Webシステムにおける暗黙的非同期処理アーキテクチャー

小宮 聖則　今井 博一　齋藤 幸二郎

Implicit Asynchronous Processing Architecture for Web-based Systems

Kiyonori Komiya, Hirokazu Imai and Kohjiroh Saitoh

Webシステムにおいて十数秒を越えるような処理時間のかかるトランザクションは, 1回のHTTP要求/応答で完結させず, 非同期処理として実装すべきである. それは高負荷時のシステム資源の枯渇を防止し, スループットを維持させるためである. しかしながら非同期処理の実装は複雑で, 開発の負担は大きい. 本論文では, Webシステムにおける非同期処理方式として『暗黙的非同期処理』と名づけたアーキテクチャーを提案する. それは, 同期型で開発した処理を実行時に非同期処理化させるアーキテクチャーである.『暗黙的非同期処理』は, 周知のデザイン・パターンや一般的なシステム構成で実現可能なものである.

Generally, long running web transactions, such as more than 10 seconds, should be implemented with asynchronous processing instead of synchronous processing. However, the development of asynchronous processing in web systems requires highly skilled developers because of the complexity of the system and programming environment. The paper proposes an "Implicit Asynchronous Processing Architecture" for web based systems. This architecture enables the running of "synchronous" web transactions asynchronously. In other words, the developer is able to build asynchronous web transactions without thinking about asynchronous processing or programming techniques. This architecture is based on well-known architecture and design patterns.

Key Words & Phrases：非同期処理, Webシステム, Javaバッチ, コンポーネント化
asynchronous processing, web based systems, Java batch environment, component

1. はじめに

近年のWeb技術の向上とWebコンテンツのリッチ・クライアント化により, GUIを有するフロントエンド・システムはWebベースのシステム開発が主流となった. 企業内においてもフロントエンド・システムの大半がクライアント・サーバー型（以降C/S型と記述）などからイントラネットのWebシステムへ移行されてきている. イントラネット・システムは, 金融機関の基幹システムに見られるように, 膨大なデータ検索や複雑なビジネス・ロジックの実行, そしてバックエンドのレガシー・システム連携が必要な場合が多く, HTTP処理時間が十数秒を越えるような場合もある. 特に, ファット・クライアントで重いSQL処理が多く実装されてきたPCアプリケーションとデータベース（以降DBと記述）・サーバーのC/S型システムをWebシステムに移行する場合に多く見られる.

提出日：2007年3月16日　　再提出日：2007年7月17日

Web技術の基盤となっているHTTPプロトコルは, 本来ブラウザーとサーバー間でHTMLテキスト文書を送受信するものであり, ブラウザーから要求（HTTPリクエスト）をサーバーに送付し, 結果（HTTPレスポンス）を通常はHTML文書として受信する同期処理のプロトコルである［1］. 従って, 長時間を要する処理には本来向かず, それらは非同期処理として実装すべきである. しかし一般的には同期処理として設計/実装するWebアプリケーションを非同期処理とすることは決して容易ではなく, 開発者への負担が大きい.

本論文は, Webシステムにおける非同期処理開発の課題を解決するために考案した『暗黙的非同期処理』と名付けたアーキテクチャーを紹介する. それは, 同期処理として設計/開発したビジネス・ロジックを実行時に非同期処理させる技術である.

以下, 2章でWebシステムにおける非同期処理の重要性と課題について説明し, 3章でその課題を解決するための『暗黙的非同期処理』の実現法を紹介する. そして4章でその効果と考慮点を述べる.

2．Web非同期処理の重要性と課題

本章では，銀行のイントラネットWebシステムの例を挙げて，処理時間が長くなりうる現状を示す．そして，Webシステムでの非同期処理の重要性について説明する．

2.1　イントラシステムのトランザクション時間

第一章で述べたとおり，企業内C/S型システムをWebシステム化する場合，サーバー・サイドの処理時間を考慮する必要がある．銀行の稟議システムの場合を例として，処理時間の長いトランザクションを挙げてみる．

(1) 稟議案件作成

1000を超える大量の動的SQL処理や複雑な計算ロジックを経て稟議案件の作成画面を表示し，稟議書の作成工程に入る．案件の規模にもよるが，明細の多いものでは十数秒以上の処理時間がかかる．

(2) 帳票生成

稟議書データ相当のDBレコードを検索・抽出し，帳票（PDF）生成を行い，ブラウザーにダウンロードする．帳票の複雑度や案件の規模によるが，これも十数秒以上かかりうる．

その他各種計算処理，稟議案件登録等，大量のレコード検索・更新処理やホスト連携処理などにより，数秒を上回る処理は多い．これら長い処理について，1回のHTTPトランザクションで完結させる実装はWebシステムとして危険である．

2.2　処理時間の長いHTTP要求の危険性

C/S型の場合は，十数秒以上かかるような処理時間であってもユーザビリティーの低下を犠牲にすれば，システム的に問題となる場合は少ない．それは，基本的にはクライアント側の処理性能，DBサーバー負荷および複数タスクが同時並行で動作するDBシステムとしての行/表ロックの考慮により対応できるからである．一方Webシステムの場合，HTTP処理中はシステム資源（アプリケーション・サーバー内で並行処理を行うスレッド，DB接続等）が一部占有されるため，長時間のトランザクションがピーク時に大量に投入されると，処理遅延やシステム障害の発生原因にもなる［2］．このような危険を回避するため，実行時間の長い処理は非同期処理での実装を考慮することが重要である．

