D-035

永続化フレームワークにおけるデータベースの隠蔽とレスポンスの高速化

Database hiding in framework and speed-up of response

西　宏昭†　東海林　健志†　宮川　治‡　大山　実†

Hiroaki Nishi†　Takeshi Shoji†　Osamu Miyakawa‡　Minoru Ohyama†

1. はじめに

現在，Java や C#などの言語の登場によって，クライアント・サーバアプリケーションをこれらの言語で開発するようになった．もともと Java や C#はこれらのアプリケーションの作成に対応できるように J2EE や.NET などの大規模アプリケーション向けの仕様が用意されている．現在この仕様に準拠したクライアント・サーバアプリケーションの開発が一般的になってきた．ここで問題となるのがオブジェクトの永続化である．スタンドアローンアプリケーションと違い，クライアント・サーバアプリケーションにはクライアントから送られてきた情報を管理する必要がある．この情報の管理方法として，現在最も広く普及しているのが RDB である．しかし，RDB は Java や C#といったオブジェクト指向言語との親和性が低く，オブジェクト指向の利点を生かしきれないだけではなく，SQL 文があらゆるコードに散在し，コード管理の難しさなどの問題がある．

これに対処するにはオブジェクトと RDB の差をうめる技術とデータベースへのアクセスを他の機能とは隔離し，モジュールとして分離することが必要となる．

また、サイトパフォーマンスのひとつの基準に 8 秒ルール，ユーザを 8 秒以上待たせてはいけないというルールがあるが,高速ネットワーク回線の普及に伴い，「6 秒ルール」や「3 秒ルール」といった言葉もすでに過去のものになりつつある[1]．レスポンスはユーザビリティにつながる大きな要素のひとつといえる．システムの運用において，多くのリクエストが発生したときにレスポンスが悪くなれば，ユーザは使い勝手が悪いと感じるであろう．このことから，アプリケーション自身の高速化が必要と同時にユーザビリティを考慮したレスポンスの高速化技法が重要となってきている．上記のように現在のクライアント・サーバアプリケーションに求められる要求は高度化し，どれも重要な要素となってきた．

本論文では

- クライアント･サーバアプリケーションにおけるデータベースの隠蔽
- ユーザビリティ向上の観点からのレスポンスの高速化技法

この 2 点について順次述べる．

†東京電機大学大学院　情報環境学研究科，Tokyo Denki University，Graduate School of Information Environment

‡東京電機大学　情報環境学部，Tokyo Denki University，The School of information Environment

2. オブジェクトの永続化とその必要性

純粋なオブジェクト指向の世界では，あらゆる情報がオブジェクトとしてモデリングされ，クラスとして記述される．クラスから生成される多数のオブジェクトが情報の実態を持つことになる．しかし，この方法には重大な問題がある．通常，オブジェクトはメモリ上に生成されるものであり，プログラムを終了すると消えてしまう．どのようなすばらしい情報を持ったオブジェクトを生成しても，プログラムの終了とともに情報が消失してはクライアント・サーバアプリケーションとしての機能を満たさない．

オブジェクトの永続化とはプログラムが終了してもオブジェクトの情報を何らかの方法（ファイルに記録する，データベースに記録する）で保存し，再び必要とされたときオブジェクトを，すでにそこにあったかのように元通りに復活させることである．

3. 従来型の永続化

現在，アプリケーションへの要求は高度化してきており，従来から行われてきたデータ中心による開発では要求の変化についていけない場合がある．要求はさまざまな状況によって変化し続ける．従来，次のような技法が用いられることが多かった．

3.1 従来型永続化技法

従来のデータ中心による開発では，システム全体において永続化する必要のあるデータをはじめに洗い出し，それらのデータをデータベースに保存していくために次のような作業を行う必要があった．

- データ型の決定
- key の設定
- リレーションの決定
- 正規化

このようにデータベース側の処理を行ったうえで Entity（情報の対象領域）を決定して，Entity 自身に永続化のための SQL を記述し，システム全体をボトムアップ方式で組み立てる．

