正しい構文は **-Dcom.ibm.cics** です。

com.ibm.cics は、このプロパティーが CICS 環境の IBM JVM に固有のものであることを示します。

com.ibm は、より広い範囲で使用される一般的な JVM プロパティーであることを示します。

java.ibm も、より広い範囲で使用される一般的な JVM プロパティーであることを示します。

一般的なプロパティーについては、 216 ページの『JVM プロファイルの検証およびプロパティー』を参照してください。

プロパティーのコーディング規則

プロパティーは一連のコーディング規則に従って指定する必要があります。規則について詳しくは、 217 ページの『プロファイルのコーディング規則』を参照してください。

さまざまな用途の **JVM** サーバーに対するプロパティーの使用可否

JVM サーバーには、OSGi、Liberty、および Classpath の 3 つのタイプがあります。Classpath JVM サーバーはさらに、Axis2 対応、Security Token Server (STS) 対応、Batch 対応、および Mobile 対応に分けられます。それぞれの特定の機能に適用されるオプションを、次の表に示します。この表には、JVM サーバーの特定の用途に対して各プロパティーがサポートされているかどうかも示されています。一部のプロパティーは読み取り専用であることに注意してください。「読み取り専用」プロパティーを変更すると、ランタイム環境で障害が起きる可能性があります。これらのプロパティーについて詳しくは、 237 ページの『読み取り専用プロパティー』を参照してください。

表 *31. JVM* サーバーの用途別オプション

システム・プロパティー	**OSGi**	**Liberty**	クラスパス
com.ibm.cics.json.enableAxis2Handlers	サポート対象外	サポート対象外	サポートの有無
com.ibm.cics.jvmserver.applid	サポートの有無	サポートの有無	サポートの有無
com.ibm.cics.jvmserver.cics.product.name	サポート対象外	サポートの有無	サポート対象外
com.ibm.cics.jvmserver.cics.product.version	サポート対象外	サポートの有無	サポート対象外
com.ibm.cics.jvmserver.cmci.user.agent.white.list	サポート対象外	サポートの有無	サポート対象外
com.ibm.cics.jvmserver.cmci.user.agent.white.list.monitor.interval	サポート対象外	サポートの有無	サポート対象外
com.ibm.cics.jvmserver.cmci.user.agent.white.list.reject.text	サポート対象外	サポートの有無	サポート対象外
com.ibm.cics.jvmserver.configroot	サポートの有無	サポートの有無	サポートの有無
com.ibm.cics.jvmserver.controller.timeout	サポートの有無	サポートの有無	サポート対象外
com.ibm.cics.jvmserver.local.ccsid	サポートの有無	サポートの有無	サポートの有無
com.ibm.cics.jvmserver.name	サポートの有無	サポートの有無	サポートの有無
com.ibm.cics.jvmserver.override.ccsid	サポートの有無	サポートの有無	サポートの有無
com.ibm.cics.jvmserver.supplied.ccsid	サポートの有無	サポートの有無	サポートの有無
com.ibm.cics.jvmserver.threadjoin.timeout	サポートの有無	サポートの有無	サポート対象外

表 31. JVM サーバーの用途別オプション (続き)

システム・プロパティー	**OSGi**	**Liberty**	クラスパス
com.ibm.cics.jvmserver.trace.filename	サポートの有無	サポートの有無	サポートの有無
com.ibm.cics.jvmserver.trace.format	サポートの有無	サポートの有無	サポートの有無
com.ibm.cics.jvmserver.trigger.timeout	サポートの有無	サポートの有無	サポート対象外
com.ibm.cics.jvmserver.unclassified.tranid	サポートの有無	サポートの有無	サポート対象外
com.ibm.cics.jvmserver.unclassified.userid	サポートの有無	サポートの有無	サポート対象外
com.ibm.cics.jvmserver.wlp.args	サポート対象外	サポートの有無	サポート対象外
com.ibm.cics.jvmserver.wlp.autoconfigure	サポート対象外	サポートの有無	サポート対象外
com.ibm.cics.jvmserver.wlp.bundlepart.timeout	サポート対象外	サポートの有無	サポート対象外
com.ibm.cics.jvmserver.wlp.defaultapp	サポート対象外	サポートの有無	サポート対象外
com.ibm.cics.jvmserver.wlp.install.dir	サポート対象外	サポートの有無	サポート対象外
com.ibm.cics.jvmserver.wlp.jdbc.driver.location	サポート対象外	サポートの有無	サポート対象外
com.ibm.cics.jvmserver.wlp.jta.integration	サポート対象外	サポートの有無	サポート対象外
com.ibm.cics.jvmserver.wlp.latebinding	サポート対象外	サポートの有無	サポート対象外
com.ibm.cics.jvmserver.wlp.optimize.static.resources	サポート対象外	サポートの有無	サポート対象外
com.ibm.cics.jvmserver.wlp.optimize.static.resources.extra	サポート対象外	サポートの有無	サポート対象外
com.ibm.cics.jvmserver.wlp.server.config.dir	サポート対象外	サポートの有無	サポート対象外
com.ibm.cics.jvmserver.wlp.server.host	サポート対象外	サポートの有無	サポート対象外
com.ibm.cics.jvmserver.wlp.server.http.port	サポート対象外	サポートの有無	サポート対象外
com.ibm.cics.jvmserver.wlp.server.https.port	サポート対象外	サポートの有無	サポート対象外
com.ibm.cics.jvmserver.wlp.server.name	サポート対象外	サポートの有無	サポート対象外
com.ibm.cics.jvmserver.wlp.server.output.dir	サポート対象外	サポートの有無	サポート対象外
com.ibm.cics.jvmserver.wlp.wab	サポート対象外	サポートの有無	サポート対象外

表 31. JVM サーバーの用途別オプション (続き)

システム・プロパティー	OSGi	Liberty	クラスパス
com.ibm.cics.jvmserver.wlp.xml.format	サポート対象外	サポートの有無	サポート対象外
com.ibm.cics.sts.config	サポート対象外	サポート対象外	サポート対象 (STS のみ)
com.ibm.ws.logging.console.log.level	サポート対象外	サポートの有無	サポート対象外
com.ibm.ws.zos.core.angelName	サポート対象外	サポートの有無	サポート対象外
com.ibm.ws.zos.core.angelRequired	サポート対象外	サポートの有無	サポート対象外
console.encoding	サポートの有無	サポートの有無	サポート対象
file.encoding	サポートの有無	サポートの有無	サポート対象
java.security.manager	サポートの有無	サポート対象外	サポートの有無
java.security.policy	サポートの有無	サポート対象外	サポートの有無
org.osgi.framework.system.packages.extra	サポートの有無	サポートの有無	サポート対象外
osgi.compatibility.bootdelegation	サポートの有無	サポートの有無	サポート対象外

読み取り専用プロパティー

com.ibm.cics.json.enableAxis2Handlers
JSON データを処理するとき、JVM には Axis2 ハンドラー・プログラムを実行する機能が必要であることを示します。このプロパティーは、`JAVA_PIPELINE=YES` が指定され、JSON パイプラインをサポートするように構成された JVM にのみ関係しています。このオプションは、CICS の z/OS Connect には関係していません。この機能が必要な場合にのみ有効にしてください。このオプションを有効にすると、Axis2 ハンドラー・プログラムは JSON ワークロード中に実行できるようになります。ただし、このオプションを有効にするとパフォーマンスが低下する可能性があり、マッピング・レベル 4.2 以降の WSBind ファイルの機能の一部は使用できなくなります。

com.ibm.cics.jvmserver.applid
CICS 領域アプリケーション ID (APPLID) を指定します。これは読み取り専用プロパティーです。アプリケーションでこのプロパティーを使用することはできますが、変更することはできません。

com.ibm.cics.jvmserver.cics.product.name
Liberty を実行する CICS 製品の名前を指定します。これは読み取り専用プロパティーです。アプリケーションでこのプロパティーを使用することはできますが、変更することはできません。

com.ibm.cics.jvmserver.cics.product.version
Liberty を実行する CICS 製品のバージョンを指定します。これは読み取り専用プロパティーです。アプリケーションでこのプロパティーを使用することはできますが、変更することはできません。

com.ibm.cics.jvmserver.configroot
JVM サーバーの JVM プロファイルなどの構成ファイルがある場所を指定します。これは読み取り専用プロパティーです。アプリケーションでこのプロパティーを使用することはできますが、変更することはできません。

com.ibm.cics.jvmserver.local.ccsid
JCICS API が使用される場合のファイル・エンコード用のコード・ページを指定します。これは読み取り専用プロパティーです。アプリケーションでこのプロパティーを使用することはできますが、変更することはできません。

com.ibm.cics.jvmserver.name
JVM サーバーの名前を指定します。これは読み取り専用プロパティーです。アプリケーションでこのプロパティーを使用することはできますが、変更することはできません。

com.ibm.cics.jvmserver.supplied.ccsid
ローカル領域のデフォルト CICS を指定します。これは読み取り専用プロパティーです。アプリケーションでこのプロパティーを使用することはできますが、変更することはできません。

com.ibm.cics.jvmserver.trace.filename
JVM サーバー・トレース・ファイルの名前を指定します。これは読み取り専用プロパティーです。アプリケーションでこのプロパティーを使用することはできますが、変更することはできません。

com.ibm.cics.jvmserver.wlp.install.dir
Liberty インストールのロケーションを指定します。これは読み取り専用プロパティーです。アプリケーションでこのプロパティーを使用することはできますが、変更することはできません。

com.ibm.cics.jvmserver.wlp.server.config.dir
Liberty 構成ディレクトリーの場所を指定します。これは読み取り専用プロパティーです。アプリケーションでこのプロパティーを使用することはできますが、変更することはできません。

com.ibm.cics.jvmserver.wlp.server.output.dir
Liberty ログがある Liberty 出力ディレクトリーの場所を指定します。これは読み取り専用プロパティーです。アプリケーションでこのプロパティーを使用することはできますが、変更することはできません。

変更可能なプロパティー

com.ibm.cics.jvmserver.cmci.user.agent.white.list={*file_path*}

注: このプロパティーは、CMCI JVM サーバーのみを対象としています。

クライアント・ホワイトリスト・ファイルの場所を指定し、CMCI JVM サーバーでのホワイトリスト処理を使用可能にします。

com.ibm.cics.jvmserver.cmci.user.agent.white.list.monitor.interval={*time*|10s}

注: このプロパティーは、CMCI JVM サーバーのみを対象としています。

CMCI JVM サーバーが実行する Liberty キャッシュ・ファイルのモニター検査の間隔を指定します。これにより、クライアント・ホワイトリスト・ファイルから取得したユーザー・エージェントのホワイトリスト値のキャッシュがリフレッシュされます。

com.ibm.cics.jvmserver.cmci.user.agent.white.list.reject.text={*text*}

注: このプロパティーは、CMCI JVM サーバーのみを対象としています。

使用されているシステム管理クライアントがクライアント・ホワイトリストに含まれていないために CMCI への接続要求が拒否されるとき、ユーザーに返すカスタム応答メッセージを指定します。

com.ibm.cics.jvmserver.controller.timeout={*time*|90000ms}

注: このプロパティーは、IBM サービス担当者の指示があった場合にのみ使用してください。このプロパティーはいつでも変更される可能性があります。

com.ibm.cics.jvmserver.override.ccsid=

注: このプロパティーは上級ユーザー向けです。
これは JCICS API が使用されるときのファイル・エンコード用のコード・ページをオーバーライドします。デフォルトでは、JCICS は **LOCALCCSID** システム初期設定パラメーターの値をファイルのエンコード方式として使用します。この値をオーバーライドすることを選択した場合には、このプロパティーでコード・ページを設定します。 EBCDIC コード・ページを使用してください。新しいコード・ページにアプリケーションが整合していることを確認する必要があります。そうでない場合、エラーが発生する可能性があります。有効な CCSID の詳細については、LOCALCCSID システム初期設定パラメーターを参照してください。

com.ibm.cics.jvmserver.threadjoin.timeout={*time*|30000ms}
スレッドを待機している要求がサービスのキューに入れられているときのタイムアウト値を制御します。

com.ibm.cics.jvmserver.trace.format={FULL|SHORT|ABBREV}
トレースの形式を制御します。目的に応じてトレース形式を変更できますが、診断情報を IBM サービス担当者に送るときには FULL に設定する必要があります。

com.ibm.cics.jvmserver.trigger.timeout={*time*|500ms}

注: このプロパティーは、IBM サービス担当者の指示があった場合にのみ使用してください。このプロパティーはいつでも変更される可能性があります。

com.ibm.cics.jvmserver.unclassified.tranid={transaction id}
JVM サーバーで実行された未分類の作業に使用するデフォルトのトランザクションを指定します。

- Liberty JVM サーバーでは、com.ibm.cics.jvmserver.unclassified.tranid プロパティーを指定した場合を除き、未分類の作業はトランザクション CJSU で実行されます。

- OSGI JVM サーバーでは、`com.ibm.cics.jvmserver.unclassified.tranid` プロパティーを指定した場合を除き、未分類の作業はトランザクション CJSA で実行されます。

ユーザーは、CJSA または CJSU トランザクションを複製して、指定したトランザクション ID を CICS に定義する必要があります。

com.ibm.cics.jvmserver.unclassified.userid={user id}
JVM サーバーで未分類の作業が CICS タスクとして実行されるデフォルト・ユーザー ID を、ユーザーが変更できるようにします。指定されていない場合は、CICS のデフォルト・ユーザー ID が使用されます。指定されるユーザー ID は、RACF® に定義されていて、作業の実行に必要な権限を持っている必要があります。

未分類の作業とは、Liberty の HTTP 分類コンポーネントで識別されない要求のことです。例えば、JMS、インバウンド JCA、EJB 要求などです。

com.ibm.cics.jvmserver.wlp.args=

注: このプロパティーは、IBM サービス担当者の指示があった場合にのみ使用してください。

com.ibm.cics.jvmserver.wlp.autoconfigure={false|true}
必要な Liberty ディレクトリー `server.xml` およびその他の構成ファイルがまだ存在しない場合に、CICS がそれらを作成するかどうかを指定します。

com.ibm.cics.jvmserver.wlp.bundlepart.timeout={time|30000ms}
バンドル部分のインストール中に CICS Liberty によって使用されるタイムアウト値を制御します。操作がタイムアウトになると、バンドル部分 (および関連付けによってバンドル) が無効状態になります。

Liberty がインストール・フェーズを確認すると、Liberty がアプリケーションを完全に開始するまで、バンドル部分は使用可能状態のままになります。タイムアウトは、この状態に達したバンドル部分には影響しません。

重要: これは Liberty 構成のモニター間隔より大きい値でなければなりません。そうでない場合、バンドル部分が誤ってタイムアウトになることがあります。

com.ibm.cics.jvmserver.wlp.defaultapp={false|true}

Liberty サーバーが適切にインストールされて始動したことを確認するために使用できる、デフォルトの CICS Web アプリケーションをインストールする `defaultApp-1.0` 機能を `server.xml` に追加するように、CICS に指示します。

ヒント: このプロパティーは、**com.ibm.cics.jvmserver.wlp.autoconfigure=true** の場合のみに使用されます。

com.ibm.cics.jvmserver.wlp.jdbc.driver.location=
DB2 JDBC ドライバーが含まれるディレクトリーの場所を指定します。この場所には、DB2 JDBC ドライバーの `classes` および `lib` ディレクトリーが含まれている必要があります。自動構成プロパティー **com.ibm.cics.jvmserver.wlp.autoconfigure=true** を設定した場合、JVM サーバーを有効にすると、`server.xml` 内の既存のサンプル構成がデフォルトの構成に置き換えられ、ユーザーによる更新内容は失われます。

com.ibm.cics.jvmserver.wlp.jta.integration={false|true}
Java Transaction API (JTA) を使用して CICS 統合を有効にします。JTA インターフェースによって作成されたトランザクションが有効な場合、CICS 作業単位は Java Transaction Manager に従属します。

ヒント: このプロパティーは、**com.ibm.cics.jvmserver.wlp.autoconfigure=true** の場合のみに使用されます。

com.ibm.cics.jvmserver.wlp.latebinding={NONHTTP|COMPATIBILITY}

注: このプロパティーは、IBM サービス担当者の指示があった場合にのみ使用してください。このプロパティーはいつでも変更される可能性があります。

com.ibm.cics.jvmserver.wlp.optimize.static.resources={false|true}

静的コンテンツに対する要求を非 CICS スレッドで処理できるようにします。静的なものとして認識されるファイルのタイプは、`.css` `.gif` `.ico` `.jpg` `.jpeg` `.js` および `.png` です。

com.ibm.cics.jvmserver.wlp.optimize.static.resources.extra=

最適化のための追加の静的リソースを含めたカスタム・リストを指定します。項目をコンマで区切ってピリオドで始める必要があります (例えば `.css`, `.gif`, `.ico`)。

ヒント: この値は、**com.ibm.cics.jvmserver.wlp.optimize.static.resources=true** である場合にのみ有効です。

com.ibm.cics.jvmserver.wlp.server.host={*|*hostname*|*IP_address*}
Web アプリケーションにアクセスするための HTTP 要求のホストの名前または IP アドレス IPv4 または IPv6 形式を指定します。Liberty JVM サーバーは、デフォルト値として `*` を使用します。この値は CICS で Web アプリケーションを実行する場合は適していないため、このプロパティーを使用して別の値を指定するか、`server.xml` ファイルを更新してください。

ヒント: このプロパティーは、**com.ibm.cics.jvmserver.wlp.autoconfigure=true** の場合のみに使用されます。

com.ibm.cics.jvmserver.wlp.server.http.port={9080|*port_number*}
Java Web アプリケーションに対する HTTP 要求を受け入れるためのポートを指定します。 CICS は、Liberty から提供されるデフォルト値を使用します。Liberty JVM サーバーは TCPIPSERVICE リソースを使用しません。z/OS システムでこのポート番号が空いていること、または共用されていることを確認してください。

ヒント: このプロパティーは、**com.ibm.cics.jvmserver.wlp.autoconfigure=true** の場合のみに使用されます。

com.ibm.cics.jvmserver.wlp.server.https.port={9443|*port_number*}
Java Web アプリケーションに対する HTTPS 要求を受け入れるためのポートを指定します。 CICS は、Liberty から提供されるデフォルト値を使用します。Liberty JVM サーバーは TCPIPSERVICE リソースを使用しないので、z/OS システムでポート番号が空いているか、共用されていることを確認してください。

ヒント: このプロパティーは、
com.ibm.cics.jvmserver.wlp.autoconfigure=true の場合のみに使用されます。

com.ibm.cics.jvmserver.wlp.server.name={defaultServer|*server_name*}
Liberty サーバーの名前を指定します。Liberty サーバーの構成ファイル、出力ファイル、およびディレクトリーの場所に影響するため、このプロパティーを指定してはいけません。

com.ibm.cics.jvmserver.wlp.wab={false|true}
Liberty 機能 `wab-1.0` を `server.xml` に挿入します。Java EE 8 を標準モードの Liberty で使用するには、このオプションを `com.ibm.cics.jvmserver.wlp.wab=false` に設定した上で、必要な Java EE 8 の機能を追加する必要があります。

com.ibm.cics.jvmserver.wlp.xml.format={false|true}
`server.xml` の読みやすさを改善するために、その中の空白を CICS がフォーマット設定できるようにします。

com.ibm.cics.sts.config=*path*
STS 構成ファイルの場所と名前を指定します。

com.ibm.ws.logging.console.log.level={INFO|AUDIT|WARNING|ERROR|OFF}
どんなメッセージが Liberty によって JVM サーバー STDOUT ファイルに書き込まれるかを制御します。このプロパティーの設定にかかわらず、Liberty コンソール・メッセージは Liberty `messages.log` ファイルにも書き込まれます。

com.ibm.ws.zos.core.angelName=*named_angel*
Liberty JVM サーバーが接続する名前付きのエンジェル・プロセスを指定します。 **com.ibm.ws.zos.core.angelName** を指定しない場合は、必要に応じて、Liberty JVM サーバーの始動にデフォルトのエンジェル・プロセスが使用されます。

com.ibm.ws.zos.core.angelRequired={false|true}
Liberty JVM サーバーの始動にエンジェル・プロセスが必要かどうかを示します。

console.encoding=
JVM サーバー出力ファイルのエンコード方式を指定します。

file.encoding=
JVM で文字を読み書きする場合のコード・ページを指定します。デフォルトでは、z/OS 上の JVM は EBCDIC コード・ページ IBM1047 (または cp1047) を使用します。

- OSGi 用に構成されているプロファイルには、JVM でサポートされているどのコード・ページでも指定できます。 JCICS は文字エンコードに CICS 領域のローカル CCSID を使用するので、CICS はどのコード・ページでも許容します。
- Liberty JVM サーバー用に構成されているプロファイルには、ISO-8859-1 がデフォルト値として提供されます。UTF-8 も使用できます。その他のコード・ページはサポートされていません。
- Axis2 用に構成されているプロファイルには、EBCDIC コード・ページを指定しなければなりません。

java.security.manager={default| "" | |*other_security_manager*}

JVM に使用可能にする Java セキュリティー・マネージャーを指定します。デフォルトの Java セキュリティー・マネージャーを使用可能にするには、次のいずれかの形式でこのシステム・プロパティーを組み込みます。

- `java.security.manager=default`
- `java.security.manager=""`
- `java.security.manager=`

これらのステートメントはすべて、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ます。 **java.security.manager** システム・プロパティーを JVM プロファイルに組み込まない場合、JVM は Java セキュリティーが無効な状態で実行されます。

java.security.policy=

JVM のセキュリティー・ポリシーを判別するためにセキュリティー・マネージャーで使われる、追加のポリシー・ファイルの場所を記述します。デフォルトのポリシー・ファイルは、`/usr/lpp/java/J8.0_64/lib/security/java.policy` で JVM に提供されます。ここで、`java/J8.0_64` サブディレクトリー名は、IBM 64-bit SDK for z/OS, Java テクノロジー・エディション をインストールする際のデフォルト値です。デフォルトのセキュリティー・マネージャーは常にこのデフォルト・ポリシー・ファイルを使って JVM のセキュリティー・ポリシーを判別します。 **java.security.policy** システム・プロパティーを使用すると、デフォルト・ポリシー・ファイルに加えて、セキュリティー・マネージャーで考慮される任意のポリシー・ファイルを指定することができます。

Java セキュリティーがアクティブ状態のときに CICS Java アプリケーションを正常に実行できるようにするには、アプリケーション実行に必要な権限を CICS に与える追加のポリシー・ファイルを少なくとも 1 つ指定してください。

Java セキュリティーの有効化については、Java セキュリティー・マネージャーの有効化を参照してください。

org.osgi.framework.system.packages.extra=

このオプションは、Liberty を含む OSGi 対応 JVM サーバーに固有のもので、JRE とカスタム Java パッケージの拡張機能を OSGi フレームワークを通して公開して、以後のバンドル・インポート解決に利用できるようにします。JVM ベンダーは各種拡張機能を JRE に提供できます。IBM JVM サーバーでは、このオプションは、CICS が IBM JRE から公開するように選択したパッケージのセットを含むように補強されています。必要に応じて、このプロパティーを設定して追加のパッケージを定義できます。詳しくは、OSGi Alliance Specificationsを参照してください。

osgi.compatibility.bootdelegation={false|true}

このオプションは、OSGi の Equinox 実装に固有のものです。これは、Liberty を含む OSGi 対応 JVM サーバーに適用されます。*true* に設定した場合、OSGi フレームワークは、通常の OSGi バンドル依存関係解決メカニズムで見つからなかったパッケージに対して、最終手段である bootdelegation 戦略を使用します。このオプションを使用すると、開発時に明示的な依存関係を見落としていても OSGi ランタイムが許容してくれます。最終手段のアルゴリズムでは、直接

的な解決方法 (パッケージを Import-Package バンドル・ヘッダーに明示的にリストする方法) と比較すると若干のオーバーヘッドが発生します。

OSGi への厳密な準拠、ポータビリティーの向上、最適なパフォーマンスのためには、このオプションを *false* に設定し、OSGi バンドルで使用するすべてのパッケージをバンドル `MANFEST.MF` で明示的に宣言してください。

JVM サーバーでの時間帯の設定**:**

TZ 環境変数はシステムの「ローカル」時間を指定します。この変数を JVM または Node.js プロファイルに追加することで、JVM サーバーまたは Node.js アプリケーションのローカル時間を設定できます。TZ 変数を設定しないと、システムによってデフォルトで UTC に設定されます。

JVM サーバーまたは Node.js アプリケーションの時間帯を設定する際には、以下の点に注意する必要があります。

- JVM または Node.js プロファイルの TZ 変数が GMT を基準にしたローカル MVS システムのオフセットと一致するようにします。ローカル MVS システム・オフセットを表示および設定する方法について詳しくは、「z/OS Communications Server: IP 構成解説書」の『TIMEZONE ステートメント』および『z/OS MVS シスプレックスのセットアップ』の『地方時の調整』を参照してください。
- 時間帯のカスタマイズはサポートされていないため、カスタマイズすると UTC にフェイルオーバーするか、または時間帯が混在した状態で JVMTRACE ファイル (JVM サーバーの場合) または TRACE ファイル (Node.js アプリケーションの場合) に出力されます。
- LOCALTIME を時間帯ストリングとして表示すると、構成に不整合が生じます。この不整合は、ローカル MVS 時間と設定する TZ の間、またはローカル MVS 時間と JVM または Node.js プロファイルのデフォルト設定の間に生じる可能性があります。各項目は正しくても、時間帯が混在した状態で出力されます。

POSIX タイム・ゾーン形式の使用

POSIX タイム・ゾーン形式には、省略形と完全形があります。どちらも TZ 環境変数の設定に使用できますが、省略形を使用すると、入力ミスの可能性を減らせます。

完全形のサンプル:

```
TZ=GMT0BST,M3.5.0,M10.5.0
```

省略形のサンプル:

```
TZ=GMT0BST
```

システムが実行されているタイム・ゾーンを調べるには、USS にログオンし、`echo $TZ` と入力します。結果は、TZ 環境変数が設定されている値の完全形になります。

```
/u/user:>echo $TZ
GMT0BST,M3.5.0,M10.4.0
```

POSIX タイム・ゾーン形式の詳細については、IBM developerWorks サイトの POSIX および Olson のタイム・ゾーンの形式を参照してください。

第 5 章 Java アプリケーションの管理

Java アプリケーションを使用可能にした後、CICS 領域をモニターして、アプリケーションの動作を理解することができます。アプリケーションのパフォーマンスを最適化するために環境を調整することができます。

このタスクについて

統計とモニターを使用して、CICS 領域における Java アプリケーションの動作に関する情報を収集することができます。特に、JVM の動作を確認することができます。情報を収集した後、JVM または Language Environment エンクレーブに変更を加えて、パフォーマンスを改善することができます。

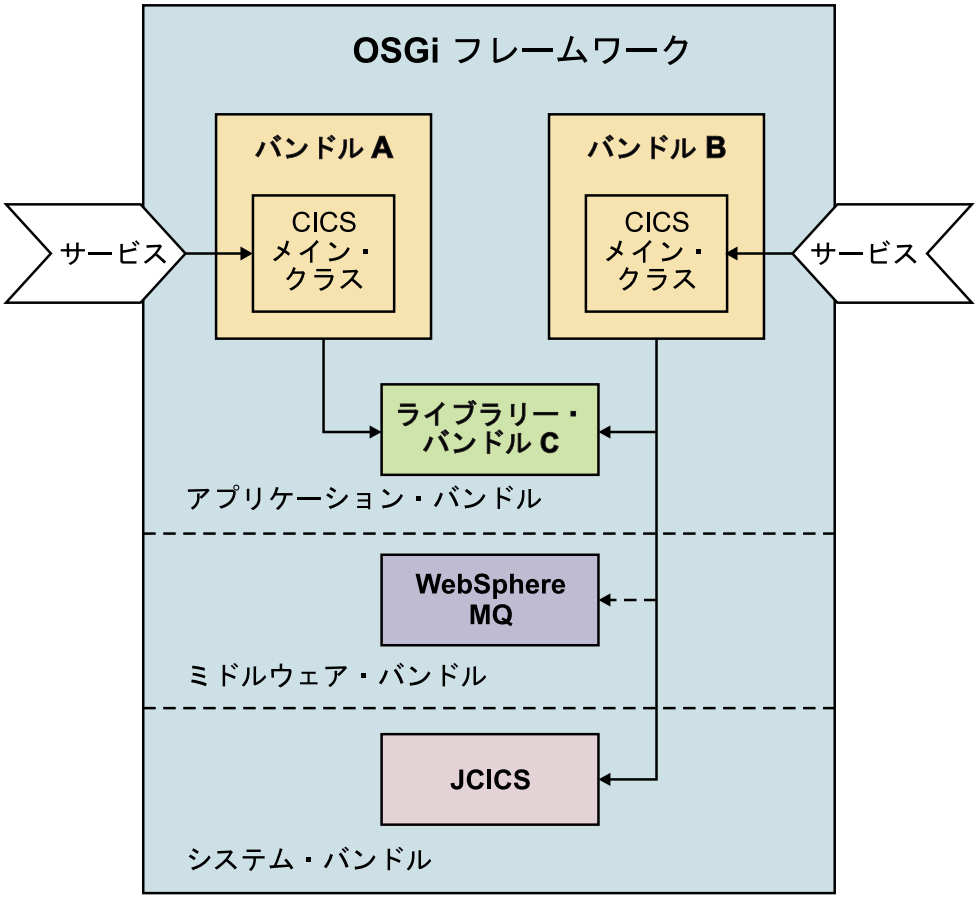
JVM サーバーにおける OSGi バンドルの更新

OSGi フレームワーク内の OSGi バンドルを更新するプロセスは、バンドルのタイプおよびその依存関係によって異なります。JVM サーバーを再始動することなく、アプリケーションの OSGi バンドルを更新できます。ただし、ミドルウェア・バンドルを更新するには、JVM サーバーの再始動が必要です。

このタスクについて

標準的な JVM サーバーでは、次の図に示されているように、OSGi フレームワークに OSGi バンドルの混合が入っています。

© Copyright IBM Corp. 1999, 2018



バンドル A とバンドル B は別々の Java アプリケーションであり、別々の CICS バンドルの OSGi バンドルとしてパッケージされています。両方のアプリケーションは、バンドル C でパッケージされている共通ライブラリーに依存しています。バンドル C は別個に管理され、更新されます。さらに、バンドル B は、IBM MQ ミドルウェア・バンドルと JCICS システム・バンドルに依存しています。

バンドル A と B の両方は、フレームワーク内の他のバンドルに影響を与えることなく、独立して更新できます。ただし、バンドル C を更新すると、それに依存する両方のバンドルに影響を与える可能性があります。バンドル C のエクスポートされたパッケージはいずれも OSGI フレームワークのメモリー内に残ります。このため、バンドル C での変更をピックアップするには、バンドル A と B もこのフレームワークで更新する必要があります。

ミドルウェア・バンドルは、フレームワーク・サービスを含み、JVM サーバーのライフサイクルで管理されます。例えば、ネイティブ・コードをフレームワークに 1 回ロードさせたり、IBM MQ などの別の製品にアクセスするためにドライバーを追加したりすることができます。

CICS には、OSGi フレームワークとの対話を管理するためにシステム・バンドルが用意されています。これらのバンドルは、製品の一部として IBM から提供されます。システム・バンドルの例として、CICS サービスにアクセスするための JCICS API のほとんどを提供する `com.ibm.cics.server.jar` ファイルがあります。

OSGi JVM サーバーの OSGi バンドルの更新

Java 開発者から、更新されたバージョンの OSGi バンドルが提供された場合、このバンドルを参照している CICS バンドルを完全に置き換えるか、または新しいバージョンの OSGi バンドルを段階的に導入 (フェーズイン) することができます。

このタスクについて

使用する更新方式は、以下の要因によって決まります。

- 更新時のサービス停止が許容されるかどうか。
- 更新時の CICS リソース変更が許容されるかどうか。

CICS バンドル PHASEIN を使用して、CICS リソースを変更せずに OSGi バンドルを動的に更新する

更新時のサービスの停止および CICS リソースの変更が許容されない場合は、この更新方法を使用して新しいバージョンの OSGi バンドルを段階的に導入します。

始める前に

新しいバージョンの OSGi バンドルの JAR ファイルは、古いバージョンの OSGi バンドルと同じ zFS ディレクトリー、つまり関連付けられている `osgibundle bundlepart` ファイルと同じディレクトリーに置く必要があります。デフォルトでは、このディレクトリーは BUNDLE リソース定義で指定されているディレクトリーです。この新しい OSGi バンドルのバージョンは、OSGi フレームワークに現在インストールされているものより上位のバージョンでなければならず、かつ OSGi バンドル参照が CICS バンドル・プロジェクトに追加されたときに定義されたバージョン範囲内になければなりません。

手順

OSGi JVM サーバーに新しいバージョンの OSGi バンドルを段階的に導入するには、以下の手順を使用します。

1. CICS Explorer の「バンドル」ビューで、OSGi バンドルを含む CICS バンドルを右クリックして「フェーズイン」をクリックし、次に「**OK**」をクリックします。新しいバージョンの OSGi バンドルが段階的に導入 (フェーズイン) され、新しいバージョンの OSGi バンドルによって実装されているすべてのサービスの新しいバージョンが OSGi フレームワークにインストールされ、古いバージョンのサービスが OSGi フレームワークから削除されます。
2. CICS Explorer の「OSGi サービス」ビューで、新しいバージョンの OSGi バンドルのすべての OSGi サービスがアクティブ状態であることを確認します。
3. CICS Explorer の「OSGi バンドル」ビューで、新しいバージョンの OSGi バンドルがリストされていてアクティブ状態であることを確認します。

タスクの結果

すべての新しいサービス要求に、新しいバージョンの OSGi バンドルが使用されます。既存の要求には引き続き古いバージョンが使用されます。 OSGi バンドルのシンボル・バージョンが増え、Java コードが更新されたことを示します。

次のタスク

新しいバージョンの OSGi サービスが正常に動作していることが確認されたら、作業は完了です。必要に応じて、古いバージョンの OSGi JAR ファイルを zFS から削除できますが、これは必須ではありません。新しいバージョンで問題が起きた場合に古いバージョンに戻せるように、古いバージョンの OSGi JAR ファイルを残しておくのも良いでしょう。

新しいバージョンの OSGi サービスに満足できず、古いバージョンに戻したい場合は、以下の手順を使用します。

1. 新しいバージョンの OSGi バンドル JAR を zFS から削除します。
2. CICS Explorer の「バンドル」ビューで、OSGi バンドルを含む CICS バンドルを右クリックして「フェーズイン」をクリックし、次に「**OK**」をクリックします。古いバージョンの OSGi バンドルが現時点で zFS にある最上位のバージョンなので、そのバージョンが OSGi フレームワークに再インストールされて、問題のある新しいバージョンは削除されます。
3. CICS Explorer の「OSGi サービス」ビューで、古いバージョンの OSGi バンドルの OSGi サービスだけがリストされていて、それらがみなアクティブ状態であることを確認します。
4. CICS Explorer の「OSGi バンドル」ビューで、古いバージョンの OSGi バンドルだけがリストされていてアクティブ状態であることを確認します。

CICS のコールド・リスタート、ウォーム・リスタート、または緊急リスタートがある場合には、新しいバージョンの OSGi バンドルが自動的に復元されます。この動作が行われるようにするために、CICS リソース定義を変更する必要はありません。

CICS リソースの変更を伴う OSGi バンドルの段階的な導入 (フェーズイン)

更新処理時のサービスの停止が許容されない場合には、この更新方法を使用して新しいバージョンの OSGi バンドルを段階的に導入します。更新中に新しい CICS リソースが作成されます。

始める前に

新しいバージョンの OSGi バンドルを含む CICS バンドルは既に定義済みで、zFS にエクスポートされています。この新しい OSGi バンドルのバージョンは、OSGi フレームワークに現在インストールされているバージョンより上位のバージョンとして OSGi バンドル・マニフェストに指定されている必要があります。同時に両方の OSGi バンドルをフレームワーク内で実行することが可能です。

手順

OSGi JVM サーバーに新しいバージョンの OSGi バンドルを段階的に導入するには、以下の手順を使用します。

1. CICS Explorer の「バンドル定義」ビューで、任意の場所を右クリックしてから「新規」をクリックし、 zFS 上に新しい CICS バンドルを取得するためのバンドル・プロジェクトを作成します。
2. CICS Explorer の「バンドル定義」ビューで、前の手順で作成した BUNDLE リソースを右クリックして「インストール」をクリックします。インストールの

ターゲットを選択してから「**OK**」をクリックして、 OSGi バンドルと CICS バンドルの OSGi サービスを OSGi フレームワークにインストールします。

3. CICS Explorer の「OSGi バンドル」ビューで、OSGi バンドルの状況を確認します。2 つのバージョンの OSGi バンドルがアクティブな状態でリストされます。
4. CICS Explorer の「OSGi サービス」ビューで、両方のバージョンの OSGi バンドルによって実装されている OSGi サービスがみなアクティブ状態であることを確認します。新しいサービス呼び出しには、最上位のセマンティック・バージョンを持つ OSGi バンドルを参照する OSGi サービスが使用されます。