2.3　一般的な非同期処理実装方式と課題

一般的なWebシステムにおける非同期処理実現方式を以下に列挙する．

(1) J2EE/JSR（Java™ Specification Request）機能による非同期プログラミング・モデルの利用［3］［4］
- J2EE MDB（Message Driven Bean）
- JSR 237 Work Manager（J2EE環境のスレッド・プログラミング・モデル）
- J2EE EJB（Enterprise Java Beans）Timer サービス
- JSR 236 Timer

これらの実装技術の詳細についてはそれぞれの参考文献（［3］［4］）を参照されたい．

これらの技術はJ2EE仕様に準拠し，高可用性の効率的な並列処理を実装することが可能である．また高度なネットワーク・プログラミング技術は不要であり，特にMDBの場合はマルチスレッドのライフサイクル管理をEJBコンテナーが行うため，基本的なマルチスレッドの動作原理を理解していれば実装は可能である．

しかしながら，業務開発者はJMSデザイン・パターンやこれら技術のAPIなどの習熟が必要であり，環境構築においても非同期処理を意識した設計が必要となり，考慮点は多い．

(2) 製品/IPアセットの利用
- WebSphere® XD J2EEバッチ［5］
- JBeX（Batch Execution and Control Environment for Java）

WebSphere XD は，IBMのJ2EE アプリケーション・サーバー製品である WebSphere Application Server に，Javaバッチ実行環境や高度なサーバー資源管理/負荷分散機能が拡張された製品である［5］．

JBeXはJavaによるバッチ実行管理の基盤機能とバッチ処理プログラムの実行制御用APIを提供するフレームワークで，IBMが提供する有償ソフトウェア資産（IPアセット）のサービス・コンポーネントである．

これらも（1）と同じくJ2EEアプリケーション・サーバーで動作するJavaバッチの実行環境である．製品として実行環境が提供されるため，その仕様が確定しており，環境面の設計，構築に関しての考慮点は少ない．

業務開発の観点からは，バッチ処理を実装することを意識する必要があり，実行方法など製品に対する習熟が必要になる．

(3) 独自実装
- DB連携オンライン・バッチ方式
- その他MOM（Message Oriented Middleware）等を利用した独自方式

“DB連携オンライン・バッチ方式”は，ブラウザーからのHTTP要求に応じてDBにバッチ処理要求を書き込む．その処理要求は，常駐型もしくは定期起動型バッチ・ジョブが認識し，実行する典型的な方式である．またMOM（IBM WebSphere MQSeries®など）を使用

することにより, メッセージ駆動型アプリケーションを作成することが可能である.

実績のあるDBやMOMを使用するため, 過去に開発した資産を活用しやすく, 製品面での信頼性も高い. 反面, 製品に対する専門技術を必要とし, また独自仕様のために他環境との親和性も低い.

上記(1)～(3)いずれかの技術/方式を利用すれば, Webシステムにおいても非同期処理は確かに実現可能である. しかしながら, これらの実装方式には以下の課題がある.

課題1:オンライン処理ノードとバッチ・ジョブ実行ノードの分離

長い処理時間のために非同期化する処理は, 通常, CPUやメモリ資源を多く消費する. 従って, オンライン処理とは別のバッチ専用ノードで稼働させる必要がある. J2EE/JSR方式のスレッドおよびTimer系のプログラミング・モデルを採用する場合は, 別途メッセージング・エンジンや, リモートEJB呼び出し等と組み合わせて実現する必要がある.

課題2:開発者の必要スキル・レベルの高さ

いずれの方式も, 開発者は非同期処理であることを意識して開発する必要がある. 本来ビジネス・ロジックに専念すべき業務アプリケーション開発者が, 非同期処理のプログラミング・モデルや製品機能についての高度なスキルが必要となる. 従って, 実際のプロジェクトで採用する場合は開発者の教育や確保について十分考慮しなければならない.

課題3:開発/テストの生産性が低い

課題2にも関連するが, 非同期処理の開発は, 本処理から派生する非同期処理の起動のみならず, 本処理と非同期処理との待ち合わせやエラー処理などが複雑で難易度が高い. また, 実行環境にも依存しており, 開発/テストの生産性は低い.

課題4:ユーザビリティーの低下

非同期処理との待ち合わせ画面や処理状況照会画面が必要となる. 展開する画面数が増えることにより, ユーザビリティーの低下に繋がる.

筆者は, 非同期処理開発におけるこれらの課題を解決するための『暗黙的非同期処理』アーキテクチャーを考案した. ここで言う『暗黙的』とは, 同期処理として開発したHTTPトランザクションが, 実行時に暗黙のうちに非同期処理されることを意味している. つまり, 業務アプリケーション開発者が, HTTPトランザクションの非同期処理の実装を意識せずに, 実行時にフレームワーク機能により非同期処理されることである.

3. 暗黙的非同期処理の実現

本章では『暗黙的非同期処理』実現のための基本的なシステム要件を示し, アプリケーション構造図, オペレーショナル・モデルによるウォークスルー, そしてアプリケーション・アーキテクチャーについて説明する. **図1**に, 前章で提示した課題に対する解決策として本章で説明する内容を総括しておく. なお, システム・アーキテクチャー, アプリケーション・アーキテクチャーにて提示している個々の要素技術やアーキテクチャーは, 全て実装/実証済みであり, 暗黙的非同期処理への応用はフレームワーク機能の一部拡張のみによって実現可能である.