3.2 問題点

前述したような，一般的な開発にはいくつかの問題点が存在している．

- データベースの構造を先に決定するために Entity 自体の柔軟性にかける
- オブジェクト指向の扱える多種多様なデータ型とデータベースの扱えるデータの変換が必要となる（O/R マッピングにおけるインピーダンス不整合）

- 正規化，key の設定,リレーションの設定をただしく行うには高度なデータベースのスキルが必要となる
- SQL 処理が散在する
- Entity の構造の変更による SQL 処理の変更
- トランザクション処理

これらの問題の一部を解決するために SOC（Separation of concern）[2]の概念を導入して Locator（Entity をメモリ上で管理するもの）に SQL の処理を記述する方法がとられた．しかし，この方法をとっても Entity の構造の変化によって Locator の SQL の記述の変更が必要になる．さらに Entity の種類ごとに Locator も存在することになるので，結局 SQL の処理はいたるところに散在してしまう．これらの SQL は Entity との結びつきが強いために Locator ごとに SQL の処理も異なる．結局 Entity の構造が変わったときにはデータベースの構造を変更しなければならない．このような状況を図１に示す．

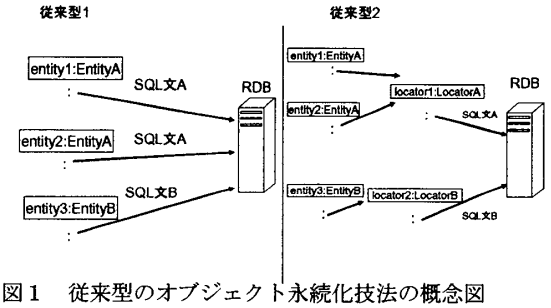

図１　従来型のオブジェクト永続化技法の概念図

4. 提案する技法

上記のような問題を解決するために我々は XML を用いた解決方法を提案する．

4.1 解決技法

XML を用いることでオブジェクトの状態を表すことが可能である．XML はオブジェクトと相性がよく，オブジェクトの複雑な状態やデータ型すらも，スキーマをうまく定義することによって吸収できる[3]．これによって O/R マッピングにおけるインピーダンス不整合は解決できる．我々のとったアプローチはオブジェクトの状態を XML に変換し，それをそのまま文字列情報として RDB に格納する方法である．このようにすると，オブジェクトの状態が増えたとしても変換される XML の文字数が増えるだけでデータベースの構造を変える必要がない．また，発行する SQL 文すらも変える必要がなくなる．このことから，データベースへの書き込みの処理を他のオブジェクトから隔離でき，データベースの処理を完全に隠蔽することが可能になる．つまり，Entity の構造の変化によって SQL の処理は変わることなくテーブルの構造を一意に決定でき，すべての Entity に対して同じ SQL の処理で対応できる．

XML の解析によって生成されたオブジェクトは Entity という形でメモリ上に存在することになる．それぞれの Entity は元になった XML の種類（モデル）ごとに Locator に登録される．Locator はメモリ上で Entity の一元管理を行い永続化の処理を Recorder（データベースの処理を隠蔽しているオブジェクト）へと委譲する．Entity は変更されると Pull 型の Observer パターン[4]によって Locator へ Entity 自身が変更されたことを知らせる．Locator はその通知をそのまま Recorder に委譲し Recorder は Entity を再び XML に変換しオブジェクトの情報を記録する．このような方法をとることで永続化機構の存在を意識することなく永続化が行われる．さらにこの方法をとると，どこに永続化されているのかさえ気にする必要がない．Strategy パターン[4]によって MySQL なのか，PostgreSQL なのか，ORACLE なのか，それどころか RDB に記録されているのか XMLDatabase なのか，アプリケーションが動いている同じコンピュータなのかさえ気にする必要がない．従来，一般的だった Entity ごとに記述されていた SQL の処理は Entity の種類に関係なく Recorder で一意の SQL 文によって処理される．これらの状況を図２に示す．