タスクの結果

OSGi フレームワークでは、更新された OSGi バンドルは、古いバージョンの OSGi バンドルと共に使用可能です。

すべての新しいサービス要求に、新しいバージョンの OSGi バンドルが使用されます。既存の要求には引き続き古いバージョンが使用されます。

次のタスク

新しいバージョンの OSGi サービスが正常に動作していることが確認されたら、以下の手順を使用して、古いバージョンを OSGi フレームワークから削除します。

1. 旧バージョンの OSGi バンドルを指し示す BUNDLE リソースを使用不可にします。CICS Explorer の「バンドル」ビューで、古い BUNDLE を右クリックして「使用不可」をクリックし、次に「**OK**」をクリックします。その OSGi バンドルによって実装されているすべてのサービスの古いバージョンが OSGi フレームワークから削除され、新しいバージョンのサービスのみが「OSGi サービス」ビューにリストされるようになります。「OSGi バンドル」ビューで、古い OSGi バンドルの状態がすべて解決済みになります。
2. 古いバージョンの OSGi バンドルを指し示す BUNDLE リソースを破棄します。CICS Explorer の「バンドル」ビューで、古い BUNDLE を右クリックして「破棄」をクリックし、次に「**OK**」をクリックします。「OSGi バンドル」ビューでは、古い OSGi バンドルが OSGi フレームワークから削除され、新しいバージョンの OSGi バンドルのみがリストされます。

新しいバージョンの OSGi バンドルがインストールされたことを確認するため、CICS 領域のコールド・スタートがある場合は、必ず、古いバージョンの BUNDLE 定義を含む CSD グループを参照している CICS グループ・リスト (GRPLIST システム初期化パラメーター) を、手順 1 で作成した新しい BUNDLE 定義を含む CSD グループを参照するように更新してください。

古いバージョンの OSGi バンドルを復元する場合は、上記の 2 つの手順を使用して、新しいバージョンの OSGi バンドルを指し示す BUNDLE リソースを使用不可にして破棄します。古いバージョンのサービスが CICS Explorer の「OSGi サービス」ビューにリストされ、このサービスが新しいサービス呼び出しに使用されます。

OSGi JVM サーバーの OSGi バンドルの置き換え

更新処理時のサービスの停止が許容される場合には、この更新方法を使用します。新しい CICS リソースは作成されませんが、既存の BUNDLE リソース定義を更新することが必要になる場合があります。

始める前に

CICS バンドルを完全に置き換えるには、新しいバージョンの OSGi バンドルを含む、更新された CICS バンドルが zFS に存在しなければなりません。

手順

OSGi JVM サーバーにある既存の OSGi バンドルを新しいバージョンの OSGi バンドルに置き換えるには、以下の手順を使用します。

1. CICS Explorer の「バンドル」ビューで、更新する CICS バンドルの BUNDLE リソースを使用不可にして破棄します。その CICS バンドルの一部である OSGi サービスは、OSGi フレームワークから除去され、CICS Explorer の「OSGi サービス」ビューにリストされなくなります。

   注**:** OSGi バンドルによって実装されているサービスはこの時点から手順 3 が完了するまで OSGi フレームワークで使用できなくなるため、このサービスのすべてのユーザーがサービス停止の影響を被ります。
2. オプション: 更新された CICS バンドルを zFS 内の別のディレクトリーにデプロイした場合は、BUNDLE リソース定義を編集します。
3. CICS Explorer の「バンドル」ビューで、この BUNDLE リソース定義をインストールして、変更された OSGi バンドルを取得します。 CICS バンドル内の OSGi バンドルとサービスが、OSGi フレームワークにインストールされます。
4. CICS Explorer の「OSGi バンドル」ビューで、OSGi バンドルの状況を確認します。新しいバージョンの OSGi バンドルがアクティブな状態でリストされます。
5. CICS Explorer の「OSGi サービス」ビューで、新しいバージョンの OSGi バンドルによって実装されているすべての新しいバージョンの OSGi サービスがみなアクティブ状態であることを確認します。

タスクの結果

すべての新しいサービス要求に、新しいバージョンの OSGi バンドルが使用されます。既存の要求には引き続き古いバージョンが使用されます。 OSGi バンドルのシンボル・バージョンが増え、Java コードが更新されたことを示します。

共通ライブラリーを含むバンドルの更新

他の OSGi バンドルで使用するための共通ライブラリーを含む OSGi バンドルは、特定の順序で更新される必要があります。

始める前に

新しいバージョンの OSGi バンドルを含む、更新された CICS バンドルが、zFS に存在しなければなりません。共通のライブラリーを別の CICS バンドルで管理する場合は、それらのライブラリーのライフサイクルを、それらに依存しているアプリ

ケーションとは別個に管理することができます。

このタスクについて

通常、OSGi バンドルには、別の OSGi バンドルへの依存関係についてサポートされるバージョンの範囲を指定します。範囲を使用することで、より柔軟にフレームワーク内で共存可能な変更を行うことができます。共通ライブラリーを含むバンドルを更新する場合、OSGi バンドルのバージョン番号が増えます。ただし、実行中のアプリケーションは、依存関係に対応するバージョンのバンドルを既に使用しています。最新バージョンのライブラリーを取得するには、アプリケーションの OSGi バンドルをリフレッシュする必要があります。そのため、特定のアプリケーションは別のバージョンのライブラリーを使用するように更新して、その他のアプリケーションは古いバージョンを実行する状態で残すことが可能です。

共通ライブラリーを含む OSGi バンドルを更新する場合、その CICS の BUNDLE リソースを完全に置き換えることができます。しかし、クラスがライブラリーにロードされていないと、従属するバンドルがエラーを受け取る可能性があります。そのため、代わりの方法として、新しいバージョンのライブラリーをインストールして、元のバージョンのライブラリーと一緒にフレームワーク内で実行することができます。OSGi バンドルに異なるバージョン番号がある場合、OSGi フレームワークは両方のバンドルを並行して実行できます。

手順

OSGi JVM サーバーで既存の OSGi バンドルを置き換えるには、以下のようにします。

1. 新しいバージョンの CICS バンドル (共通ライブラリーを定義する OSGi バンドルを含む) を指し示す CICS BUNDLE リソースを定義してインストールします。CICS は、新しいバージョンの OSGi バンドルを OSGi フレームワークに定義します。既存の OSGi バンドルは、引き続き前のバージョンのライブラリーを使用します。
2. CICS Explorer の「OSGi バンドル」ビューで (操作 > **Java** > **OSGi** バンドル)、OSGi バンドルの状況を確認します。リストには、フレームワーク内で実行中の、シンボル名が同じでバージョンが異なる 2 つの OSGi バンドルの項目が表示されます。
3. 従属 Java アプリケーションの新規バージョンのライブラリーを取得するには、以下のいずれかの手順を使用します。
   - Java アプリケーションの CICS バンドルを置き換えます。
     a. Java アプリケーションの CICS BUNDLE リソースを使用不可にして破棄します。
     b. Java アプリケーションの CICS BUNDLE リソースを再インストールします。
   - 新しいバージョンの Java アプリケーションを段階的に導入 (フェーズイン) します。
     a. Java 開発者に、OSGi バンドルのバージョン情報を更新するよう依頼します。新しいバージョンの OSGi バンドルは、OSGi バンドル・マニフェストで指定された上位バージョンでなければならず、OSGi バンドル・プロジェクトが CICS バンドルに追加されたときに指定されたバージョ

ン範囲内である必要もあります。必要に応じて、新しいバージョンの OSGi バンドルの依存関係を、共通ライブラリーが定義されている新しいバージョンの OSGi バンドルを必要とするように変更することもできます。

b. 新しいバージョンの OSGi バンドルの JAR を CICS BUNDLE リソースのルート・ディレクトリーにコピーします。

c. CICS Explorer の「操作」 > 「バンドル」ビューで、OSGi バンドルを含む CICS バンドルを右クリックして「フェーズイン」をクリックし、次に「**OK**」をクリックして、新しいバージョンの OSGi バンドルを段階的に導入します。

OSGi バンドルがフレームワークにロードされると、バンドルは最新バージョンの共通ライブラリーを取得します。

4. CICS Explorer の「バンドル」ビューで (「操作」 > 「バンドル」)、CICS BUNDLE リソースの状況を確認します。

タスクの結果

共通ライブラリーを含む OSGi バンドルを更新し、最新バージョンのライブラリーを使用するように Java アプリケーションを更新しました。

OSGi ミドルウェア・バンドルの更新

OSGi フレームワークで実行中のミドルウェア・バンドルを更新するには、JVM サーバーを停止してから、再始動する必要があります。

このタスクについて

OSGi ミドルウェア・バンドルは、JVM サーバーの初期化中に OSGi フレームワークにインストールされます。例えば、パッチを適用したり、新しいバージョンを使用したりするために、ミドルウェア・バンドルを更新したい場合、変更されたバンドルを取得するために、JVM サーバーを停止してから、再始動する必要があります。

CICS Explorer を使用することによって、JVM サーバーのライフサイクルを管理したり、JVM プロファイルを編集したりすることができます。

手順

1. 新しいバージョンのミドルウェア・バンドルが、CICS が読み取りおよび実行アクセス権限を持つ、zFS 上のディレクトリーにあることを確認します。CICS には、ファイルへの読み取りアクセス権限も必要です。
2. zFS ディレクトリー名またはファイル名が JVM プロファイルに指定された値と異なる場合、JVM サーバーの JVM プロファイルの `OSGI_BUNDLES` オプションを編集します。

   a. CICS Explorer の「JVM サーバー」ビューを開き、zFS 内の JVM プロファイルの名前と場所を見つけます。 JVMSERVER リソースを参照するには、選択した領域または CICSplex に接続されている必要があります。

   b. 「z/OS UNIX ファイル」ビューを開き、JVM プロファイルがあるディレクトリーを参照します。

c. JVM プロファイルを編集して、OSGI_BUNDLES オプションを更新します。

3. JVMSERVER リソースを使用不可にして、JVM サーバーをシャットダウンします。 JVMSERVER を使用不可にすると、その JVM サーバーにインストールされている OSGi バンドルを含むすべての BUNDLE リソースも使用不可になります。
4. JVMSERVER リソースを使用可能にして、更新された JVM プロファイルを使用して JVM サーバーを始動します。 JVM サーバーが始動し、新しいバージョンのミドルウェア・バンドルを OSGi フレームワークにインストールします。また、CICS は、使用不可になった BUNDLE リソースを使用可能にし、更新されたフレームワークに OSGi バンドルとサービスをインストールします。

タスクの結果

OSGi フレームワークには、更新されたミドルウェア・バンドル、および JVM サーバーをシャットダウンする前にインストールされていた、Java アプリケーション用の OSGi バンドルとサービスが入っています。

JVM サーバーからの OSGi バンドルの除去

JVM サーバーから OSGi バンドルを除去したい場合は、CICS Explorer を使用して BUNDLE リソースを使用不可にして破棄します。

このタスクについて

BUNDLE リソースは、CICS バンドルで定義される OSGi バンドルと OSGi サービスの集合に対するライフサイクル管理を行います。OSGi フレームワークから OSGi バンドルを除去しても、インストールされている他の OSGi バンドルやサービスの状態に自動的に影響を与えることはありません。別のバンドルの前提条件であるバンドルを除去しても、そのバンドルを明示的にリフレッシュするまで従属バンドルの状態は一般的には変わりません。シングルトン・バンドルを使用する場合は例外です。他のバンドルが依存しているシングルトン・バンドルをアンインストールした場合、従属バンドルは、アンインストールされたバンドルのサービスを使用できません。報告される CICS BUNDLE リソースの状況は、OSGi バンドルの状況を正確には反映していない場合があります。

手順

1. 「**Operations**」 > 「**Java**」 > 「**OSGi Bundles**」をクリックして、OSGi バンドルが入っている BUNDLE リソースを検出します。
2. 「**Operations**」 > 「**Bundles**」をクリックして、BUNDLE リソースを使用不可にします。 CICS は、CICS バンドルで定義されている各リソースを使用不可にします。 OSGi バンドルとサービスについては、CICS は JVM サーバーの OSGi フレームワークに要求を送信して OSGi サービスの登録を解除し、OSGi バンドルを解決済み状態にします。すべての未完了トランザクションが完了しますが、CICS アプリケーションから OSGi サービスへの新しいリンクはすべて、エラーで戻ります。
3. BUNDLE リソースを破棄します。 CICS は OSGi フレームワークに要求を送信して、JVM サーバーから OSGi バンドルを除去します。

タスクの結果

OSGi バンドルとサービスを OSGi フレームワークから除去しました。

次のタスク

OSGi フレームワークに存在しなくなった OSGi サービスを指す PROGRAM リソースがある場合、それらの PROGRAM リソースを使用不可にし、破棄する必要がある可能性があります。

JVM サーバーへのアプリケーションの移動

プールされた JVM で Java アプリケーションを実行している場合、JVM サーバーで実行するようにそれらのアプリケーションを移動することができます。JVM サーバーは、同じ JVM 内で Java アプリケーションに対する複数の要求を処理できるので、同じワークロードの実行に必要な JVM 数を減らすことができます。

始める前に

アプリケーションがスレッド・セーフであり、1 つ以上の OSGi バンドルとしてパッケージされていることを確認してください。これらの OSGi バンドルは、1 つの CICS バンドルで zFS にデプロイされ、正しいターゲット JVMSERVER リソースを指定する必要があります。

Java 開発者は、CICS Explorer に組み込まれている CICS SDK for Java を使用することにより、OSGi を使用して Java アプリケーションを再パッケージすることができます。サード・パーティーの JAR を使用するアプリケーションのマイグレーション方法について詳しくは、Java 環境のアップグレードを参照してください。

このタスクについて

アプリケーション用に既存の JVM サーバーを使用するか、アプリケーション用に JVM サーバーを作成することができます。スレッド限度と使用量が既に高い JVM サーバーにアプリケーションを移動しないでください。その JVM サーバーでロッキングの競合が生じる可能性があるからです。

手順

1. JVM サーバーを作成または更新します。
   - JVM サーバーを作成する場合は、Liberty JVM サーバーの構成を参照してください。プールされた JVM の JVM プロファイルの設定の多くは、JVM サーバーに適用されません。プールされた JVM プロファイルから DFHOSGI プロファイルにコピーできるオプションは、`LIBPATH_SUFFIX` オプションのみです。
   - 既存の JVM サーバーを使用する場合は、JVMSERVER リソースの THREADLIMIT 属性を増やして追加アプリケーションを処理するか、JVM サーバー・プロファイルのオプションを更新する必要がある可能性があります。JVM プロファイルを変更する場合、変更を有効にするために JVM サーバーを再始動します。
2. zFS でデプロイされたバンドルを指す BUNDLE リソースを作成します。BUNDLE リソースをインストールすると、CICS は、JVM サーバーの OSGi

フレームワークに OSGi バンドルをロードします。OSGi フレームワークは、OSGi バンドルを解決し、OSGi サービスを登録します。 CICS Explorer を使用して、BUNDLE リソースが使用可能であることを確認してください。また、「OSGi Bundles」ビューおよび「OSGi Services」ビューを使用して、OSGi バンドルとサービスの状態を確認することもできます。

3. アプリケーションの PROGRAM リソースを更新します。
   a. EXECKEY 属性が CICS に設定されていることを確認します。 すべての JVM サーバーの作業は CICS キーで実行されます。
   b. JVM プロファイル名を除去し、JVMSERVER リソースの名前を入力します。
   c. JVMCLASS 属性が、Java アプリケーションの OSGi サービスと一致することを確認します。
   d. アプリケーションの PROGRAM リソースを再インストールします。

   PROGRAM リソースは、OSGi サービスを使用して、OSGi バンドルを JVM サーバー外部の他の CICS アプリケーションから使用可能にします。

タスクの結果

Java アプリケーションが呼び出されると、JVM サーバーで実行されます。

次のタスク

CICS Explorer の JVM サーバー・ビュー、および CICS 統計を使用して、JVM サーバーをモニターすることができます。パフォーマンスが最適でない場合は、スレッド限度を調整してください。

Liberty JVM サーバーでの Java EE アプリケーションの更新

Liberty JVM サーバーで Java EE アプリケーションを更新するには、CICS バンドルをリフレッシュする方法、drop-ins フォルダー内のアプリケーションを更新する方法、`<application>` エレメントを使用する方法の、3 つの方法があります。

このタスクについて

Liberty サーバーでの Java EE アプリケーションの更新プロセスは、アプリケーションのデプロイ方法によって決まります。

- アプリケーションは、CICS バンドルでデプロイされている

  このシナリオでは、アプリケーションは、CICS Explorer を使用してバンドルの一部として CICS バンドル・プロジェクトに追加され、その後、z/OS File System (zFS) にエクスポートされる必要があります。それから、エクスポートされたプロジェクトを参照する BUNDLE 定義を使用して CICS にインストールされます。
- アプリケーションは、Liberty の `drop-ins` フォルダーに直接デプロイされている

  このシナリオでは、Java アーカイブは、事前に定義されている `drop-ins` ディレクトリーに直接コピーされます。

- アプリケーションは、`<application>` エレメントで `server.xml` にデプロイされている

  このシナリオでは、アプリケーションの参照が、追加のアプリケーション属性および記述エレメントと共に `server.xml` に追加されます。

手順

CICS バンドル内にデプロイされたアプリケーション

- CICS バンドルをリフレッシュするには、Java EE アプリケーションを含むバンドルが、CICS 領域に既にインストールされており、使用可能になっている必要があります。詳しくは、CICS バンドル内の Java EE アプリケーションの Liberty JVM サーバーへのデプロイを参照してください。
  1. CICS Explorer の「バンドル」ビューで、更新する CICS バンドルの BUNDLE リソースを使用不可にします。

     注: その CICS バンドルの一部であるアプリケーションは Liberty サーバー・ランタイムから削除され、その時点から最後のステップが完了するまで使用できなくなります。このサービスのすべてのユーザーがサービス停止の影響を受けます。
  2. Java EE アプリケーションを含む CICS バンドルの新しいバージョンを、古いバージョンと同じ zFS ロケーションにエクスポートします。
  3. CICS Explorer の「バンドル」ビューで BUNDLE リソース定義を使用可能にして、Java EE アプリケーションを取得します。アプリケーションが Liberty サーバーに再インストールされます。
  4. CICS Explorer で CICS バンドルの状況を確認します。CICS バンドルがアクティブな状態でリストされます。

  新しいバージョンの Java EE アプリケーションがアクティブになると、すべての新しい要求にそのバージョンが使用されます。

アプリケーションは、Liberty の `drop-ins` フォルダーに直接デプロイされている

- Liberty サーバーの「`drop-ins`」ディレクトリーを使用するには、`server.xml` 構成を更新してこの機能を使用可能にする必要があります。詳しくは、Liberty JVM サーバーへの Java EE アプリケーションの直接デプロイを参照してください。
  1. Eclipse 環境から、新しいバージョンのアーカイブ (WAR、EAR、または EBA) をエクスポートします。

     注: CICS バンドルの一部であるアプリケーションは、Liberty サーバー・ランタイムから除去されます。このサービスのすべてのユーザーがサービス停止の影響を受けます。
  2. この新しいアーカイブを `drop-ins` ディレクトリーにコピーし、元のバージョンを置き換えます。

     Liberty サーバーはこのディレクトリーをスキャンし、前のバージョンをアンインストールして、新しいバージョンをインストールします。

新しいバージョンの Java EE アプリケーションがアクティブになると、すべての新しい要求にそのバージョンが使用されます。

アプリケーションは、`<application>` エレメントで `server.xml` にデプロイされている

- アプリケーションの動的な更新を可能にするには、`<applicationMonitor>` エレメントの `updateTrigger` 属性を `polled` に設定する必要があります。詳しくは、Controlling dynamic updatesを参照してください。
  1. Eclipse 環境から、新しいバージョンのアーカイブ (WAR、EAR、または EBA) をエクスポートします。

     注: CICS バンドルの一部であるアプリケーションは、Liberty サーバー・ランタイムから除去されます。このサービスのすべてのユーザーがサービス停止の影響を受けます。
  2. この新しいアーカイブを、`<application>` エレメントに指定されたロケーションにコピーします。

     Liberty サーバーは、変更がないかファイルをスキャンし、変更が検出されると、前のバージョンをアンインストールして新しいバージョンをインストールします。

  新しいバージョンの Java EE アプリケーションがアクティブになると、すべての新しい要求にそのバージョンが使用されます。

JVM サーバーのスレッド限度の管理

JVM サーバーでは、Java アプリケーションの実行に使用できるスレッド数が制限されています。また、各スレッドが 1 つの T8 TCB を使用するため、CICS 領域にもスレッド数の制限があります。 CICS 統計を使用してスレッド限度を調整すると、領域内の JVM サーバー数と、各 JVM サーバーで実行されるアプリケーションのパフォーマンスとのバランスを取ることができます。

このタスクについて

各 JVM サーバーには、Java アプリケーションを実行するために最大 256 個のスレッドがあります。 CICS 領域には最大 2000 個のスレッドを備えることができます。 CICS 領域で多数の JVM サーバー (例えば、8 つ以上) を実行している場合、どの JVM サーバーにも最大値を設定できるわけではありません。各 JVM サーバーのスレッド限度を調整して、CICS 領域内の JVM サーバー数と、Java アプリケーションのパフォーマンスとのバランスを取ることができます。

スレッド限度は JVMSERVER リソースで設定されるので、初期値を設定し、CICS 統計を使用して、Java ワークロードのテスト時にスレッド数を調整してください。

手順

1. JVMSERVER リソースを使用可能にし、Java アプリケーションのワークロードを実行します。
2. 適切な統計間隔を使用して JVMSERVER リソース統計を収集します。 CICS Explorer で「操作」 > 「**Java**」 > 「**JVM** サーバー」ビューを使用するか、DFH0STAT 統計プログラムを使用することができます。

3. タスクでスレッドを待機した回数と時間を確認します。 「JVMSERVER thread limit waits」フィールドと「JVMSERVER thread limit wait time」フィールドに、この情報が含まれています。
   - これらのフィールドの値が高く、多数のタスクが JVMTHRD 待機で中断する場合、JVM サーバーには使用可能な十分なスレッドがありません。スレッド数を増やすと、プロセッサーの使用量が増える可能性があるので、十分な MVS リソースが使用可能であることを確認してください。
   - これらのフィールドの値が低く、ピーク時のタスク数が使用可能な最大スレッド数より少ない場合、スレッド限度を減らして、他の JVM サーバー用にスレッドを解放することができます。
4. MVS リソースが使用可能かどうかを確認するために、ディスパッチャー TCB プール統計と TCB モード統計を使用して、CICS 領域全体での T8 TCB の使用量を判断します。 JVM サーバーの各スレッドは 1 つの T8 TCB を使用し、1 つの領域内で 2000 個に制限されます。 T8 TCB は複数の JVM サーバー間で共用できませんが、すべての TCB は 1 つの THRD TCB プール内にあります。待機している TCB 数とプロセッサー使用量が少ない場合、十分な MVS リソースが使用可能であることを示します。
5. JVM サーバーで実行できるスレッド数を調整するには、JVMSERVER リソースの THREADLIMIT 値を変更します。
6. Java アプリケーションのワークロードを再度実行し、統計を使用して、待機しているタスクの数が減ったことを確認します。

次のタスク

JVM サーバーのパフォーマンスを調整するには、JVM サーバーのパフォーマンスの改善を参照してください。

Recovery of resources in bundles

CICS バンドル内で定義されたリソースは、CSD には保管されません。 BUNDLE リソースは定義から作成することができ、それらのリソースはカタログに保管されます。 BUNDLE リソースは、プラットフォームとアプリケーションをインストールする際にも作成できます。これらのリソースには定義はなく、カタログには保管されず、アプリケーションまたはプラットフォームを介してリカバリーされます。

BUNDLE 定義のリカバリー

CICS のコールド・スタートを実行すると、BUNDLE リソース定義はカタログから削除され、バンドルが再作成されることはありません。

CICS のウォーム・リスタートまたは緊急時再始動を実行すると、初期設定後プロセス中に CICS はリスタート前にインストールされたすべての BUNDLE リソース定義の再作成を試行し、それらの定義を CICS 領域が停止した時と同じ使用可能化状態でインストールしようとします。

定義から作成されてカタログに格納されている BUNDLE リソースの場合、CICS は CRLR 提供のトランザクションを使用して、すべての必要な CICS リソースの動的再作成を含め、バンドル・マニフェストで定義されているすべてのリソースを解決するプログラムを開始します。動的に作成されるほとんどのリソースはカタロ

グ内に定義されていませんが、EVENTBINDING、EPADAPTER、および EPADAPTERSET の各リソースはカタログに保管されています。

CICS がリソースを動的に作成したり使用可能にしたりできない場合、BUNDLE リソースは使用不可の状態でインストールされ、警告メッセージが発行されます。複数のリソースのうちの 1 つだけがインストールできなかった場合でも、BUNDLE リソースは使用不可の状態でインストールされます。 CICS Explorer の「バンドル・パーツ」ビューで、BUNDLE リソース内のすべてのリソースの状態を確認できます。

アプリケーション・エントリー・ポイントが含まれるスタンドアロンの BUNDLE リソースの場合、 CICS 領域の再始動時にバンドルの可用性状況もリカバリーされます。

アプリケーションとプラットフォームによって生成された BUNDLE リソースのリカバリー

CICSplex にプラットフォームをインストールする場合や、プラットフォームにアプリケーションをインストールする場合、デプロイメントの一部であるすべての CICS バンドルが CICSPlex SM によって適切な CICS 領域に動的に作成されます。各 BUNDLE リソースは動的に作成され、固有の名前がそれぞれに与えられます。各 BUNDLE リソースには、プラットフォームの名前、アプリケーション、およびアプリケーションのバージョンが入った BASESCOPE 値も含まれます。 BUNDLE リソースは動的に作成されるため、CSD 内に同等の定義を持たず、カタログには格納されません。

プラットフォームまたはアプリケーションの各 CICS バンドルの内部に定義されているリソースは、動的に作成された BUNDLE リソースのインストール時に CICS 領域に動的に作成されます。これらのリソースも、CSD 内に同等の定義を持たず、カタログに格納されません。

アプリケーションまたはプラットフォームをインストールすると、CICSPlex SM はデータ・リポジトリー内にプラットフォームまたはアプリケーションのレコードを作成します。これは、そのプラットフォームまたはアプリケーションに関連付けられた CICS バンドルのリカバリー処理で使用されます。アプリケーションの CICS バンドルは、最初は使用不可の状態で CICS 領域内にインストールされます。アプリケーションを使用可能にすると、CICSPlex SM が CICS 領域内のバンドルを使用可能にします。アプリケーションを使用可能にすると、呼び出し元がそのアプリケーションを、そのアプリケーション・エントリー・ポイントから呼び出すことができるようになります。

プラットフォームの一部として定義されている CICS 領域を始動または再始動すると、CICSPlex SM は zFS 内のそのプラットフォームおよびアプリケーションのバンドルの情報を読み取り、適切な CICS バンドルの適切なバージョンを CICS 領域にインストールします。CICS バンドルは、CICS 領域のコールド・スタート、ウォーム・スタート、または緊急時再始動のいずれの場合も、同じ方法でインストールされます。バンドル内に生成されたリソースは、自動インストールの状況や、アプリケーション・バンドルの一部ではない CICS 領域の類似のリソースのカタログに関係なくインストールされます。

- プラットフォームに関連付けられている CICS バンドルは、インストールに問題がない限り、プラットフォームと同じ状態でインストールされます。プラットフォームが使用不可の場合、関連付けられている CICS バンドルも使用不可で、プラットフォームが使用可能な場合は、バンドルも使用可能です。
- アプリケーション・バージョンが CICS 領域の始動または再始動の前に使用可能ではなかった場合、その CICS バンドルは使用不可の状態でインストールされます。アプリケーション・バージョンが使用可能であった場合、インストールに問題がない限り、その CICS バンドルは使用可能な状態でインストールされます。
- アプリケーション・バージョンの可用性状況は CICS 領域の始動時に適用され、CICS 領域の再始動時に復元されます。

CICS バンドルと、それがインストールされている CICS 領域との間の関係は、アプリケーションまたはプラットフォームの管理パートに格納されます。管理パートは、アプリケーション・インストール・プロセスで自動的に作成される MGMTPART レコードです。アプリケーションの全体の状況情報は、そのアプリケーションの個々の管理パートの状況から導き出されます。プラットフォームの全体の状況情報は、領域タイプの状況と、プラットフォームの個々の管理パートの状況から導き出されます。アプリケーションまたはプラットフォームの状況を確認するには、CICS Explorer の「クラウド・エクスプローラー」ビューを使用します。

ユーザー・リソースのリカバリー

CICS の外で処理されるリソース・タイプ (例えばベンダー・リソース) がバンドル内に含まれる場合、初期設定後プログラミング中にコールバック・プログラムを登録してリソース・タイプを再作成するために、バンドルの登録プログラムが使用可能である必要があります。

登録プログラムによってコールバック・プログラムが登録されない場合や、コールバック・プログラムを使用してユーザー・リソースを再作成できない場合は、BUNDLE リソースが使用不可の状態でインストールされ、ユーザー・リソースが使用不能の状態でインストールされます。 BUNDLE リソースの破棄と再作成を実行する前に、登録プログラムとコールバック・プログラムの両方が、 CICS で使用可能である必要があります。

バンドル内のファイルのリカバリー

CICS バンドル内で定義されているファイルをインストールすると、そのファイルはカタログに追加されます。CICS は、ウォーム・リスタートまたは緊急リスタート時に、バンドルによってインストールされたファイルをカタログからリカバリーします。これらのリカバリーされたファイルは、CICS リカバリーでのみ使用されます。CICS によるリカバリーが完了すると、これらのすべてのファイルは削除されます。バンドルが再作成されるときに、CICS はバンドル・ディレクトリーから定義を取得して新しいファイルを作成します。カタログからリカバリーされたファイルに保持ロックがある場合、そのファイルはリカバリーの完了時に削除できません。その結果、ファイルは孤立し (どのバンドルにも属さない)、CICS のリスタート後に手動で削除する必要があります。

JVM stdout および stderr 出力をリダイレクトするための Java クラスの作成

JVM プロファイルの USEROUTPUTCLASS オプションを使用して、JVM からの stdout ストリームと stderr ストリームを代行受信する Java クラスを指定します。このクラスを更新すると、適当なタイム・スタンプとレコード・ヘッダーを指定し、出力をリダイレクトすることができます。

CICS は、この目的に使用できるサンプル Java クラス com.ibm.cics.samples.SJMergedStream および com.ibm.cics.samples.SJTaskStream を用意しています。/usr/lpp/cicsts/cicsts55/samples/com.ibm.cics.samples ディレクトリーに、これらの両方のクラスのサンプル・ソースが提供されています。/usr/lpp/cicsts/cicsts55 ディレクトリーは、z/OS UNIX で CICS ファイル用のインストール・ディレクトリーです。このディレクトリーは、DFHISTAR インストール・ジョブの **USSDIR** パラメーターで指定されます。また、これらのサンプル・クラスは、クラス・ファイル com.ibm.cics.samples.jar として出荷時に付属しています。このクラス・ファイルは、/usr/lpp/cicsts/cicsts55/lib ディレクトリーにあります。これらのクラスを変更するか、サンプルに基づいて独自のクラスを作成することができます。

JVM 出力、ログ、ダンプ、およびトレースの場所の制御には、以下に関する情報が記載されています。

- JVM からの出力のタイプで、USEROUTPUTCLASS オプションによって指定されるクラスによって代行受信されるものと、されないもの。使用するクラスは、代行受信する可能性があるすべてのタイプの出力を処理できなければなりません。
- 提供されたサンプル・クラスの動作。com.ibm.cics.samples.SJMergedStream クラスは、JVM 出力用とエラー・メッセージ用にマージされた 2 つのログ・ファイルを作成します。各レコードには、アプリケーション ID、日付、時刻、トランザクション ID、タスク番号、およびプログラム名を含むヘッダーが付けられます。これらのログ・ファイルは、一時データ・キューが使用可能な場合は、その一時データ・キューを使用して作成されます。一時データ・キューが使用不可であるか、Java アプリケーションで使用できない場合は、z/OS UNIX ファイルを使用して作成されます。com.ibm.cics.samples.SJTaskStream クラスは、単一のタスクからの出力を z/OS UNIX ファイルに送信し、タイム・スタンプとヘッダーを追加して、単一のタスクに固有の出力ストリームを提供します。

JVM サーバーが出力リダイレクト・クラスを使用するには、出力リダイレクト・クラスを含む OSGi バンドルを作成する必要があります。バンドル・アクティベーターがクラスのインスタンスをフレームワーク内のサービスとして登録し、プロパティー com.ibm.cics.server.outputredirectionplugin.name=*class_name* を確実に設定しておく必要があります。定数 com.ibm.cics.server.Constants.CICS_USER_OUTPUT_CLASSNAME_PROPERTY を使用して、プロパティー名を取得できます。次のコードの抜粋は、バンドル・アクティベーターでサービスを登録する方法を示しています。

```
Properties serviceProperties = new Properties();
    serviceProperties.put(Constants.CICS_USER_OUTPUT_CLASSNAME_PROPERTY, MyOwnStreamPlugin.class.getName());
    context.registerService(OutputRedirectionPlugin.class.getName(), new MyOwnStreamPlugin(), serviceProperties);
```

OSGi バンドルを JVM プロファイルの OSGI_BUNDLES オプションに追加するか、または最初のタスクの実行時にバンドルがフレームワークに確実にインストールされるようにすることができます。どちらの方法を使用しても、USEROUTPUTCLASS オプションでクラスを指定する必要があります。

独自のクラスを作成する場合は、以下を理解している必要があります。

- OutputRedirectionPlugin インターフェース
- 出力の予想される宛先
- 出力リダイレクト・エラーと内部エラーの処理

出力リダイレクト・インターフェース

CICS は、com.ibm.cics.server.jar に com.ibm.cics.server.OutputRedirectionPlugin と呼ばれるインターフェースを備えています。このインターフェースは、JVM からの stdout および stderr 出力を代行受信するクラスによって実装できます。提供のサンプルはこのインターフェースを実装します。

以下のサンプル・クラスが用意されています。

- このインターフェースを実装するスーパークラス com.ibm.cics.samples.SJStream。
- JVM プロファイルで指定されるクラスである、サブクラス com.ibm.cics.samples.SJMergedStream および com.ibm.cics.samples.SJTaskStream。

サンプル・クラスのように、ご使用のクラスがインターフェース OutputRedirectionPlugin を直接実装するか、このインターフェースを実装するクラスを拡張することを確認してください。スーパークラス com.ibm.cics.samples.SJStream から継承するか、同じインターフェースを持つクラス構造を実装することができます。どちらのメソッドを使用する場合でも、クラスは java.io.OutputStream を拡張する必要があります。

initRedirect() メソッドは、1 つ以上の出力リダイレクト・クラスで使用される 1 組のパラメーターを受け取ります。次のコードはインターフェースを示しています。

```
package com.ibm.cics.server;

import java.io.*;

public interface OutputRedirectionPlugin {

  public boolean initRedirect( String inDest,
                               PrintStream inPS,
                               String inApplid,
                               String inProgramName,
                               Integer inTaskNumber,
                               String inTransid
                              );
  }
```

スーパークラス com.ibm.cics.samples.SJStream には、com.ibm.cics.samples.SJMergedStream および com.ibm.cics.samples.SJTaskStream の共通コンポーネントが含まれています。これに含まれている initRedirect() メソ

ッドは「false」を戻します。これは、このメソッドがサブクラス内の別のメソッドによってオーバーライドされる場合を除いて、出力のリダイレクトを事実上使用不可にします。これは、writeRecord() メソッドを実装しません。このようなメソッドは、出力リダイレクト・プロセスを制御するために任意のサブクラスによって提供されなければなりません。独自のクラス構造でこのメソッドを使用することができます。出力リダイレクトの初期化も、initRedirect() メソッドではなく、コンストラクターを使用して実行できます。

inPS パラメーターには、JVM のオリジナルの System.out 印刷ストリームまたはオリジナルの System.err 印刷ストリームのどちらかが入っています。基礎となるこれらのロギング宛先のどちらにでもロギングを書き込むことができます。これらの印刷ストリームのどちらかで close() メソッドを呼び出してはなりません。印刷ストリームは完全にクローズされたままになり、将来使用できないからです。

出力の予想される宛先

CICS 提供のサンプル・クラスは、JVM からの出力を、CICS 領域に固有のディレクトリーに送信します。このディレクトリー名は、その CICS 領域に関連したアプリケーション ID を使用して作成されます。独自のクラスを作成する場合、必要に応じて、複数の CICS 領域から、同じ z/OS UNIX ディレクトリーまたはファイルに出力を送信することができます。

例えば、単一のファイルを作成して、複数の異なる CICS 領域で実行される特定アプリケーションに関連した出力をそのファイルに含めることができます。

Thread.start() を使用してプログラマチックに開始されるスレッドでは、CICS 要求を行うことはできません。これらのアプリケーションの場合、JVM からの出力は、USEROUTPUTCLASS に指定されたクラスによって代行受信されますが、CICS 機能 (一時データ・キューなど) を使用してリダイレクトすることはできません。提供されたサンプル・クラスの場合と同様に、これらのアプリケーションからの出力を z/OS UNIX ファイルに送信することができます。

出力リダイレクト・エラーと内部エラーの処理

ご使用のクラスで、CICS 機能を使用して出力をリダイレクトする場合、それらのクラスには、これらの機能を使用する際のエラーを処理するために適切な例外処理が含まれていなければなりません。

例えば、一時データ・キュー CSJO および CSJE に書き込もうとするときに、これらのキューに CICS 提供の定義を使用する場合、次の例外が TDQ.writeData によってスローされます。

- IOErrorException
- LengthErrorException
- NoSpaceException
- NotOpenException

ご使用のクラスが出力を z/OS UNIX ファイルに送信する場合、それらのクラスには、z/OS UNIX への書き込み時に発生するエラーを処理する適切な例外処理が含まれていなければなりません。これらのエラーで最も一般的な原因は、セキュリティー例外です。