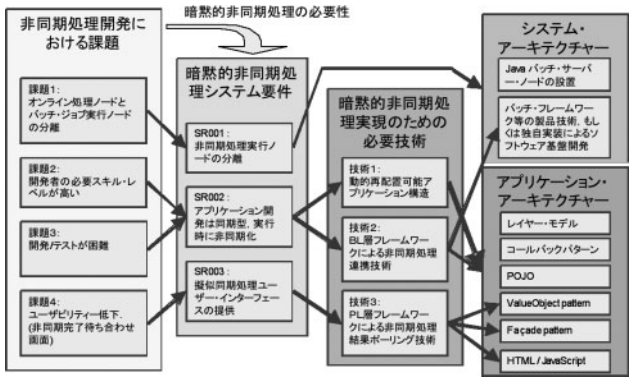

図1. 非同期処理開発における課題と解決策

3.1 暗黙的非同期処理システム要件

SR001:非同期処理実行ノードの分離

『暗黙的非同期処理』は, オンライン処理トランザクションへの影響を与えないため, バッチ処理専用ノードで実行すること.

SR002:アプリケーション開発は同期型

業務アプリケーションは非同期処理であることを意識しない実装が可能であり, 単体テスト(以降UTと記述)は同期処理として実行可能なこと. 開発者に要求されるスキル・レベルの軽減と開発生産性向上のためである.

SR003:擬似同期処理ユーザー・インターフェースの提供

エンドユーザーから見たユーザー・インターフェースでは, 非同期処理されていることを意識せず, 同期処理されているように見えること. 同期処理と同じユーザビリティーを維持するためである.

3.2 アプリケーション構造と基本技術

前節に示したシステム要件を満たすため, まず, **図**

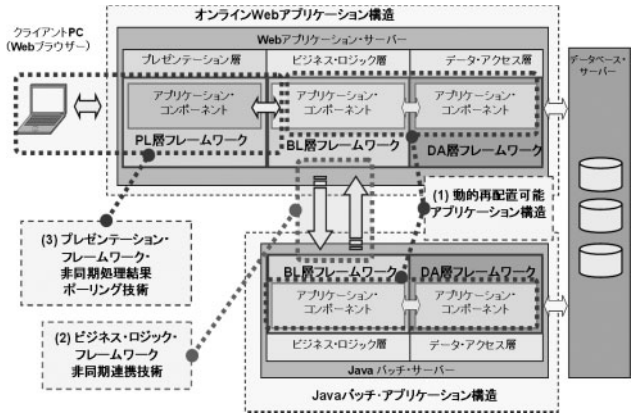

図2.　アプリケーション構造図

2に示すようにアプリケーションはレイヤー構造とし, コンポーネント化する. さらに, 以下の3つの基本技術が必要である.

(1)動的再配置可能アプリケーション構造
（業務開発者が意識しなければならないアプリケーション構造）
(2)ビジネス・ロジック層フレームワークによる非同期処理連携技術
(3)プレゼンテーション層フレームワークによる非同期処理結果ポーリング技術

これら3つの技術は, 周知の枯れた要素技術で実現できる. 次節において, IBMのシステムアーキテクチャー・モデリング手法であるIBM Global Service Method（以下 GS Methodと記述）のオペレーショナル・モデル[6]にて, その3つの技術を取り込み説明する.

3.3　オペレーショナル・モデル

説明を容易にするため, 主要なサーバー・ノード・コンポーネントとそのノード上での機能を実現する配置ユニット（DU: Deployment Unit）を示した仕様レベルのオペレーショナル・モデルを**図3**に示す.

主な配置ユニット（以降DUと記述）については図中の説明を参照してほしいが, Javaバッチ・フレームワーク（E_BAT_FW）について補足する.

E_BAT_FWは, ビジネス・ロジックフレームワーク（E_BL_FW）と連携してビジネス・ロジック層の非同期処理連携機能を実現する. E_BL_FWの実装方式は, 前章『2.3 一般的な非同期処理実装方式と課題』で提示した各種方式および組み合わせにより実装可能であるが, ここではIPアセットのJBeXを製品コンポーネントとして配置している. JBeXは, ジョブ・クライアント側（Webアプリケーション・サーバー）から, Javaバッチ・サーバーで稼動するジョブの投入（submit）や状況取

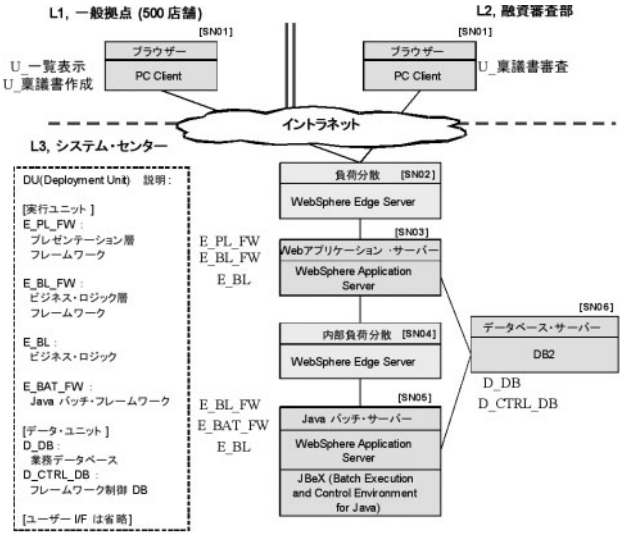

図3.　オペレーショナル・モデル

得（query）を行うAPIを提供する.