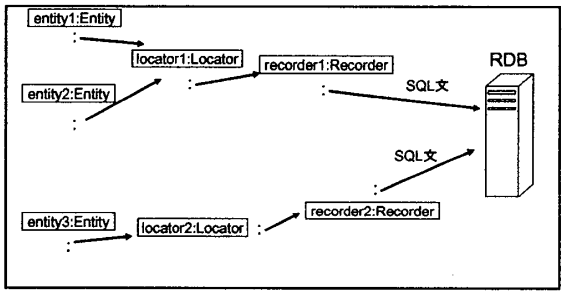

図２　我々の提案するオブジェクト永続化技法の概念図

このような技法を用いることによって，JDBC やデータベースの高度なスキルを必要とすることなく永続化が可能となる．

4.2 効果

Locator 層の背後に，もうひとつ Recorder 層を設けることによってデータベースの処理を隠蔽することが可能となった．

また現在，アプリケーションの開発工数の最大 4 割がオブジェクトの永続化を解決するための作業だといわれている[5]．しかし，我々の提案する，データベース処理の隠蔽と O/R マッピングの解決方法を使うことで，アプリケーション本来の機能に絞って開発をすることが可能になる．さらに，軽量なアプリケーションを作成したところ約 30%のコーディング量の減少もみられた．

5. 他の O/R マッピングツールとの比較

オブジェクトの永続化はクライアント・サーバアプリケーションでは必須の条件となっている．

上述の問題点を解決するために，複数のフレームワークも存在し，様々なアプローチを取っている．いくつかの一般的な永続化フレームワークでは SQL レスや他の技術を覚える必要がないと説明しているが，いくつかの問題点がある．オブジェクトを問い合わせるために SQL に似た新しい言語を覚える必要や，Entity ごとにデータベ

ースの構造やデータ型を強く意識した XML ファイルを定義して，それからソースコードを生成したり，完全にデータベースや SQL を排除し切れていない．

我々の技法ではオブジェクトを問い合わせる場合はフィルターオブジェクトを使用すればよいだけで，SQL も必要なければ SQL に似た言語を習得する必要もない．次にデータベースの存在は Recorder によって完全に隠蔽され，データベースの構造やデータ型を意識することはない．必要なのは Entity にどのような状態があればよいのか，また，その型はどのような型なのかといった普通のオブジェクト指向分析・設計の知識である．

一般的なオブジェクト永続化フレームワークは DAO（Data Access Object）パターン[6]を利用している場合が多い．J2EE パターン[7]ではデータソースへアクセスするコードをビジネスロジック層に直接書くべきではないといっている．これに従い，DAO パターンは Data Access Object という中間層を定義し，ビジネスロジック層から永続化処理を取り除こうとしたものである．しかし，DAO パターンでは永続化したい Entity そのものを扱うよりも，Data Access Object を扱うことが多くなる．これでは Entity が変更されると思われる部分全てに DAO パターンをもちいたコードを書く必要があり，同じソースコードがあらゆるところに出てくる可能性が存在する．さらに，それらのコードは単純作業になることが考えられ，コピーしたり，書き忘れたりと，フォールトの原因となることも考えられる．

6. レスポンスの問題点

これまでに述べたような技法を用いることで，Recorder だけにデータベース処理の隠蔽をすることが可能になった．しかし，我々のフレームワークは JDBC を直接扱うコーディングよりもレスポンスが悪化した（図 4）．そこで，Recorder のレスポンスの高速化をすることでアプリケーション全体のレスポンス向上を図る必要があった．

7. レスポンス高速化技法

一般的な永続化処理モジュールや我々の作成した永続化フレームワークでは，データベースの書き込み終了を待ち，書き込まれたことを確認してからレスポンスを返すようになっている．しかし，それでは高負荷時において，アプリケーションのボトルネックといわれることの多いデータベースの書き込みに時間がかかりすぎ，レスポンスが悪くなるという問題が起こる．そこで，データベースへの書き込み終了を待たずにレスポンスを返すことでレスポンスタイムを短縮する方法を提案する．

具体的には，永続化処理モジュールである Recorder にマルチスレッドを利用し，メインスレッドはデータベースへの書き込み終了を待たず，そのデータベースへの書き込み作業を別スレッドで行うことによってメインスレッドより切り離すことにした．

7.1 マルチスレッド

シングルスレッドがひとつのスレッドからなるプログラムであるのに対し，マルチスレッドは複数のスレッドが協調して動作をするプログラムのことである．

しかし，プログラムしだいでは，タイミングや順番などを誤るとデータが壊れることもあり，また制御しあうことによるデッドロックが起こる危険性もある．それらの危険性を考慮し，プログラムを作成する必要がある．