クラスを USEROUTPUTCLASS オプションで指定する JVM で実行される Java プログラムには、クラスでスローされる可能性がある例外を処理する適切な例外処理が含まれていなければなりません。CICS 提供のサンプル・クラスは、例外を内部で処理します。そのために、Try/Catch ブロックを使用してすべてのスロー可能な例外をキャッチしてから、問題を報告する 1 つ以上のメッセージを書き込みます。出力メッセージのリダイレクト中にエラーが検出されると、これらのエラー・メッセージは `System.err` に書き込まれ、リダイレクトに使用可能にします。しかし、エラー・メッセージのリダイレクト中にエラーが検出される場合、この問題を報告するメッセージは、要求を処理する JVM で使用される JVM プロファイルの STDERR オプションによって指定されるファイルに書き込まれます。このようにサンプル・クラスはすべてのエラーをトラップするため、これは、呼び出し側プログラムが出力リダイレクト・クラスによってスローされる例外を処理する必要がないことを意味します。このメソッドを使用すると、呼び出し側プログラムへの変更を避けることができます。クラスで発行されるエラー・メッセージを、失敗した宛先にリダイレクトしようとして、出力リダイレクト・クラスをループに入れないように注意してください。

第 6 章 Java アプリケーションのセキュリティー

Java アプリケーションを保護して、許可されたユーザーだけがアプリケーションのデプロイやインストールを行ったり、これらのアプリケーションに Web からアクセスしたり CICS を経由してアクセスしたりできるようにすることが可能です。Java セキュリティー・マネージャーを使用して、Java アプリケーションが安全でない可能性のあるアクションを実行しないようにすることもできます。

Java アプリケーションのライフサイクル内のさまざまな時点で、以下のようなセキュリティーを追加できます。

- Java アプリケーション・リソースの定義やインストールに関するセキュリティー検査を実装します。Java アプリケーションは CICS バンドルにパッケージ化されているので、JVM サーバーにアプリケーションをインストールすることを許可されているユーザーが、このタイプのリソースをインストールできることを確認しなければなりません。
- 許可されたユーザーだけがアプリケーションにアクセスできるようにするための、アプリケーション・ユーザーに関するセキュリティー検査を実装します。
- CICSExecutorService を使って開始される CICS Java タスクに関するセキュリティー検査を実装します。このような CICS タスクはすべて、CJSA トランザクションおよびデフォルト・ユーザー ID の下で実行されます。
- Java セキュリティー・マネージャーを使用して、Java API に対するセキュリティー制限を実装します。

Java アプリケーションは、OSGi フレームワーク内か Liberty サーバー内で実行できます。Liberty は、Web アプリケーションをホストするように設計されており、OSGi フレームワークが組み込まれています。Liberty には独自のセキュリティー・モデルがあるので、Liberty サーバーのセキュリティー構成は異なります。

OSGi アプリケーションのセキュリティーを構成するには、CICS リソース・セキュリティーを使用して、JVMSERVER と Java アプリケーションのライフサイクルを管理できるユーザーを許可します。 CICS トランザクション・セキュリティーを使用して、アプリケーションにアクセスできるユーザーを判別します。

OSGi アプリケーションに関するセキュリティーの構成

CICS リソース・セキュリティーを使用して、JVMSERVER と Java アプリケーションのライフサイクルを管理できるユーザーを許可します。 CICS トランザクション・セキュリティーを使用して、アプリケーションにアクセスできるユーザーを判別します。

手順

- 必要に応じて、JVMSERVER リソースや BUNDLE リソースの作成、表示、更新、削除を行うための権限を、アプリケーション開発者やシステム管理者に与えます。JVMSERVER リソースは JVM サーバーの可用性を制御します。

© Copyright IBM Corp. 1999, 2018

BUNDLE リソースは、Java アプリケーションのデプロイメントの単位で、このアプリケーションの可用性を制御します。

- ユーザーにアプリケーションを実行する権限を与えます。これを行うには、アプリケーションの実行に使用されるトランザクションに、関連するユーザー ID で接続できるようにします。

タスクの結果

OSGi フレームワーク内で実行される Java アプリケーションに関するセキュリティーを正常に構成しました。

Liberty JVM サーバーに関するセキュリティーの構成

CICS Liberty セキュリティー機能を使用すると、Java Platform, Enterprise Edition ロール (Java EE ロール) を介して、ユーザーを認証したり、Web アプリケーションへのアクセスを許可したりできます。このようにして、CICS トランザクションおよびリソース・セキュリティーとの統合が可能になります。また、CICS リソース・セキュリティーを使用して、CICS BUNDLE リソースにデプロイされる JVMSERVER リソースと Java Web アプリケーションの両方のライフサイクルを管理する権限を適切なユーザーに与えることができます。このトピックでは、認証によって所定のユーザーの ID が検証されます。一般的には、ユーザーにユーザー名とパスワードの入力を要求することによって行われます。許可によって、認証済みユーザーの ID に基づいてアクセス制御権限が付与されます。

始める前に

1. CICS 領域で、SAF セキュリティーを使用するための構成が完了していることと、システム初期設定パラメーターとして SEC=YES が定義されていることを確認します。
2. アプリケーション開発者とシステム管理者に、Web アプリケーションを Liberty JVM サーバーにデプロイするために、JVMSERVER リソースと BUNDLE リソースの作成、表示、更新、および除去を行う権限を与えます。

JVMSERVER リソースは JVM サーバーの可用性を制御します。BUNDLE リソースは Java アプリケーションのデプロイメント単位であり、このアプリケーションの可用性を制御します。 CICS TS セキュリティー機能 `cicsts:security-1.0` のデフォルトの動作は SAF レジストリーを使用することです。LDAP レジストリーを使用する場合、SAF レジストリーは作成されません。詳しくは、分散 ID マッピングを使用した Liberty JVM サーバーのセキュリティーの構成を参照してください。基本ユーザー・レジストリー (`quickStartSecurity` でも使用) は、単純なセキュリティー・テストにのみ適しています。基本ユーザー・レジストリーを構成し、それを使用して実行し、`cicsts:security-1.0` に切り替える必要がある場合は、セッション・トークンを削除する必要があることに注意してください。

このタスクについて

このタスクでは、Liberty JVM サーバー用セキュリティーの構成方法、および Liberty セキュリティーと CICS セキュリティーの統合方法について説明します。Link to Liberty のセキュリティーの構成方法については、CICS プログラムから Java EE アプリケーションへのリンクを参照してください。

Web 要求を実行するためのデフォルトのトランザクション ID は CJSA です。ただし、JVMSERVER タイプの URIMAP を使用して、別のトランザクション ID で Web 要求を実行するように CICS を構成することもできます。通常、URIMAP を Web アプリケーションの一般コンテキスト・ルート (URI) に一致するように指定し、トランザクション ID の有効範囲が、そのアプリケーションを構成するサーブレットのセットになるようにします。あるいは、具体性の高い URI を使用して、個々のサーブレットを別々のトランザクションで実行することも可能です。

Web 要求を実行するためのデフォルトのトランザクション ID は、CICS のデフォルト・ユーザー ID です。URIMAP が使用可能で、静的ユーザー ID が含まれる場合、それがデフォルト・ユーザー ID よりも優先して使用されます。Web 要求のセキュリティー・ヘッダーにユーザー ID が含まれる場合、他のすべてのメカニズムより優先されます。

Liberty から発生し、Web 要求として分類されないタスクは、デフォルトでは CJSU トランザクションで実行されます。このようなタイプのタスクには URIMAP スタイルのメカニズムはありませんが、JVM プロファイル・プロパティー `com.ibm.cics.jvmserver.unclassified.tranid` を使用してデフォルト・トランザクション ID をオーバーライドし、JVM プロファイル・プロパティー `com.ibm.cics.jvmserver.unclassified.userid` を使用してデフォルト・ユーザー ID をオーバーライドすることができます。

注**:** ユーザー ID には、指定されたトランザクションに接続する権限が必要です。詳しくは、トランザクション・セキュリティーを参照してください。

手順

1. Liberty のエンジェル・プロセスを、認証サービスおよび許可サービスを Liberty JVM サーバーに提供するように構成します。 Liberty サーバーのエンジェル・プロセスを参照してください。

   ヒント**:** 名前付きエンジェル・プロセスがある場合は、JVM プロファイルに次の行を追加して、このエンジェル・プロセスに接続するように Liberty JVM サーバーを構成する必要があります。

   ```
   -Dcom.ibm.ws.zos.core.angelName=<named_angel>
   ```

2. オプション: JVM プロファイルに次の行を追加することによって、Liberty JVM サーバーが使用可能にされるときに Liberty エンジェル・プロセスに接続するという要件を適用します。

   ```
   -Dcom.ibm.ws.zos.core.angelRequired=true
   ```

   このオプションを使用すると、エンジェル・プロセスを使用できない場合には Liberty JVM サーバーが始動しなくなります。 これは CICS に対して、Liberty エンジェル検査 API を呼び出して、Liberty JVM サーバーの始動にエンジェル・プロセスを使用できるかどうかを検証するよう指示します。

   エンジェル・プロセスを使用できない場合、CICS は次の処理を行います。

   - Liberty JVM サーバーが CEMT トランザクションを介して使用可能にされる場合は、メッセージが発行され、Liberty JVM サーバーが使用不可になります。

- Liberty JVM サーバーが **SET JVMSERVER** SPI コマンドによって使用可能にされるか、または CICS Explorer を介した CMCI を使用して使用可能にされる場合は、メッセージが発行され、Liberty JVM サーバーが使用不可になります。
- Liberty JVM サーバーが CICS CREATE SPI、BAS、または GRPLIST によって使用可能にされる場合は、メッセージが発行され、CICS は 30 秒待機してから Liberty エンジェル検査 API 呼び出しを再試行します。 5 回試行してもエンジェル・プロセスを使用できない場合は、WTOR メッセージが発行され、そのまま待機するか JVMSERVER リソースを使用不可にするかを選択するオプションがオペレーターに示されます。

3. `cicsts:security-1.0` 機能を、`server.xml` の `featuremanager` リストに追加します。

```
<featureManager>
   ...
   <feature>cicsts:security-1.0</feature>
</featureManager>
...
```

4. 次の例を使用して、System Authorization Facility (SAF) レジストリーを `server.xml` に追加します。

```
<safRegistry id="saf" enableFailover="false"/>
```

5. `server.xml` に対する変更を保管します。
6. オプション: あるいは、Liberty JVM サーバーを自動構成する場合に CICS 領域で **SEC** システム初期設定パラメーターが YES に設定されていると、Liberty JVM サーバーの再始動時に、Liberty JVM セキュリティーをサポートするように Liberty JVM サーバーが動的に構成されます。詳しくは、Liberty JVM サーバーの構成を参照してください。 **SEC** システム初期設定パラメーターが NO に設定されていれば、認証または SSL サポートに依然として Liberty セキュリティーを使用できます。CICS セキュリティーがオフになっている場合に、Liberty セキュリティーを使用するには、`server.xml` ファイルを手動で構成する必要があります。
   a. `appSecurity-2.0` 機能を `featuremanager` リストに追加します。
   b. ユーザーを認証するユーザー・レジストリーを追加します。 Liberty セキュリティーでは、SAF、LDAP、および基本ユーザー・レジストリーがサポートされます。詳しくは、Liberty のユーザー・レジストリーの構成を参照してください。
   c. セキュリティー・ロール定義を追加して、アプリケーション・リソースへのアクセスを許可します。 275 ページの『ユーザーに Liberty JVM サーバー内のアプリケーションを実行する権限を与える』を参照してください。

タスクの結果

Web コンテナーは、Liberty の z/OS® セキュリティー機能を使用するように自動的に構成されます。 SAF レジストリーが認証に使用され、Java EE ロールが許可のために考慮されます。許可制約とセキュリティー・ロールによって、アプリケーションにアクセスできるユーザーが決定されます。これらは通常、アプリケーションのデプロイメント記述子 (`web.xml`) で定義されますが、ソース・コードのセキュリティー・アノテーションとして定義されている場合もあります。通常、ユーザーとグループは、`server.xml` 内のアプリケーションの *<application-bnd>* エレメント

によってロールにマップされます。あるいは、*<safAuthorization>* エレメントが `server.xml` で構成されている場合、マッピングは SAF に保持されます (RACF の EJBROLES として)。

次のタスク

- Liberty アプリケーション・セキュリティー認証規則を構成します。 272 ページの『Liberty JVM サーバーでのユーザー認証』を参照してください。
- Web アプリケーションの許可規則を定義します。 275 ページの『ユーザーに Liberty JVM サーバー内のアプリケーションを実行する権限を与える』および 281 ページの『SAF ロール・マッピングを使用した許可』を参照してください。
- Liberty 認証キャッシュを変更します。

Secure Sockets Layer (SSL) を使用するための詳細については、 287 ページの『Java 鍵ストアを使用した Liberty JVM サーバー用の SSL (TLS) の構成』を参照してください。

Liberty のエンジェル・プロセス

`cicsts:security-1.0` 機能を組み込むと、CICS Liberty JVM サーバーは、エンジェル・プロセスを使用して、System Authorization Facility (SAF) などの z/OS 許可サービスを呼び出します。

オプションで、エンジェル・プロセスに名前を付けることができます。エンジェル・プロセスに名前を指定しない場合、それがデフォルト・エンジェル・プロセスになります。デフォルトのエンジェル・プロセスは 1 つだけ作成できます。別のプロセスを作成しようとすると、その開始時に失敗します。 z/OS イメージ上で実行されるすべての Liberty サーバーで、1 つのエンジェル・プロセスを共用できます。各サーバーが実行しているコードのレベルや、各サーバーが CICS JVM サーバーで実行されるかどうかは関係ありません。名前付きエンジェル・プロセスについて詳しくは、名前付きエンジェルを参照してください。

重要: バンドルされている製品に関係なく、最新バージョンのエンジェル・プロセスをインストールします。最新バージョンは他の IBM ソフトウェアにバンドルされている可能性があります。また、CICS にバンドルされているバージョンを置き換えている可能性があります。

エンジェル・プロセスの開始タスクの実行

1. USSHOME ディレクトリー内の開始タスクの JCL プロシージャーを見つけます (例えば、`/usr/lpp/cicsts55/wlp/templates/zos/procs/bbgzangl.jcl`)。
2. JCL プロシージャーを変更し、JES プロシージャー・ライブラリーにコピーします。 **ROOT** には、`USSHOME/wlp` の値を設定できます。例えば、`ROOT=/usr/lpp/cicsts55/wlp` のようにします。
3. エンジェル・プロセスを開始します。以下の例では、`[.identifier]` は最大 8 文字のオプションの ID を示します。
   a. エンジェル・プロセスに名前を付けずに開始するには、次のコマンドを使用します。

      ```
      START BBGZANGL[.identifier]
      ```

b. エンジェル・プロセスを名前付きエンジェル・プロセスとして開始するには、オペレーターの START コマンドに NAME パラメーターを指定します。以下に例を示します。

```
START BBGZANGL[.identifier],NAME=<named_angel>
```

エンジェル・プロセス名は 1 文字から 54 文字までで、次の文字のみを使用してください。A-Z 0-9 ! # $ + - / : < > = ? @ [ ] ■ _ ` { } | ■

注: Liberty サーバーでは、独自の名前付きエンジェル・プロセスを使用できます。この分離の利点の 1 つは、エンジェル・プロセスが LPAR 上の他の Liberty サーバー・インスタンスに影響を与えずにサービスを提供できることです。エンジェル・プロセスは、Liberty JVM サーバーが始動する前に実行する必要があります。

4. Liberty JVM サーバーを始動します。デフォルトでは、サーバーは名前のないエンジェル・プロセス (使用可能な場合) に接続します。特定のエンジェル・プロセスに接続するには、com.ibm.ws.zos.core.angelName プロパティーを設定します。以下に例を示します。

```
-Dcom.ibm.ws.zos.core.angelName=named_angel
```

5. com.ibm.ws.zos.core.angelRequired プロパティーを true に設定すると、エンジェル・プロセスを有効にする前に、実行中のエンジェル・プロセスの有無を CICS で検査するように指定できます。以下に例を示します。

```
-Dcom.ibm.ws.zos.core.angelRequired=true
```

始動中にエンジェル・プロセスを使用できない場合、サーバーは失敗します。このプロパティーを使用すると、失敗になる状況をより迅速かつスムーズに検出できます。

エンジェル・プロセスの開始タスクとの対話

以下の例では、[.identifier] は最大 8 文字のオプションの ID を示します。

- エンジェル・プロセスを停止します。

```
STOP BBGZANGL[.identifier]
```

- 以下のコンソール・コマンドを使用して、エンジェル・プロセスに接続されている Liberty JVM サーバーを表示します。

```
MODIFY BBGZANGL[.identifier],DISPLAY,SERVERS,PID
```

ジョブ名とプロセス ID (PID) のリストが表示されます。

```
15.48.45 STC82204  CWWKB0067I ANGEL DISPLAY OF ACTIVE SERVERS
15.48.45 STC82204  CWWKB0080I ACTIVE SERVER ASID 5c JOBNAME IYK3ZNA1 PID 83953428
15.48.45 STC82204  CWWKB0080I ACTIVE SERVER ASID 5c JOBNAME IYK3ZNA1 PID 33621002
```

各 Liberty JVM サーバーは、固有の PID の下で実行し、CICS コマンド INQUIRE JVMSERVER によって返されます。

エンジェル・プロセスで使用される SAF プロファイル

- Liberty JVM サーバーは、CICS 領域ユーザー ID の権限の下で実行されます。許可サービスを使用するには、このユーザー ID でエンジェル・プロセスに接続

できる必要があります。エンジェル・プロセスの実行に使用するユーザー ID には、SAF STARTED プロファイルに対するアクセス権限が必要です。以下に例を示します。

```
RDEFINE STARTED BBGZANGL.* UACC(NONE) STDATA(USER(WLPUSER))
SETROPTS RACLIST(STARTED) REFRESH
```

- Liberty JVM サーバーでエンジェル・プロセスに接続するには、**SERVER** クラスにエンジェル (**BBG.ANGEL**。ただし、名前付きエンジェル・プロセスを使用している場合は **BBG.ANGEL.<namedAngelName>**) 用のプロファイルを作成します。例えば RACF で、それに対するアクセス権限を CICS 領域ユーザー ID (*cics_region_user*) に付与します。

  ```
  RDEFINE SERVER BBG.ANGEL UACC(NONE)
  PERMIT BBG.ANGEL CLASS(SERVER) ACCESS(READ) ID(cics_region_user)
  ```

- Liberty サーバーで z/OS 許可サービスを使用するには、許可モジュール **BBGZSAFM** 用の **SERVER** プロファイルを作成し、プロファイルに対して CICS 領域ユーザー ID (*cics_region_user*) を付与します。

  ```
  RDEFINE SERVER BBG.AUTHMOD.BBGZSAFM UACC(NONE)
  PERMIT BBG.AUTHMOD.BBGZSAFM CLASS(SERVER) ACCESS(READ) ID(cics_region_user)
  ```

- CICS 領域ユーザー ID (*cics_region_user*) の権限の下で実行される Liberty JVM サーバーに対して、**SERVER** クラスの SAF ユーザー・レジストリーと SAF 許可サービス (**SAFCRED**) へのアクセス権限を付与します。例えば、RACF では以下のように指定します。

  ```
  RDEFINE SERVER BBG.AUTHMOD.BBGZSAFM.SAFCRED UACC(NONE)
  PERMIT BBG.AUTHMOD.BBGZSAFM.SAFCRED CLASS(SERVER) ACCESS(READ) ID(cics_region_user)
  ```

- **IFAUSAGE** サービス (**PRODMGR**) 用の **SERVER** プロファイルを作成し、そこへのアクセス権限を CICS 領域ユーザー ID に付与します。これにより、Liberty JVM サーバーは、CICS JVM サーバーが使用可能であれば **IFAUSAGE** に登録し、使用不可であれば登録抹消できるようになります。以下に例を示します。

  ```
  RDEFINE SERVER BBG.AUTHMOD.BBGZSAFM.PRODMGR UACC(NONE)
  PERMIT BBG.AUTHMOD.BBGZSAFM.PRODMGR CLASS(SERVER) ACCESS(READ) ID(cics_region_user)
  ```

- **SERVER** リソースをリフレッシュします。

  ```
  SETROPTS RACLIST(SERVER) REFRESH
  ```

次の表は、CICS JVM サーバーで実行される Liberty サーバーによって使用される SAF セキュリティー・プロファイルを要約したものです。

表 32. *CICS Liberty* セキュリティーの *SAF* プロファイル表

クラス	**Profile (プロファイル)**	対象	**CICS 領域ユーザー ID** 1	非認証ユーザー **ID** 2	認証済みユーザー **ID** 3
SERVER	BBG.ANGEL	Liberty サーバー始動時のエンジェル・プロセスの登録	READ		
SERVER	BBG.ANGEL.<namedAngelName>	Liberty サーバー始動時のエンジェル・プロセスの登録	READ		

表 32. *CICS Liberty* セキュリティーの *SAF* プロファイル表 (続き)

クラス	**Profile** (プロファイル)	対象	**CICS** 領域ユーザー **ID** 1	非認証ユーザー **ID** 2	認証済みユーザー **ID** 3
SERVER	BBG.AUTHMOD.BBGZSAFM	Liberty サーバー始動時のエンジェル・プロセスの登録	READ		
SERVER	BBG.AUTHMOD.BBGZSAFM.SAFCRED	Liberty サーバー始動時のエンジェル・プロセスの登録	READ		
SERVER	BBG.AUTHMOD.BBGZSAFM.PRODMGR	Liberty サーバー始動時のエンジェル・プロセスの登録	READ		
SERVER	BBG.SECPFX.BBGZDFLT 4	認証または許可	READ		
APPL	BBGZDFLT 4	認証または許可		READ	READ
EJBROLE	BBGZDFLT.<resource>.<role> 5	認証または許可			READ

1. CICS ジョブまたは開始タスクに関連付けられているユーザー ID。
2. Liberty で非認証要求に使用するユーザー ID。この値は、<safCredentials> エレメントの unauthenticatedUser 属性を使用して制御されます。この値は、デフォルトで WSGUEST になります。
3. Liberty サーバーによって認証されたユーザー ID。
4. BBGZDFLT は、<safCredentials> エレメントの profilePrefix 属性を使用して設定されたセキュリティー・プロファイル接頭部のデフォルト値です。例えば、<safCredentials profilePrefix="BBGZDFLT"/> のようになります。
5. <safAuthorization> エレメントが構成されている場合は、EJBROLE プロファイルが必要です。プロファイルのデフォルト・パターンは、SAF ロール・マッパー・エレメントによって制御されます。このエレメントは、デフォルトで <safRoleMapper profilePattern="%profilePrefix%.%resource%.%role%"/> になります。

詳しくは、z/OS のプロセス・タイプを参照してください。

Liberty JVM サーバーでのユーザー認証

Liberty JVM サーバーで実行されるすべての Web アプリケーションに対して CICS セキュリティーを構成できますが、Web アプリケーションは、セキュリティー制約が Web アプリケーションに含まれている場合にのみユーザーを認証します。セキュリティー制約は、動的 Web プロジェクトまたは OSGi アプリケーション・プロジェクトのデプロイメント記述子 (web.xml) 内に、アプリケーション開発者が定義します。セキュリティー制約とは、保護対象 (URL) および保護に使用するロールを定義するものです。

<login-config> エレメントは、ユーザーが Web コンテナーへのアクセスを獲得する方法、および認証に使用する方式を定義します。サポートされるメソッドは、

HTTP 基本認証、フォーム・ベース認証、または SSL クライアント認証のいずれかです。 CICS 用のアプリケーション・セキュリティーを定義する方法について詳しくは、CICS Explorer 製品資料内の『SSL security for Explorer connections』を参照してください。web.xml 内のエレメントの例を以下に示します。

```
<!-- Secure the application -->
<security-constraint>
 <display-name>com.ibm.cics.server.examples.wlp.tsq.web_SecurityConstraint</display-name>
  <web-resource-name>com.ibm.cics.server.examples.wlp.tsq.web</web-resource-name>
  <description>Protection area for com.ibm.cics.server.examples.wlp.tsq.web</description>
  <url-pattern>/*</url-pattern>
 </web-resource-collection>
 <auth-constraint>
  <description>Only SuperUser can access this application</description>
  <role-name>SuperUser</role-name>
 </auth-constraint>
 <user-data-constraint>
  <!-- Force the use of SSL -->
  <transport-guarantee>CONFIDENTIAL</transport-guarantee>
 </user-data-constraint>
</security-constraint>

<!-- Declare the roles referenced in this deployment descriptor -->
<security-role>
 <description>The SuperUser role</description>
 <role-name>SuperUser</role-name>
</security-role>

<!--Determine the authentication method -->
<login-config>
  <auth-method>BASIC</auth-method>
</login-config>
```

注: あるサーブレットから別のサーブレットに要求を転送するために RequestDispatcher.forward() メソッドを使用すると、セキュリティー検査は、クライアントから要求された最初のサーブレットだけに行われます。

Liberty セキュリティーを使用して CICS で認証されたタスクでは、CICS でのトランザクションおよびリソース・セキュリティー検査を許可するために Liberty アプリケーション・セキュリティー・メカニズムのいずれかから派生したユーザー ID を使用できます。 CICS ユーザー ID は、以下の基準に従って決定されます。

1. Liberty アプリケーション・セキュリティー認証。

   SAF ユーザー・レジストリーとの統合は、CICS Liberty セキュリティー機能の一部として提供されます (分散 ID マッピングが使用されている場合を除く)。Liberty でサポートされているアプリケーション・セキュリティー・メカニズムは、いずれも CICS でサポートされます。これには、HTTP 基本認証、フォーム・ログイン、SSL クライアント証明書認証に加え、カスタム・ログイン・モジュール、JACC、JASPIC、または Trust Association Interceptor (TAI) を使用した ID アサーションが含まれます。 Liberty によって認証されるすべての SAF ユーザー ID には、Liberty JVM サーバーの APPL クラス・プロファイルへの読み取り権限が付与される必要があります。この名前は、Liberty サーバー構成ファイル server.xml の safCredentials エレメントの profilePrefix 設定によって決定されます。

   ```
   <safCredentials profilePrefix="BBGZDFLT"/>
   ```

APPL クラスは、特定の CICS 領域へのアクセスを制御するために CICS 端末ユーザーによっても使用され、Liberty JVM サーバーは、セキュリティー要件に応じて、CICS APPLID と同じプロファイルを使用できます。このエレメントを指定しない場合、デフォルトの profilePrefix である BBGZDFLT が使用されます。

APPLID は定義する必要があります。また、ユーザーはその APPLID へのアクセス権限を持っている必要があります。 APPL クラスの BBGZDFLT プロファイルを構成し、アクティブ化するには、次のようにします。

```
RDEFINE APPL BBGZDFLT UACC(NONE)
SETROPTS CLASSACT(APPL)
```

ユーザーを認証するには、ユーザーに APPL クラスの BBGZDFLT プロファイルへの読み取り権限が付与されている必要があります。ユーザー AUSER が BBGZDFLT APPL クラス・プロファイルに対して認証することを許可するには、次のようにします。

```
PERMIT BBGZDFLT CLASS(APPL) ACCESS(READ) ID(AUSER)
```

Liberty SAF 非認証ユーザー ID には、APPL クラス・プロファイルへの読み取り権限を与える必要があります。SAF で認証しないユーザー ID を Liberty サーバー構成ファイル server.xml の safCredentials エレメントで指定することができます。

```
<safCredentials unauthenticatedUser="WSGUEST"/>
```

このエレメントを指定しない場合、デフォルトの unauthenticatedUser は WSGUEST です。 SAF 非認証ユーザー ID に APPL クラスの BBGZDFLT プロファイルへの WSGUEST 読み取り権限を付与するには、以下のようにします。

```
PERMIT BBGZDFLT CLASS(APPL) ACCESS(READ) ID(WSGUEST)
```

WLP z/OS システム・セキュリティー・アクセス・ドメイン (WZSSAD) は、Liberty サーバーに付与された権限を参照します。これらの権限によって、サーバーがユーザーの認証と許可を行う際に、どの System Authorization Facility (SAF) アプリケーション・ドメインおよびリソース・プロファイルの照会を許されるかを制御します。 CICS 領域ユーザー ID には、認証呼び出しを行うために WZSSAD ドメイン内で権限を付与する必要があります。認証する権限を付与するには、CICS 領域 ID が SERVER クラスの BBG.SECPFX.<APPL> プロファイルに対する読み取り権限を付与されている必要があります。

```
RDEFINE SERVER BBG.SECPFX.BBGZDFLT UACC(NONE)
PERMIT BBG.SECPFX.BBGZDFLT CLASS(SERVER) ACCESS(READ) ID(cics_region_user)
```

詳しくは、WZSSAD を使用した z/OS セキュリティー・リソースへのアクセスを参照してください。

2. 認証されないサブジェクトが Liberty から提供された場合は、URIMAP に定義されている USERID が使用されます。
3. USERID が URIMAP に定義されていない場合は、CICS のデフォルトのユーザー ID で要求が実行されます。

注:

Liberty トランザクションのセキュリティー処理が CICS トランザクションの接続処理中に据え置かれるという方法のために、CICS Monitoring Facility (CMF) レコード、z/OS Workload Manager (WLM) 種別、タスク関連データ、および XAPADMGR 出口の UEPUSID グローバル・ユーザー出口フィールドで使用されるユーザー ID は次のように決定されます。HTTP セキュリティー・ヘッダーのユーザー ID、またはそれがない場合は、対応する URIMAP から取られたユーザー ID。どちらも存在しない場合、CICS のデフォルト・ユーザー ID が使用されます。

Liberty では、認証済みユーザー ID がキャッシュに入れられ、CICS とは異なり、キャッシュ期間中にユーザー ID の有効期限切れ検査がないことに注意してください。標準 Liberty 構成処理を使用して、キャッシュ・タイムアウトを構成できます。 Liberty の認証キャッシュの構成を参照してください。

ユーザーに Liberty JVM サーバー内のアプリケーションを実行する権限を与える

Java EE アプリケーション・セキュリティー・ロールを使用すると、Java EE アプリケーションへのアクセスを許可できます。また、Liberty JVM サーバーでは、CICS トランザクションおよびリソース・セキュリティーを使用して、トランザクションへのアクセスをさらに制限できます (アプリケーションの一部として実行)。

このタスクについて

アプリケーションは、デプロイメント記述子 (`web.xml`) 内の許可制約 (`<auth_constraint>` エレメント) を提供することによって保護されます。これが提供されている場合、許可されたロールのメンバーであるユーザーのみがアプリケーションにアクセスできるようになります。 Java EE ロールのユーザーまたはグループのメンバーシップは、次の 2 つの方法のいずれかで決定されます。

- `server.xml` の `<application>` エレメント内の `<application-bnd>` エレメントを使用して、ユーザー/グループからロールへのマッピングを XML に直接記述します。
- `server.xml` の `<safAuthorization>` を使用して、ユーザー/グループのロール・メンバーシップを SAF でマップできるようにします (通常は EJBROLES を使用)。

詳しくは、SAF ロール・マッピングを使用した許可を参照してください。

CICS セキュリティーを使用すると、既存のセキュリティー・プロシージャーを再利用できますが、個別の Web アプリケーションにはそれぞれ異なる URIMAP からアクセスする必要があります。ロール・ベースのセキュリティーを使用すると、別の Java EE アプリケーション・サーバーの既存の標準 Java EE セキュリティー定義を使用できます。詳しくは、 272 ページの『Liberty JVM サーバーでのユーザー認証』を参照してください。

CICS トランザクションとリソース許可を排他的に使用する場合、またはコードの詳細な注釈ベースのロールの検査を使用する場合は、以下の例に示すように、特別なサブジェクトの ALL_AUTHENTICATED_USERS ロールを使用して、それらのコ

ンポーネントに対する許可決定を据え置くことができます。 Liberty アプリケーションを CICS バンドルでデプロイする場合、これは CICS によって自動的に構成されます。

注: 宣言セキュリティー・アノテーション、および CICS のトランザクションとリソースのセキュリティーに対するアクセス検査は、構成済みの制約 (web.xml) の検証後にのみ実行されます

```
<application id="com.ibm.cics.server.examples.wlp.tsq.app"
    name="com.ibm.cics.server.examples.wlp.tsq.app" type="eba"
    location="${server.output.dir}/installedApps/com.ibm.cics.server.examples.wlp.tsq.app.eba">
 <application-bnd>
  <security-role name="cicsAllAuthenticated">
   <special-subject type="ALL_AUTHENTICATED_USERS"/>
  </security-role>
 </application-bnd>
</application>
```

この特別なサブジェクトを使用して Web アプリケーションのデプロイメント記述子 (web.xml) 内のすべての URL に cicsAllAuthenticated ロール・アクセスを与えると、認証された任意のユーザー ID を使用して Web アプリケーションにアクセスできるようになるため、トランザクションに対する許可は、CICS トランザクション・セキュリティーを使用して制御する必要があります。アプリケーションを dropins ディレクトリーに直接デプロイする場合、dropins はセキュリティーをサポートしないため、アプリケーションは CICS セキュリティーを使用するようには構成されません。

safAuthorization を使用すると、<application-bnd> はユーザー ID からロールへのマッピングのソースとして機能しなくなります。代わりに、SAF の EJBROLE によって、どの SAF ユーザーがどのロール (EJBROLE) を持つかが決まります。safAuthorization を指定すると、<application-bnd> は無視されます。同じ効果を実現し、すべての認証ユーザーにアプリケーションの実行を許可するには、web.xml の <auth-constraint> で特別なロール ** を使用する必要があります。例を以下に示します。

```
<auth-constraint>
        <description>special role for all authenticated users</description>
        <role-name>**</role-name>
</auth-constraint>
```

- 特別なロール名 ** は、ロールに依存しない認証済みユーザーの省略表現です。
- 特別なロール名 * は、デプロイメント記述子に定義されているすべてのロール名の省略表現です。

特別なロール名 ** が許可制約に含まれている場合は、ロールに依存しない認証済みユーザーが制約付き要求の実行を許可されていることを示します。特別なロールでは、web.xml に追加の <security-role> 宣言は必要ありません。

CICS トランザクションまたはリソースのセキュリティーを使用するには、以下の手順に従う必要があります。

手順

1. 各 Web アプリケーションにタイプ JVMSERVER の URIMAP を定義します。通常、URIMAP を Web アプリケーションの一般コンテキスト・ルート (URI) に一致するように指定し、トランザクション ID の有効範囲が、そのアプリケ

ーションを構成するサーブレットのセットになるようにします。あるいは、より正確な URI を使用して、個別のトランザクションでそれぞれのサーブレットが実行されるようにします。

2. CICS トランザクションまたはリソースのセキュリティー・プロファイルを使用して、URIMAP に指定したトランザクションを使用することを Web アプリケーションのすべてのユーザーに許可します。

OAuth 2.0 を使用したアプリケーションの許可

OAuth 2.0 は、委任された許可のオープン・スタンダードです。 OAuth 許可フレームワークを使用すると、ユーザーは、自分のアクセス権限や自分のデータの全範囲を共用せずに、別の HTTP サービスで保管された情報へのアクセス権限をサード・パーティー・アプリケーションに付与することができます。

WebSphere Liberty は OAuth 2.0 をサポートし、OAuth サービス・プロバイダー・エンドポイントおよび OAuth 保護リソース適用エンドポイントとして使用できます。 Liberty は、持続 OAuth 2.0 サービスをサポートします。 パーシスタント OAuth 2.0 サービスの構成を参照してください。localStore エレメントおよび client エレメントを使用して、クライアントをローカルで定義できます。以下の手順では、ローカル・クライアントを使用して、OAuth 2.0 許可を有効にします。

始める前に

SAF セキュリティーは CICS での一般的なユースケースであり、この手順では例で SAF を使用します。

CICS 領域で、SAF セキュリティーを使用するための構成が完了していることと、システム初期設定パラメーターとして SEC=YES が定義されていることを確認します。

オプションで、SAF EJBROLE `BBGZDFLT.com.ibm.ws.security.oauth20.clientManager` へのアクセス権限を管理者ユーザーに付与できます。セキュリティー・ロール `clientManager` は、管理インターフェースへのアクセスを制御して、ローカル・クライアントに対する照会と持続ローカル・クライアントの作成を行えるようにします。管理者ユーザーは、OAuth 2.0 ローカル・クライアントを制御します。

認証サービスおよび許可サービスを Liberty JVM サーバーに提供するように、Liberty のエンジェル・プロセスを構成します。 Liberty サーバー・エンジェル・プロセスを参照してください。

OAuth について詳しくは、oauth-2.0を参照してください。

このタスクについて

次の手順では、以下の方法について説明します。

- Liberty JVM サーバーで提供される OAuth 2.0 サービスを作成する。
- ローカルに構成されたクライアントを作成する。

- このローカル・クライアントを使用して、リライング・パーティー・アプリケーション (サード・パーティー Web アプリケーションとも呼ばれる) に OAuth 2.0 トークンを付与する。
- このトークンを使用して、アプリケーション内の保護リソースにアクセスする。

制約事項: Db2 JDBC タイプ 2 接続は、持続 OAuth 2.0 サービスではサポートされません。

手順

1. OAuth 2.0 サービス・プロバイダーを構成します。
   a. `oauth-2.0` および `cicsts:security-1.0` の各機能を `server.xml` の `featureManager` エレメントに追加します。

```
<featureManager>
    ...
    <feature>oauth-2.0</feature>
    <feature>cicsts:security-1.0</feature>
</featureManager>
...
```

   b. `server.xml` で OAuth 2.0 プロバイダーを構成します。

```
<oauthProvider id="myProvider">
</oauthProvider>
```