ウォークスルーの最初として, 同期処理の動的モデルを示す（**図4**）. なお, 負荷分散ノードについては説明を簡潔にするため, ここでは省略している. 次の2つのステレオタイプを使用している.

<<Framework>> フレームワーク機能であり, 特定業務に依存しない業務基盤機能のコンポーネント.
<<業務>> 業務担当者の開発するアプリケーション・コンポーネント.

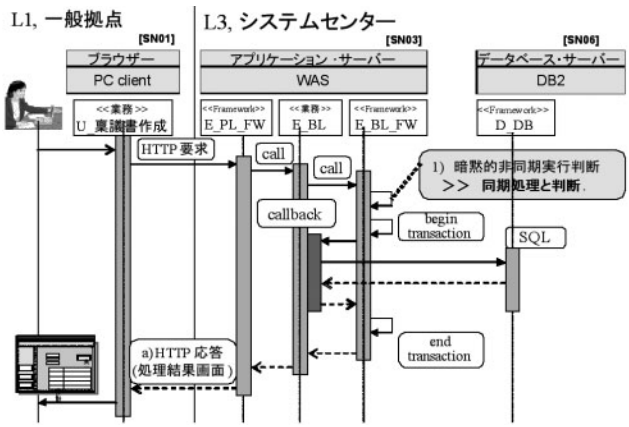

図4.　ウォークスルー ― 同期処理モデル

図4より分かるとおり, プレゼンテーション・ロジック（以降PLと記述）層フレームワーク（E_PL_FW）とE_BL_FWの間にビジネス・ロジック（E_BL）が介在する形になっており, 両フレームワークの依存関係は低い. さらに, E_BLが, E_BL_FWからコールバックされる形式になっており, トランザクションと暗黙的非同期処理の制御がE_BL_FWで可能となっている. ここでは暗黙的非同期処理の実行判断を行った結果, 同期処理として判断している.

図5では，アプリケーション・サーバー（SN03）のE_BL_FWが，暗黙的非同期処理の実行可否を判断し，E_BAT_FWを利用してJavaバッチ・サーバー上で稼動するジョブを投入している．その際，E_BLのプロパティーを直列化（Serialize）[7]し，制御DB（D_CTRL_DB）に格納している（図中2，3）．

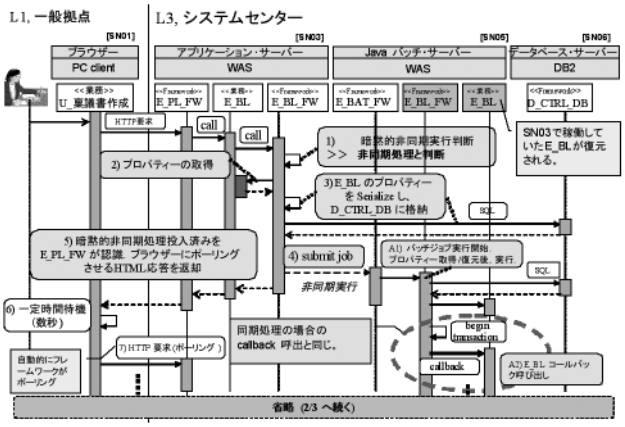

図5.　ウォークスルー ― 非同期処理モデル（1/3）

Javaバッチ・サーバー（SN05）では，非同期ジョブが起動され，D_CTRL_DBから直列化されたE_BLのプロパティーを復元し，E_BLのコールバックを呼び出している．このようにしてビジネス・ロジックの動的再配置を実現している．

一方，ジョブを起動したアプリケーション・サーバー（SN03）では，E_PL_FWが暗黙的非同期で起動された状態であることを認識し，ブラウザーから結果待ちのポーリングをさせるHTML応答を返却する（5）．ポーリングをさせるHTML応答は，Webページ描画後，一定時間後に再度HTTP要求を送付するHTMLのメタ・タグやJavaScriptによる技術で実現可能である．画面には，進行状況を表示することも可能である．