7.2 技法

レスポンスの高速化を図るために，マルチスレッド環境におけるデザインパターンである WorkerThread パターン[8]を利用し，設計を行った．図 3 は設計構造をあらわし，オブジェクト図の表記に準ずる．以後，太字の文字列はクラス名やインスタンス名を表現することとする．

それぞれの役割を示す．

- 「仕事を与える役」：メインスレッド
- 「仕事を蓄える役（キュー）」：**Queue**，**Map**
- 「仕事を取り，実行する役」：他複数スレッド

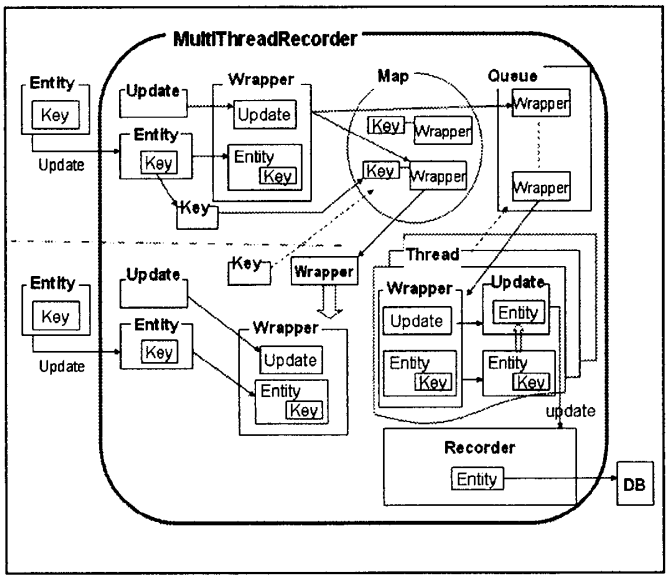

図 3 Multi Thread Recorder 設計構造

「仕事を蓄える役」であるキューは同じオブジェクトが格納できないようにする必要があったため，**Queue** だけでなく，**Map** も併用することで，**Queue** に同じオブジェクトが 2 つ以上格納できないようにした．（図 3 の右上）

まず，「仕事を与える役」であるメインスレッドの動きから説明する．（図 3 の左側）ユーザがデータに対する操作を行うと，そのデータを持つ **Entity** が変更される．変更されると，その情報は Observer パターンによって自動的に **MultiThreadRecorder** に届く．**MultiThreadRecorder** では，その変更された **Entity** を「操作の種類（図では **Update**）」と一緒に **Wrapper** でセットにする．ここで，**Entity** の **Key** を使用して **Map** にセットした **Wrapper** が存在するかどうかをチェックする．

存在しなかった場合，その **Wrapper** を **Map** と **Queue** の両方に格納して終わる．（図 3 の左上）

存在した場合，**Map** から存在した **Wrapper** を取り出し，セットにしてある **Entity** を「操作の種類」を新しい **Entity** と「操作の種類 **Update**」に替えて終わる．（図 3 の左下）**Map** から取り出したその **Wrapper** の中身を替え

るだけですむのは，同じ **Wrapper** が **Queue** にも格納されており，参照を持っているので **Queue** に格納した順番は前の操作の順番のまま，中身だけ最新の操作のものに替えられるからである．

このようにすることで，前の操作を飛ばして最新のデータ状況をデータベースに反映でき，より早く現在のオブジェクトの状態と同期をとれるようにした．

次に，「仕事を取り，実行する役」であるほかの複数スレッドの動きを説明する．（図 3 の右下）

Queue には **Wrapper** が順番を保たれた状態で格納されているので，これらのスレッドはそれぞれ **Queue** から **Wrapper** を取り出し，**Queue** を同じ状態にするため，**Map** からも同じ **Wrapper** を消す．取り出した **Wrapper** にセットされていた **Entity** を「操作の種類 **Update**」に渡す．図では **Update** になっているのでこの場合，**Entity** の情報をデータベースとのコネクションを持つ **Recorder** に update させる．これらの複数スレッドははじめに立ち上げられ常に待機しており，**Queue** に **Wrapper** があれば仕事をし，なければくるまで再び待機する．複数スレッドをはじめに立ち上げ，待機させることによって，スレッドオブジェクトを生成するコストを避けることができる．