2. リライング・パーティー・アプリケーションのローカル・クライアントを構成します。ローカル・クライアントは、リライング・パーティー・アプリケーションの詳細 (アプリケーションの名前、秘密パスワード、リダイレクト URI を含む) を定義します。
   a. 分かりやすいローカル・クライアント名を定義し、サーバーが許可に使用する秘密パスワードを作成します。ローカル・クライアント・アプリケーションが URI を listen し、サーバーが許可コードを提供します。
   b. `server.xml` の `oauthProvider` エレメントで、ローカル・クライアント ID、秘密パスワード、およびリダイレクト URI を指定して、OAuth 2.0 ローカル・クライアントを構成します。

```
<oauthProvider id="myProvider">
    <localStore>
        <client id="myClient" redirect="https://client.example.ibm.com/webApp/redirect" secret="mySecret" />
    </localStore>
</oauthProvider>
```

   重要:

   この例では示されていませんが、パスワードをエンコードして、`server.xml` 構成へのアクセスを制限することが重要です。パスワードは、`USS_HOME/wlp/bin/securityUtility` にある Liberty `securityUtility` を使用してエンコードできます。詳しくは、securityUtility コマンドを参照してください。

   注: `localStore` エレメント内には、複数のローカル・クライアントを構成できます。

3. リライング・パーティー・アプリケーションでサーバー上の保護リソースへのアクセスが必要になる場合、ユーザーは、最初にこれらのリソースへのアクセスを許可する必要があります。

a. リライング・パーティー・アプリケーションでは、ユーザーが、サーバーで認証を行う必要があり、また、以下のようにユーザーを許可エンドポイントにリンクまたはリダイレクトすることにより、リライング・パーティー・アプリケーションのアクセスのタイプを選択する必要があります。

```
https://hostname:port/oauth2/endpoint/provider_name/authorize
```

または

```
https://hostname:port/oauth2/declarativeEndpoint/provider_name/authorize
```

URL の照会パラメーターには、追加のパラメーターが必要です。ステップ 2 で構成したローカル・クライアントの場合は、以下の GET 要求が必要です (すべて 1 行で記述)。

```
https://zos.example.ibm.com/oauth2/endpoint/myProvider/authorize?response_type=code
   &client_id=myClient&client_secret=mySecret&redirect_uri=https://client.example.ibm.com/webApp/redirect
```

ユーザーは、リライング・パーティー・アプリケーションのアクセス権限を選択すると、以下のリダイレクト URI を使用してリライング・パーティー・アプリケーションにリダイレクトされます。

```
https://client.example.ibm.com/webApp/redirect?code=access_code
```

リライング・パーティー・アプリケーションは、OAuth トークンを要求するためにこのアクセス・コードを保管する必要があります。

注: ローカル・クライアントの場合、Liberty JVM サーバーのユーザー・レジスターにユーザーが存在している必要があります。 Liberty JVM サーバーでのユーザー認証について詳しくは、Liberty JVM サーバーでのユーザー認証を参照してください。

b. リライング・パーティー・アプリケーションは、サーバーに POST 要求を送信して OAuth 2.0 トークンを要求します。

```
https://hostname:port/oauth2/endpoint/provider_name/token
```

リライング・パーティー・アプリケーションは、許可エンドポイントから受け取った許可コード、ローカル・クライアント ID、および秘密パスワードを POST データで送信します (`grant_type` はすべて 1 行で記述)。

```
POST https://zos.example.ibm.com/oauth2/endpoint/myProvider/token HTTP/1.1
Content-Type: application/www-form-urlencoded

grant_type=authorization_code&code=code&client_id=myClient
   &client_secret=mySecret&redirect_url=https://client.example.ibm.com/webApp/redirect
```

これにより、トークンを含んだ JSON 文書が返されます。

4. このトークンを使用して、保護リソースにアクセスします。

   a. HTTP 要求の `Authorization` ヘッダーにトークンを追加します。

   ```
   Authorization: Bearer <token>
   ```

タスクの結果

ユーザーは、OAuth 2.0 許可フローを介して、Liberty JVM サーバー内の保護リソースにアクセスする権限をサード・パーティー・アプリケーションに付与できます。 Liberty JVM サーバーは、これらのトークンのプロバイダーを構成して、ローカルに構成されたクライアントを作成することができます。

トークンを付与するためにいくつかの方式を使用できます。詳しくは、OAuth 2.0 サービス呼び出しを参照してください。

持続 OAuth 2.0 サービスの構成

WebSphere Liberty は、データベースでの OAuth 2.0 ローカル・クライアントとトークンの永続化をサポートしています。持続 OAuth 2.0 を使用すると、許可されたローカル・クライアントは、再始動後も引き続き OAuth 2.0 サービスにアクセスできます。

始める前に

SAF セキュリティーは CICS での一般的なユースケースであり、この手順では例で SAF を使用します。

- データベース内の表の作成とこれらの表への読み取り/書き込みに必要なアクセス権限を取得して、Liberty の `server.xml` で構成します。
- SAF EJBROLE `BBGZDFLT.com.ibm.ws.security.oauth20.clientManager` へのアクセス権限を管理者ユーザーに付与して、OAuth 2.0 ローカル・クライアントを制御します。
- Liberty の `server.xml` で OAuth 2.0 プロバイダーを作成します。詳しくは、OAuth 2.0 を使用した許可を参照してください。

このタスクについて

以下の手順で、持続 OAuth 2.0 ローカル・クライアントを作成します。このローカル・クライアントは、OAuth 2.0 トークンを付与するために使用します。

制約事項: Db2 JDBC タイプ 2 接続は、持続 OAuth 2.0 サービスではサポートされません。

手順

1. パーシスタント OAuth サービス用の IBM Db2を参考にして、必要な表を作成します。
2. 次の URL に POST 要求を送信して、持続ローカル・クライアントを作成します。

   ```
   https://hostname:port/oauth2/endpoint/provider_name/registration
   ```

   クライアント登録要求を受け入れるための OpenID Connect プロバイダーの構成の最初の表で説明されている JSON 文書を使用します。以下に例を示します。

   ```
   {
       "client_id": "client_id",
       "client_secret": "client_secret",
       "grant_types": [ "authorization_code", "refresh_token" ],
       "redirect_uris": [ "https://client.example.ibm.com/webApp/redirect" ]
   }
   ```

タスクの結果

持続 OAuth 2.0 ローカル・クライアントが作成されます。このローカル・クライアントを使用してトークンを生成すると、トークンがデータベースに保持されま

す。サーバーが再始動したときに、持続ローカル・クライアントとトークンは引き続き有効になります。

SAF ロール・マッピングを使用した許可

Java EE ロールからユーザーおよびグループへのマッピングは、さまざまな方法で行うことができます。分散システムでは、基本レジストリーまたは LDAP レジストリーを、通常はアプリケーション固有の `<application-bnd>` エレメントと組み合わせて使用して、これらのレジストリーに基づいてユーザーをロール にマップします。アプリケーションのデプロイメント記述子によって、アプリケーションのどの部分にどのロールがアクセスできるかが決まります。

このタスクについて

z/OS には、追加のレジストリー・タイプである System Authorization Facility (SAF) レジストリーがあります。`cicsts:security-1.0` 機能がインストールされると、LDAP を使用するように構成されていない限り、Liberty JVM サーバーはこのタイプを認証に暗黙的に使用します。SAF 許可を使用することを選択できます。SAF 許可を使用する場合、SAF ロール・マッパーを使用してロールを EJBROLE リソース・プロファイルにマップするために、ユーザーからロールへのマッピングが使用されます。サーバーは SAF に照会して、EJBROLE リソース・プロファイルに対する必要な READ 権限をユーザーが備えているかどうかを判別します。

Liberty JVM サーバーで SAF 許可なしで Java EE ロールを使用する場合、CICS バンドルを使用してアプリケーションをインストールすることはできません。その理由は、CICS バンドルがインストールされているアプリケーションの場合、`<application-bnd>` エレメントが自動的に作成され、特別なサブジェクト `ALL_AUTHENTICATED_USERS` が使用されるので、ユーザー自身でこのエレメントを定義することがないためです。代わりに、`server.xml` に `<application>` エレメントを直接作成し、必要なロールとユーザーを含めて `<application-bnd>` を構成する必要があります。

ただし、Java EE ロールと SAF 許可を使用することを選択した場合は、引き続き CICSバンドルを Web アプリケーションのライフサイクルに対して使用できます。SAF レジストリーで判別されるロール・マッピングを使用することを優先して、Liberty で `<application-bnd>` が無視されます。ロール・マッピングは、EJB ロールに属するユーザーによって決まります。

ヒント: SAF 許可が使用可能になっている場合、特別なサブジェクト `ALL_AUTHENTICATED_USERS` と `EVERYONE` は使用できません。

ヒント: CICS 領域を開始する前に、EJB ロールの作成または更新を行うことが推奨されます。Liberty は、z/OS の最小バージョンをサポートするために、`GOBAL=NO` を指定して `RACROUTE REQUEST=LIST` を発行します。再始動 (または開始) されるまで、アドレス・スペースには更新が表示されません。

手順

1. `<safAuthorization id="saf"/>` エレメントを `server.xml` に追加します。`cicsts:distributedIdentity-1.0` 機能を使用している場合、これは自動的に定義されます。

2. オプション: 前のステップで racRouteLog="ASIS" をエレメントに追加できます。 これにより、Liberty からの RACF EJBROLE ロギングを確認できます。
3. 記述されている接頭部スキームへの参照を使用して、RACF で EJB ロールを作成します。
4. ユーザーをこれらの EJB ロールに追加します。

   デフォルトでは、SAF 許可を使用すると、ユーザーが特定のロールに属しているかどうかを判別するために <profile_prefix>.<resource>.<role> というパターンがアプリケーションにより使用されます。profile_prefix 部分はデフォルトでは BBGZDFLT ですが、<safCredentials> エレメントを使用して変更できます。詳しくは、WZSSAD を使用した z/OS セキュリティー・リソースへのアクセスを参照してください。

   ロール・マッピングの設定は、server.xml の <safRoleMapper> エレメントを使用して変更できます。以下に例を示します。

   ```
   <safRoleMapper profilePattern="myprofile.%resource%.%role%" toUpperCase="true"/>
   ```

   以下の RACF コマンドを使用して、特定の EJB ロールに対する権限をユーザーに許可できます。ここで、WEBUSER は認証済みユーザー ID です。

   ```
   RDEFINE EJBROLE BBGZDFLT.MYAPP.ROLE UACC(NONE)
   PERMIT BBGZDFLT.MYAPP.ROLE CLASS(EJBROLE) ACCESS(READ) ID(WEBUSER)
   ```

5. オプション: CICS サーブレット・サンプルをデプロイし、Java EE ロール・セキュリティーを SAF 許可とともに使用する場合は、デプロイしたサーブレットごとに SAF EJBROLE を作成します。例えば、BBGZDFLT のデフォルト APPL クラスを使用する場合は、RACF コマンドを使用して、以下の EJBROLE セキュリティー定義を定義します。

```
RDEFINE EJBROLE BBGZDFLT.com.ibm.cics.server.examples.wlp.hello.war.cicsAllAuthenticated UACC(NONE)
RDEFINE EJBROLE BBGZDFLT.com.ibm.cics.server.examples.wlp.tsq.app.cicsAllAuthenticated UACC(NONE)
RDEFINE EJBROLE BBGZDFLT.com.ibm.cics.server.examples.wlp.jdbc.app.cicsAllAuthenticated UACC(NONE)
SETROPTS RACLIST(EJBROLE) REFRESH
```

   許可を必要とする Web ユーザー ID ごとに、定義したロールに読み取り権限を付与します。

```
PERMIT BBGZDFLT.com.ibm.cics.server.examples.wlp.hello.war.cicsAllAuthenticated
       CLASS(EJBROLE) ID(user) ACCESS(READ)
PERMIT BBGZDFLT.com.ibm.cics.server.examples.wlp.tsq.app.cicsAllAuthenticated
       CLASS(EJBROLE) ID(user) ACCESS(READ)
PERMIT BBGZDFLT.com.ibm.cics.server.examples.wlp.jdbc.app.cicsAllAuthenticated
       CLASS(EJBROLE) ID(user) ACCESS(READ)
SETROPTS RACLIST(EJBROLE) REFRESH
```

タスクの結果

ロールと、ロールに含めるユーザーを定義することにより、CICS セキュリティー、Java EE ロール・セキュリティー、あるいはその両方を使用して、Web アプリケーションに対するアクセス権限を付与できます。

LDAP レジストリーを使用した Liberty JVM サーバーのセキュリティーの構成

Liberty は、ユーザー・レジストリーを使用して、ユーザーを認証し、認証および許可を含むセキュリティー関連操作を実行するためのユーザーおよびグループに関する情報を取得します。CICS Liberty のデフォルト・セキュリティーでは、SAF レジストリーが使用されます。ただし、CICS で実行される多くのトランザクションは、分散アプリケーション・サーバー上で ID を認証するユーザーによって開始されるため、CICS は Liberty での Lightweight Directory Access Protocol (LDAP) レジストリーの使用もサポートします。LDAP を使用するには、`server.xml` を手動で構成する必要があります。

始める前に

- CICS 領域で、SAF セキュリティーを使用するための構成が完了していることと、システム初期設定パラメーターとして `SEC=YES` が定義されていることを確認します。
- アプリケーション開発者とシステム管理者に、Web アプリケーションを Liberty JVM サーバーにデプロイするために、JVMSERVER リソースと BUNDLE リソースの作成、表示、更新、および除去を行う権限を与えます。JVMSERVER リソースは JVM サーバーの可用性を制御します。BUNDLE リソースは Java アプリケーションのデプロイメント単位であり、このアプリケーションの可用性を制御します。

このタスクについて

このタスクでは、Liberty JVM サーバーの LDAP セキュリティーを構成し、Liberty セキュリティーを CICS セキュリティーに統合する方法を説明します。分散 ID マッピングは、SAF ユーザー ID を分散 ID に関連付けるために使用できます。CICS 分散 ID マッピング機能を使用して、分散 ID マッピングをセットアップできます。次に、ユーザーは、LDAP サーバーによって認証された自分の分散 ID を使用して CICS Web アプリケーションにログオンできます。z/OS セキュリティー製品で定義されたフィルター (RACMAP) が、この ID の SAF ユーザー ID へのマッピングを判別します。さらに、この SAF ユーザー ID を使用して、JEE アプリケーション・ロール・セキュリティーを介した Web アプリケーションへのアクセスを許可できます。そのようにして、CICS のトランザクションおよびリソース・セキュリティーとの統合が提供されます。SAF ユーザー ID は 1 つ以上の分散 ID にマップできます。

Web 要求を実行するためのデフォルトのトランザクション ID は CJSA です。JVMSERVER タイプの URIMAP を使用して、別のトランザクション ID で Web 要求を実行するように CICS を構成できます。URIMAP を Web アプリケーションの一般コンテキスト・ルート (URI) と一致するように指定して、トランザクション ID のスコープを、そのアプリケーションを構成するサーブレットのセットにすることができます。あるいは、具体性の高い URI を使用して、個々のサーブレットを別々のトランザクションで実行することを選択できます。

このタスクには、以下の 3 つのシナリオがあります。

- シナリオ 1 – SAF 許可を使用した分散 ID マッピング
- シナリオ 2 – SAF 許可を使用しない分散 ID マッピング

- シナリオ 3 – 認証と許可のための LDAP

手順

1. **SAF** 許可を使用した分散 **ID** マッピング

   CICS 分散 ID マッピング機能 `cicsts:distributedIdentity-1.0` を使用して、LDAP 分散 ID が SAF ユーザー ID にマップされるようにすることができます。CICS セキュリティー機能 `cicsts:security-1.0` とともに使用すると、Liberty LDAP セキュリティーが認証に使用され、EJB ロール・マッピングからの JEE アプリケーション・ロール・セキュリティーが許可のために考慮されます。 CICS トランザクションは、マップされた SAF ユーザー ID で実行され、CICS トランザクションとリソース・セキュリティーとの統合が提供されます。

   a. 認証サービスと許可サービスを Liberty JVM サーバーに提供するように、WebSphere Liberty エンジェル・プロセスを構成します。詳しくは、Liberty サーバーのエンジェル・プロセスを参照してください。

   b. `cicsts:security-1.0` 機能および `cicsts:distributedIdentity-1.0` 機能を、`server.xml` 内の featureManager リストに追加します。

   ```
   <featureManager>
      ...
      <feature>cicsts:security-1.0</feature>
      <feature>cicsts:distributedIdentity-1.0</feature>
   </featureManager>
   ...
   ```

   c. 例えば、`server.xml` で LDAP サーバーを定義することにより、Liberty が LDAP 認証を使用するように構成します。

   ```
   <ldapRegistry id="ldap"
       host="host.domain.com" port="389"
       ldapType="IBM Tivoli Directory Server"
       baseDN="ou=users,dc=domain,dc=com"
       ignoreCase="true">
   </ldapRegistry>
   ```

   Liberty での LDAP ユーザー・レジストリーの構成について詳しくは、Liberty での LDAP ユーザー・レジストリーの構成を参照してください。

   d. safRegistry エレメントを削除します (存在する場合)。`server.xml` に対する変更を保管します。

   e. 必要な RACF 定義を行います。これには、Liberty での LDAP ユーザー・レジストリーの構成の説明に従って、分散 ID を SAF ユーザー ID にマップするように RACMAP をセットアップすることや、 281 ページの『SAF ロール・マッピングを使用した許可』の説明に従って、それらのユーザー ID が適切な EJBROLES にアクセスできるようにすることが含まれます。CICS は、SAF の許可および mapDistributedIdentities 属性を safCredentials 構成エレメントで構成します。

   `cicsts:distributedIdentity-1.0` 機能が `cicsts:security-1.0` 機能とともに使用されている場合、Liberty LDAP セキュリティーが認証に使用され、EJB ロール・マッピングからの JEE アプリケーション・ロール・セキュリティーが許可

のために考慮されます。 CICS トランザクションは、RACMAP がマップしたユーザー ID で実行され、CICS トランザクションとリソース・セキュリティーとの統合が提供されます。次のタスク

先頭に戻る

2. **SAF** 許可を使用しない分散 **ID** マッピング

   アプリケーションの `<application-bnd>` エレメントで構成されたロールを考慮すると同時に、CICS トランザクションに、RACMAP がマップしたユーザー ID で実行することを許可することができます。これは、分散した Liberty から CICS Liberty に作業をマイグレーションする際に役立ちます。CICS バンドルを使用すると、ユーザー定義の `<application-bnd>` は、CICS 生成の `<application-bnd>` によって上書きされることに注意してください。ロール・マッピングを使用した SAF 許可を使用することをお勧めします。詳細については、 281 ページの『SAF ロール・マッピングを使用した許可』を参照してください。

   a. 認証サービスおよび許可サービスを Liberty JVM サーバーに提供するように WebSphere Liberty エンジェル・プロセスを構成します。詳しくは、Liberty サーバーのエンジェル・プロセスを参照してください。

   b. `cicsts:security-1.0` 機能および `ldapRegistry-3.0` 機能を、`server.xml` の featureManager リストに追加します。

   ```
   <featureManager>
      ...
      <feature>cicsts:security-1.0</feature>
      <feature>ldapRegistry-3.0</feature>
   </featureManager>
   ...
   ```

   c. 例えば、`server.xml` で LDAP サーバーを定義することにより、Liberty が LDAP 認証を使用するように構成します。

   ```
   <ldapRegistry id="ldap"
       host="host.domain.com" port="389"
       ldapType="IBM Tivoli Directory Server"
       baseDN="ou=users,dc=domain,dc=com"
       ignoreCase="true">
   </ldapRegistry>
   ```

   Liberty での LDAP ユーザー・レジストリーの構成について詳しくは、Liberty での LDAP ユーザー・レジストリーの構成を参照してください。

   d. 分散 ID フィルターを使用して分散 ID を SAF ユーザー ID にマップするように Liberty を構成します。それには、`safCredentials` 構成エレメントの `mapDistributedIdentities` 属性を `server.xml` で `true` に設定します。

   e. safRegistry エレメントを削除します (存在する場合)。`server.xml` に対する変更を保管します。

   f. 必要な RACF 定義を行います。これには、 Liberty での LDAP ユーザー・レジストリーの構成の説明に従って、分散 ID を SAF ユーザー ID にマップするように RACMAP をセットアップすることが含まれます。

   g. EJB ロールからの JEE アプリケーション・ロール・セキュリティーが許可に必要な場合は、 281 ページの『SAF ロール・マッピングを使用した許可』のトピックを参照してください。

Liberty LDAP セキュリティーが認証に使用され、<application-bnd> エレメントの JEE アプリケーション・ロール・セキュリティーが分散 ID の許可のために考慮されます。CICS では、トランザクションは RACMAP がマップしたユーザー ID で実行され、CICS トランザクションとリソース・セキュリティーとの統合が提供されます。次のタスク

先頭に戻る

3. 認証と許可のための **LDAP**

   LDAP セキュリティーは、JEE アプリケーション・ロール・セキュリティーを使用した認証と許可の両方のために、CICS Liberty JVM サーバーで使用できます。その後、URIMAP 定義を使用して、トランザクションの実行ユーザー ID を設定できます。このシナリオでは、mapDistributedIdentities 属性は設定されません。

   このシナリオは、セキュリティー・リソースを大幅に変更する必要なしに、分散アプリケーションを CICS Liberty JVM サーバーにマイグレーションする場合に役立ちます。

   a. cicsts:security-1.0 機能および ldapRegistry-3.0 機能を、server.xml の featureManager リストに追加します。

      ```
      <featureManager>
         ...
         <feature>cicsts:security-1.0</feature>
         <feature>ldapRegistry-3.0</feature>
      </featureManager>
      ...
      ```

   b. 例えば、server.xml で LDAP サーバーを定義することにより、Liberty が LDAP 認証を使用するように構成します。

      ```
      <ldapRegistry id="ldap"
          host="host.domain.com" port="389"
          ldapType="IBM Tivoli Directory Server"
          baseDN="ou=users,dc=domain,dc=com"
          ignoreCase="true">
      </ldapRegistry>
      ```

      Liberty での LDAP ユーザー・レジストリーの構成について詳しくは、Liberty での LDAP ユーザー・レジストリーの構成を参照してください。

   c. safRegistry エレメントを削除します (存在する場合)。server.xml に対する変更を保管します。

   d. EJB ロールからの JEE アプリケーション・ロール・セキュリティーが許可に必要な場合は、 281 ページの『SAF ロール・マッピングを使用した許可』のトピックを参照してください。

   アプリケーションは Liberty LDAP セキュリティーを認証に使用し、<application-bnd> エレメントの JEE アプリケーション・ロール・セキュリティーが許可のために考慮されます。CICS トランザクションでは、必要に応じて、URIMAP ユーザー ID または CICS DFLTUSER ユーザー ID で実行します。次のタスク

   先頭に戻る

次のタスク

以下は 3 つのシナリオすべてに適用されます。

- Liberty 認証キャッシュを変更します。
- Web アプリケーション URL をトランザクション ID にマップするように URIMAP 定義をセットアップします。

以下はシナリオ 1 とシナリオ 2 に適用されます。

- マップされたユーザー ID に基づいて URI へのアクセスを許可するように、CICS トランザクション・セキュリティー定義をセットアップします。

先頭に戻る

Java 鍵ストアを使用した Liberty JVM サーバー用の SSL (TLS) の構成

SSL を使用してデータを暗号化するように Liberty JVM サーバーを構成できます。オプションとして、クライアント証明書を使用してサーバーで認証するように構成することもできます。証明書は、Java 鍵ストアか、または RACF などの SAF 鍵リングに保管できます。

このタスクについて

Liberty JVM サーバーで SSL を使用可能にするには、**transportSecurity-1.0** Liberty フィーチャー、鍵ストア、および HTTPS ポートを追加する必要があります。CICS では自動的に `server.xml` ファイルが作成され、更新されます。自動構成では、必ず Java 鍵ストアが作成されます。

Liberty JVM サーバーへの Web 要求では、CICS ソケット・ドメインではなく、TCP/IP ソケットおよび SSL 処理の JVM サポートが使用されることを理解することが重要です。

手順

SSL を構成するために自動構成を使用するには、以下のステップを実行します。

1. JVM プロファイルで JVM システム・プロパティー **-Dcom.ibm.cics.jvmserver.wlp.autoconfigure=true** を使用して、自動構成を有効にします。
2. JVM プロファイルで JVM システム・プロパティー **-Dcom.ibm.cics.jvmserver.wlp.server.https.port** を設定して、SSL ポートを設定します。
3. JVM サーバーを再始動して、必要な構成エレメントを `server.xml` に追加します。

タスクの結果

Liberty JVM サーバーのための SSL が正常に構成されます。

RACF を使用した Liberty JVM サーバー用の SSL (TLS) の構成

SSL を使用してデータを暗号化するように Liberty JVM サーバーを構成できます。オプションとして、クライアント証明書を使用してサーバーで認証するように構成することもできます。証明書は、Java 鍵ストアか、または RACF などの SAF 鍵リングに保管できます。

このタスクについて

Liberty JVM サーバーで SSL を使用可能にするには、**transportSecurity-1.0** Liberty フィーチャー、鍵ストア、および HTTPS ポートを追加する必要があります。server.xml ファイルを編集して、必要なエレメントと値を追加します。RACF 鍵リングを使用する場合は、手動の手順に従う必要があります。

Liberty JVM サーバーへの Web 要求では、CICS ソケット・ドメインではなく、TCP/IP ソケットおよび SSL 処理の JVM サポートが使用されることを理解することが重要です。

手順

SSL を手動で構成するには、署名証明書を作成する必要があります。この署名証明書を使用して、サーバー証明書を作成します。そして、その署名証明書を、署名証明書を使用してサーバー証明書の認証を行うクライアントの Webブラウザーにエクスポートします。

1. 認証局 (CA) 証明書 (署名証明書) を作成します。RACF コマンドを使用する例を以下に示します。

   ```
   RACDCERT GENCERT
     CERTAUTH
     SUBJECTSDN(CN('CICS Sample Certification Authority')
      O('IBM')
      OU('CICS'))
      SIZE(2048)
     WITHLABEL('CICS-Sample-Certification')
   ```

   証明書の SIZE は、少なくとも 2048 ビットにする必要があります。詳しくは、RACF RACDCERT GENCERT (証明書の生成) コマンドを参照してください。

2. ステップ 2 の署名証明書を使用したサーバー証明書を作成します。ここで、<userid> は CICS 領域ユーザー ID です。hostname は、Liberty サーバー HTTPS ポートで使用するように構成されたサーバーのホスト名です。

   ```
   RACDCERT ID(<userid>) GENCERT
     SUBJECTSDN(CN('<hostname>')
      O('IBM')
      OU('CICS'))
      SIZE(2048)
      SIGNWITH (CERTAUTH LABEL('CICS-Sample-Certification'))
     WITHLABEL('<userid>-Liberty-Server')
   ```

   証明書の SIZE は、少なくとも 2048 ビットにする必要があります。詳しくは、RACF RACDCERT GENCERT (証明書の生成) コマンドを参照してください。

3. 署名証明書およびサーバー証明書を RACF 鍵リングに接続します。 以下のコマンドで RACF を使用できます。<keyring> の値は、使用する鍵リングの名前に置き換えてください。<userid> の値は CICS 領域ユーザー ID に置き換えてください。

```
RACDCERT ID(<userid>) CONNECT(RING(<keyring>)
       LABEL('CICS-Sample-Certification')
       CERTAUTH)

RACDCERT ID(<userid>) CONNECT(RING(<keyring>)
       LABEL('<userid>-Liberty-Server'))
```

署名証明書を CER ファイルにエクスポートします。

```
RACDCERT CERTAUTH EXPORT(LABEL('CICS-Sample-Certification'))
       DSN('<userid>.CERT.LIBCERT')
       FORMAT(CERTDER)
       PASSWORD('password')
```

エクスポートされた証明書を、FTP を使用してバイナリー形式でワークステーションに転送し、認証局証明書としてブラウザーにインポートします。

4. server.xml ファイルを編集し、SSL 機能と鍵ストアを追加します。HTTPS ポート (次の例の値は 9443) を設定し、CICS 領域を再始動します。SAF 鍵リングを URL 形式 safkeyring://<userid>/<keyring> で指定する必要があります。<userid> 値には CICS 領域ユーザー ID を設定し、<keyring> 値には鍵リングの名前を設定する必要があります。パスワード・フィールドは、SAF 鍵リングへのアクセスには使用されないため、password に設定する必要があります。

```
<featureManager>
  ...
       <feature>transportSecurity-1.0</feature>
</featureManager>
...
<httpEndpoint host="*" httpPort="9080" httpsPort="9443"
    id="defaultHttpEndpoint"/>
...
.
<keyStore filebased="false" id="racfKeyStore"
        location="safkeyring://<userid>/<keyring>"
        password="password"
        readOnly="true"
        type="JCERACFKS"/>
<ssl id="defaultSSLConfig" keyStoreRef="racfKeyStore"
      sslProtocol="SSL_TLS"
      serverKeyAlias="<userid>-Liberty-Server" />
```

タスクの結果

Liberty JVM サーバーのための SSL が正常に構成されます。

Liberty JVM サーバーでの SSL (TLS) クライアント証明書認証のセットアップ

SSL クライアント証明書認証を使用することで、クライアントおよびサーバーは証明書を相手に提供して相互に確認できます。この方法は、セキュリティーについての懸念のために追加レベルの認証が必要となる状況でよく使用されます。

始める前に

RACF を使用した Liberty JVM サーバー用の SSL (TLS) の構成のタスクを完了する必要があります。まだ CICS Liberty のセキュリティーをセットアップ済みでない場合は、先に進む前に、Liberty JVM サーバーのセキュリティーの構成 (Configuring security for a Liberty JVM server)を完了させてください。

このタスクについて

以下のセットアップ情報では、RACF 鍵ストアを使用して、SSLクライアント証明書認証の証明書を保管していることを想定しています。

手順

1. 署名証明書を使用して個人証明書を作成し、この個人証明書を RACF ユーザー ID に関連付けます。 次に、個人証明書を CER 形式でデータ・セットにエクスポートしてから、ワークステーションにバイナリーで FTP 転送します。その個人証明書を Web ブラウザーに個人証明書としてインポートします。証明書が Web ブラウザーにインポートされると、SSL クライアント証明書を提供して Liberty サーバーの HTTPS ポートに接続できるようになります。次の RACF コマンドを実行します。ここで `<clientuserid>` は、RACF ユーザー ID を `<hostname>` はクライアント・コンピューターのホスト名を表します。

   ```
   RACDCERT ID(<clientuserid>) GENCERT
     SUBJECTSDN(CN('<hostname>')
      O('IBM')
      OU('CICS'))
      SIZE(2048)
      SIGNWITH (CERTAUTH LABEL('CICS-Sample-Certification'))
     WITHLABEL('<clientuserid>-certificate')
   ```

   この手順で既に行ったように、個人証明書をエクスポートします。

   ```
   RACDCERT ID(<clientuserid>)
     EXPORT(LABEL('<clientuserid>-certificate'))
      DSN('USERID.CERT.CLICERT')
      FORMAT(PKCS12DER)
      PASSWORD('password')
   ```

   SSL クライアント証明書認証をサポートするように、`server.xml` の SSL 要素を更新します。

   ```
   <ssl id="defaultSSLConfig" keyStoreRef="racfKeyStore"
      sslProtocol="SSL_TLS"
      serverKeyAlias="<userid>-Liberty-Server"
      clientAuthenticationSupported="true"/>
   ```

   さらに、すべてのクライアントが有効な SSL クライアント証明書を必ず提供するようにしたい場合は、次のようにして **clientAuthentication** 属性を SSL 要素に追加します。

   ```
   <ssl id="defaultSSLConfig" keyStoreRef="racfKeyStore"
      sslProtocol="SSL_TLS"
      serverKeyAlias="<userid>-Liberty-Server"
      clientAuthenticationSupported="true"
      clientAuthentication="true"/>
   ```

2. 手順 2 のクライアント・ユーザー ID の ID を使用して CICS で Web 要求を認証できます。その後、`web.xml` に CLIENT-CERT の `login-config` 要素を指

定して、Web アプリケーションをデプロイします。web.xml ファイルは、デプロイする Web アプリケーションのソース・ファイルにあります。

```
<login-config>
  <auth-method>CLIENT-CERT</auth-method>
</login-config
```

代わりに、SSL クライアント証明書認証が構成されていない場合に HTTP 基本認証へのフェイルオーバーを許可するには、server.xml に webAppSecurity 要素を追加します。

```
<webAppSecurity allowFailOverToBasicAuth="true" />
```

3. 最後に、CICS トランザクション・セキュリティーをセットアップして、CICS トランザクションへのアクセスを認証済みクライアント・ユーザー ID に基づいて許可します。 詳細は、 275 ページの『ユーザーに Liberty JVM サーバー内のアプリケーションを実行する権限を与える』を参照してください。

Java セキュリティー・マネージャーの有効化

デフォルトで、Java アプリケーションでは、Java API に要求されるアクティビティーにセキュリティーの制限がありません。Java セキュリティーを使用して、安全でない可能性があるアクションを Java アプリケーションが実行しないようにするために、そのアプリケーションが実行される JVM に対してセキュリティー・マネージャーを有効にすることができます。

このタスクについて

セキュリティー・マネージャーは、コード・ソースに割り当てられる 1 組の権限 (システム・アクセス権) であるセキュリティー・ポリシーを実施します。 Java プラットフォームにはデフォルトのポリシー・ファイルが用意されています。ただし、Java セキュリティーがアクティブであるときに、Java アプリケーションを CICS で正常に実行できるようにするには、アプリケーションの実行に必要な権限を CICS に付与する追加のポリシー・ファイルを指定する必要があります。

この追加のポリシー・ファイルは、セキュリティー・マネージャーが有効になっている JVM の種類ごとに指定する必要があります。CICS には、独自のポリシーを作成するために使用できるサンプルが用意されています。

注: Liberty JVM サーバーでは、Java セキュリティー・マネージャーの有効化はサポートされていません。

- OSGi セキュリティー・エージェントのサンプルは、プロジェクトに com.ibm.cics.server.examples.security という名前の OSGi ミドルウェア・バンドルを作成します。その中には、セキュリティー・プロファイルが含まれています。このプロファイルは、インストール先のフレームワーク内にあるすべての OSGi バンドルに適用されます。
- example.permissions ファイルには、JVM サーバーで実行中のアプリケーションに固有の許可が含まれています。それには、アプリケーションが System.exit() メソッドを使用していないことを確認するチェックが含まれます。
- CICS に、zFS の中で OSGi バンドルの配置先となるディレクトリーに対する読み取りアクセス権限と実行アクセス権限が必要です。

JVM サーバーの OSGi フレームワークで実行されるアプリケーションの場合:

手順

1. IBM CICS SDK for Java でプラグイン・プロジェクトを作成して、提供される OSGi セキュリティー・エージェントのサンプルを選択します。
2. プロジェクトで、`example.permissions` ファイルを選択して、セキュリティー・ポリシーの許可を編集します。
   a. CICS zFS および Db2 のインストール・ディレクトリーが正しく指定されていることを確認します。
   b. 必要に応じて他の権限を追加します。
3. zFS 内の適切なディレクトリー (例えば `/u/bundles`) に OSGi バンドルをデプロイします。
4. JVM サーバーの JVM プロファイルを編集して、他のすべてのバンドルより前になるように、`OSGI_BUNDLES` オプションに OSGi バンドルを追加します。

   ```
   OSGI_BUNDLES=/u/bundles/
   com.ibm.cics.server.examples.security_1.0.0.jar
   ```

5. JVM プロファイルに次の Java プロパティーを追加して、セキュリティーを有効にします。

   ```
   -Djava.security.policy=all.policy
   ```

6. JVM プロファイルに次の Java 環境変数を追加して、OSGi フレームワーク内でセキュリティーを有効にします。

   ```
   org.osgi.framework.security=osgi
   ```

7. OSGi フレームワークを Java 2 セキュリティーで開始できるようにするには、以下のポリシーを追加します。

   ```
   grant { permission java.security.AllPermission; };
   ```

8. 変更を保管し、JVMSERVER リソースを使用可能にして、ミドルウェア・バンドルを JVM サーバーにインストールします。
9. オプション: Java 2 セキュリティーをアクティブにします。
   a. Java 2 セキュリティー・ポリシー・メカニズムをアクティブにするには、それを適切な JVM プロファイルに追加します。また、Java 2 セキュリティー・ポリシーを編集して、適切な権限を付与する必要があります。
   b. Java 2 セキュリティー・ポリシー・メカニズムがアクティブな状態で Java アプリケーションから JDBC または SQLJ を使用するには、IBM Data Server Driver for JDBC and SQLJ を使用します。
   c. Java 2 セキュリティー・ポリシー・メカニズムをアクティブにするには、JVM プロファイルを編集します。
   d. JDBC ドライバーに対する権限を付与するには、例 1 に示されている行を追加して、Java 2 セキュリティー・ポリシーを編集します。`db2xxx` の代わりに、すべての db2 ライブラリーが配置されているディレクトリーを指定してください。権限は、このレベルより下のすべてのディレクトリーとファイルに適用されます。これにより、JDBC および SQLJ を使用できるようになります。
   e. 例 2 に示されている行を追加して、Java 2 セキュリティー・ポリシーを編集して読み取り許可を付与します。読み取り許可を追加しないで Java プロ

グラムを実行すると、AccessControlExceptions と予測不能な結果が生成されます。 Java 2 セキュリティー・ポリシーで JDBC および SQLJ を使用できます。

例 **1:**

```
grant codeBase "file:/usr/lpp/db2xxx/-" {
  permission java.security.AllPermission;
};
```

例 **2:**

```
grant {

// allows anyone to read properties
permission java.util.PropertyPermission "*", "read";

};
```