ポーリング開始後の暗黙的非同期処理終了待ち，そして非同期ジョブの実行終了の様子は**図6**，**図7**中

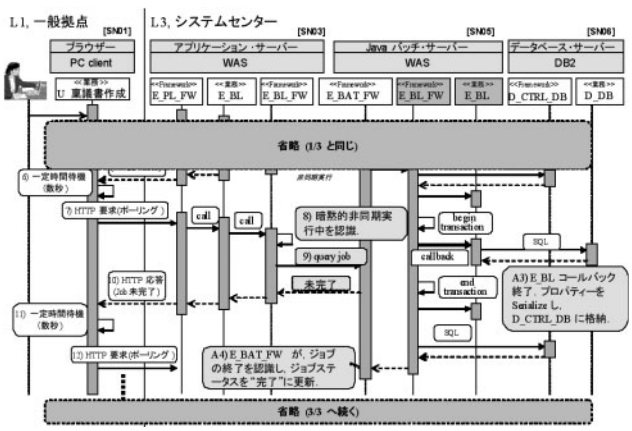

図6.　ウォークスルー ― 非同期処理モデル（2/3）

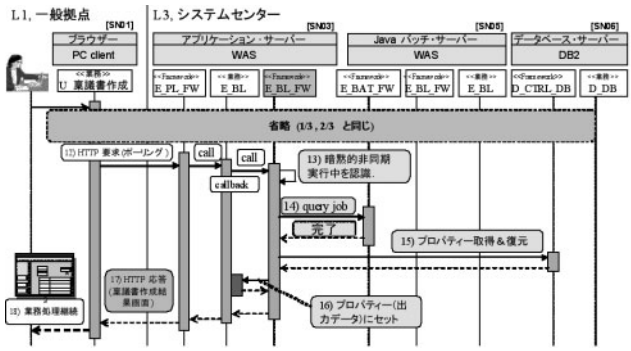

図7.　ウォークスルー ― 非同期処理モデル（3/3）

の説明を参照されたい．

図5～図7の暗黙的非同期処理のウォークスルーで分かるとおり，非同期処理の制御部分はE_BL_FWとE_BAT_FWの連携処理で実装され，E_BLは一切関与していない．つまり，アプリケーションが同期処理として開発可能なことを意味している．図4で示した同期モデルのウォークスルーを見直してほしい．『1）暗黙的非同期実行判断』のロジックで“同期処理と判断”しているが，開発環境においては常に“同期処理と判断”するようにE_BL_FWが振舞うことにより，非同期処理を意識しない開発が可能である．

3.4　アプリケーション・アーキテクチャー

ここでは『暗黙的非同期処理』実現の前提となるアプリケーション・アーキテクチャーについて整理する．合わせてビジネス・ロジック層周りの分析モデル・クラス図を**図8**に示す．

（1）レイヤー・モデル

ビジネス・ロジック層とプレゼンテーション層を明確に分離し，その間のインターフェースを一つとしていることにより，Javaバッチ・サーバーへのビジネス・ロジック処理の連携を容易に実現している．データアクセス層のコンポーネントは，ビジネス・ロジック層のコンポーネントと共にJavaバッチ・サーバー上でも稼動する．

（2）コールバック・パターン

E_BLはコールバック・パターンでE_BL_FWから呼び出されることにより，E_BL_FWによる非同期処理制御とトランザクション制御が同時に可能となる．

（3）POJO（Plain Old Java Object）[8]

J2EEや非同期処理技術などのテクノロジーに関する実装は全てフレームワークで吸収し，E_BLからは排除した．これにより将来の技術変化に対応しやすくなると共に，開発/UTが容易になり品質向上に貢献する事ができる．

(4) Session Facade パターン [9]

E_BLのクラス構造として採用し, 画面イベントにより呼び出される形式とした. これにより, BL層とPL層が疎結合となり, ビジネス・ロジックのJavaバッチ・サーバーへの連携が容易となる. PL層からBL層へのパラメータは, Value Objectにより伝達する. また, E_PL_FWによる非同期処理待ちポーリング処理も, E_BLの入り口が一つに絞られているため容易である.

(5) Value Object パターン [9]

ビジネス・ロジック(E_BL)に採用し, PL層フレームワークから, 一つのパラメータ・クラス(Value Object)の設定で, ビジネス・ロジックを呼び出す方式とした. また, E_BLはValue Object等, Serialize可能なインスタンス変数のみをプロパティーとして保持しているため, Javaバッチ・サーバーとの処理の連携を可能とした.

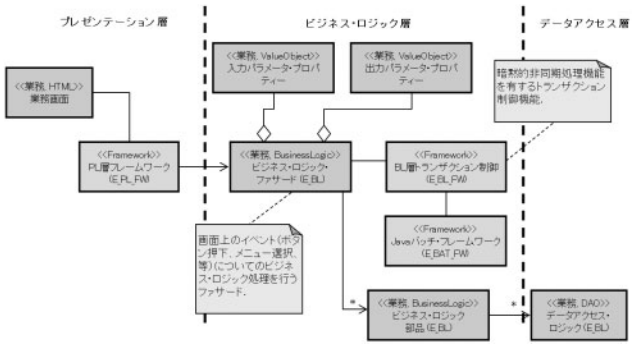

図8. ビジネス・ロジック層分析モデル・クラス図

4. 暗黙的非同期処理による効果と考慮点

本章では,『暗黙的非同期処理』を実施した場合の効果と実装上および運用上の考慮点, そしてパフォーマンスについての評価をまとめる.

4.1 アプリケーション開発生産性と品質向上

ビジネス・ロジック(E_BL)の開発は, BL層フレームワークからコールバックするメソッドの実装が主である. 開発者はビジネス・ロジックの実行形式(同期/非同期)をフレームワークに委譲しているため, 非同期処理を意識する必要はない. またPOJOでの開発であるため, ビジネス・ロジックの実装に専念することが可能である. そのためにビジネス・ロジックはシンプルな設計・実装となる.

ビジネス・ロジック層コンポーネントのUTについては, POJOで実行可能であるため, JUnit等のUTツールを利用したカバレッジ・レポートを取得することが可能となる. プロジェクトでのテスト方式標準化に繋がる.