このように，メインスレッドはキューに仕事をいれ，別スレッドにあとの処理を任せて終了し，レスポンスを返すことができる．これにより，レスポンスが改善される．

7.3 検証

レスポンスにおいてはいくつかの条件で実際に検証を行った．

7.3.1 負荷条件

50KB のデータを同時に書き込ませる数（スレッド数）を変化させ，5 秒の間隔をあけ，10 回繰り返した平均を計測した．これを 3 回行い，その平均値を記録した．

7.3.2 結果

図 4 のグラフの 3 本のラインのうち，一番上のライン「Single Thread」は我々のオブジェクト永続化フレームワークをそのまま利用した場合，真ん中のライン「JDBC」はデータベースへの書き込み部分を JDBC を使用し，SQL 処理をしデータベースへ書き込みを行った場合，一番下のライン「Multi Thread」が我々のオブジェクト永続化フレームワークに，このプログラムを組み込み使用した場合である．

「Multi Thread」のラインが一番下にあるということは，レスポンスタイムが一番短かったということである．

また，高負荷時の読み込み時において最大で約 7 倍の改善が見られることもあった．CPU1GHz，メモリ 768MByte を搭載したコンピュータ上で，軽量なアプリケーションを動かした場合において毎秒 100 スレッド，一日平均 150 万アクセスを超える高負荷時においても平均レスポンス 0.27 秒を記録した．

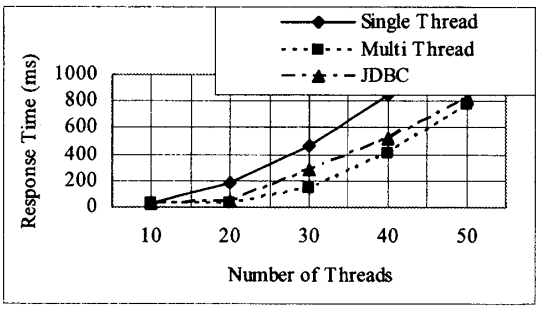

図 4　書き込み時のパフォーマンス

8. まとめ

O/R マッピングにおけるインピーダンス不整合を解決するために XML を導入することで，Entity の状態が増減しても SQL を変えることなく永続化をすることができるようになった．さらに Recorder 層を導入することによってデータベースの処理を完全にビジネスロジックから隠蔽することができた．

マルチスレッドを利用した我々のフレームワークの検証を行った結果，レスポンスが向上され，オブジェクトの永続化フレームワークがユーザビリティの観点において改善された．また，SQL を直接書き込み，データベースへアクセスした場合と比べてもレスポンスが改善され，マルチスレッドにしたことで，データベースへかかる負荷を軽減することが出来た．また、同時アクセス限界数を大幅に高められた．課題としてはマルチスレッド環境における安全性の確認である．マルチスレッド環境における安全性は確認することが非常に難しく，コードレビューの詳細化などを通して，さらなる安全性を追求していく必要がある．

参考文献

[1]e-Word
http://ewords.jp/w/8E7A792E383ABE383BCE383AB.html

[2] Aspects and advanced separation of concerns http://trese.ewi.utwente.nl/oldhtml/aspects_asoc/

[3] 東海林健志，久住貴史，西宏昭，宮川治，大山実，オブジェクトの永続化と RDB 設計からの解放，FIT2005，投稿中

[4] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, "Design Patterns", ADDISON-WESLEY PUBLISHING COMPANY, 1995

[5]@IT[FYI]
http://www.atmarkit.co.jp/ad/sonic/sonic2004/sonic0407.html

[6] 岡本　隆史，吉田　英嗣，金子　崇之，権藤　夏男，"Light weight Java"，毎日コミュニケーションズ 2005

[7] ディーバック・アラー，"J2EE パターン-明暗を分ける設計の戦略"，ピアソンエデュケーション 2002

[8] 結城　浩，"Java 言語で学ぶデザインパターン入門　マルチスレッド編"，Softbank publishing 2002