タスクの結果

Java アプリケーションが呼び出されると、JVM は、クラスのコード・ソースを判別し、セキュリティー・ポリシーを調べてから、適切な権限をクラスに付与します。

第 7 章 Java パフォーマンスの改善

Java アプリケーションとそれらのアプリケーションが実行される JVM のパフォーマンスを改善するために、さまざまなアクションを実行することができます。

このタスクについて

CICS 自体の微調整に加えて、以下の方法で Java アプリケーションのパフォーマンスをさらに高めることができます。

- Java アプリケーションが適切に作成されていることを確認する
- Java ランタイム環境 (JVM) を調整する
- JVM を実行する言語を調整する

手順

1. Java ワークロードのパフォーマンス目標を決定します。 最も一般的な目標には、プロセッサー使用率やアプリケーション応答時間の最小化があります。目標を決定したら、Java 環境を調整することができます。
2. Java アプリケーションを分析して、その Java アプリケーションが効率よく動作し、過剰なガーベッジを生成していないことを確認します。 IBM が提供するツールは、Java アプリケーションを分析して、特定の方法およびアプリケーション全体の効率とパフォーマンスを改善するのに役立ちます。
3. JVM サーバーを調整します。 統計と IBM ツールを使用すれば、ストレージの設定、ガーベッジ・コレクション、タスクの待機などの情報を分析して、JVM のパフォーマンスを調整することができます。
4. JVM が実行される Language Environment エンクレーブを調整します。 JVM が使用する MVS ストレージは、MVS Language Environment サービスの呼び出しによって取得されます。Language Environment のランタイム・オプションを変更して、MVS によって割り振られるストレージを調整することができます。
5. オプション: z/OS 共用ライブラリー領域を使用して、異なる CICS 領域内の JVM 間で DLL を共用する場合、ストレージの設定を調整できます。

Java ワークロードにおけるパフォーマンス・ゴールの判別

所定のアプリケーション・ワークロードの全体的なパフォーマンスが最良のものになるように CICS JVM を調整するには、いくつかの異なる要因が関係しています。Java ワークロードに必要なパフォーマンス特性を決定する必要があります。これらの特性を設定すると、変更が必要なパラメーターとその変更方法を判別することができます。

Java ワークロードの最も一般的なパフォーマンス目標は、以下のとおりです。

最小限の総プロセッサー使用率
: この目標は、使用可能なプロセッサー・リソースを最も効率的に使用することに重点を置いています。この目標を達成するようにワークロードを調整す

© Copyright IBM Corp. 1999, 2018

ると、ワークロード全体のプロセッサーの合計使用率は最小化されますが、個々のタスクはプロセッサーを著しく消費する可能性があります。総プロセッサー使用率が最小限になるよう調整するには、JVM に大きいストレージ・ヒープ・サイズを指定して、ガーベッジ・コレクション数を最小化することが必要です。

アプリケーションの最小応答時間
: この目標は、アプリケーション・タスクをできるだけ素早く呼び出し元に戻すことに重点を置いています。達成しなければならないサービス・レベル・アグリーメントが存在する場合、この目標は特に重要である可能性があります。この目標を達成するようにワークロードを調整すると、アプリケーションは一貫して素早く応答します。ただし、ガーベッジ・コレクションのためにプロセッサー使用率が高くなることがあります。アプリケーションの最小応答時間のための調整には、ヒープ・サイズを小さくしておくことと、おそらく gencon ガーベッジ・コレクション・ポリシーを使用することが必要です。

最小限の **JVM** ストレージ・ヒープ・サイズ
: この目標は、JVM によって使用されるストレージの量を削減することに重点を置いています。 JVM で使用されるストレージの量を削減するには、JVM ヒープ・サイズを小さくします。

注**:** JVM ヒープ・サイズを小さくすると、ガーベッジ・コレクション・イベントの頻度が高まる可能性があります。

その他の要因が、アプリケーションの応答時間に影響を与える場合があります。その中でも最も重要なのが、Just In Time (JIT) コンパイラーです。 JIT コンパイラーは、実行時にアプリケーション・コードを動的に最適化し、多くの利点を提供しますが、これを行うには一定量のプロセッサー・リソースが必要になります。

IBM Health Center を使用した Java アプリケーションの分析

Java アプリケーションのパフォーマンスを改善するには、IBM Health Center を使用してそのアプリケーションを分析することができます。このツールは、アプリケーションのパフォーマンスと効率の改善に役立つ推奨事項を提供します。

このタスクについて

IBM Health Center は、IBM Support Assistant Workbench で使用できます。これらの無料のツールは、IBM Health Center の入門ガイドに記載されているとおりに IBM からダウンロードできます。JVM でアプリケーションを単独で実行してみてください。JVM サーバーで混合ワークロードを実行している場合は、特定アプリケーションの分析がさらに難しくなることがあります。

手順

1. 必要な接続オプションを JVM サーバーの JVM プロファイルに追加します。IBM Health Center の資料に、ツールから JVM に接続するために追加する必要があるオプションが記述されています。

2. IBM Health Center を開始し、実行中の JVM に接続します。 IBM Health Center は JVM のアクティビティーをリアルタイムで報告するので、IBM Health Center が JVM をモニターするまでしばらく待ってください。
3. 「**Profiling**」リンクを選択して、アプリケーションのプロファイルを作成します。 さまざまなメソッドで費やす時間を確認できます。使用率が最大のメソッドを確認して、潜在的な問題を探してください。

   ヒント**:** 「**Analysis and Recommendations**」タブでは、最適化の候補になりうる特定のメソッドを識別できます。
4. 「**Locking**」リンクを選択して、アプリケーションにロッキングの競合がないか確認します。 Java ワークロードが、使用可能なすべてのプロセッサーを使用できるとは限らない場合、ロッキングが原因である可能性があります。アプリケーションでのロッキングにより、実行できる並列スレッドの量が減る可能性があります。
5. 「**Garbage Collection**」リンクを選択して、ヒープ使用量とガーベッジ・コレクションを確認します。 「**Garbage Collection**」タブでは、ヒープの使用量、およびガーベッジ・コレクションを実行するために JVM が一時停止する頻度が表示されます。
   a. ガーベッジ・コレクションで費やされた時間の割合を確認します。 この情報は「Summary」セクションに表示されます。ガーベッジ・コレクションで費やされた時間が 2% を超える場合、ガーベッジ・コレクションの調整が必要な可能性があります。
   b. ガーベッジ・コレクションの一時停止時間を確認します。 一時停止時間が 10 ミリ秒を超える場合、ガーベッジ・コレクションがアプリケーションの応答時間に影響を与えている可能性があります。
   c. ガーベッジ・コレクションの比率をトランザクション数で除算して、各トランザクションが生成するおおよそのガーベッジ量を確認します。 ガーベッジの量がアプリケーションにとって多いと思われる場合は、さらにアプリケーションを調査する必要がある可能性があります。

次のタスク

アプリケーションを分析した後、Java ワークロードに合わせて Java 環境を調整することができます。

ガーベッジ・コレクションおよびヒープ拡張

ガーベッジ・コレクションおよびヒープ拡張は、JVM の操作において重要な部分を占めています。JVM におけるガーベッジ・コレクションの頻度は、JVM で実行するアプリケーションによって作成される不要情報、つまりオブジェクトの量に影響されます。

割り振り失敗

JVM がストレージ・ヒープでスペース不足になり、それ以上のオブジェクトを割り振ることができない (割り振り失敗) 場合には、ガーベッジ・コレクションがトリガーされます。ガーベッジ・コレクターは、アプリケーションによって参照されなくなったストレージ・ヒープのオブジェクトをクリーンアップし、スペースの一部を

解放します。ガーベッジ・コレクションは、ガーベッジ・コレクション・サイクルの期間に JVM で実行中の他のすべての処理を停止するため、ガーベッジ・コレクションに費やされる時間は、アプリケーションの実行には使用されていない時間ということになります。 JVM のガーベッジ・コレクション処理の詳しい説明については、世代別並行ガーベッジ・コレクターおよびIBM SDK, Java Technology Edition バージョン 8 の z/OS ユーザーズ・ガイドを参照してください。

割り振り失敗によってガーベッジ・コレクションが起動しても、十分なスペースが解放されなかった場合、ガーベッジ・コレクターはストレージ・ヒープを拡張します。ガーベッジ・コレクターは、ヒープ拡張時に、ヒープ用に予約されるストレージの最大量 (`-Xmx` オプションで指定された量) からストレージを取り、それをヒープのアクティブな部分 (`-Xms` オプションで指定されたサイズで始まる) に追加します。開始時に、`-Xmx` オプションで指定されたストレージの最大量が既に JVM に割り振られているため、ヒープ拡張によって、JVM に必要なストレージの量が増えることはありません。 `-Xms` オプションの値が、アプリケーションのヒープのアクティブな部分に十分なストレージを提供する場合、ガーベッジ・コレクターがヒープ拡張を実行する必要はありません。

JVM の存続時間中のある時点で、ガーベッジ・コレクターはストレージ・ヒープの拡張を停止します。これは、ヒープが、ガーベッジ・コレクションの頻度および処理によって解放されるスペースの量に関して、ガーベッジ・コレクターが適切であると判断する状態に達したためです。ガーベッジ・コレクターは割り振り失敗を除去するわけではありません。このため、ガーベッジ・コレクターがストレージ・ヒープの拡張を停止した後も、割り振り失敗により一部のガーベッジ・コレクションを起動することができます。パフォーマンス目標に応じて、ガーベッジ・コレクションの頻度が高すぎないかどうか考慮することができます。

ガーベッジ・コレクションのオプション

ガーベッジ・コレクションにはさまざまなポリシーを使用でき、これらのポリシーは、アプリケーションとシステム全体のスループットと、ガーベッジ・コレクションが原因の一時停止時間とのトレードオフを行います。ガーベッジ・コレクションは `-Xgcpolicy` オプションによって制御されます。

`-Xgcpolicy:optthruput`
このポリシーは、高いスループットをアプリケーションで実現しますが、その代わりに、ガーベッジ・コレクションの処理時に一時停止が生じることがあります。

`-Xgcpolicy:gencon`
このポリシーは、ガーベッジ・コレクションの一時停止で費やされる時間を最小限に抑えるのに役立ちます。このガーベッジ・コレクション・ポリシーは JVM サーバーで使用してください。 JVMSERVER リソースを調べると、JVM サーバーで使用されているポリシーを確認することができます。JVM サーバー統計にあるフィールドは、ガーベッジ・コレクションのメジャー・イベントとマイナー・イベントの数、およびガーベッジ・コレクションで費やされたプロセッサー時間を示します。

`-XX:+HeapManagementMXBeanCompatibility`
Java 8 SR5 以上を使用している場合、デフォルトではこのポリシーが設定されます。このポリシーにより、以前のレベルの Java と一貫性のあるガー

ベッジ・コレクション統計が使用されます。詳しくは、-XX:[+|-]HeapManagementMXBeanCompatibilityを参照してください。

-XX:-HeapManagementMXBeanCompatibility
このポリシーを使用することを、オプトインによって選択できます。Java 8 SR5 以上では、このポリシーを使用してデフォルトのヒープを変更できますが、場合によっては、ヒープの使用量が最大ヒープ・サイズよりも大きいことがガーベッジ・コレクション統計に示されることがあります。詳しくは、-XX:[+|-]HeapManagementMXBeanCompatibilityを参照してください。

JVM プロファイルを更新することによって、ガーベッジ・コレクション・ポリシーを変更できます。すべてのガーベッジ・コレクション・オプションについて詳しくは、IBM SDK の『ガーベッジ・コレクション・ポリシーの指定』を参照してください。

JVM サーバーのパフォーマンスの改善

JVM サーバーで実行されているアプリケーションのパフォーマンスを改善するには、ガーベッジ・コレクションやヒープ・サイズを始めとする、環境のさまざまな部分を調整することができます。

このタスクについて

CICS が提供する JVM サーバーに関する統計レポートには、タスクでスレッドを待機する時間、ヒープ・サイズ、ガーベッジ・コレクションの頻度、およびプロセッサー使用率の詳細が記載されています。また、JVM のモニターと分析を直接行って JVM サーバーを調整し、問題診断に役立つ、追加の IBM ツールを使用することもできます。統計を使用すると、JVM が効率よく動作していること、特にヒープ・サイズが適切で、ガーベッジ・コレクションが最適化されていることを確認することができます。

手順

1. JVM サーバーで使用されているプロセッサー時間の量を確認します。 ディスパッチャー統計は、T8 TCB が使用しているプロセッサー時間の量を示すことができます。JVM サーバー統計は、JVM がガーベッジ・コレクションに費やす時間、およびガーベッジ・コレクションの回数を示します。JVM ガーベッジ・コレクションは、アプリケーションの応答時間とプロセッサー使用率に悪影響を与える可能性があります。
2. CICS アドレス・スペースに使用可能なストレージ容量が十分にあることを確認します。CICS アドレス・スペースには、JVM サーバーに必要な Language Environment ヒープ・サイズが含まれます。
3. JVM におけるガーベッジ・コレクションとヒープを調整します。 ヒープが小さいと、ガーベッジ・コレクションの頻度が非常に高くなる可能性があります。一方、ヒープが大きすぎると、MVS ストレージの使用効率が悪くなるおそれがあります。 IBM Health Center を使用すると、ガーベッジ・コレクションの視覚化と調整を行い、それに応じてヒープを調整することができます。

次のタスク

メモリー使用量とヒープ・サイズをより詳しく分析するには、IBM Support Assistant の Memory Analyzer ツールを使用すると、Java プロセスのシステム・ダンプまたはヒープ・ダンプのスナップショットを使用して Java ヒープ・メモリーを分析することができます。

CICS 領域で 1 つ以上の JVM サーバーを始動するには、CICS に割り振られているストレージ容量の他に、JVM で使用できるストレージ容量が十分にあることを確認する必要があります。

JVM サーバーによるプロセッサー使用量の確認

CICS モニター機能を使用すると、JVM サーバーで実行中のトランザクションで使用されるプロセッサー時間をモニターすることができます。JVM サーバー内の CICS 対応スレッドは T8 TCB で実行されます。

このタスクについて

DFH$MOLS ユーティリティーを使用すると、SMF レコードを印刷したり、CICS Performance Analyzer などのツールを使用して SMF レコードを分析したりすることができます。

手順

1. CICS 領域でモニターをオンにして、パフォーマンス・クラスのモニター・データを収集します。
2. パフォーマンス・データ・グループ DFHTASK を確認します。 特に、次のフィールドを調べることができます。

フィールド **ID**	フィールド名	説明
283	MAXTTDLY	CICS 領域が使用可能なスレッドの限界に達したため、ユーザー・タスクが T8 TCB を取得するために待っている間に経過した時間。スレッドの限度は CICS 領域ごとに 2000 で、各 JVM サーバーは最大で 256 のスレッドを持つことができます。
400	T8CPUT	ユーザー・タスクが、CICS T8 モードの TCB 上の CICS ディスパッチャー・ドメインによってディスパッチされている間のプロセッサー時間。 T8 TCB に 1 つのスレッドが割り振られると、処理が完了するまでそのスレッドに対して同じ TCB が関連付けられた状態が続きます。
401	JVMTHDWT	CICS システムが CICS 領域内の JVM サーバーに関するスレッドの限界に達したため、ユーザー・タスクが JVM サーバー・スレッドを取得するために待っている間に経過した時間。 これは、Liberty JVM サーバーには適用されません。

3. プロセッサー使用量を改善するために、可能ならば、トレースの使用を削減または除去します。

a. 実稼働環境では、CICS マスター・システムのトレース・フラグをオフに設定して、CICS 領域を稼働させることを検討してください。 このフラグをオンに設定すると、Java プログラムの実行によってプロセッサーの消費が著しく増加します。SYSTR=OFF で CICS を初期化するか、CETR トランザクションを使用することによって、フラグをオフに設定することができます。

b. 特殊なトランザクションのみの JVM トレースをアクティブにしていることを確認します。 JVM トレースは、短時間で大量の出力を生成することがあり、プロセッサー・コストを増加させます。JVM トレースの制御について詳しくは、Java の診断を参照してください。

4. 実稼働環境で JVM プロファイルでの `USEROUTPUTCLASS` オプションを使用しないでください。 このオプションを指定すると、JVM のパフォーマンスに悪影響が出ます。 `USEROUTPUTCLASS` オプションによって、同一の CICS 領域を使用する開発者は、JVM 出力を分離し、適切な宛先に送信することができます。ただし、このためには、追加クラス・インスタンスの作成および呼び出しが必要です。

JVM サーバーのストレージ所要量の計算

CICS 領域で JVM サーバーを正常に実行するには、JVM とその配置済みのアプリケーションの両方が使用するために十分な空き MVS ストレージがあることを確認する必要があります。同じ領域で実行される CICS とその他のコンポーネントには、大量の MVS ストレージが必要になる場合があります。

このタスクについて

JVM サーバーとそのサーバー内の Java アプリケーションに必要なストレージは、DSA、EDSA、GDSA などの CICS 管理ストレージ域からは取得されません。一部のストレージ域は、C コードで実行される **`malloc()`** などの要求を処理する Language Environment で管理されます。それ以外のストレージ域は、**`IARV64`** などの z/OS ストレージ管理要求を使用して、JVM で直接管理されます。どちらのストレージ域管理タイプも、使用可能な MVS 専用領域からのストレージを使用します。24 ビット、31 ビット、64 ビットの各アドレッシング域に、十分な量の未割り振りの専用領域ストレージが存在することが重要です。専用領域ストレージが不足している場合、CICS はストレージ不足メカニズムを使用できません。

Java アプリケーションで使用されるストレージのほとんどは、Java ヒープで制御されます。その他の Java コンポーネントは MVS ストレージ域を使用します。次に例を示します。

- Java クラスのロード
- JIT コンパイル
- ネイティブ・スタック
- Java モニター
- Java スレッド

Java ヒープ以外で使用される MVS ストレージは、JVM に処理するように要求するものによって決まります。このストレージは、ロードされる Java クラスのワー

クロードと数に関係します。Java の使用量に加えて、IBM MQ、Db2、サード・パーティー・ツールなど、領域内で JVM と対話する他のコンポーネントにも注意する必要があります。

JVM サーバーを実行する場合は、必要な 31 ビットと 24 ビットのストレージ量を考慮する必要があります。これらの領域には制限があるからです。64 ビットのストレージの使用量は z/OS **MEMLIMIT** パラメーターによって制限されます。まずは故意に高い値に設定しておき、必要に応じて減らすことができます。ステップ 1 から 3 は、必要なストレージの見積もり方を示しています。ステップ 4 および 5 は、システムでサンプルの統計プログラムを実行する方法を示しています。

手順

1. 24 ビット・ストレージを計算します。

   JVM スレッドごとに、4 KB バイトの 24 ビット・ストレージが必要です。単一の JVM サーバーで 50 を超えるバックグラウンド・デーモン・スレッドを開始できます。この数に、 JVMSERVER の THREADLIMIT 属性で定義される CICS 管理 JVM サーバーのスレッド数は含まれません。Liberty JVM サーバーを使用している場合、デーモン・スレッドの数は 100 以上です。

   UNIX System Services では、新規スレッドの作成処理時に、連続する 24 ビット・ストレージが 256 KB 一時的に必要になります。24 ビットの最小要件は、以下のように計算されます。 `256KB + (4KB*number_of_threads)`

2. 31 ビット・ストレージを計算します。 共用ライブラリー領域は、UNIX System Services のダイナミック・リンク・ライブラリー (DLL) ファイルのロードのパフォーマンスを向上させるアドレス・スペースを有効にする z/OS® の機能です。共用ライブラリーを使用する最初のプロセスが領域内で開始されると、各共用ライブラリー領域が、31 ビットの高専用領域内にストレージを予約します。z/OS V2R3 を使用している場合、Java 8 SR5 (JIT コード・キャッシュでの 64 ビット・アプリケーションの常駐モード (RMODE64) をサポート) にアップグレードすることによって、31 ビット専用領域により多くの空き容量を確保できます。コンパイルされた JIT コードは 64 ビット専用領域に保管されます。詳しくは、 315 ページの『z/OS 共用ライブラリー領域の調整』を参照してください。

   デフォルトでは、CICS JVM サーバーでは Java 圧縮参照が使用されます。圧縮参照は、JVM に、小さいオブジェクトを作成するように命令します。小さいオブジェクトを使用することでパフォーマンスを改善できます。圧縮参照を使用すると、Java クラスが 31 ビット・ストレージ内の `LE HEAP31` ストレージ域にロードされます。31 ビット・ストレージ内のスペースが足りない場合は、クラスのロードが失敗し、JVM が終了します。JVM コマンド行オプション `–Xnocompressedrefs` を設定すると、圧縮参照を使用できなくなり、31 ビット・ストレージの使用量が減ります。圧縮参照を無効にすると、JVM のパフォーマンスが低下します。

   JIT コンパイラーが、Java バイト・コードをコンパイルして、継続的に最適化を行います。実行可能コードは JIT コード・キャッシュ内に保管され、静的データは JIT データ・キャッシュ内に保管されます。JIT データ・キャッシュは 64 ビット・ストレージ内に保管され、コード・キャッシュは 31 ビット・スト

レージ内に保管されます。Java アプリケーションの数と JIT アクティビティーの量によっては、JIT コード・キャッシュを、JVM 設定 `-Xcodecachetotal` で指定したサイズに拡張できます。コード・キャッシュのデフォルト・サイズは、Java 8 SR5 を使用していなければ 128 MB です (Java 8 SR5 を使用している場合のデフォルト値は 256 MB)。デフォルト値は変更することができます。

JIT コード・キャッシュの拡張時には、コード・キャッシュ用のストレージが 2 MB チャンクずつ動的に割り振られます。キャッシュが満杯になると、JIT プロセスは停止しますが、JVM は稼働し続けます。通常は、パフォーマンスが低下します。31 ビット専用領域の空きスペースを増やすには、z/OS V2R3 に移行してください。z/OS V2R3 で Java 8 SR5 以降を使用する場合、JIT コード・キャッシュはデフォルトで 64 ビット・アプリケーション用の常駐モード (**RMODE64**) を使用します。つまり、JIT コード・キャッシュは 64 ビットの専用領域内に保管されます。

3. 64 ビット・ストレージを計算します。 Java ヒープ用のストレージは、言語環境 HEAP64 領域から事前に割り振られ、ガーベッジ・コレクション・プロセスを使用して JVM で管理されます。ヒープ使用量は、JVM サーバー統計または IBM Health Center を使用してモニターできます。詳しくは、JVM サーバー・ヒープとガーベッジ・コレクションの調整を参照してください。

   JVM サーバーが使用できる 64 ビット z/OS ストレージの量は、次のように見積もります。

   - Java ヒープの最大サイズを定義する `-Xmx` 値。
   - スレッドにつき最大 20 MB が割り振られます。この総量は次のように計算されます。

     `Language Environment STACK64 最大スタック・サイズ + 4 MB`

     例えば、`STACK64(1M,1M,16M)` を使用すると、初期スタック・サイズ 1 MB、増分サイズ 1 MB、最大サイズ 16 MB が指定されます。したがって、各スレッドには最大 `16MB + 4MB = 20MB` の 64 ビット・ストレージが割り振られます。
   - 残りの 64 ビットの所要量は見積もることができません。それらには、Java クラス、JIT キャッシュ、Java スタック、および Java スレッドとモニターが含まれます。256 MB から 1 GB の間の妥当な値を追加できます。

   結果の数値は次の GB に切り上げる必要があります。切り上げるのは、CICS の GDSA 拡張時に **MEMLIMIT** が正しく認識されるようにするためです。

   ヒント: アプリケーションのワークロードが増えるに連れて、Language Environment ストレージが徐々に増えることがあります。こうした増加はストレージ不足のように見えることがありますが、ストレージの増加量と総量は、この手順の残りの部分に基づいて Language Environment HEAP64 実行時オプションの 64 ビットおよび 31 ビット・パラメーターを調整することで訂正できます。

4. サンプル統計プログラム `DFH0STAT` を実行して、MVS ストレージの見積もりに使用する値を取得します。

- **1** Private Area storage available below 16 Mb の値をメモします。これは、領域内で現在使用可能な 24 ビット専用ストレージです。
- **2** Private Area storage available above 16 Mb の値をメモします。これは、領域内で現在使用可能な 31 ビット専用ストレージです。
- **3** MEMLIMIT minus usable within Private Memory Objects の値をメモします。これは、領域内で現在使用可能な 64 ビット専用ストレージです。

```
Storage BELOW 16MB
_________________
  Private Area Region size below 16Mb . . . . . . . . . :      10,216K
    Max LSQA/SWA storage allocated below 16Mb (SYS) . :         660K
    Max User storage allocated below 16Mb (VIRT). . . :       5,460K
    System Use. . . . . . . . . . . . . . . . . . . . :          20K
    RTM . . . . . . . . . . . . . . . . . . . . . . . :         250K
  ____________________________________________________________________
  Private Area storage available below 16Mb . . . . . :       3,826K   1

Storage ABOVE 16MB
_________________
  Private Area Region size above 16Mb . . . . . . . . . :   1,417,216K
    Max LSQA/SWA storage allocated above 16Mb (SYS) . :      84,500K
    Max User storage allocated above 16Mb (EXT) . . . :     987,936K
  ____________________________________________________________________
  Private Area storage available above 16Mb . . . . . :     344,780K   2


Storage ABOVE 2GB
_________________
  MEMLIMIT Size. . . . . . . . . . . . :              200G
  MEMLIMIT Set By. . . . . . . . . . . :               JCL

  Current Address Space active (bytes) :      3,780,116,480
  Current Address Space active . . . . :             3,605M
  Peak Address Space active. . . . . . :             3,661M

  MEMLIMIT minus Current Address Space active. . . . :         201,195M
  MEMLIMIT minus usable within Private Memory Objects:         196,408M  3
  Number of Private Memory Objects . . . . . . . . . . :             728
    ....minus Current GDSA extents . . . . . . . . . . :             727
  Bytes allocated to Private Memory Objects. . . . . . :           8,392M
    ....minus Current GDSA allocated . . . . . . . . . :           7,368M
  Bytes hidden within Private Memory Objects . . . . . :           4,787M
    ....minus Current GDSA hidden. . . . . . . . . . . :           4,786M
      ....minus CICS Internal Trace Table hidden . . . :           3,794M
  Bytes usable within Private Memory Objects . . . . . :           3,605M
  Peak bytes usable within Private Memory Objects. . . :           3,681M
  Current GDSA Allocated . . . . . . . . . . . . . . . :           1,024M
  Peak GDSA Allocated. . . . . . . . . . . . . . . . . :           1,024M
```

5. JVM サーバーを開始し、代表的な Java ワークロードを実行します。 使用可能な各専用ストレージ域の値の変化を監視し、専用ストレージ域が制約を受けていないことを確認します。

JVM サーバー・ヒープとガーベッジ・コレクションの調整

JVM サーバーのガーベッジ・コレクションは、JVM によって自動的に処理されます。ガーベッジ・コレクション処理とヒープ・サイズを調整して、アプリケーションの応答時間とプロセッサー使用率が最適であることを確認できます。

このタスクについて

ガーベッジ・コレクション処理は、アプリケーションの応答時間とプロセッサー使用率に影響を与えます。ガーベッジ・コレクションは、JVM におけるすべての作業を一時的に停止するので、アプリケーションの応答時間に影響を与える場合があります。小さいヒープ・サイズを設定する場合、メモリーを節約できますが、ガーベッジ・コレクションの頻度が高くなり、ガーベッジ・コレクションで費やされるプロセッサー時間が増える可能性があります。設定するヒープ・サイズが大きすぎると、JVM は、MVS ストレージの使用効率が悪くなるので、データ・キャッシュ・ミスやページングまでもが生じる可能性があります。CICS が提供する統計を使用すると、JVM サーバーを分析することができます。また、有利なデータ分析と調整オプションの推奨を行う IBM Health Center を使用することもできます。

手順

1. 適切な間隔で JVM サーバー統計とディスパッチャー統計を収集します。JVM サーバー統計により、メジャー・ガーベッジ・コレクションとマイナー・ガーベッジ・コレクションの数、およびガーベッジ・コレクション実行のために費やされた時間を知ることができます。ディスパッチャー統計は、CICS 領域全体で T8 TCB のプロセッサー使用率に関する情報を示すことができます。
2. T8 TCB のディスパッチャー TCB モード統計を使用して、JVM サーバー・スレッドで費やされたプロセッサー時間を確認します。 「Accum CPU Time / TCB」フィールドは、この TCB モードで接続中であるか、または接続されていたすべての TCB に要したプロセッサー時間の累積を示します。「TCB attaches」フィールドは、統計間隔で使用された T8 TCB の数を示します。これらの数値を使用して、各 T8 TCB が使用したおおよそのプロセッサー時間を算出してください。
3. JVM サーバー統計を使用して、ガーベッジ・コレクションで費やされた時間のパーセンテージを検出します。 統計間隔の時間を、ガーベッジ・コレクションで費やされた経過時間で除算してください。ガーベッジ・コレクションにおけるプロセッサー使用率を 2% 未満にすることを目標にします。パーセンテージがこれより高い場合、ガーベッジ・コレクションの頻度が低くなるようにヒープのサイズを大きくすることができます。
4. 解放されたヒープの値を、その間隔内で実行されたトランザクション数で除算して、トランザクションごとに収集されている不要情報の量を確認します。 T8 TCB のディスパッチャー統計を調べると、実行されたトランザクション数を確認できます。JVM サーバー内の各スレッドは 1 つの T8 TCB を使用します。
5. オプション: verbosegc ログ・データをファイルに書き込みます。それには、パラメーター **-Xverbosegclog:***path_to_file* を指定します。このデータは、Garbage Collection and Memory Visualizer という別の ISA ツールによって分析することが可能です。 JVM は、XML のガーベッジ・コレクション・メッセージを、JVM プロファイルの STDERR オプションで指定されるファイルに書き込みます。メッセージの例と説明は、Troubleshooting and support に記載されています。

   ヒント**:** Memory Analyzer ツールのファイルを使用すると、より詳細な分析を実行することができます。

タスクの結果

調整結果は、ユーザーの Java ワークロード、CICS および IBM SDK for z/OS の保守レベル、およびその他の要因によって異なります。ストレージやガーベッジ・コレクションの設定と JVM の調整の可能性について詳しくは、Troubleshooting and support に記載されています。

IBM Health Center および Memory Analyzer は、Java でのモニターと診断のための 2 つの IBM ツールです。それらは、IBM サポート・アシスタント・ワークベンチによって提供されます。これらのツールは、IBM Support Assistant の Web サイトから無償でダウンロードできます。

JVM サーバー始動環境の調整

複数の JVM サーバーを実行している場合、JVM 始動環境を調整することによりパフォーマンスが改善されます。

このタスクについて

JVM サーバーの始動時に、サーバーでは `/usr/lpp/cicsts/cicsts55/lib` ディレクトリーに 1 セットのライブラリーがロードされる必要があります。多数の JVM サーバーを同時に始動すると、必要なライブラリーをロードするために時間がかかり、一部の JVM サーバーが、タイムアウトになったり、始動までに長時間かかったりする場合があります。 JVM サーバーの起動時間を短縮するために、JVM 開始環境を調整する必要があります。

手順

1. JVM サーバー用の共用クラス・キャッシュを作成して、ライブラリーを 1 回ロードします。 共用クラス・キャッシュを使用するには、**-Xshareclasses** オプションを各 JVM サーバーの JVM プロファイルに追加します。詳しくは、IBM SDK の『JVM 間でのクラス・データの共用』を参照してください。
2. OSGi フレームワークのタイムアウト値を増やします。 DFHOSGI.jvmprofile に含まれている `OSGI_FRAMEWORK_TIMEOUT` オプションは、CICS が JVM サーバーの始動およびシャットダウンを待機する時間の長さを指定します。値を超過した場合、JVM サーバーは正しく初期化またはシャットダウンされません。デフォルト値は 60 秒であるため、ご使用の環境に応じてこの値を増やしてください。

JVM の Language Environment エンクレーブ・ストレージ

JVM サーバーでは、(主に 64 ビット・ストレージで) 静的ストレージ要件と動的ストレージ要件の両方が適用されます。場合によっては、相当な量の 31 ビット・ストレージが使用されます。

注: 使用される 31 ビット・ストレージの量は、以下に示すいくつかの要因で決まります。

- 構成パラメーター
- 他の製品の設計および使用
- JVM の設計

- Java ワークロード

例えば、**-Xcompressedrefs** を使用するとパフォーマンスが向上する可能性があります。ただし、31 ビット・ストレージが必要になるため、必ず、**-XXnosuballoc32bitmem** と共に使用して、JVM が必要に応じて圧縮参照のために 31 ビット・ストレージを動的に割り振れるようにする必要があります。これらのオプションについて詳しくは、IBM SDK の『JVM のデフォルト設定』を参照してください。ジャストインタイム・コンパイル (JIT) でも、コンパイルされるクラス・コードのために 31 ビット・ストレージが必要になります。

JVM は、Language Environment 事前初期設定モジュール **CELQPIPI** を使用して作成される Language Environment エンクレーブで、z/OS UNIX システム・サービスのプロセスとして実行されます。

JVM のストレージ要求が Language Environment によって処理されると、定義済みのランタイム・オプションに基づいて z/OS ストレージが割り振られます。

Language Environment ランタイム・オプションは、DFHAXRO で設定されます。これらのプログラムによって JVM エンクレーブに提供されるデフォルト値を表 33 に示しています。

表 *33. CICS* で *JVM* エンクレーブに使用される *Language Environment* のランタイム・オプション

Language Environment のランタイム・オプション	**JVM** サーバーのサンプル値
ヒープ・ストレージ	`HEAP64(256M,4M,KEEP,4M,1M,FREE,1K,1K,KEEP)`
ライブラリー・ヒープ・ストレージ	`LIBHEAP64(5M,3M)`
ストレージ内のどこにでも常駐可能なライブラリー・ルーチン・スタック・フレーム	`STACK64(1M,1M,16M)`
マルチスレッド・アプリケーションに対するオプションのヒープ・ストレージ管理 (64 ビット)	`HEAPPOOLS64(ALIGN)`
マルチスレッド・アプリケーションに対するオプションのヒープ・ストレージ管理 (31 ビット)	`HEAPPOOLS(ALIGN)`
ストレージ不足の状態のために予約されたストレージ量、および割り振り時と解放時のストレージの初期内容	`STORAGE(NONE,NONE,NONE)`

注**:** 現行の JVM サーバーの値については、ライブラリー SDFHSAMP 内の DFHAXRO メンバーを参照してください。

HEAP64 などの Language Environment ランタイム・オプションは、そのタイプのストレージの初期値の原則に基づいて機能します (256 MB 64 ビットなど)。HEAP64 で新規要求が収まらない場合は、指定されたサイズ (4 MB 以上) か、または、要求のサイズに制御情報を加えたサイズのいずれか大きい方だけ増分して割り振られます。要求を満たすために、適宜、追加で増分して割り振られます。増分が空になると、Language Environment は、ランタイム値に基づいて z/OS ストレージを保持 (KEEP) または解放 (FREE) します。

Language Environment ランタイム・オプションについて詳しくは、「z/OS Language Environment カスタマイズ」を参照してください。

多少の増分は許容されますが、できる限り、31 ビットと 64 ビットの初期サイズに、31 ビットと 64 ビットの合計のストレージ要件が収まるようにしてください。そうすれば、増分が多い場合と比較して、z/OS ストレージ要件と CPU 時間の両方が全体的に削減されます。

HEAP64 31 ビットの増分サイズは 1M より小さく設定されるべきではなく、FREE オプションが使用される必要があります。前の例では、31 ビットのパラメーターは 4M、1M、および FREE に設定されています。

Language Environment の 31 ビットおよび 64 ビットのヒープ使用率は、DFHAXRO で RPTO(ON) オプションおよび RPTS(ON) オプションをアクティブにすることによって確認できます。Language Environment ストレージ・レポートは、JVM サーバーが停止されると作成されます。

Language Environment ランタイム・オプションは、サンプル・プログラム DFHAXRO を変更して再コンパイルすることでオーバーライドできます (313 ページの『DFHAXRO による JVM サーバーのエンクレーブの変更』を参照)。このプログラムは JVMSERVER リソースに対して設定するため、さまざまな名前を使用できます。これが、必要に応じて、JVM サーバーごとに別々のオプションが存在する理由です。

Language Environment エンクレーブ内で 1 つの JVM が必要とするストレージの量によっては、**REGION** および **MEMLIMIT** サイズの制限に使用するインストール・システム出口 IEALIMIT または IEFUSI の変更が必要になることがあります。すべての Java プログラム要求のルート先として、Java 所有領域 (JOR) を使用する方法が考えられます。このような領域で Java ワークロードのみを実行することにより、必要な CICS DSA ストレージの量を最小化し、最大量の MVS ストレージを JVM に割り振り可能にします。

JVM サーバーの Language Environment ストレージ必要量の特定

実際のストレージ必要量が分かれば、提供されている **DFHAXRO** オプションを変更する必要があるかどうかを判断できます。それによって、ストレージを増分割り振りする必要のない、あるいは、増分割り振りの回数を許容レベルに抑える値を選択できます。

このタスクについて

DFHAXRO の HEAP64 ランタイム・オプションは、JVM サーバーの Language Environment エンクレーブのヒープ・サイズを制御します。このオプションには、64 ビット・ストレージ、31 ビット・ストレージ、および 24 ビット・ストレージの設定が含まれています。必要に応じて、DFHAXRO の代わりに独自のプログラムを使用できます。このプログラムは JVMSERVER リソースで指定する必要があります。