これらに示したシンプルな実装(=POJO)により通常のWeb開発と同等レベルの生産性を確保する事を可能としている. 加えて, テストの標準化により, バグの混入を低減させると同時に問題判別の容易性を確保している. 結果としてソフトウェアの品質向上となる.

4.2 問題判別手法

通常, Webアプリケーション・サーバーのログ・ファイルには, 投入されるトランザクション毎に実行したトランザクション種別, 業務処理結果, エラーの情報等を出力し, 問題判別に利用する. 非同期処理に関しても同様であるが, バッチ・サーバーでの実行となるため, ブラウザーからの要求を受け入れたオンライン・アプリケーション・サーバーと, バッチ・サーバーのログ・ファイルの突合せによる問題判別が必要となる. また, ミッション・クリティカル・システムの場合はクラスタリングによる冗長構成をとるため, 複数のサーバーのログの中から目的のログを抽出し, 収集するシステム機能もしくは作業手順が必要となる. 以下2つの機能としてまとめる.

(1) マルチノード・ログ検索/抽出機能

これは同期処理の場合でも同じことが言えるが, クラスタリング構成のアプリケーション・サーバーやバッチ・サーバーのうち, どのノードで調査対象のトランザクションが処理されたかを特定する必要がある. 処理手順としては以下の様になる.

① 冗長構成されたアプリケーション・サーバーのログ・ファイルの中からクライアント情報や, トランザクションのIDをキーとして目的のログを抽出する.
② そのログに記録されているトランザクション通番(次項参照)をキーとして, バッチ・サーバー上のログから目的のログを抽出する.

(2) トランザクション通番採番/連携機能

アプリケーション・サーバーにトランザクション要求が投入された時点で, そのトランザクション通番を割り振る. その通番を, アプリケーション・サーバーおよびバッチ・サーバーでログ・ファイルに記録し, 問題判別時にログ抽出の突合せのキーとして利用する. このログ出力は, フレームワークにて実施し, ビジネス・ロジックには意識させない.

これらの機能/処理手順を可能な限り自動化(システム化)することが, 問題判別のスピードを上げることに繋がり, ミッションクリティカル・システムにおいては重要である.

4.3 パフォーマンス評価

同期処理に対して暗黙的非同期処理は, 直列化/非直列化, DB入出力, オンライン・アプリケーション・

サーバーとバッチサーバー間連携処理のオーバーヘッドがある．ここでは，それらパフォーマンスに関する考察を行う．**図9**に，テスト・ドライバーによる直列化と非直列化処理時間の実測値を示す．

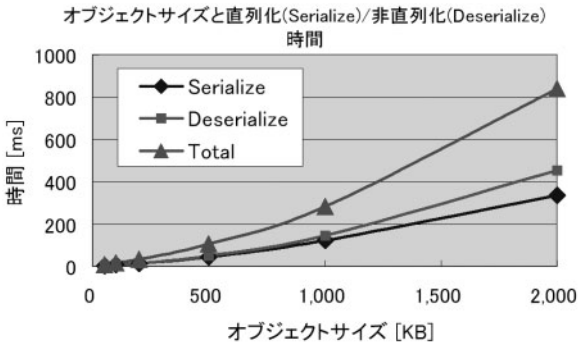

図9.　オブジェクト直列化/非直列化の実測値

- 実行環境 IBM AIX®5.3（CPU: IBM System p5 1.9GHz×1, Memory: 4GB）
- 使用JVM: IBM JDK 5.0
- 直列化対象オブジェクト（50byte長×20フィールドのSerializableのデータクラスを利用し，合計サイズが50kB, 100kB,200kB,500kB,1MB,2MBで計測）

当該環境の検証においては，本論文で取り上げた稟議業務としては最大規模である2MBのオブジェクトの直列化＋非直列化で約0.8秒の処理時間となっている．オブジェクトサイズと処理時間の関係は指数関数曲線を描くことが分かる．

テスト・ドライバーの処理は，図8に示したビジネス・ロジック・ファサード（E_BL）の入出力パラメータ・プロパティのValue Object群を，直列化/非直列化する処理を行い，処理時間を計測している．

DB入出力時間については，DB物理構成やネットワーク帯域などの環境面に大きく左右される．ギガ・ビットEthernetとキャッシュが有効に効いた構成のDBサーバー（CPU：IBM System p5™ 1.9GHz×0.2CPU, Memory：2GB）において，2MBのBLOB（Binary Large Object）のINSERT + SELECTをJDBCで呼び出した場合，約0.13秒（INSERT：0.08秒，SELECT：0.05秒）の処理時間であった．ここではそれを0.2秒とし，同期処理，暗黙的非同期処理それぞれの応答時間とオンライン・アプリケーション・サーバーの処理時間について比較検証してみる．まず，それらを数式化する．

［応答時間］
　同期処理：
　　BL層処理時間（α秒）
　＋PL層処理時間（0.5秒）
　＋クライアント処理時間（1.5秒）
　＝α＋2.0［秒］

暗黙的非同期処理：
　　BL層処理時間（α秒）
　＋PL層処理時間（0.5秒）
　＋クライアント処理時間（1.5秒）
　＋Serialize/Deserialize＋DB I/O時間（2.0秒）
　（オンラインサーバー，バッチサーバーの2回分）
　＋ポーリングによる終了検知遅延時間（β秒）
＝α＋β＋4.0［秒］