手順

1. DFHAXRO に `RPTO(ON)` および `RPTS(ON)` オプションを設定します。 これらのオプションは、DFHAXRO の指定されたソースでコメント化されています。これらのオプションを指定すると、Language Environment は、ストレージ・オプションについて報告し、実際に使用されているストレージを示すストレージ・レポートを作成します。
2. JVMSERVER リソースを使用不可にします。 JVM サーバーがシャットダウンし、Language Environment エンクレーブが除去されます。
3. JVMSERVER リソースを使用可能にします。 CICS は、DFHAXRO の Language Environment ランタイム・オプションを使用して、JVM サーバーのエンクレーブを作成します。JVM も開始します。
4. JVM サーバーで Java ワークロードを実行して、Language Environment エンクレーブで使用されるストレージに関するデータを収集します。
5. DFHAXRO から `RPTO(ON)` オプションと `RPTS(ON)` オプションを除去します。
6. JVMSERVER リソースを使用不可にして、ストレージ・レポートを生成します。 このストレージ・レポートには、初期の Language Environment エンクレーブ・ヒープ・ストレージの提案が含まれています。64 ビット・ユーザー・ヒープ統計の「Suggested initial size」項目に推奨値が含まれ、JVM サーバーで使用された Language Environment エンクレーブ・ヒープ・ストレージの合計量と等しくなります。

タスクの結果

ストレージ・レポートは、z/OS UNIX の `stderr` ファイルに保存されます。また、**JOBLOG** または **DD:**// 転送構文を使用して、CICS JES 出力に送ることもできます。ディレクトリーは、JVM プロファイルで JVM の出力をリダイレクトしたかどうかによって異なります。リダイレクトが存在しない場合、このファイルは、JVM の作業ディレクトリーに保管されます。プロファイルで WORK_DIR に値が設定されていない場合、このファイルは `/tmp` ディレクトリーに保管されます。

ストレージ・レポートの情報を使用して、**DFHAXRO HEAP64** オプションの Language Environment エンクレーブ・ヒープ・ストレージに適切な値を選択してください。ストレージ要件は CICS の実行ごとに異なる場合があり、通常は、1 つの DFHAXRO を共有する個々の CICS システムごとに異なるものです。そのため、歩み寄りが必要です。

通常は、HEAP64 の初期割り振りを推奨サイズに設定することにより、増分を回避するか、その回数を減らすことを目指します。増分割り振りが行われる回数が多くなるほど、アクティブに使用されている Language Environment ストレージに対する z/OS ストレージの比率が増えます。増分割り振りが多いと、HEAP64 のストレージ要求を管理するために Language Environment が使用する CPU 時間の量も増加することになります。Java 7.0 は、Java ヒープを Language Environment の 64 ビットのヒープ・ストレージに割り振ります。その結果、推奨される初期割り振りが大きくなる可能性があります。この場合は、推奨サイズから **-Xmx** の値を引いた残りの値 (これは **-Xmx** より小さくなるはずです) を初期割り振りとして使用します。これで、Language Environment がそれ自体のメモリー・オブジェクト内に Java ヒープを Language Environment ヒープ増分として割り振るように強制されます。Java の後続のリリースは、Language Environment ストレージ要求からでは

なく、IARV64 を介して JVM ヒープをメモリー・オブジェクトとして割り振ります。 **-Xmx** を含む初期割り振りで Java マイグレーションが行われる場合は通常 Java ヒープに使用されるストレージが 2 倍になるため、結果として MEMLIMIT が小さすぎる可能性があります。

増分割り振りを何度も行うと、ストレージ不足の影響が出る可能性があります。これは、時間の経過とともに z/OS ストレージでの継続的な増加として明らかになります。実際には、これはストレージ・クリープである場合が多いものです。ストレージ・クリープでは、その特徴として z/OS で割り振られたストレージと Language Environment のフリー・ストレージの両方が増加します。ストレージ不足の場合は、z/OS で使用されているストレージと Language Environment で使用されているストレージの両方に継続的な増加が見られます。31 ビット HEAP64 のストレージは、z/OS のサブプール 1 に割り振られます。Java 7.0 では、JIT ストレージは z/OS サブプール 1 に割り振られ、2MB よりやや大きいサイズの増分で時間の経過とともに自然に増大していきます。後続の Java のリリースでは、JIT ストレージは z/OS サブプール 2 に、2MB の増分で割り振られます。

修正されたオプションの影響は、少なくとも 1 回は評価し、必要に応じて調整する必要があります。アプリケーションの変更やその他の変更によるストレージ使用量の変更の影響を評価するために、チューニングも適切な間隔で繰り返される必要があります。また、ストレージ使用量のパターンが変更される可能性があるため、CICS リリースまたは Java リリースが変更されるたびに、チューニングを繰り返す必要があります。

注: **HEAP64** の 31 ビットの初期サイズを増やす場合は、**HEAPPOOLS** の 31 ビット・ストレージの割り振りが過剰にならないように、**HEAPP** も変更する必要があります。下記の例では、HEAPPOOLS のパーセント値を 10% から 1% に減らす必要があります。

HEAPPOOLS および HEAPPOOLS64 はデフォルトの DFHAXRO でアクティブであり、構成時に有効にすることができますが、正しい値はワークロードによって異なるため、正確なチューニングは困難な場合があります。

ストレージの最大使用量が、定義されている限度 (通常は 16 MB) に迫っていないか、STACK64 を確認してください。この限度を超えると、ランタイム・エラーになります。

STACK64(1M,1M,16M) を使用することで JVM スレッド・スタックの拡張のための安全な値が割り振られるものの、その結果として、 GDSA 拡張時に CICS で 2 GB 境界より上の SOS が発生することを防ぐために MEMLIMIT の値を大きくすることが必要になる場合があるということは、LE RPTSTG 出力からはわかりません。最大値 16MB の場合は、JVM スレッドごとに 3 つのメモリー・オブジェクトで 20 MB が割り振られます (それぞれのサイズは 16+1 MB、2 MB、および 1 MB)。通常は 3 MB のみが使用可能で (17 MB が隠しストレージとして残ります)、これが z/OS による IARV64 MEMLIMIT チェックでのカウント対象になります。これに対し、CICS は、MEMLIMIT を超えずに GB の倍数単位で GDSA を拡張できるかどうか判断するために、20 MB をすべてカウントします。単一の JVM サーバーは 200 個を超えるスレッドを正当に使用でき、200 個のスレッドは CICS の MEMLIMIT チェックではおよそ 4 GB と見なされます。これはバグではありません。そのため、STACK64 の最大値を、まだある程度の拡張が可能な程度

まで小さくすると、MEMLIMIT のサイズを減らすことができ、また 2 GB 境界より上の SOS が発生する可能性も減らすことができます。STACK64 の最大値を小さくする場合は、適切なロード・テスト環境で確認してから、実稼働環境で使用することをお勧めします。

例

以下の例は、これらの DFHAXRO オプションに基づいた **RPTOPTS** 出力です。

```
HEAPPOOLS(ALIGN,8,10,32,10,128,10,256,10,1024,10,2048,10,0,10,0,10,0,10,0,10,0,10,0,10)
HEAPPOOLS64(ALIGN,8,4000,32,2000,128,700,256,350,1024,100,2048,50,3072,50,4096,50,8192,
            25,16384,10,32768,5,65536,5)
HEAP64(256M,4M,KEEP,4194304,1048576,KEEP,1024,1024,KEEP)
LIBHEAP64(3M,3M,FREE,16384,8192,FREE,8192,4096,FREE)
STACK64(1M,1M,16M)
THREADSTACK64(OFF,1M,1M,128M)
```

以下の例は、**RPTSTG** 出力の一部です。

```
STACK64 statistics:
Initial size:                                    1M
Increment size:                                  1M
Maximum used by all concurrent threads:          1M
Largest used by any thread:                      1M - no change required
Number of increments allocated:                  0
THREADSTACK64 statistics:
Initial size:                                    1M
Increment size:                                  1M
Maximum used by all concurrent threads:          0M
Largest used by any thread:                      0M - not used
Number of increments allocated:                  0
64bit User HEAP statistics:
Initial size:                                    256M
Increment size:                                  4M
Total heap storage used:                         730857472
Suggested initial size:                          697M - use this
Successful Get Heap requests:                    783546
Successful Free Heap requests:                   780785
Number of segments allocated:                    135 - too many increments
Number of segments freed:                        0
31bit User HEAP statistics:
Initial size:                                    4194304
Increment size:                                  1048576
Totalheap storage used (suggested initial size):  137165672 - use this
Successful Get Heap requests:                    1345332
Successful Free Heap requests:                   1345260
Number of segments allocated:                    125 - too many increments
Number of segments freed:                        0
64bit Library HEAP statistics:
Initial size:                                    3M
Increment size:                                  3M
Total heap storage used:                         4640032
Suggested initial size:                          5M
Successful Get Heap requests:                    113381
Successful Free Heap requests:                   112860
Number of segments allocated:                    1 - low, so no change required
Number of segments freed:                        0
31bit Library HEAP statistics:
Initial size:                                    16384
Increment size:                                  8192
Total heap storage used (suggested initial size): 520
Successful Get Heap requests:                    33725
Successful Free Heap requests:                   33725
Number of segments allocated:                    1 - low, so no change required
```

```
Number of segments freed:                        0

Suggested Percentages for current CellSizes:
HEAPP(ALIGN,8,1,32,1,128,1,256,1,1024,1,2048,1,0)
```

RPTSTG 出力を確認する際には、**HEAP64** の増分サイズは Language Environment が割り振る最小のストレージ量であって、実際の増分はこの値よりもかなり大きい可能性があることに注意してください。したがって、1 回以上の増分割り振りが行われた場合に使用された z/OS ストレージの量を正確に算定することは不可能です。64 ビット HEAP には実際の増分割り振りの回数 (つまり 135) が報告されていますが、31 ビット **HEAP** については、レポートに示されている値より 1 つ少ない値 (つまり 125 ではなく 124) が実際の増分の回数です。

増分が行われるときの Language Environment のストレージ管理の仕組みが原因で、割り振られる 31 ビットと 64 ビットの z/OS ストレージの量は、**RPTSTG** の「maximum used (最大使用量)」に示されている値よりもかなり大きくなる可能性があります。

推奨されている **DFHAXRO** の変更は、以下のとおりです。

```
* Heap storage
DC     C'HEAP64(700M,'         Initial 64bit heap - change (Note 1)
DC     C'4M,'                   64bit Heap increment
DC     C'KEEP,'                 64bit Increments kept
DC     C'128M,'                 Initial 31bit heap - change (Note 2)
DC     C'2M,'                   31bit Heap increment - change (Note 3)
DC     C'FREE,'                 31bit Increments freed - change (Note 4)
DC     C'1K,'                   Initial 24bit heap
DC     C'1K,'                   24bit Heap increment
DC     C'KEEP) '                24bit Increments kept

* Heap  pools
DC     C'HP64(ALIGN) '
DC     C'HEAPP(ALIGN,8,1,32,1,128,1,256,1,1024,1,2048,1,0) ' - change (Note 5)

* Library Heap storage
DC     C'LIBHEAP64(3M,3M) '     Initial 64bit heap - do not change (Note 6)

* 64bit stack storage
DC     C'STACK64(1M,1M,16M) ' - consider a change (Note 7)
```

注:

1. **RPTSTG** 出力の 64 ビットの「Suggested initial size (推奨初期サイズ)」に示されている値と、若干の増加分。
2. **RPTSTG** 出力の 31 ビット「Suggested initial size (推奨初期サイズ)」に示されているとおり。ただし、FREE を使用しているので多少減少しています。
3. 31 ビット **HEAP** 増分は、1 M ではなく 2 M の値にするのが適切です。
4. 必要に応じて、31 ビット に**HEAP FREE** を使用すると、KEEP を指定した場合に比べて、「Total heap storage used (使用済み合計ヒープ・ストレージ)」に対応する z/OS ストレージの割り振り量が減少する可能性があります。
5. **RPTSTG** 出力で **HEAPPOOLS** 統計の後に推奨内容が示されていますが、さらに最適化すると効果がある可能性があります。31 ビット・ヒープの初期サイズ 128 MB の 10% (デフォルト値) だと、割り振られるストレージの量が過剰になる可能性があります。 最低 6 つのプールがそれぞれ 128 MB の初期ヒープ・サイズの 10% を取ると、77 MB が割り振られることになります。プールの何パーセントが生産的に使用されているかに関わりなく、この割り振り分は「Total

heap storage used (使用済み合計ヒープ・ストレージ)」値に含まれます (なぜなら、**HEAPPOOLS** ストレージ・エクステントがそこに割り振られるからです)。256 バイトより大きい **HEAPPOOLS** セル・サイズを使用すると、Language Environment ヒープ・ストレージの使用効率が悪くなる可能性があります。

6. 増分は 1 回しか必要ありませんでした。これは問題ではありません。
7. 使用された最大サイズは 1MB でした。最大値 16MB を 8 MB やそれより小さい値に減らすと、新しい GDSA 範囲を割り振ることができるかどうか確認する際に、CICS が **MEMLIMIT** までの量の中にカウントする STACK64 ストレージの量が大幅に減少します。 STACK64 の変更は、実稼働環境に移行する前に十分にテストする必要があります。

以下に、もう一度同じ JVM サーバーで 31 ビット **HEAP FREE** を使用して実行した例を示します。「Number of segments (セグメント数)」は実行された **GETMAIN** と **FREEMAIN** の数です。JVM サーバーがアクティブであったとき、これは低い値を示していました。差が 2 であるということは、エンクレーブが初期割り振りと増分 1 回のみで終了したことを示しています。この量は「Total heap storage (合計ヒープ・ストレージ)」よりも少ない可能性があり、FREE が有効であったことを示しています。「Total heap storage used (使用済み合計ヒープ・ストレージ)」はより大きな値でした。しかし、JVM サーバーを実行するごとに合計は変化するので、1 つの **RPTSTG** だけに基づいて変更しても、ベストな設定が得られるとは限りません。

```
31bit User HEAP statistics:
Initial size: 134217728
Increment size: 2097152
Total heap storage used (suggested initial size): 154056664
Successful Get Heap requests: 3253239
Successful Free Heap requests: 3253176
Number of segments allocated: 149
Number of segments freed: 147
```

RPTSTG 出力を正しく解釈するには、「Language Environment デバッグ・ガイド」を読むことが重要です。

DFHAXRO による JVM サーバーのエンクレーブの変更

DFHAXRO は、JVM サーバーが実行される Language Environment エンクレーブにデフォルトの実行時オプション・セットを提供するサンプル・プログラムです。例えば、JVM ヒープとスタックにストレージ割り振りパラメーターを定義します。すべてのワークロード用に最適化されたデフォルトのランタイム・オプションを提供することは不可能です。実際のストレージ使用量を調べ、必要に応じてデフォルト値をオーバーライドして、使用されるストレージと割り振られるストレージの比率を最適化することを検討してください。

このタスクについて

このサンプル・プログラムを更新して Language Environment エンクレーブを調整するか、サンプルに基づいて独自のプログラムを作成することができます。このプログラムは JVMSERVER リソースで定義され、JVM サーバー用に作成される Language Environment エンクレーブの CELQPIPI 事前初期設定段階時に呼び出されます。

アセンブリー言語でこのプログラムを作成する必要があり、CICS 変換プログラムでこれを変換してはなりません。オプションは文字ストリングとして指定され、2 バ

イトのストリングの長さと、それに続く実行時オプションで構成されます。すべての Language Environment 実行時オプションの最大長は 255 バイトです。したがって、各オプションの省略バージョンを使用し、変更内容を全体で 200 バイト未満に制限してください。

手順

1. DFHAXRO プログラムを新規ロケーションにコピーして、実行時オプションを編集します。 CICS 領域に保守が適用される場合、プログラムの変更を反映したい場合があります。DFHAXRO のソースは、CICSTS55.CICS.SDFHSAMP ライブラリーにあります。
2. 各オプションの省略形を使用して、実行時オプションを編集します。 「z/OS Language Environment プログラミング・ガイド」に、Language Environment 実行時オプションに関する詳細情報が記載されています。 CICS はいくつかのオプションをこのリストに追加するため、高速処理のためにはオプションのリストのサイズを最小限に保つ必要があります。
   - HEAP64 オプションを使用して、64 ビット、31 ビット、および 24 ビット・ストレージの Language Environment ヒープ領域の初期のヒープ割り振りを指定します。

     例えば次の HEAP64 設定 `HEAP64(256M,4M,KEEP,4M,1M,FREE,1K,1K,KEEP)` では、64 ビット・ヒープの初期サイズは 256 MB で getmained ストレージが 4 MB ずつ増加し、31 ビット・ヒープの初期ストレージ・サイズは 4 MB で getmained が 1 MB ずつ増加し、 24 ビット・ヒープの初期ストレージ・サイズは 1 KB で getmained が 1 KB ずつ増加するようになります。
   - ALL31 オプション、POSIX オプション、および XPLINK オプションは、CICS によって強制的にオンにされます。 ABTERMENC オプションは、CICS によって (ABEND) に設定され、TRAP オプションは (ON,NOSPIE) に設定されます。
   - RPTO(ON) および RPTS(ON) の値を使用して、LE オプションおよび LE ストレージ使用量についてのレポートを生成します。生成される出力は、それぞれの JVM 終了時に JVM サーバーの `stderr` ストリームに書き込まれます。
3. DFHASMVS プロシージャーを使用して、プログラムをコンパイルし、RPL に配置し、JVM サーバーを再始動します。
4. 生成された LE ストレージ・レポートを分析して、いずれかの LE ユーザー・ヒープが高い数値の増分で割り振られていないかを調べます。 増分が多いと、通常は、ストレージ要求のための CPU 時間が増えます。割り振られるセグメントが多ければ多いほど、ストレージのフラグメント化が発生する確率は高くなります。
5. いずれかのユーザー・ヒープ統計で 20 を超えるセグメントが割り振られていた場合は、関連する LE 実行時オプションで初期サイズまたは増分サイズを増やしてください。
6. チューニング調整が完了したら、実行時オプションを編集して LE オプションおよび LE ストレージ・レポートのレポート作成を無効にします。プログラムをコンパイルし、RPL に配置し、JVM サーバーを再始動します。 LE ユーザー・ヒープ統計の LE ストレージ・レポート例

```
64bit User HEAP statistics:
  Initial size:                                          256M
  Increment size:                                          4M
  Total heap storage used:                            47842496
  Suggested initial size:                                 46M
  Successful Get Heap requests:                         8214
  Successful Free Heap requests:                        8052
  Number of segments allocated:                            0
  Number of segments freed:                                0
31bit User HEAP statistics:
  Initial size:                                      4194304
  Increment size:                                    1048576
  Total heap storage used (sugg. initial size):      8583912
  Successful Get Heap requests:                          338
  Successful Free Heap requests:                          69
  Number of segments allocated:                            6
  Number of segments freed:                                0
24bit User HEAP statistics:
  Initial size:                                         1024
  Increment size:                                       1024
  Total heap storage used (sugg. initial size):            0
  Successful Get Heap requests:                            0
  Successful Free Heap requests:                           0
  Number of segments allocated:                            0
  Number of segments freed:                                0
```

タスクの結果

JVMSERVER リソースを使用可能にすると、CICS は DFHAXRO プログラムで指定された実行時オプションを使って Language Environment エンクレーブを作成します。 CICS は実行時オプションを Language Environment に渡す前に、それらの長さを検査します。この長さが 255 バイトより長い場合、CICS は JVM サーバーの始動を試みず、CSMT にエラー・メッセージを書き込みます。指定した値は、Language Environment に渡される前に CICS によって検査されません。

z/OS 共用ライブラリー領域の調整

共用ライブラリー領域は、UNIX System Services のダイナミック・リンク・ライブラリー (DLL) ファイルのロードのパフォーマンスを向上させるアドレス・スペースを有効にする z/OS® の機能です。

共用ライブラリー機能を使用すると、JVM で使用されるネイティブ・ライブラリーのロードに必要なストレージを複数の CICS® 領域で共用できるので、JVM ごとに個別に各ライブラリーをロードする必要がなくなります。それらの実行可能プログラムは、LPA のように、それぞれメガバイト境界上で割り振られ、単一ページ・テーブルの共用を許可されます。1 つの領域に多数の JVM があり、そのすべてが共用ライブラリー域を使用する場合は、各領域に大量の専用ストレージを割り振ることができます。共用ライブラリー領域を使用しない場合は、割り振るストレージを減らします。共用ライブラリー領域へのマッピングを確立するためにユーザー・アドレス・スペース内で使用されるストレージは、専用ストレージの最高位部分から取られます。詳細については、高専用領域を参照してください。

共用ライブラリー領域用に予約されているストレージ全体は、共用ライブラリーを使用する最初の処理が領域で開始されるときにそれぞれの CICS 領域に割り振られます。割り振られるストレージの量は、z/OS の **SHRLIBRGNSIZE** パラメーターによって制御されます。このパラメーターは、SYS1.PARMLIB の BPXPRMxx メンバー内に

あります。最小値は 16 MB で、z/OS のデフォルトは 64 MB です。必要なスペースの量を調査することによって、共用ライブラリー領域用に割り振られるストレージの量を調整できます。CICS 以外の他のアプリケーションが、その共用ライブラリー領域を使用しており、それに応じて **SHRLIBRGNSIZE** パラメーターを調整している可能性があることに注意してください。

共用ライブラリー領域用に割り振られるストレージの量を削減する場合は、最初に共用ライブラリー領域に無駄なスペースがないことを確認します。z/OS システムで通常の作業負荷をかけ、コマンド `D OMVS,L` を発行してライブラリー統計を表示します。共用ライブラリー領域に未使用のスペースがある場合は、**SHRLIBRGNSIZE** の設定を減らしてこのスペースを除去できます。

共用ライブラリー領域によるストレージの使用量を削減する必要がある場合、以下の 2 つの手順が検討できます。

1. CICS 領域で実行されているすべての USS プロセス (JVM サーバーおよび Node.js アプリケーションを含む) に変数 `_BPXK_DISABLE_SHLIB=YES` を設定します。この変数を設定することにより、ご使用の CICS 領域は共用ライブラリーをまったく使用しなくなります。また、CICS 領域内の共用ライブラリーにストレージが割り振られなくなります。この方法を選択した場合は、共用ライブラリー領域の共用機能を利用することはできません。この一連のアクションは、CICS 領域の 31 ビット・ストレージに保管する必要があり、使用している CICS 領域で実行されている JVM サーバーや Node.js アプリケーションが少ない場合には、最も適しています。
2. 共用ライブラリー領域サイズをファイルで必要なストレージの量より小さく設定することは可能です。共用ライブラリー領域がフルの場合には、代わりにファイルは専用ストレージにロードされ、共用機能の利点は得られません。この方法を選択した場合は、より重要なアプリケーションを先に再始動して、共用ライブラリー領域を確実に利用できるようにする必要があります。

z/OS UNIX の共用ライブラリーについて詳しくは、z/OS UNIX システム・サービス の z/OS UNIX System Services Web サイトを参照してください。

第 8 章 Java アプリケーションのトラブルシューティング

Java アプリケーションに問題がある場合は、CICS と JVM で提供される診断機能を使用して、問題の原因を調べることができます。

このタスクについて

CICS は、Java に関連した問題の診断に役立ついくつかの統計、メッセージ、およびトレースを提供します。Java で提供される診断ツールおよびインターフェースは、JVM で何が発生しているかについて CICS よりも詳しい情報を提供できます。これは、CICS が JVM 内のアクティビティーの多くを認識しないからです。

JVM の分析をリアルタイムおよびオフラインで実行する無料のツール (例えば、IBM Health Center) を使用できます。詳細については、IBM Monitoring and Diagnostic Tools for Java - Health Centerを参照してください。

Liberty JVM サーバーで実行している Web アプリケーションのトラブルシューティングについては、 323 ページの『Liberty JVM サーバーおよび Java Web アプリケーションのトラブルシューティング』を参照してください。ログ・ファイルが存在する場所については、 333 ページの『JVM 出力、ログ、ダンプ、およびトレースの場所の制御』を参照してください。

手順

1. JVM サーバーを始動できない場合は、Java インストールのセットアップが正しいことを確認してください。 CICS メッセージおよび JVM の `stderr` ファイルにあるエラーを使用して、問題の原因を判別してください。
   a. 正しいバージョンの Java SDK がインストールされていることと、CICS が z/OS UNIX 内の Java SDK にアクセスできることを確認してください。サポートされる SDK のリストについては、アプリケーション・プログラミング言語に関する CICS サポートの変更点を参照してください。
   b. **USSHOME** システム初期設定パラメーターが CICS 領域で設定されていることを確認します。 このパラメーターは、z/OS UNIX 上のファイルのホームを指定します。
   c. **JVMPROFILEDIR** システム初期設定パラメーターが CICS 領域で正しく設定されていることを確認します。 このパラメーターは、z/OS UNIX 上の JVM プロファイルの場所を指定します。
   d. JVM プロファイルが入っている z/OS UNIX ディレクトリーへの読み取りアクセス権限と実行アクセス権限が CICS 領域にあることを確認します。
   e. JVM の作業ディレクトリーへの書き込みアクセス権限が CICS 領域にあることを確認します。 このディレクトリーは、JVM プロファイルの `WORK_DIR` オプションで指定されます。
   f. JVM プロファイルの `JAVA_HOME` オプションが、Java SDK が入っているディレクトリーを指し示していることを確認します。

© Copyright IBM Corp. 1999, 2018

g. IBM MQ または Db2 の DLL ファイルを使用している場合は、これらのファイルの 64 ビット・バージョンが CICS から使用可能であることを確認してください。

h. Language Environment エンクレーブを構成するために DFHAXRO を変更する場合は、実行時オプションが 200 バイトを超えないことと、それらのオプションが有効であることを確認してください。 CICS は、指定されたオプションを Language Environment に渡す前に、それらを検証しません。 Language Environment からのエラー・メッセージがないかどうか、SYSOUT を確認してください。

2. セットアップが正しい場合は、診断情報を収集して、アプリケーションと JVM に何が起きているかを調べます。

   a. 診断を得るには、`PRINT_JVM_OPTIONS=TRUE` を使用する必要があります。このオプションのデフォルトは `PRINT_JVM_OPTIONS=FALSE` であるため、デフォルトのままにすると診断のオプションが表示されません。`PRINT_JVM_OPTIONS=TRUE` を指定すると、クラスパスの内容を始めとして、開始時に JVM に渡されるすべてのオプションが SYSPRINT に出力されます。JVM がプロファイル内でこのオプションを指定して開始されるたびに、情報が生成されます。

   b. JVM からの情報およびエラー・メッセージがないか、`dfhjvmout` ファイルと `dfhjvmerr` ファイルを調べます。 これらのファイルは、JVM プロファイルの `WORK_DIR/applid/jvmserver` オプションで指定されるディレクトリーにあります。JVM プロファイルで `STDOUT` オプションと `STDERR` オプションが変更された場合、これらのファイルの名前が異なる可能性があります。

3. アプリケーションが障害を起こしたか、アプリケーションのパフォーマンスが低下する場合は、アプリケーションをデバッグします。

   - `java.lang.ClassNotFoundException` エラーを受け取り、トランザクションが AJ05 コードで異常終了する場合は、アプリケーションが OSGi フレームワークの IBM またはベンダーのクラスにアクセスできない可能性があります。この問題の修正方法について詳しくは、Java 環境のアップグレードを参照してください。
   - CEDX トランザクションを使用して、アプリケーション・トランザクションをデバッグします。Liberty JVM サーバーの場合、URI マップを使用してインバウンド・アプリケーション要求をアプリケーション・トランザクションに対応させているとしたら、そのトランザクションをデバッグします。デフォルトのトランザクション CJSA を使用する場合、DFHEDFTC トランザクション・クラス (CEDY を使用する場合は DFHEDFTO トランザクション・クラス) で MAXACTIVE 属性を 1 に設定する必要があります。この設定が必要となる理由は、多数の CJSA タスクが実行されている場合があり、誤ったトランザクションをデバッグする可能性があるためです。実稼働環境では、CJSA トランザクションで CEDX を使用しないでください。
   - JVM サーバーでデバッガーを使用するには、JVM プロファイルでいくつかのオプションを設定する必要があります。詳しくは、 339 ページの『Java アプリケーションのデバッグ』を参照してください。
   - OSGi バンドルおよびサービスの状態を判別する場合は、OSGi コンソールを使用します。JVM プロファイルに次のプロパティーを設定します。

-Dosgi.file.encoding=ISO-8859-1、および -Dosgi.console=host:port (host は JVM サーバーが稼働しているシステムのホスト名で、port は同じシステムの空きポートです)。 osgi.console.encoding プロパティーは、OSGi コンソールで任意のエンコードを、そのエンコードに JVM 全体を含めないでも使用できるようにするために設計されたものですが、これは Equinox OSGi フレームワークの未解決のバグのために使用できないため、代わりに ASCII ベースのエンコードに file.encoding 値を設定する必要があります。OSGi JVM サーバーを使用している場合は、**OSGI_CONSOLE=TRUE** を JVM プロファイルに追加します。Liberty JVM サーバーを使用している場合は、osgiConsole-1.0 フィーチャーを server.xml に追加します。JVM プロファイルに指定したホストおよびポートのプロパティーを指定して Telnet セッションを使用することで、OSGi コンソールに接続します。

注: OSGi コンソールで exit コマンドを入力すると、JVMSERVER が動作する環境に system.exit(0) 呼び出しを実行します。 OSGi コンソールから端末を切断するコマンドは disconnect です。 system.exit(0) を使用すると、すべてのスレッドとワークロードが即時に停止され、そのまま処理を継続すれば、JVM と CICS が不確実な状態になります。このような場合、それ以降の複雑さを回避するために即時シャットダウンを実行するように、CICS は設計されています。このため、JVM プロファイルと server.xml の両方の書き込み権限を制御することが重要です。Liberty JVM サーバーは、さらに保護を提供するために、OSGi コンソールが実行可能になる前に、osgiConsole-1.0 機能を含めることを要求します。OSGi コンソールは、主に開発とデバッグの補助機能であり、実稼働環境での実行は想定されていません。

4. メモリー不足エラーが表示される場合は、JVM または CICS アドレス・スペースに十分なストレージが割り振られなかったか、アプリケーションでメモリー・リークが発生したか、またはヒープ・サイズが不十分であった可能性を示しています。
   a. CICS 統計またはツール (IBM Health Center など) を使用して、JVM をモニターします。アプリケーションにメモリー・リークがある場合、ガーベッジ・コレクション後に残っているライブ・データの量が、ヒープが使い果たされるまで時間と共に徐々に増えます。 JVM サーバー統計は、最後のガーベッジ・コレクション後のヒープのサイズ、およびヒープの最大サイズとピーク・サイズを報告します。詳しくは、IBM Health Center を使用した Java アプリケーションの分析を参照してください。
   b. Language Environment のストレージ・レポートを実行して、ストレージの量が十分かどうかを確認してください。 詳しくは、JVM 用 Language Environment エンクレーブ・ストレージを参照してください。
5. Java アプリケーションのインストール時や実行時にエンコード・エラーが表示される場合、セットアップしたコード・ページの組み合わせが競合しているか、サポートされていない可能性があります。 z/OS 上の JVM は通常、ファイルのエンコードに EBCDIC コード・ページを使用します。非 Liberty JVM サーバーのデフォルトは IBM1047 (または cp1047) ですが、JVM では必要に応じて他のコード・ページをファイルのエンコードに使用できます。 CICS では文字データの処理に EBCDIC コード・ページが必要で、すべての JCICS 呼び出

しも EBCDIC コード・ページを使用しなければなりません。コード・ページは、CICS 領域に関する **LOCALCCSID** システム初期設定パラメーターで設定されます。

a. JVM サーバー・ログを調べて、**LOCALCCSID** の値に関連した警告メッセージが発行されたかどうかを確認します。 このパラメーターが EBCDIC でないコード・ページ、JVM でサポートされていないコード・ページ、またはサポートされていない EBCDIC コード・ページ (930 など) に設定されている場合は、JVM サーバーは cp1047 を使用します。

b. JCICS 呼び出しは、**LOCALCCSID** システム初期設定パラメーターで指定されているコード・ページを使用します。 アプリケーションが別のコード・ページを想定している場合は、エンコード・エラーが表示されます。 JCICS に別のコード・ページを使用するには、JVM プロファイル内で **-Dcom.cics.jvmserver.override.ccsid=** パラメーターを設定します。

c. JVM プロファイル内で **-Dcom.cics.jvmserver.override.ccsid=** パラメーターを使用している場合は、CCSID が EBCDIC コード・ページであることを確認してください。 JCICS 呼び出しの使用時には、アプリケーションは EBCDIC を使用しなければなりません。

d. Axis2 JVM サーバー内で SOAP 処理を実行している場合は、**-Dfile.encoding** JVM プロパティーで EBCDIC プロパティーが指定されていることを確認してください。 UTF-8 などの EBCDIC でないコード・ページを指定すると、Web サービス要求は失敗し、応答には壊れたデータが含まれます。

6. 始動時のタイムアウトまたはワークロード下のタイムアウトが発生する場合、問題の解決に役立つ、調整可能なさまざまなパラメーターがあります。調整可能な値には、以下のようなものがあります。

   - HTTP 要求で JVM サーバー・スレッドを取得するために待機する時間を制御するために、**-Dcom.ibm.cics.jvmserver.threadjoin.timeout** 設定を変更します。
   - JVMSERVER リソースの **THREADLIMIT** 値を増やしてください。
   - **THREADLIMIT** が既に最大許可値に設定されている場合、単一の JVM サーバーが処理可能な作業量を超えて実行を試みている可能性があります。複数の JVM サーバーまたは複数の領域間のワークロード・バランシングを検討してください。

     または、他の制約が原因で CICS システムが応答しなくなっている可能性もあります。パフォーマンス上の問題を診断するための標準的な手順に従ってください。 CICS システムのパフォーマンスの向上を参照してください。

次のタスク

問題の原因を確定できない場合は、IBM サポートにお問い合わせください。 Java の問題を報告するための MustGather にリストされているとおりに、必要な情報を確実に提供してください。

Java の診断

通常の CICS 診断情報源の多くには、Java アプリケーションに適用される情報が含まれています。CICS 提供の情報に加えて、問題判別に使用できる、JVM 固有の複数のインターフェースがあります。

Java の CICS 診断ツール

CICS には、実行中の Java アプリケーションに関して収集できる統計とモニター・データがあります。エラーが発生すると、トランザクションは異常終了し、メッセージが該当するログに書き込まれます。JVM (SJ) ドメインに適用される異常終了とメッセージのリストについては、CICS メッセージ を参照してください。 Java に関連したメッセージの形式は DFHSJxxxx です。

また、トレースをオンにして、追加の診断情報を生成することもできます。 JVM ドメインのトレース・ポイントは、JVM および Node.js ランタイム・ドメインのトレース・ポイント にリストされています。

初期化後に最初の JVM が CICS 領域で開始すると、CICS は、メッセージ DFHSJ0207 を発行して、使用されている Java のバージョンを表示します。

Java SDK が提供する診断ツールとインターフェースは、JVM で何が発生しているかについてより詳細な情報を提供します。 JVM からのメッセージと診断情報は、JVM の `stderr` ログ・ファイルに書き込まれます。Java の問題を検出した場合は、必ずこのファイルを調べてください。例えば、CICS がメッセージを発行して、JVM が異常終了したことを示す場合、`stderr` ログ・ファイルが第一の診断情報源です。 333 ページの『JVM 出力、ログ、ダンプ、およびトレースの場所の制御』では、JVM からの出力の場所を制御する方法、および JVM 内部からのメッセージならびに JVM で実行中の Java アプリケーションからの出力のリダイレクト方法を示しています。

CICS 用の Java アプリケーションを開発する際には、CICS におけるスレッド・セーフティーとトランザクション分離の要件を考慮することが重要です。 Java アプリケーションが、最初に使用されるときに正しく機能するものの、それ以降の使用時に正しく動作しない場合、この問題はおそらく、分離の問題が原因です。

OSGi 診断ファイル

OSGi フレームワークは、JVM サーバーにおける OSGi バンドルおよびサービスの問題のトラブルシューティングに使用できる診断ファイルを zFS で作成します。

OSGi キャッシュ
: OSGi キャッシュは、JVM サーバーの *$WORK_DIR/applid/jvmserver/*`configuration/org.eclipse.osgi` ディレクトリーにあります。*$WORK_DIR* は JVM サーバーの作業ディレクトリー、*applid* は CICS APPLID、*jvmserver* は JVMSERVER リソースの名前です。OSGi キャッシュには、フレームワークのメタデータ、およびフレームワークの実行に必要なその他の情報が入っています。JVM サーバーが始動するときに、キャッシュが置き換えられます。

OSGi ログ
: OSGi フレームワークでエラーが発生すると、OSGi ログが、JVM サーバ

ーの *$WORK_DIR/applid/jvmserver/*configuration/ ディレクトリーに作成されます。ファイル拡張子は .log です。

JVM 診断ツール

CICS 資料には、一部の Java 診断ツールとインターフェースに関する情報が記載されています。

- 338 ページの『JVM サーバーのトレースの活動化と管理』では、CETR トランザクションによって提供されるコンポーネント・トレースを使用して、JVM サーバーと JVM サーバー内で実行されるタスクのライフサイクルをトレースする方法を説明しています。JVM サーバーは補助トレースも GTF トレースも使用しません。代わりに、JVM サーバーごとに固有に名前が指定される zFS 上のファイルにトレースが書き込まれます。
- 339 ページの『Java アプリケーションのデバッグ』 では、リモート・デバッガーを使用して、JVM で実行されている Java アプリケーションのアプリケーション・コードをステップスルーする方法について説明しています。また、CICS は CICS Java ミドルウェアに一連の代行受信ポイント (すなわちプラグイン) を用意しているため、デバッグ、ロギング、またはその他の目的のための追加の Java プログラムを、アプリケーション Java コードの実行直前および直後に挿入できます。詳しくは、 341 ページの『CICS JVM プラグイン・メカニズム』を参照してください。

JVM にはより多くの診断ツールとインターフェースが使用できます。JVM の問題判別に使用できる追加の機能については、Troubleshooting and support を参照してください。次の機能が、役に立つ診断情報を提供します。

- CICS が提供するインターフェースを使用することなく、JVM の内部トレース機能を直接使用できます。内部トレース機能の制御と、各種宛先への JVM トレース情報の出力に使用できるシステム・プロパティーに関する情報は、CICS トレースの使用を参照してください。これらのシステム・プロパティーを使用すると、JVM 内の任意のメソッドまたはクラスからトレースを出力し、メソッド呼び出しで任意のパラメーターと戻りの型の値を検出することができます。
- JVM にメモリー・リークが検出される場合、JVM にヒープ・ダンプを要求できます。ヒープ・ダンプは、JVM のヒープ内にあるすべてのライブ・オブジェクト (引き続き使用中のオブジェクト) のダンプを生成します。また、IBM Health Center や Memory Analyzer ツールを使用して、メモリー・リークを分析することもできます。これらのツールはどちらも、IBM Support Assistant で入手可能です。Java ツールについて詳しくは、IBM Monitoring and Diagnostic Tools for Java - Health Centerを参照してください。
- IBM 64-bit SDK for z/OS, Java テクノロジー・エディション に付属の HPROF プロファイラーは、JVM で実行されるアプリケーションのパフォーマンス情報を提供します。したがって、プログラムのどの部分が最大のメモリーまたはプロセッサー時間を使用しているかが分かります。
- JVM は、モニター、プロファイル作成、および RAS (信頼性・可用性・保守性) 用のインターフェースを提供します。

CICS 環境に固有ではなく、IBM JVM に使用可能なすべてのインターフェース、オプション、またはシステム・プロパティーでは、IBM JVM 資料を第一の情報源として使用してください。

Liberty JVM サーバーおよび Java Web アプリケーションのトラブルシューティング

Java Web アプリケーションに問題がある場合、CICS および Liberty で提供される診断を使用して、問題の原因を判別できます。

CICS は、Liberty JVM サーバーで実行中の Java Web アプリケーションに関連した問題の診断に役立つ統計、メッセージ、およびトレースを提供します。Liberty は、zFS で使用可能な診断も作成します。一般的なセットアップ・エラーおよびアプリケーションの問題については、Troubleshooting and supportを参照してください。

問題の回避

CICS は、領域 APPLID と JVMSERVER リソース名の値を使用して、固有の zFS ファイル名とディレクトリー名を作成します。許容文字の一部は、UNIX システム・サービス・シェルで特殊な意味を持ちます。例えば、ドル記号 ($) は環境変数名の先頭を意味します。これらの文字の一部を使用すると、Equinox OSGi フレームワーク内で `Exception` が発生し、JVM サーバーが開始できない可能性があります。領域 APPLID と JVM サーバー名では、非英数字は使用しないでください。UNIX システム・サービス・シェルでこれらの文字を使用する場合、エスケープ文字として円記号 (¥) を使用しなければならない場合があります。例えば、JVM サーバー MY$JVMS を呼び出し、JVM システム出力ファイルを読み取る必要がある場合は、次のようにします。

```
cat CICSPRD.MY\$JVMS.D20140319.T124122.dfhjvmout
```

Liberty JVM サーバーを始動できない

1. Liberty JVM サーバーを始動できない場合は、セットアップが正しいことを確認します。詳しくは、Liberty JVM サーバーの構成を参照してください。問題の原因として考えられる要因を判別するには、`WLP_OUTPUT_DIR` の後にある CICS システム・ログおよび Liberty `messages.log` ファイルのメッセージを使用します。
2. JVM プロファイルの中の **-Dfile.encoding** JVM プロパティーが、ISO-8859-1 または UTF-8 のどちらかを指定していることを確認します。これらの 2 つのコード・ページは、Liberty でサポートされています。それ以外の値に設定すると、JVM サーバーは始動できません。

CICS Liberty JVM サーバー内の保護された Web アプリケーションにアクセスしようとするとき、ユーザーを認証できない

CICS **JESMSGLG** ログに以下のメッセージが入っています。

```
ICH420I PROGRAM DFHSIP FROM LIBRARY hlq.SDFHAUTH CAUSED THE
ENVIRONMENT TO BECOME UNCONTROLLED
BPXP014I ENVIRONMENT MUST BE CONTROLLED FOR DAEMON (BPX.DAEMON) PROCESSING.
```

Liberty `messages.log` には、以下のメッセージが入っています。

```
CWWKS1100A: Authentication did not succeed for user ID user.
(ユーザー ID user の認証に失敗しました。)
An invalid user ID or password was specified.
(無効なユーザー ID またはパスワードが指定されました。)
```

CICS Liberty JVM サーバーのセキュリティー実装は、Liberty のエンジェル・プロセスを使用して、許可セキュリティー検査を実行します。Liberty がエンジェル・プロセスに接続できない場合は、UNIX システム・サービス・セキュリティーを使用する方法にフェイルオーバーします。このセキュリティーでは、STEPLIB と DFHRPL の連結内のすべてのメンバーがプログラムで管理されている必要があります。

重要: Liberty サーバーは、サーバーの始動時にのみエンジェル・プロセスに接続します。認証を完了するには、JVM サーバーを再始動する必要があります。

CICS Liberty JVM サーバー内の保護 Web アプリケーションにアクセスしようとするときに、ユーザー ID とパスワードでユーザーを認証できず、APPL-ID にアクセスできない

Liberty `messages.log` には、以下のメッセージが入っています。

```
com.ibm.ws.security.saf.SAFServiceResult E CWWKS2909E:
A SAF authentication or authorization attempt was rejected because the server
is not authorized to access the following SAF resource:
APPL-ID APPL-ID. Internal error code 0x03008108.
```

CICS Liberty JVM サーバー・セキュリティーには、クラス APPL および SERVER の SAF セキュリティー・プロファイルへのアクセスが必要です。アクセス権限が付与されていない場合、Liberty ではユーザー ID とパスワードを認証できません。これを構成する方法の詳細については、Liberty JVM サーバーでのユーザー認証を参照してください。

dropins ディレクトリーにデプロイされた後、Web アプリケーションが使用できない

`dfhjvmerr` で CWWK0221E エラー・メッセージを受け取っている場合、JVM プロファイルと `server.xml` でホスト名とポート番号に正しい値を設定していることを確認します。ポートが別のプロセスによって使用中であり、ポート共用が使用不可になっている可能性があります。クライアントがホスト名を解決できない可能性があります。

Liberty JVM サーバーを使用可能にした後、CICS の CPU 使用量が増大する

`server.xml` 内で *<config>* エレメントと *<applicationMonitor>* エレメントを使用して、構成とインストール済みアプリケーションの両方の更新を定期的に検査するように Liberty を構成できます。構成のポーリング・レートやアプリケーションのモニター間隔を高すぎる頻度に設定すると、CPU と入出力の使用量が過大になる可能性があります。

<config> については、*monitorInterval* 属性を使用して頻度を下げることができます。 CICS では構成変更が数秒以内に Liberty に反映される必要があるので、*updateTrigger* 属性は無効に設定しないでください。

<applicationMonitor> の場合は、*pollingRate* 属性を使用して頻度を削減し、*updateTrigger* 属性を `mbean` に変更するか、無効にできます。

詳細については、Controlling dynamic updatesを参照してください。

アプリケーションが使用可能にならない

WAR ファイルを dropins ディレクトリーにコピーしても、アプリケーションが使用可能にならないという問題が発生することがあります。Liberty messages.log ファイルの中でエラー・メッセージを確認してください。 CWWKZ0013E エラー・メッセージを受け取っている場合、既に同じ名前の Web アプリケーションが Liberty JVM サーバーで実行されています。この問題を修正するには、Web アプリケーションの名前を変更してから、dropins ディレクトリーにデプロイします。

Web アプリケーションが「Context Root Not Found (コンテキスト・ルートが見つかりません)」を返す

Liberty JVM サーバーを使用可能にし、Web アプリケーションをデプロイしました。JVM サーバーは使用可能であることを報告していますが、アプリケーションにアクセスしようとすると Context Root Not Found を受け取ります。しばらく後で Web アプリケーションにアクセスすると、成功します。これはタイミング・ウィンドウと呼ばれるもので、その期間は、サーバーは使用可能であることを報告しますが、アプリケーションはまだバックグラウンドで開始中です。シスプレックス・ディストリビューターやポート共用を使用する複数領域環境では、この状態が発生する可能性が高くなります。また、自動化を使用して enabled 状況から起動されるアプリケーションにアクセスする場合も、この状態が発生する可能性があります。シスプレックス・ディストリビューターまたはポート共用を使用している場合は、TCP/IP 自動化を使用してポートを休止し、Web アプリケーションが使用可能になった後でポートを再開できます。回避策として、自動化スクリプトに休止を追加したり、位置が分かる場合は、アプリケーションでその位置にフラグを書き込んだりする方法が考えられます。

Web アプリケーションが認証を要求しない

セキュリティーを構成しましたが、Web アプリケーションが認証を要求していません。

1. Web アプリケーションに対して CICS セキュリティーを構成できますが、Web アプリケーションは、WAR ファイルの中にセキュリティーの制限が含まれる場合に限り、セキュリティーを使用します。アプリケーション開発者によって、セキュリティーの抑制が動的 Web プロジェクトの web.xml ファイル内に定義されたことを確認します。
2. server.xml ファイルに、正しいセキュリティー情報が含まれていることを確認します。すべての構成エラーは dfhjvmerr に報告され、有用な情報がいくつか提供されていることがあります。CICS セキュリティーを使用している場合、server.xml に機能 **cicsts:security-1.0** が指定されていることを確認します。CICS セキュリティーのスイッチがオフの場合、アプリケーション・ユーザーを認証するための基本ユーザー・レジストリーを指定したことを確認します。
3. server.xml が、EJBRoles を利用するように <safAuthorization> 用に構成されているか、または <application-bnd> エレメントでローカル・ロール・マッピング用に構成されているかを確認します。 <application-bnd> エレメントは、server.xml または installedApps.xml の <application> エレメントにあります。ローカル・ロール・マッピング用に CICS によって追加されたデフォルトのセキュリティー・ロールは **cicsAllAuthenticated** です。

Web アプリケーションが HTTP 403 エラー・コードを返す

Web アプリケーションが Web ブラウザーに HTTP 403 エラー・コードを返す理由は、ユーザー ID が取り消されているか、アプリケーション・トランザクションの実行が許可されていないかのどちらかです。

1. CICS メッセージ・ログのエラー・メッセージ ICH408I を調べて、どの種類の許可障害が発生したかを確認します。問題を解決するには、ユーザー ID のパスワードが正しいことと、トランザクションの実行が許可されていることを確認します。
2. ICH408I メッセージが見つからない場合は、`messages.log` ファイルを確認します。
   - 以下のメッセージの場合:

     ```
     CWWKS3005E: A configuration exception has occurred.
     No UserRegistry implementation service is available.
     Ensure that you have a user registry configured.
     (構成例外が発生しました。使用可能な UserRegistry 実装サービスがありません。
     ユーザー・レジストリーを構成したことを確認してください。)
     ```

     `server.xml` に SAF レジストリーを構成したことを確認する必要があります。詳しくは、server.xml の手動調整を参照してください。
   - 分散 ID の使用中に以下のメッセージが表示された場合、

     ```
     CWWKS9104A: Authorization failed for user alidist:defaultRealm
     while invoking LdapTests on /basic.
     (/basic で LdapTests を呼び出し中に、
     ユーザー alidist:defaultRealm の許可に失敗しました。)
     このユーザーは必要なロールのどれに対しても
     アクセス権限が付与されていません。[testing]
     ```

     `server.xml` が **`<safAuthorization>`** 用に構成されているか、または `cicsts:distributedIdentity-1.0` 機能が含まれている場合は、RACMAPped ユーザー ID に適切な EJBRoles が定義されていることを確認します。詳しくは、SAF ロール・マッピングを使用した許可を参照してください。
     `server.xml` が **`<safAuthorization>`** 用に構成されておらず、かつ `cicsts:distributedIdentity-1.0` 機能が含まれていない場合は、**`<application-bnd>`** エレメントで、適切な配布ユーザー ID に対して適切なロールへのアクセス権限が定義されていることを確認します。詳しくは、ユーザーに Liberty JVM サーバー内のアプリケーションを実行する権限を与えるを参照してください。
3. アプリケーションがクラス com.ibm.ws.webcontainer.util.Base64Decode の例外を返す場合は、`dfhjvmerr` のエラー・メッセージを確認します。構成エラー・メッセージ (例えば、CWWKS4106E または CWWKS4000E) がある場合、サーバーは、異なるエンコード方式で作成された構成ファイルにアクセスしようとしています。このタイプの構成エラーは、**`file.encoding`** 値を変更して JVM サーバーを再始動するときに発生することがあります。この問題を解決するには、以前のエンコード方式に戻して JVM サーバーを再始動するか、構成ファイルを削除することができます。JVM サーバーは、始動するときに、正しいファイル・エンコード方式でファイルを再作成します。

Web アプリケーションが HTTP 500 エラー・コードを返す

Web アプリケーションは、Web ブラウザーに HTTP 500 エラーを返します。HTTP 500 エラーを受け取る場合、構成エラーが発生しました。

1. CICS メッセージ・ログの DFHSJ メッセージを調べてください。そのメッセージに、このエラーの特定の原因に関する詳細情報が記されていることがあります。
2. URIMAP を使用して特定のトランザクションに対するアプリケーション要求を実行している場合、URIMAP が正しいトランザクション ID を指定していることを確認します。
3. SCHEME 属性と USAGE 属性が正しく設定されていることを確認します。SCHEME は、アプリケーション要求 (HTTP または HTTPS のどちらか) と一致している必要があります。USAGE 属性は、JVMSERVER に設定されている必要があります。

Web アプリケーションが HTTP 503 エラー・コードを返す

Web アプリケーションは、Web ブラウザーに HTTP 503 エラーを返します。HTTP 503 エラーを受け取る場合、アプリケーションは使用不可です。

1. CICS メッセージ・ログの DFHSJ メッセージを調べて、追加情報を確認します。
2. アプリケーションに対して TRANSACTION リソースと URIMAP リソースが使用可能であることを確認します。これらのリソースが CICS バンドル内のアプリケーションの一部としてパッケージされている場合、BUNDLE リソースの状況を確認します。
3. 要求は、完了前に消去された可能性があります。ログ内のエラー・メッセージに、要求が消去された理由が説明されています。

分散 ID マッピングを使用して Web アプリケーションにアクセスできない

配布 ID マッピングを使用しているときに、`messages.log` ファイルに以下のメッセージが表示された場合:

```
FFDC1015I: An FFDC Incident has been created (FFDC 機能不良が発生しました):
"com.ibm.ws.security.saf.SAFException:

CWWKS2905E: SAF service IRRSIA00_CREATE did not succeed because
user null was not found in the SAF registry.
SAF 戻りコード 0x00000008。RACF 戻りコード 0x00000008。RACF reason code 0x00000010.

FFDC1015I: An FFDC Incident has been created:
"javax.security.auth.login.CredentialException: could not create SAF credential
for <distid> DistId
```

CICS メッセージ・ログのエラー・メッセージ ICH408I を調べて、どの種類の許可障害が発生したかを確認します。それが、`ICH408I USER(<userid>) GROUP(TSOUSER ) NAME(<name>) DISTRIBUTED IDENTITY IS NOT DEFINED: 776 cn= <distid> DistId,ou=users,dc=domain,dc=com LdapRegistry` の場合、アプリケーションへのアクセスに使用されている配布 ID 用の適切な RACMAP を作成する必要があります。**RACMAP QUERY** コマンドはデバッグに役立ちます。以下に例を示します。

```
RACMAP QUERY USERDIDFILTER(NAME('ou=users,dc=domain,dc=com')) REGISTRY(NAME('LdapRegistry'))
```

Web アプリケーションが例外を返す

Web アプリケーションは、Web ブラウザーに例外を返しました。例えば、アプリケーションは、クラス com.ibm.ws.webcontainer.util.Base64Decode の例外を戻しています。

1. `dfhjvmerr` でエラー・メッセージを確認します。
2. 構成エラー・メッセージ (例えば、CWWKS4106E または CWWKS4000E) がある場合、サーバーは、異なるエンコード方式で作成された構成ファイルにアクセスしようとしています。このタイプの構成エラーは、**`file.encoding`** 値を変更して JVM サーバーを再始動するときに発生することがあります。この問題を解決するには、以前のエンコード方式に戻して JVM サーバーを再始動するか、構成ファイルを削除することができます。JVM サーバーは、始動するときに、正しいファイル・エンコード方式でファイルを再作成します。

`messages.log` 内のエラー・メッセージ CWWKB0109E

Liberty が正常なシャットダウンに失敗した場合、WLP_ZOS_PLATFORM=TRUE に設定されている次の Liberty JVM サーバーで、エラー・メッセージ CWWKB0109E が `messages.log` ファイルに書き込まれます。このエラーは修正する必要はなく、無視してかまいません。

それ以外の場合、CICS 領域ユーザー ID は **`IFAUSAGE`** に対する登録と登録解除には使用できません。Liberty サーバー・エンジェル・プロセスを参照してください。

エラー・メッセージ WTRN0078Eトランザクション・マネージャーがトランザクション・リソースで開始を呼び出そうとした結果、エラーになりました。

エラー・コードは `XAER_PROTO` でした。このエラーが発生する場合、最も可能性の高いシナリオは、Liberty サーバー上でデフォルトの JTA 統合が有効であり、かつアプリケーションが `REQUIRES_NEW` として宣言された Bean メソッドを使用しているという状況の場合です。例えば、`@Transactional(value = TxType.REQUIRES_NEW) void yourMethod{}` のように XA トランザクション内で REQUIRES_NEW を使用することは、CICS ではサポートされていません。アプリケーションを実行する前に変更する必要があります。

MSGUSER でエラー・メッセージ DFHSJ1004 が発生するものの、対応する STDERR 例外がない

zFS ファイル・システム・スペースが不足する場合の症状として、対応する STDERR 例外がない DFHSJ1004 が発生する可能性があります。このメッセージは、スペース不足のために送信されますが、メッセージをファイルに書き込むスペースがないため、STDERR に例外がありません。

「z/OS UNIX システム・サービスの計画」の『ファイル・システムの管理』で説明されている技法を使用して、ファイル・システムのサイズを計画およびモニターすることができます。

productInfo スクリプトを使用して Liberty の整合性を検証

CICS のインストール後、またはサービスの適用後に、productInfo スクリプトを使用して Liberty インストールの整合性を検証できます。

1. ディレクトリーを CICS USSHOME ディレクトリーに移動します。
2. productInfo では Java を使用するので、PATH に Java が組み込まれていることを確認する必要があります。または、JVM プロファイルで **JAVA_HOME** 環境変数を **JAVA_HOME** の値に設定します。例えば、次のようにします。

   ```
   export JAVA_HOME=/usr/lpp/java/J8.0_64
   ```

3. 検証オプション wlp/bin/productInfo validate を指定して productInfo スクリプトを実行します。エラーが出ないようにしてください。 Liberty **productInfo** スクリプトについて詳しくは、Liberty プロファイルのインストールの整合性を検証を参照してください。

Liberty コマンドを実行するために wlpenv スクリプトを使用する

IBM サービスから、**productInfo** や **server dump** などの 1 つ以上の Liberty 提供コマンドの実行を依頼されることがあります。こうしたコマンドを実行するには、必要な環境を設定するためのラッパーとして wlpenv スクリプトを使用できます。このスクリプトは、JVM プロファイルが正常に構文解析された後に、Liberty JVM サーバーを使用可能にするたびに作成および更新されます。このスクリプトは各 CICS 領域の各 JVM サーバーに固有であるため、JVM プロファイルにデフォルトで指定されているように *WORK_DIR/APPLID/JVMSERVER* 内に作成され、wlpenv という名前になります。*APPLID* は、CICS 領域 APPLID の値で、*JVMSERVER* は JVMSERVER リソースの名前です。

UNIX システム・サービス・シェルで **wlpenv** スクリプトを実行するには、JVM プロファイルで指定されている *WORK_DIR* にディレクトリーを変更し、引数として Liberty コマンドを指定してスクリプトを実行します。例えば、次のようにします。

```
./wlpenv productInfo version
./wlpenv server dump --archive=package_file_name.dump.pax --include=heap
```

server dump コマンドの場合は、サーバー名を指定しません。サーバー名は、JVM サーバーが最後に使用可能になったときに設定されていた値に wlpenv スクリプトによって設定されるからです。

Liberty コマンドについて詳しくは、productInfo コマンドおよびコマンド行からの Liberty サーバー・ダンプの生成を参照してください。

Java EE アプリケーション起動時のトラブルシューティング

EXEC CICS LINK コマンドが **RESP = PGMIDERR**、**RESP2 = 1** で失敗する

1. アプリケーションを検査して、正しい成果物が生成されたか判別します。
   a. ソース・プロジェクトで注釈処理が有効であるか確認します。

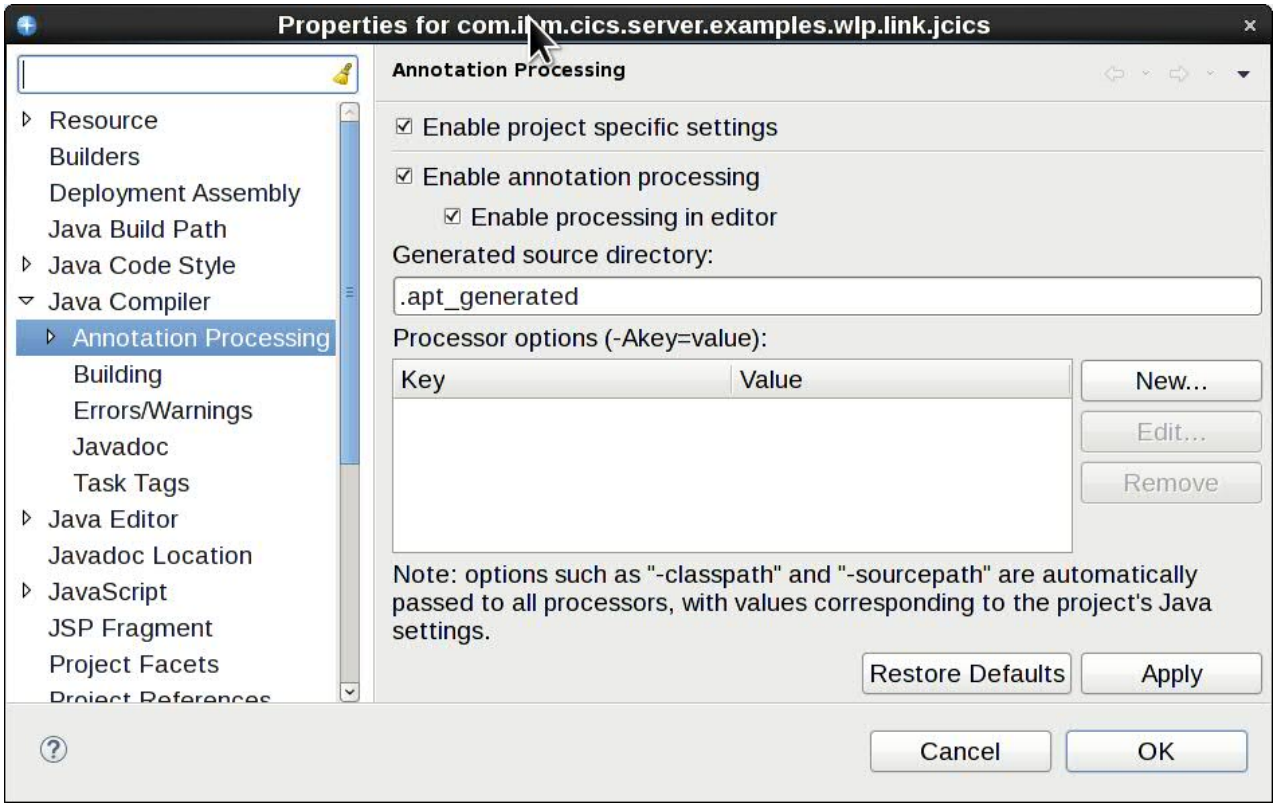


図 *12.* 注釈処理が有効であることを確認

b. `@CICSProgram` が Java メソッドに追加されており、正常にコンパイルするか確認します。

c. プロジェクトに `web.xml` が含まれる場合、それが指定するサーブレット仕様のバージョンを確認してください。少なくともバージョン 2.5 が必要です。

d. アプリケーションをエクスポートし、`com.ibm.cics.server.invocation.proxy` パッケージに生成されたコードを確認します。例えば、ワークステーション上で、アーカイブ・マネージャーを使用して WAR ファイルまたは EAR ファイルを開くか、z/OS 上で `jar -tf` コマンドを使用して、WAR ファイルまたは EAR ファイルの内容を調べます。コードが生成されていない場合、最新バージョンの CICS Explorer、CICS ビルド・ツールキット、またはアノテーション・プロセッサーを使用しているか確認します。

2. CICS メッセージ・ログを調べて、以下のようなメッセージを探します。

   - DFHSJ1204: A linkable service has been registered for class examples.TSQ.ClassOne method anotherMethod with program name LINKJCIN in JVMSERVER LINKJVM
   - DFHPG0101: Resource definition for LINKJCIN has been added. (LINKJCIN のリソース定義が追加されました。)

   これらのメッセージがない場合は、次のようにします。

a. 使用可能な状態の Liberty JVM サーバーがあることを確認します。

b. `server.xml` に `cicsts:link-1.0` 機能を構成したことを確認します。構成されている場合は、メッセージ「`J2CA7001I: Resource adapter com.ibm.cics.wlp.program.link.connectorinstalled`」が `messages.log` に出力されます。

c. CICS バンドルを使用してアプリケーションをデプロイしている場合、バンドルがインストールされて使用可能になっていることを確認します。

d. アプリケーションが Liberty にインストールされているか確認します。インストールされている場合、`messages.log` にユーザーのアプリケーションの名前を含むメッセージが表示されます。例:
`CWWKZ0001I: Application com.ibm.cics.test.javalink started.`

`EXEC CICS LINK` コマンドが RESP = PGMIDERR、RESP = 27 で失敗する

これは、CICS が Liberty 内の Java EE アプリケーションを呼び出そうとしたが、アプリケーションが成功する前にタイムアウトが発生したことを示しています。この問題の最も一般的な原因は、JVM サーバー内に使用可能なスレッドがなかったことです。これを解決するには、JVM サーバーのスレッド限度を増やすか、あるいは WLP_LINK_TIMEOUT の値を大きくして、タスクがスレッドを獲得するために待機できる時間を長くします。詳しくは、JVM プロファイルに使用するシンボルおよびManaging the thread limit of JVM servers内の WLP_LINK_TIMEOUT を参照してください。

JCICS API 呼び出しが CICSRuntimeException をスローする

```
com.ibm.cics.server.CicsRuntimeException:
DTCTSQ_READNEXT: No JCICS context is associated with the current thread.
```

この例外の最も可能性が高い原因は、特定のスレッドで JCICS オブジェクトを作成し、別のスレッドからそのインスタンス・メソッドを呼び出そうとしたことです。そのメソッドを呼び出すスレッドと同じスレッド上で JCICS オブジェクトを構成するように、アプリケーションを変更します。

別のスレッドのオブジェクトを気付かずに使用してしまうことにつながるパターンとしては、以下があります。

- `java.lang.Runnable` または `java.util.concurrent.Callable` のコンストラクター内で JCICS オブジェクトを構成する。代わりに `run()` メソッド内でオブジェクトを構成します。
- JCICS オブジェクトが静的変数に割り当てられている。代わりに、インスタンス変数を使用してください。
- JCICS オブジェクトを、別のスレッドによって実行されるメソッドにパラメーターとして渡している。そのスレッド自身が JCICS オブジェクトを構成している必要があります。

Java EE アプリケーションの呼び出しに使用中にトランザクションが AJ05 で異常終了する

次の例外が `dfhvjmerr` ファイルに記録されます。

```
com.ibm.cics.server.InvalidRequestException: CICS INVREQ Condition(RESP=INVREQ, RESP2=200)
java.lang.RuntimeException:
javax.transaction.RollbackException:
XAResource start association error:XAER_PROTO
```

Liberty へのリンクで JTA を使用することは、CICS JTA 統合が使用不可になっている場合にのみサポートされます。これを構成するには、server.xml に <cicsts_jta integration="false"/> を指定します。

Java スタック・オーバーフロー

Java エラー・メッセージ「java.lang.StackOverflowError: operating system stack overflow」は、通常、スレッドがオペレーティング・システム・スレッドの初期スタック・サイズを超えた場合に発生します。このサイズは、JVM プロファイルの JVM オプション -Xmso によって設定されます。Java Platform Debugger Architecture (JPDA) が使用可能になっている場合は、この値を増やす必要がある場合があります。

ログ内の予期しない ICH408I メッセージ

これらは標準の監査メッセージです。詳細については、「z/OS Security Server RACF 監査担当者のガイド」の『z/OS UNIX System Services の監査を管理するクラス』を参照してください。

以下のいずれかの RACF コマンドを実行することによって、発行されないようにすることができます

- SETROPTS LOGOPTIONS(NEVER(IPCOBJ))
- SETROPTS LOGOPTION(DEFAULT(IPCOBJ))

IPCOBJ は、z/OS UNIX セキュリティー・イベントの監査のためにのみ定義され、許可検査には使用されません。

Liberty Bundlepart がタイムアウトになる

JVM ログまたは STDERR ファイルに以下のメッセージがあります。

```
The application installed by bundlepart <symbolic-name> was not started
after 30000 milliseconds. Either a problem exists with the application,
or the system is busy.
This timeout can be controlled by the System Property
'com.ibm.cics.jvmserver.wlp.bundlepart.timeout=n'
where n is the value of milliseconds to wait.
```

Liberty messages.log の中で CWWKZ メッセージを確認してください。CWWKZ メッセージは、アプリケーションが開始されなかった理由に関する情報を提供している場合があります。アプリケーションに関する CWWKZ メッセージがない場合、<config> 要素が、ポーリングを使用し、モニター・インターバルがタイムアウトよりも小さくなるように構成されていることを確認してください。以下に例を示します。

```
<config updateTrigger="polled" monitorInterval="10s" />
```

モニターのインターバルがバンドル部分のタイムアウトよりも小さい場合、タイムアウト値を増加させる必要があります。タイムアウト値は、JVM システム・プロパティー com.ibm.cics.jvmserver.wlp.bundlepart.timeout によって制御されます。

JVM 出力、ログ、ダンプ、およびトレースの場所の制御

JVM サーバーで実行される Java アプリケーションからの出力は、z/OS UNIX ファイルに書き込むことができます。 JVM サーバーの STDOUT、STDERR、JVMLOG、および JVMTRACE オプションで z/OS UNIX ファイルを指定します。指定しなければ、JES ログに転送されます。Liberty JVM サーバーでは、Liberty サーバーの出力は、構成されたログ・ディレクトリーに相対的な `messages.log` にあります。

デフォルトでは、JVM サーバーで実行される Java アプリケーションからの出力は、z/OS UNIX ファイル・システムに書き込まれます。 z/OS UNIX ファイル・システムは、`$WORK_DIR/APPLID/JVMSERVER` のディレクトリー構造内でファイル名規則 `DATE.TIME.<dfhjvmxxx>` に従います。これらのオーバーライドを使用して、出力を JES ログに転送することもできます。詳しくは、DD ステートメントを使用した JVM サーバー出力の JES への転送 を参照してください。

デフォルトをオーバーライドする場合は、`STDOUT`、`STDERR`、`JVMLOG`、および `JVMTRACE` オプションに zFS ファイル名を指定します。ただし、固定のファイル名を使用した場合、その JVM プロファイルを使用して作成されたすべての JVM の出力が同じファイルに追加されます。さまざまな JVM からの出力が、レコード・ヘッダーなしで混ざり合うことになります。この状態では、問題判別の役に立ちません。

これらの値をカスタマイズする場合は、`STDOUT`、`STDERR`、`JVMLOG`、および `JVMTRACE` オプションに可変ファイル名を指定することをお勧めします。CICS 領域の存続期間中、個々の JVM ごとに固有のファイルにすることができます。さらに、追加の識別情報を含めることもできます。

- *APPLID* シンボルを使用して、ファイル名に CICS 領域の APPLID を含めることができます。

ファイル名に含められるその他の識別情報としては、*DATE* シンボルと *TIME* シンボルがあります。

DATE は、JVM サーバー始動時にプロファイルで解析された日付 (形式は Dyymmdd) に置換されます。

TIME は、JVM サーバー始動時にプロファイルで解析された時刻 (形式は Thhmmss) に置換されます。

JVMSERVER は、JVMSERVER リソースの名前に置換されます。

さらなるカスタマイズを、`USEROUTPUTCLASS` オプションを使用してプログラムのレベルで実施できますが、これは Liberty では機能しません。JVM プロファイルに指定する `USEROUTPUTCLASS` オプションには、Java クラスを指定します。Java クラスは、JVM からの出力を代行受信して、カスタム・ロケーション (CICS 一時データ・キューなど) にリダイレクトします。出力レコードにタイム・スタンプとヘッダーを追加し、JVM で実行中の個々のトランザクションからの出力を識別することができます。CICS には、これらのタスクを実行するサンプル・クラスが用意されています。

JVM から出力される JAVADUMP ファイルの場所は、JVM プロファイルの `WORK_DIR` オプションで指定した、z/OS UNIX 上の作業ディレクトリーです。

JAVADUMP ファイルは、それらの名前にあるタイム・スタンプで一意的に識別されます。これらのファイルの名前をカスタマイズすることはできません。より詳細な Java TDUMP は、JAVA_DUMP_TDUMP_PATTERN オプションで指定したファイルに書き込まれます。CICS に付属のサンプル JVM プロファイルで示されているように、この値で *APPLID*、*DATE*、*TIME*、および *JVMSERVER* シンボルを使用すると、JVM ごとに固有の名前にすることができます。

JVM は、JAVADUMP または TDUMP を生成すると、情報をその stderr ストリームに書き込みます。JAVADUMP ファイルと TDUMP ファイルの内容の詳細については、Troubleshooting and supportを参照してください。

DD ステートメントを使用した JVM サーバー出力の JES への転送

出力を特定の場所にリダイレクトするように JVM サーバーを更新できます。

JVM サーバーの STDOUT、STDERR、JVMTRACE、JVMLOG、および messages.log 出力を JES ログに転送できます。これによって、JVM サーバーのログ・ファイル出力を MSGUSR といった他の CICS ログと共に管理できるようになります。

JOBLOG パラメーターを使用する場合、STDOUT、JVMLOG、および JVMTRACE は、SYSPRINT が定義されていれば SYSPRINT に、定義されていなければ動的な SYSnnn に転送されます。JVMTRACE=JOBLOG のみが指定されている場合、JVMTRACE は現在の stdout の場所に転送されます。 STDERR は、SYSOUT が定義されていれば SYSOUT に、定義されていなければ動的な SYSnnn に転送されます。以下に例を示します。

```
STDOUT=JOBLOG
STDERR=JOBLOG
JVMTRACE=JOBLOG
JVMLOG=JOBLOG
```

出力は、JES に定義されているいずれかの MVS データ定義 (DD) に転送することもできます。例えば、以下のように CICS 領域 JCL に DD ステートメント JVMOUT、JVMERR、および MSGLOG を指定する場合です。

```
//JVMOUT   DD SYSOUT=*
//JVMERR   DD SYSOUT=*
//MSGLOG   DD SYSOUT=*    <--- redirects Liberty messages.log
```

Liberty で構成された DD ステートメントが CICS ランタイム JCL に定義されていない場合、これらのログは指定された DD 出力に自動的にリダイレクトされ、Liberty の開始時に CICS ジョブ出力にリストされます。

これにより、JVM プロファイルで以下の JVM プロファイル・オプションを使用して、stdout ストリームと stderr ストリームを宛先 JVMOUT および JVMERR に転送できます。省略すると、それらの宛先が JVM サーバーによって自動的に作成されます。JVM プロファイル構成の必要なく、MSGLOG ステートメントによって messages.log が JES に自動的にリダイレクトされます。

```
STDOUT=//DD:JVMOUT
STDERR=//DD:JVMERR
```

JVM サーバー出力の発信元を示すために、JES に転送されるすべての stdout および stderr 項目は、接頭辞として JVM サーバー名の文字列を付けて書き込まれます。これは、複数の JVM サーバーで 1 つの宛先を共用する場合に便利です。この

動作を無効にするには、JVM プロファイル・オプション IDENTITY_PREFIX を使用します。このオプションを FALSE に設定すると、接頭辞文字列は使用されなくなります。

宛先を指定しない場合、出力は zFS のデフォルト・ファイルにリダイレクトされますが、特定の zFS ファイルに送信されるように設定することも可能です。 333 ページの『JVM 出力、ログ、ダンプ、およびトレースの場所の制御』を参照してください。

JVM stdout および stderr ストリームのリダイレクト

アプリケーション開発時に、開発者は USEROUTPUTCLASS オプションを使用して、CICS 領域で開発者固有の stdout 項目および stderr 項目を分離し、開発者が選択した識別可能な宛先に送信できます。Java クラスを使用して出力をリダイレクトし、出力レコードにタイム・スタンプとヘッダーを追加できます。ただし、この方法ではダンプ出力を代行受信することはできません。

USEROUTPUTCLASS オプションを指定すると、JVM のパフォーマンスに悪影響が出ます。実稼働環境で最高のパフォーマンスを発揮するには、このオプションは使用しないでください。