［オンライン・アプリケーション・サーバー処理時間］
　同期処理：
　　BL層処理時間（α秒）
　＋PL層処理時間（0.5秒）
＝ α＋0.5［秒］

　暗黙的非同期処理：
　　PL層処理時間（0.5秒）
　＋Serialize/Deserialize＋DB I/O時間（1.0秒）
＝ 1.5［秒］　（一定）

BL層処理時間（α）と，ポーリングによる終了検知時間（β）は，暗黙的非同期処理の効果に大きく影響するため，変動要因としてパラメータ化した．その他は前述の実測値と典型的な数値で前提をおいた．また，数式には直接現れていないが，ポーリングの間隔は3.0［秒］とし，オンライン・アプリケーション・サーバーからバッチ・サーバーへの処理連携のオーバーヘッドは数十ミリ秒のオーダーでの即時連携機能があるものとし，無視することとする．ポーリング処理時間についても1回あたり数十ミリ秒のオーダーであるため，これも無視する．

図10にBL層処理時間（α）＝5.0秒，ポーリングによる終了検知遅延時間（β）＝1.5秒（ポーリング間隔3.0秒の1/2で平均値）とおいた場合の処理時間の分解図を示し，**表1**に両方式の各数値を比較し整理する．ここでα＝5.0秒とおいた根拠であるが，暗黙的非同期

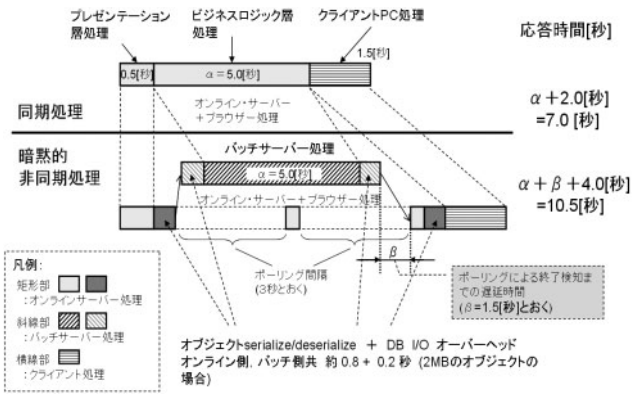

図10.　同期処理と暗黙的非同期処理の応答時間比較（α＝5.0秒，β＝1.5秒の場合）

表1.　同期処理と暗黙的非同期処理の応答時間比較
（α＝5.0秒，β＝1.5秒の場合）

BL層処理時間によるオンライン･サーバー占有率の違い

［秒］

処理方式	オンラインサーバー				バッチサーバー		クライアント	応答時間	オンラインサーバー処理時間
	PL層処理	BL層処理	Serialize/Deserialize + DB I/O処理	ポーリングによる終了検知遅延	BL層処理	Serialize/Deserialize + DB I/O処理	画面展開		
同期処理	0.5	5.0	－	－	－	－	1.5	7.0	5.5
暗黙的非同期処理	0.5	－	1.0 (0.8+0.2)	0～ポーリング間隔（平均 1.5）	5.0	1.0 (0.8+0.2)	1.5	10.5	1.5

処理化を行う一つの判断基準として考えた．オンライン・アプリケーション・サーバー内で，処理中として滞留時間を許容する最大時間を5.0秒とすると，PL層処理時間，NW処理時間，そしてブラウザーでのHTML展開時間を考慮すると応答時間は7.0秒となり，ユーザーとしてはストレスを感じ始める時間と言える．また，サーバー・サイドのシステム構成にもよるが，大量に5.0秒の滞留時間のトランザクションが投入された場合，サーバー・システム資源枯渇の危険領域といえる滞留時間である．

図10および表1の結果で明白なとおり，1回のトランザクションの応答時間を比較すると，同期処理が7.0秒のところ，非同期処理は10.5秒かかるのがわかる．一方，オンライン・アプリケーション・サーバーの処理時間で比較すると，同期処理が5.5秒，非同期処理が1.5秒となり，結果は逆転する．

この結果により，暗黙的非同期処理は，他のトランザクションが少なく，リソースに余裕のある場合には，同期処理より処理時間が長くなるが，同時リクエスト数が多く，リソースの競合が発生しやすい場合にはBL層処理時間の影響を受けにくく，同期処理のような大幅な遅延が避けられる．これは，オンライン・アプリケーション・サーバーのスループット向上に大きく貢献することを意味している．**図11**にポーリングによる終了検知遅延時間（β）を1.5秒に固定し，BL層処理時間（α）を5秒から40秒まで変動した場合の同期処理と暗黙的非同期処理によるオンライン・アプリケーション・サーバー処理時間の違いを示す．同期処理は，BL層処理時間の増加に比例してオンライン・アプリケーション・サーバーの処理時間が増加するが，暗黙的非同期処理は，BL層処理時間によらず一定であることが分かる．

この検証により，2.2節『処理時間の長いHTTP要求の危険性』に示したWebシステムにおけるサーバー資源の長時間占有の危険性に関して，暗黙的非同期処理はビジネス・ロジック処理時間に左右されずその危険を回避していることを証明している．