アプリケーションまたはシステム・コードのどちらかによって System.out() または System.err() に書き込まれた出力は、出力リダイレクト・クラスによってリダイレクトできます。ただし、JVM から発行されるいくつかのメッセージには、JVM プロファイルの STDOUT オプションと STDERR オプションで指定した z/OS UNIX ファイルが使用されます。また、これらのファイルは、USEROUTPUTCLASS オプションで指定されたクラスが目的の宛先にデータを書き込めない場合にも使用されます。したがって、これらのファイルに適切なファイル名を指定する必要があります。

USEROUTPUTCLASS オプションを使用するには、適当な Java クラスの名前を指定して、JVM プロファイルで USEROUTPUTCLASS=[java class] を指定してください。このクラスは java.io.OutputStream を拡張します。提供されたサンプル JVM プロファイルには、コメント化されたオプション USEROUTPUTCLASS=com.ibm.cics.samples.SJMergedStream が含まれています。このオプションは、提供されたサンプル・クラスを指定します。com.ibm.cics.samples.SJMergedStream クラスを使用して JVM からの出力をそのプロファイルで処理するには、このオプションをアンコメントしてください。また、CICS は代わりのサンプル Java クラス com.ibm.cics.samples.SJTaskStream も提供します。

JVM サーバーの場合、OSGi フレームワークで出力リダイレクト・クラスを実行するには、そのクラスを OSGi バンドルとしてパッケージ化します。詳細については、JVM stdout および stderr 出力をリダイレクトするための Java クラスの作成を参照してください。

注: 出力リダイレクト・サンプルは、OSGi およびクラスパス JVM サーバーで機能するものであり、Liberty JVM サーバーでは機能しません。

サンプル・クラス com.ibm.cics.samples.SJMergedStream および com.ibm.cics.samples.SJTaskStream

CICS 要求を発行できる Java アプリケーション・スレッドの場合、JVM からの出力を代行受信し、その出力を一時データ・キューに書き込むことができます。JVM のアクティビティーと CICS のアクティビティーを相互に関連付けるログが生成されます。

出力を代行受信するときに、タイム・スタンプ、タスク ID とトランザクション ID、およびプログラム名を追加できます。したがって、複数の JVM からの出力が含まれる、マージされたログ・ファイルを作成することが可能です。このログ・ファイルを使用すると、JVM のアクティビティーを CICS のアクティビティーと相互に関連付けることができます。サンプル・クラス com.ibm.cics.samples.SJMergedStream は、マージされたログ・ファイルを作成するようにセットアップされています。

com.ibm.cics.samples.SJMergedStream クラスは、JVM からの出力を一時データ・キュー CSJO (`stdout` ストリームの場合) および CSJE (`stderr` ストリームおよび内部メッセージの場合) に送信します。これらの一時データ・キューは、グループ DFHDCTG で提供され、CSSL にリダイレクトされますが、必要に応じて再定義することができます。

このクラスは、出力のリダイレクトによって、日付、時刻、APPLID、TRANSID、タスク番号、およびプログラム名を含むヘッダーを各レコードに追加します。その結果、JVM 出力用とエラー・メッセージ用に 2 つのマージされたログ・ファイルが作成されます。これらのログ・ファイルでは、出力とメッセージの送信元を容易に特定できます。

これらのクラスは、`/usr/lpp/cicsts/cicsts55/lib` ディレクトリーにあるファイル `com.ibm.cics.samples.jar` に出荷時に入っています。ここで、`/usr/lpp/cicsts/cicsts55` は、z/OS UNIX 上の CICS ファイルのインストール・ディレクトリーです。これらのクラスのソースもサンプルとして提供されているので、必要に応じてそれらのクラスを変更するか、サンプルに基づいて独自のクラスを作成することができます。これらのクラスは OSGi バンドル JAR としてパッケージされます。これらのクラスは、CLASSPATH JVM サーバーにデプロイするか、または OSGi JVM サーバーに OSGI_BUNDLES JVM サーバー・オプションを使用したミドルウェア・バンドルとしてデプロイすることができます。詳細については、JVM stdout および stderr 出力をリダイレクトするための Java クラスの作成を参照してください。

CICS によって接続されたスレッド以外のスレッドで実行される Java アプリケーションは、CICS 要求を発行できません。JVM からの出力は、CICS 機能を使用してリダイレクトできません。com.ibm.cics.samples.SJMergedStream クラスは、引き続き出力を代行受信し、各レコードにヘッダーを追加します。この出力は、前述のとおり、z/OS UNIX ファイル `/`*`work_dir`*`/`*`applid`*`/stdout/CSJO` および `/`*`work_dir`*`/`*`applid`*`/stderr/CSJE` に書き込まれます。これらのファイルが使用不可である場合、出力は、JVM プロファイルの `STDOUT` および `STDERR` オプションで指定された z/OS UNIX ファイルに書き込まれます。

JVM 出力用にマージされたログ・ファイルを作成する代わりに、単一のタスクからの出力を z/OS UNIX ファイルに送信することもできます。また、タイム・スタンプとヘッダーを追加して、単一のタスクに固有の出力ストリームを提供できます。CICS に付属のサンプル・クラス com.ibm.cics.samples.SJTaskStream は、この目的のためにセットアップされています。このクラスは、各タスクの出力を 2 つの z/OS UNIX ファイルに送信します。一方は、`stdout` ストリーム用、もう一方は `stderr` ストリーム用です。これらのストリーム内の各出力項目には、タスク番号を使用して固有の名前が付けられます (`YYYYMMDD.task.`*`tasknumber`* という形式)。これらの z/OS UNIX ファイルは、`STDOUT` および `STDERR` というディレクトリーにそれぞれ保管されます。このプロセスは、CICS によって接続されたスレッドで実行される Java アプリケーションでも、その他のスレッドで実行される Java アプリケーションでも同じです。

エラー処理

JVM から出されるメッセージの長さは可変です。CSSL キューの最大レコード長 (133 バイト) では、受信したメッセージによっては格納できない可能性があります。キューの最大レコード長を超えるメッセージを受信した場合、このサンプルの出力リダイレクト・クラスはエラー・メッセージを発行します。メッセージのテキストが影響を受ける可能性があります。

133 バイトより長いメッセージを JVM から受信することが分かった場合には、CSJO と CSJE を別々の一時データ・キューとして再定義します。それらのキューを区画外宛先にして、キューのレコード長を増やしてください。そのキューを物理データ・セットまたはシステム出力データ・セットに割り振ることができます。この場合、システム出力データ・セットの方が便利な場合があります。出力を表示するためにキューをクローズする必要がないためです。一時データ・キューの定義方法については、TDQUEUE リソースを参照してください。CSJO と CSJE を再定義する場合は、グループ DFHDCTG で定義される一時データ・キューと同じように、コールド・スタート時にできるだけ早くそれらのキューがインストールされるようにしてください。

一時データ・キュー CSJO および CSJE にアクセスできない場合、出力は、z/OS UNIX ファイル `/`*`work_dir`*`/`*`applid`*`/stdout/CSJO` および `/`*`work_dir`*`/`*`applid`*`/stderr/CSJE` に書き込まれます。ここで、*work_dir* は JVM プロファイルの `WORK_DIR` オプションで指定されたディレクトリー、*applid* は CICS 領域に関連付けられたアプリケーション ID です。これらのファイルが使用不可である場合、出力は、JVM プロファイルの `STDOUT` および `STDERR` オプションで指定された z/OS UNIX ファイルに書き込まれます。

サンプルの出力リダイレクト・クラスでエラーが検出された場合、1 つ以上のエラー・メッセージが発行されます。出力メッセージの処理中にエラーが発生した場合、エラー・メッセージは `System.err` に送信され、リダイレクトの対象になります。ただし、エラー・メッセージの処理中にエラーが発生した場合は、JVM プロファイルの `STDERR` オプションで指定されたファイルに、新しいエラー・メッセージが送信され、Java クラスでの再帰的ループが回避されます。これらのクラスは、呼び出し側の Java プログラムに例外を戻しません。

Java ダンプ・オプションの制御

JVM にダンプ・オプションを指定するには、JVM プロファイルで -Xdump オプションを使用できます。

Java ダンプ・オプションに関する情報は、Troubleshooting and support に記載されています。

JVM サーバーの CICS コンポーネント・トレース

Java によって作成されるログに加えて、CICS は、SJ (JVM) および AP ドメインで、トレース・レベル 0、1、および 2 において、いくつかの標準トレース・ポイントを提供します。これらのトレース・ポイントは、CICS が JVM サーバーのセットアップと管理を行う際に取るアクションをトレースします。

CETR トランザクションを使用して、SJ および AP ドメイン・トレース・ポイントをレベル 0、1 および 2 で活動化できます。SJ ドメインのすべての標準トレース・ポイントの詳細については、JVM および Node.js ランタイム・ドメインのトレース・ポイントを参照してください。

SJ および AP コンポーネント・トレース

SJ コンポーネントは、SJ ドメインでの例外および処理をトレースし、内部トレース・テーブルに書き込みます。AP コンポーネントは OSGi フレームワークの OSGi バンドルのインストールをトレースします。SJ のレベル 3、レベル 4、および レベル 5 のトレースは、Java ログを作成し、zFS のトレース・ファイルに書き込みます。トレース・ファイルの名前と場所は、JVM プロファイル内の `JVMTRACE` オプションによって決まります。

SJ のレベル 4 および 5 トレースは、トレース・ファイルの中に詳細なロギング情報を作成します。このトレース・レベルを使用する場合、zFS 内にファイル用の十分なスペースがあることを確認する必要があります。トレースの起動および管理について詳しくは、『JVM サーバーのトレースの活動化と管理』を参照してください。

JVM サーバーのトレースの活動化と管理

SJ および AP コンポーネントのトレースをオンにすると、JVM サーバーのトレースを活動化できます。少量のトレースが内部トレース・テーブルに書き込まれますが、Java はまた、ロギング情報を JVM サーバーごとに zFS の固有のファイルに書き込みます。このファイルはラップしないので、そのサイズを zFS で管理する必要があります。

このタスクについて

JVM サーバーのトレースは、補助トレースも GTF トレースも使用しません。CICS は、内部トレース・テーブルに一部の情報を書き込みます。しかし、ほとんどの診断情報は Java によって記録され、zFS 内のファイルに書き込まれます。このファイルには、各 JVM サーバーに由来する一意的な名前が付けられます。デフォルトのファイル名の形式は *&DATE;.&TIME;*.dfhjvmtrc です。JVMSERVER リソースを使用可能にすると、このファイルが `$WORK_DIR/&APPLID;/&JVMSERVER;` ディレクト

リーに CICS によって作成されます。JVM プロファイルでトレース・ファイルの名前と場所を変更できます。JVM サーバーの実行時にトレース・ファイルを削除または名前変更すると、CICS はそのファイルを再作成せず、ロギング情報が別のファイルに書き込まれることもありません。

手順

1. CETR トランザクションを使用して、JVM サーバーのトレースを活動化します。 JVM サーバーのトレースおよびロギングの情報を作成するには、2 つのコンポーネントを使用できます。
   - JVM サーバーを始動および停止するために CICS が行うアクションをトレースするには、SJ コンポーネントを選択します。JVM は、診断情報を zFS ファイルに記録します。
   - OSGi バンドルのインストールをトレースするには、AP コンポーネントを選択します。
2. SJ および AP コンポーネントのトレース・レベルを設定します。
   - SJ のレベル 0 は、例外のみのトレースを作成します。例えば、JVM サーバーの初期化時のエラーや、OSGi フレームワーク内の問題です。SJ のレベル 1 とレベル 2 は、SJ ドメインからさらに CICS トレースを作成します。このトレースは、内部トレース・テーブルに書き込まれます。
   - SJ のレベル 3 は、JVM から追加のログを作成します。例えば、OSGi フレームワーク内の警告メッセージや情報メッセージです。この情報は、zFS のトレース・ファイルに書き込まれます。
   - SJ のレベル 4、5 および AP のレベル 2 は、CICS および JVM からデバッグ情報を作成します。これは、JVM サーバー処理に関するより詳細な情報を提供します。この情報は、zFS のトレース・ファイルに書き込まれます。
3. 各トレース項目には、日時のタイム・スタンプがあります。`JVMTRACE` プロファイル・オプションを使用して、このトレース・ファイルの名前と場所を変更できます。
4. デフォルトの `JVMTRACE` 設定を使用している場合、JVMSERVER リソースを有効にすると、CICS は JVM の存続期間にわたって新しい固有のトレース・ファイルを作成します。 JVMSERVER リソースを無効にすると、トレース・ファイルを削除することができます。情報を別個に保持する場合はファイルの名前を変更することができます。
5. ファイルの数を管理するには、LOG_FILES_MAX オプションを設定することで、JVM サーバー始動時に保持される古いトレース・ファイルの数を制御できます。

Java アプリケーションのデバッグ

CICS における JVM は、Java Platform で提供される標準デバッグ・メカニズムである Java Platform Debugger Architecture (JPDA) をサポートします。

このタスクについて

JDPA をサポートする任意のツールを使用して、CICS で実行される Java アプリケーションをデバッグすることができます。例えば、z/OS 上の Java SDK に付属の

Java Debugger (JDB) を使用できます。JPDA リモート・デバッガーを接続するには、JVM プロファイルでいくつかのオプションを設定する必要があります。

注: JPDA を使用するには、オペレーティング・システム・スレッドのスタック・サイズを大きくすることが必要になる場合があります。動作スレッドのスタック・サイズは、JVM プロファイルでオプション -Xmso を使用して構成できます。

IBM は、Health Center を含む、Java 用のモニターおよび診断ツールを提供しています。 IBM Health Center は、IBM Support Assistant Workbench で使用できます。これらの無料のツールは、IBM ヘルス・センターの入門ガイドに記載されているとおりに、IBM からダウンロードできます。

手順

1. 以下のように、デバッグ・オプションを JVM プロファイルに追加して、JVM をデバッグ・モードで開始します。

   ```
   -agentlib:jdwp=transport=dt_socket,server=y,address=port,suspend=n
   ```

   リモートでデバッガーに接続するための空きポートを選択します。 JVM プロファイルが複数の JVM サーバーで共用される場合、別の JVM プロファイルをデバッグに使用できます。

   注: suspend のデフォルト値は y です。この値は JVM を中断し、処理を続行する前にリモート・クライアント・デバッガーが接続するのを待機します。値 n を指定すると、JVM サーバーが中断されなくなります。

2. Liberty トレースでのホット・スワップの複雑さを避けるには、Liberty JVM サーバーのデバッグを行うときに、以下のプロパティーを JVM プロファイルに追加します。これは、Liberty がデバッグ認識モードで作動しなくてはならないことを Liberty に対して示すことにもなります。

   ```
   -Dwas.debug.mode=true
   -Dcom.ibm.websphere.ras.inject.at.transform=true
   ```

3. デバッガーを JVM に接続します。 接続中にエラーが発生する場合 (ポート値が間違っている場合など)、JVM の標準出力ストリームと標準エラー・ストリームにメッセージが書き込まれます。
4. デバッガーを使用して、JVM の初期状態を確認します。例えば、開始されたスレッドの ID やロードされたシステム・クラスを確認します。
5. 完全 Java クラス名とソース・コード行番号を指定して、Java アプリケーションの適切なポイントでブレークポイントを設定します。デバッガーが、このブレークポイントの活動化が延期されることを示した場合、それはクラスがまだロードされていない可能性があるためです。 JVM を CICS ミドルウェア・コードを使用してアプリケーションのブレークポイントまで実行させてください。このポイントで JVM は再度実行を中断します。
6. ロードされたクラスと変数のソース・コードを調べ、追加のブレークポイントを設定して、必要に応じてコードをステップスルーします。
7. デバッグ・セッションを終了します。アプリケーションを最後まで実行させることができます。その時点でデバッガーと CICS JVM 間の接続はクローズします。一部のデバッガーは、JVM の強制終了をサポートします。その結果、異常終了し、CICS システム・コンソールにエラー・メッセージが表示されます。

CICS JVM プラグイン・メカニズム

JVM の標準 JPDA デバッグ・インターフェースに加えて、CICS Java ミドルウェアには CICS 提供の 1 組の代行受信ポイント (プラグイン) があります。これは、アプリケーションのデバッグに役立つ場合があります。これらのプラグインを使用して、アプリケーション Java コードの実行直前と直後に追加の Java プログラムを挿入できます。

アプリケーションに関する情報 (例えば、クラス名やメソッド名) はプラグインで使用できます。プラグインは JCICS API を使用して、アプリケーションに関する情報を取得することもでき、また、標準 JPDA インターフェースと併用して、CICS 専用の追加デバッグ機能を提供することもできます。プラグインは、CICS ユーザー出口と同様に、デバッグ以外の目的にも使用できます。

Java 出口は、Java プログラムが呼び出される直前と直後に呼び出されるメソッドを提供する CICS Java ラッパー・プラグインです。

プラグインをデプロイするには、そのプラグインを OSGi バンドルとしてパッケージ化します。詳細については、JVM サーバーにおける OSGi バンドルのデプロイを参照してください。

2 つの Java プログラミング・インターフェースが提供されます。

両方のインターフェースが `com.ibm.cics.server.jar` で提供され、Javadoc で文書化されます。

Java プログラミング・インターフェースは次のとおりです。

- DebugControl: `com.ibm.cics.server.debug.DebugControl`。このプログラミング・インターフェースは、ユーザーによって提供される実装に対して行うことができるメソッド呼び出しを定義します。
- Plugin: `com.ibm.cics.server.debug.Plugin`。これは、プラグイン実装の登録に使用する汎用プログラミング・インターフェースです。

以下は DebugControl インターフェースの例です。

```
public interface DebugControl
{
    // called before an application object method or program main is invoked
    public void startDebug(java.lang.String className,java.lang.String methodName);

    // called after an application object method or program main is invoked
    public void stopDebug(java.lang.String className,java.lang.String methodName);

    // called before an application object is deleted
    public void exitDebug();

}
public interface Plugin
{
    // initaliser, called when plug-in is registered
    public void init();
}
```

以下は DebugControl および Plugin インターフェースの実装例です。

```
import com.ibm.cics.server.debug.*;

public class SampleCICSDebugPlugin
    implements Plugin, DebugControl
{
    // Implementation of the plug-in initialiser
    public void init()
    {
        // This method is called when the CICS Java middleware loads and
        // registers the plug-in. It can be used to perform any initialization
        // required for the debug control implementation.
    }

    // Implementations of the debug control methods
    public void startDebug(java.lang.String className,java.lang.String methodName)
    {
        // This method is called immediately before the application method is
        // invoked. It can be used to start operation of a debugging tool. JCICS
        // calls such as Task.getTask can be used here to obtain further
        // information about the application.
    }

    public void stopDebug(java.lang.String className,java.lang.String methodName)
    {
        // This method is called immediately after the application method is
        // invoked. It can be used to suspend operation of a debugging tool.
    }

    public void exitDebug()
    {
        // This method is called immediately before an application object is
        // deleted. It can be used to terminate operation of a debugging tool.
    }

    public static void main(com.ibm.cics.server.CommAreaHolder ca)
    {
    }
}
```

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。この資料の他の言語版を IBM から入手できる場合があります。ただし、これを入手するには、本製品または当該言語版製品を所有している必要がある場合があります。

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。 IBM 製品、プログラムまたはサービスに代えて、IBM の知的所有権を侵害することのない機能的に同等のプログラムまたは製品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。

〒*103-8510*
東京都中央区日本橋箱崎町*19*番*21*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

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、お客様自身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。

© Copyright IBM Corp. 1999, 2018

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 Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119 Armonk,
NY 10504-1785
United States of America

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、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。

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。

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。これらの名前はすべて架空のものであり、類似する個人や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。これらのサンプル・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、お客様の当該サンプル・プログラムの使用から生ずるいかなる損害に対しても一切の責任を負いません。

プログラミング・インターフェース情報

CICS には、プログラミング・インターフェースと見なすことのできる資料と、プログラミング・インターフェースと見なすことのできない資料があります。

オンライン製品資料の以下のセクションには、CICS Transaction Server for z/OS, バージョン 5 リリース 5 のサービスを取得するプログラムをお客様が作成するためのプログラミング・インターフェースが含まれています。

- アプリケーションの開発
- Developing system programs
- 保護の概要
- 外部インターフェースに向けた開発
- リファレンス: アプリケーション開発h
- リファレンス: システム・プログラミング
- リファレンス: 接続

オンライン製品資料の以下のセクションには、CICS Transaction Server for z/OS, バージョン 5 リリース 5 のプログラミング・インターフェースとして意図されていない (プログラミング・インターフェースと誤解される可能性のある) 情報が含まれています。

- Troubleshooting and support
- リファレンス: 診断

PDF 形式のマニュアルで CICS 資料にアクセスする場合は、CICS Transaction Server for z/OS, バージョン 5 リリース 5 のサービスを取得するプログラムをお客様が作成するためのプログラミング・インターフェースが以下のマニュアルに含まれています。

- アプリケーション・プログラミング・ガイドおよびアプリケーション・プログラミング・リファレンス
- Business Transaction Services
- Customization Guide
- C++ OO Class Libraries
- Debugging Tools Interfaces Reference
- Distributed Transaction Programming Guide
- External Interfaces Guide
- Front End Programming Interface Guide
- IMS Database Control Guide
- インストール・ガイド
- セキュリティー・ガイド
- Supplied Transactions
- CICSPlex SM Managing Workloads
- CICSPlex SM Managing Resource Usage
- CICSPlex SM アプリケーション・プログラミング・ガイドおよび CICSPlex SM アプリケーション・プログラミング・リファレンス
- Java Applications in CICS

PDF 形式のマニュアルで CICS 資料にアクセスする場合は、CICS Transaction Server for z/OS, バージョン 5 リリース 5 のプログラミング・インターフェース

として意図されていない (プログラミング・インターフェースと誤解される可能性のある) 情報が以下のマニュアルに含まれています。

- Data Areas
- Diagnosis Reference
- Problem Determination Guide
- CICSPlex SM Problem Determination Guide

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 Corporation の商標または登録商標です。他の製品名およびサービス名等は、それぞれ IBM または各社の商標である場合があります。現時点での IBM の商標リストについては、 http://www.ibm.com/legal/copytrade.shtml 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。

インテル、Intel、Intel ロゴ、Intel Inside、Intel Inside ロゴ、Intel Centrino、Intel Centrino ロゴ、Celeron、Intel Xeon、Intel SpeedStep、Itanium、および Pentium は、Intel Corporation または子会社の米国およびその他の国における商標または登録商標です。

Java およびすべての Java 関連の商標およびロゴは Oracle やその関連会社の米国およびその他の国における商標または登録商標です。

Linux は、Linus Torvalds の米国およびその他の国における登録商標です。

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。

製品資料に関するご使用条件

これらの資料は、以下のご使用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。

適用範囲
: IBM Web サイトの「ご利用条件」に加えて、以下のご使用条件が適用されます。

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。

商用使用
: 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。

権利　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 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 オンラインでのプライバシー・ステートメント

サービス・ソリューションとしてのソフトウェアも含めた IBM ソフトウェア製品 (「ソフトウェア・オファリング」) では、製品の使用に関する情報の収集、エンド・ユーザーの使用感の向上、エンド・ユーザーとの対話またはその他の目的のために、Cookie はじめさまざまなテクノロジーを使用することがあります。多くの場合、ソフトウェア・オファリングにより個人情報が収集されることはありません。IBM の「ソフトウェア・オファリング」の一部には、個人情報を収集できる機能を持つものがあります。ご使用の「ソフトウェア・オファリング」が、これらの Cookie およびそれに類するテクノロジーを通じてお客様による個人情報の収集を可能にする場合、以下の具体的事項をご確認ください。

CICSPlex SM Web ユーザー・インターフェース **(**メイン・インターフェース**)** の場合**:**　このソフトウェア・オファリングは、展開される構成に応じて、セッション管理、認証、お客様の利便性の向上、または利用の追跡または機能上の目的のために、それぞれのお客様のユーザー名、およびその他の個人情報を、セッションごとの Cookie および持続的な Cookie を使用して収集する場合があります。これらの Cookie を無効にすることはできません。

CICSPlex SM Web ユーザー・インターフェース **(**データ・インターフェース**)** の場合**:**　このソフトウェア・オファリングは、展開される構成に応じて、セッション管理、認証、または利用の追跡または機能上の目的のために、それぞれのお客様のユーザー名またはその他の個人情報を、セッションごとの Cookie を使用して収集する場合があります。これらの Cookie を無効にすることはできません。

CICSPlex SM Web ユーザー・インターフェース **(**「**Hello World**」ページ**)** の場合**:**　このソフトウェア・オファリングは、展開される構成に応じて、個人情報を収集しないセッションごとの Cookie を使用する場合があります。これらの Cookie を無効にすることはできません。

CICS Explorer の場合**:**

このソフトウェア・オファリングは、展開される構成に応じて、セッション管理、お客様の利便性の向上、または利用の追跡または機能上の目的のために、それぞれのお客様のユーザー名、およびその他の個人情報を、セッションごとの設定および持続的な設定を使用して収集する場合があります。これらの設定を無効にすることはできませんが、ユーザー・パスワードの暗号化形式でのディスクへの保管は、サインオン中にチェック・ボックスにチェック・マークを付けることによるユーザーの明示的な操作によってのみ有効化することができます。

この「ソフトウェア・オファリング」が Cookie およびさまざまなテクノロジーを使用してエンド・ユーザーから個人を特定できる情報を収集する機能を提供する場合、 お客様は、このような情報を収集するにあたって適用される法律、ガイドライン等を遵守する必要があります。これには、エンドユーザーへの通知や同意の要求も含まれますがそれらには限られません。

このような目的での Cookie を含む様々なテクノロジーの使用の詳細については、『IBM オンラインでのプライバシー・ステートメント』(http://www.ibm.com/privacy/details/jp/ja/) の『クッキー、ウェブ・ビーコン、その他のテクノロジー』および「IBM Software Products and Software-as-a-Service Privacy Statement」(http://www.ibm.com/software/info/product-privacy) を参照してください。

索引

日本語, 数字, 英字, 特殊文字の順に配列されています。なお, 濁音と半濁音は清音と同等に扱われています。

[ア行]

アクセス制御リスト (ACL) 188
アプリケーション
　更新 245
　OSGi 16
アプリケーションのプロファイル作成 296
アプリケーション・プログラム、Java 43
一時データ・キュー CSJO および CSJE 336
エラーと例外
　JCICS 44
エンクレーブ・ストレージ 308
エンコード 48

[カ行]

ガーベッジ・コレクション
　JVM server (JVM サーバー) 305
開発
　制約事項 75
　ベスト・プラクティス 42, 76
開発環境 79
開発者ツールのインストール 79
回復
　バンドル・リソース 258
概要
　OSGi 3
カスタマイズ
　JVM プロファイル 191
既存の JAR ファイルからの OSGi プラグイン・プロジェクトの作成 169
既存の Java プロジェクトのプラグイン・プロジェクトへの変換 168
既存のバイナリー JAR ファイルからの OSGi プラグイン・プロジェクトの作成 172
共通ライブラリー
　デプロイ
　　Liberty 181
共用クラス・キャッシュ
　定義 8
共用ライブラリー領域 12
グループ ID (GID) 188
計画 1, 16
コード・ページ 48
更新
　OSGi バンドル 245
構成
　Axis2 213
　CICS セキュリティー・トークン・サービス 215
　Liberty JVM サーバー 209
　OSGi フレームワーク 192
コンテナー
　作成 57
　JCICS サポート 56
コンテナー・プラグイン、Java アプリケーションのデバッグ用 341

[サ行]

サンプル
　チャネルとコンテナー 59
サンプル JVM プロファイル 8
時間帯 244
　記号 244
出力リダイレクト
　サンプル 336
新規 247, 248
ストレージ 187
スレッド 46
　JVM サーバー 257
スレッド管理 12
スレッドとタスク
　JCICS サポート 69
スレッドの管理 12
制限 75
制約事項 75
セキュリティー 128
　CICS デフォルト Web アプリケーション 200
セキュリティー・ポリシーの適用 291
セキュリティー・ポリシーの有効化 291
セキュリティー・マネージャー
　セキュリティー・ポリシーの適用 291
　セキュリティー・ポリシーの有効化 291

[タ行]

ターゲット・プラットフォーム 33
大容量の COMMAREA 56
大容量の COMMAREA としてのチャネル 56
タスク管理 12
チャネル
　作成 57
　JCICS サポート 56
調整
　Java 295
　JVM server (JVM サーバー) 299
直列化可能クラス、JCICS 45
ツール 296
データベースへのアクセス 148
データ・ソース 209
デフォルト Web アプリケーション
　Liberty 200
デプロイ
　共通ライブラリー
　　Liberty 181
　JVM サーバーへのアプリケーション 175
トレース 127

[ハ行]

バイト配列の処理 48
始めに 1
バッチ・モード JVM 166
パフォーマンス
　アプリケーションの分析 296
　Java 295
　JVM server (JVM サーバー) 299
バンドル 33
バンドル・リカバリー 258
ヒープ拡張 305
プールされた JVM
　JVM サーバーへの移動 254
プールされた JVM から JVM サーバーへの移動 254
複数のスレッド 46
プラグイン
　CICS JVM における
　　概要 341
　　コンテナー・プラグイン 341
　　ラッパー・プラグイン 341
　　DebugControl インターフェース 341
　　Plugin インターフェース 341
プラグイン・プロジェクトの作成 34
ベスト・プラクティス
　開発 42, 76

[マ行]

マッピング 43

© Copyright IBM Corp. 1999, 2018

ミドルウェア・バンドル
更新 252

［ヤ行］

ユーザー ID (UID) 188

［ラ行］

ラッパー・プラグイン、Java アプリケーションのデバッグ用 341
リソース定義
JCICS の 45
リソース・アダプター 116
リンク
OSGi サービス 182
例 49

［ワ行］

割り振り失敗 305

［数字］

32 K 以上の COMMAREA 56
32 K 超の COMMAREA 56

A

Axis2
構成 213

C

CEEPIPI Language Environment 事前初期設定モジュール 11
CICS Explorer SDK
Java アプリケーションの開発 33
CICS での Java プログラミング
データベースへのアクセス 148
JCICS の使用 43
インターフェース 44
エラーと例外 44
クラス 44
スレッド 46
直列化可能クラス 45
引数 45
JavaBeans 43
JCICS コマンド解説 49
JCICS ライブラリー構造 44
PrintWriter 46
Task.err 46
Task.out 46
CICS の非 OSGi Java プロジェクトの配置 184
CICS バンドル 33
CICS-MainClass 宣言のマニフェストへの追加 36
com.ibm.cics 200
com.ibm.cics.jvmserver 200
com.ibm.cics.samples.SJMergedStream 336
com.ibm.cics.samples.SJTaskStream 336
CSJE 一時データ・キュー 336
CSJO 一時データ・キュー 336

D

Db2 アクセスの構成 191
DebugControl インターフェース、Java アプリケーションのデバッグ用 341
DFHAXRO 306, 308
DFHJVMAX JVM プロファイル 9
DFHJVMAX プロファイル 191
DFHJVMCD JVM プロファイル 186
DFHJVMPR JVM プロファイル 186
DFHJVMST JVM プロファイル 9
DFHOSGI JVM プロファイル 9
DFHOSGI プロファイル 191
DFHWLP JVM プロファイル 9
DFHWLP プロファイル 191

E

EAR ファイル 180
ECI 116

G

GID 188

I

IBM Health Center 296
IPIC 接続 124, 125, 127

J

Java
パフォーマンス 295
Java EE アプリケーションの更新 255
Java EE アプリケーションへのアクセスの制御 85
Java Message Service 107
Java アプリケーションの開発 33
Java アプリケーションの接続性 166
Java アプリケーションのデプロイ 33
Java オプション
記号 220
Java セキュリティー・マネージャー 291
Java ツール 296
Java でのプログラミング 43
Java の CICS タスク 12
Java の開発
CICS Explorer SDK 33
javadoc 167
java.security.policy 291
JCA 124, 125, 127, 128, 167
チャネル 120
トレース 122
リソース・アダプター 115, 122, 123
CCI 113, 115, 121, 122, 123
ECI 116, 120, 121, 122
JCAServlet 125, 127
JCICS
異常終了 54
一時記憶 70
インターフェース 44
エラー処理 54
エラーと例外 44
クラス 44
クラス・ライブラリー 43
現行チャネルの受け取り 58
現行チャネルのブラウズ 58
コマンド解説 49
コンテナーからのデータの取得 58
コンテナーの作成 57
条件処理 50
診断サービス 59
ストレージ・サービス 69
スレッドとタスク 69
スレッドの使用 46
端末管理 71
チャネルとコンテナー 56
チャネルの作成 57
直列化可能クラス 45
引数 45
ファイル制御 63
プログラム制御 67
プログラム例 59
変換
データから XML へ 71
XML からデータへ 71
ライブラリー構造 44
リソース定義 45
例外処理 49, 51
例外マッピング 52
ABEND 処理 49, 51
ADDRESS 60
APPC 55
BMS 55
CANCEL コマンド 68
DEQ コマンド 69

JCICS (続き)
DOCUMENT サービス 59
ENQ コマンド 69
HANDLE コマンド 50
HTTP サービス 66
INQUIRE SYSTEM 62
INQUIRE TASK 62
INQUIRE TERMINAL または NETNAME 63
JavaBeans 43
Javadoc 43
JCICS Class Reference 43
PrintWriter 46
RETRIEVE コマンド 68
START コマンド 68
Task.err 46
Task.out 46
UOW 73, 96, 108
Web サービス 74
JCICS エンコード 48
JCICS を使用した Java の開発
概要 43
JDBC 191
JMS 107
JMS クライアント 107
JVM 185, 245
インストール 8
クラス 9
アプリケーション 9
システムまたは原始 9
クラスパス 9
標準 (CLASSPATH_PREFIX、CLASSPATH_SUFFIX) 9
ライブラリー・パス 9
構造 9
出力リダイレクト
サンプル 336
使用 245
ストレージ・ヒープ 11
セットアップ 185
調整 305, 306
デバッグ 321
トレース 321
ネイティブ・ライブラリー 9
ヒープ 11
プラグイン、Java アプリケーションのデバッグ用 341
問題判別 321
DFHAXRO 306
JVM プロファイル 8, 186
JVMPROFILEDIR システム初期設定パラメーター 186
Language Environment エンクレーブ 11, 306
z/OS 共用ライブラリー領域 12
JVM profile directory (JVM プロファイル・ディレクトリー) 186
JVM server (JVM サーバー) 31
ガーベッジ・コレクション 305
からの移動、プールされた 254
新規 OSGi バンドル 247, 248
スレッド 257
セットアップ 191
パフォーマンス 299
ヒープ拡張 305
ミドルウェア・バンドルの更新 252
割り振り失敗 305
Axis2 の構成 213
CICS セキュリティー・トークン・サービスの構成 215
Java EE アプリケーション 255
Language Environment エンクレーブ 308
Liberty の構成 209
OSGi の構成 192
OSGi バンドルの更新 247, 250
OSGI バンドルの除去 253
JVM からの出力のリダイレクト
サンプル 336
JVM サーバー
デプロイ先 175
ベスト・プラクティス 42, 76
OSGi サービス 182
OSGi バンドルのインストール 175
WAR ファイルのデプロイ 180
JVM サーバーの作成 191
JVM サーバーのスレッドの制限 257
JVM サーバーのセットアップ 191
JVM システム・プロパティー 8
JVM におけるクラスのタイプ 9
JVM のクラスパス 9
JVM のトレース 321
JVM の問題判別 321
JVM プロパティー・ファイル 8
JVM プロファイル 8
位置指定 186
オプション 216
検証 216
選択 8
大/小文字の考慮事項 186
プロパティー 216
CICS によって提供されるサンプル 8
DFHJVMAX 9, 191
DFHJVMCD 186
DFHJVMPR 186
DFHJVMST 9
DFHOSGI 9, 191
DFHWLP 9, 191
JVMPROFILEDIR 186
JVM プロファイル・オプション USEROUTPUTCLASS、出力リダイレクト 335
JVM 用のシステム初期設定パラメーター JVMPROFILEDIR 186
JVMPROFILEDIR システム初期設定パラメーター 186
jvmserver 200

L

Language Environment 308
Language Environment エンクレーブ、JVM 用 306
Liberty 19, 79, 180, 210
JVM server (JVM サーバー) 255
Liberty JVM サーバー 180
構成 209
Liberty のデフォルト・アプリケーション セキュリティー 200
Liberty プロファイル 191

M

MOM 107

O

OSGi 76
OSGi Service Platform 3
OSGi サービス
呼び出し 182
OSGi セキュリティー 291
OSGi バンドル 33
インストール 175
更新 247, 250
除去 253
段階的な導入 247, 248
OSGi バンドルのデプロイ 175
OSGi フレームワーク
構成 192

P

Plugin インターフェース、Java アプリケーションのデバッグ用 341
plugin-cfg 134
POJO 3

S

SAML
構成 215
SQLJ 191

SSL 128

T

TCPIPSERVICE 124
traceRequests 127
TZ 244

U

UID 188
UNIX システム・サービスのアクセス 188
UNIX ファイル・アクセス 188
USEROUTPUTCLASS JVM プロファイル・オプション 335

W

WAR ファイル 180
  インストール 180
WAR ファイルのデプロイ 180
Web サーバー 134
Web サーバー・プラグイン 134
WebSphere Developer Tools 125, 127
WebSphere MQ classes for Java
  OSGi JVM サーバー
    構成 195
    UOW のコミット 166
WebSphere MQ classes for JMS
  OSGi JVM サーバー
    構成 194
    プログラミング 161

Z

z/OS 共用ライブラリー領域 12

［特殊文字］

-Xinitsh 11
-Xms 11
-Xmx 11

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