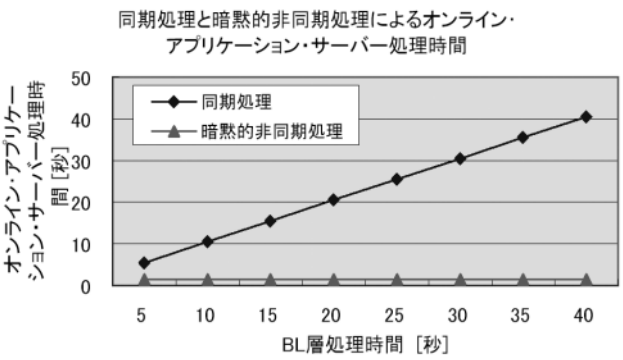

図11.　同期処理と暗黙的非同期処理によるオンライン・アプリケーション・サーバー処理時間の違い

5．おわりに

本論文では，Webシステム開発において，同期処理として開発し，実行環境において非同期処理させることが可能な『暗黙的非同期処理』アーキテクチャーを提案し，その有効性を確認した．高度なスキルを要しない開発者でも非同期処理の開発が可能となり，プロジェクト全体の品質向上，および開発者のスキル要求レベルの軽減に繋がる．また，暗黙的非同期処理実現のための要素技術は，周知のアーキテクチャーやデザイン・パターンであり，実現性を示した．

昨今，システム運用の現場では，データ量の増加に伴い夜間に実行する定例バッチジョブの処理時間が増加する傾向にあり，朝のオンライン開始時間を越えてしまう，いわゆる『突き抜け』の障害事例が多くなっているとも聞いている［10］．そのため，オンライン・トランザクションに連動した非同期処理，つまりオンライン・バッチ処理の必要性は今後増加してくる．その実装方式について，本論文で主張した『暗黙的非同期処理』アーキテクチャーが効果的である．

謝辞

本論文を執筆するにあたり，日本アイビーエム株式会社 アセット＆SOAコンピテンシー 松原 武司氏には，JBeXを始めバッチ処理についての貴重なご意見をいただいた．ここに深く感謝の意を示したい．

参考文献

［1］IETF：“ RFC2616 HTTP/1.1, ”
http://www.ietf.org/rfc/rfc2616.txt,（2007.3.12）.
［2］小宮 聖則：“ ミッション・クリティカル・システムにおけるオートノミック・コンピューティング機能の構築, ”
PROVISION, No.45, pp.67-74（2005）.
［3］神野 稚子 ：“ パフォーマンスチューニングを睨んだWebアプリケーション設計, ”
http://www.ibm.com/jp/software/websphere/developer/

websphere/ise/techcon2005/pdf/jinno.pdf, (2007.6.13).
[4] Kenji Kojima:" Java Message Service(JMS), " http://www.ibm.com/jp/software/websphere/developer/w40/iw/pdf/8_1.pdf, (2007.6.13).
[5] 山本 宏:" WebSphere XDのご紹介, " http://www.ibm.com/jp/software/websphere/developer/was/xd/v6/1_3.html, (2007.6.13).
[6] 山本久好, 榊原彰:" オペレーショナル・モデル・モデリングにおける非機能要件の効果的検証方法, " PROVISION, No.41, pp.85-92 (2004).
[7] Sun Microsystems:" java.io.Serialize, " http://java.sun.com/j2se/1.5.0/ja/docs/ja/api/java/io/Serializable.html, (2007.6.13).
[8] Martin Fowler:" Plain Old Java Object, " http://martinfowler.com/bliki/POJO.html, (2007.6.24).
[9] ディーパック・アラー, ジョン・クルーピ , ダン・マークス:J2EEパターン, 株式会社ピアソン・エデュケーション, ISBN4-894-71434-5, (2002).
[10] 日経BP社:" 解体!レガシー・バッチ, " 日経コンピュータ(2006年8月21日号), pp.32-47 (2006).

日本アイ・ビー・エム株式会社
GBS AIS銀行ソリューション
ICPシニアITアーキテクト

小宮 聖則 Kiyonori Komiya

[プロフィール]

1987年日本IBM入社. 金融機関のお客様を担当するSEとして配属され, 米国IBMでの金融ミドルウェア開発, 帰国後各種C/Sシステム開発プロジェクトを経験. 1999年より某都市銀行様担当SEとして, 分散系, Web系のシステム開発プロジェクトに携わる. ミッション・クリティカル・システムにおけるシステム基盤, ミドルウェア, フレームワーク, アプリケーションに渡るアーキテクチャー構築に従事.

komiya@jp.ibm.com

日本アイ・ビー・エム株式会社
GBS ソフトウェア・エンジニアリング
ITスペシャリスト

今井 博一 Hirokazu Imai

[プロフィール]

2002年日本IBM入社. 入社後, Web系フレームワーク, Javaコンポーネント, Java開発支援ツールなどの開発を主に担当. 同時に製造業を中心としたWeb系システム開発を経験した後, 2006年より某都市銀行様のWebアプリケーション開発の技術支援やフレームワーク設計/開発に従事している.

HIIMAI@jp.ibm.com

日本アイ・ビー・エム株式会社
GBS メインフレームシステム開発
ITスペシャリスト

齋藤 幸二郎 Kohjiroh Saitoh

[プロフィール]

2003年日本IBM入社. 金融機関のお客様のシステム開発, 保守プロジェクトに携わり, オープン系およびメインフレーム系システムのインフラ(プラットフォーム, ミドルウェア)を担当. 2006年より某都市銀行様のWebアプリケーション開発の技術支援やフレームワーク設計/開発に従事している.

kojiros@jp.ibm.com