EJB パターンを用いた性能設計の方法とその評価

2000MT042 菊池 紀子　　2000MT074 荻山 信

指導教員:青山幹雄

1. はじめに

1.1. 背景

J2EE (Java 2 Platform Enterprise Edition) は企業向けの大規模な分散サーバを構築するための技術や概念を集約したものである. J2EE システムはクライアント層や Web 層(Web サーバ), EJB 層(EJB サーバ)など複数の層にまたがって構成されているために, 各層間の関係を考慮したシステム設計が必要となり, 設計の複雑化が問題となる. この問題を解決するために Sun Java Center により J2EE パターンが提案された. J2EE パターンを参照することで, 設計の大枠や考慮点を実装に先がけて手に入れることができる.

しかし J2EE パターンの内容には熟練した開発者でなければ理解が困難であると考えられるものも多く含まれる.

1.2. 研究課題

そこで我々は J2EE パターンの中の EJB パターンに焦点をしぼり, これを EJB システム開発経験の浅い者にも理解しやすいように文書化しようと考えた. そこで各 EJB パターンの実装例, パフォーマンスを表すグラフ, 設計時の考慮点の 3 点についてまとめる.

2. J2EE

2.1. 構成要素

J2EE は, 分散型エンタープライズ Java アプリケーションを開発するための, 種々の技術を統合した開発基盤である. J2SE と同じ"Write once, Run Anywhere"の概念を基盤に, JSP (Java Server Pages), Servlet, EJB (Enterprise Java Beans), JDBC といった技術群からなる. J2EE アーキテクチャは論理的に 5 層に分離されており, クライアント層, プレゼンテーション層(Web サーバ), ビジネス層(EJB サーバ), インテグレーション層, リソース層(DB や外部システム)からなる.

2.2. EJB コンポーネントと EJB コンテナ

EJB は J2EE システムのビジネスロジック部分を実装するコンポーネント化技術である. EJB はワークフローを実装する Session Bean, データの永続化を行う Entity Bean, メッセージサーバからの非同期処理を受け付けるメッセージ駆動型 Bean の 3 種類からなる. EJB コンテナは EJB コンポーネントを格納する容器であり, 実行環境である.

3. EJB パターン

パターンとは, 繰り返し起こる問題とそれに対する最良の解決策を記述したものをいう. Sun Java Center は, J2EE システム設計のための 15 のパターンを J2EE パターンとして提供している. この中で, EJB 技術を用いたシステム設計のための 7 つのパターンを EJB パターンと呼ぶこととする. EJB パターンの中から本研究で用いた 4 つのパターンの説明を以下に示す.

1) Session Façade パターン
   ビジネスオブジェクト間の複雑な関係を隠蔽し, 階層間の独立性を図る. また, 階層間のリモートメソッド呼び出しの回数を減少させる.
2) Value Object パターン
   複数のデータをひとつにまとめて, データ交換時のネットワークトラフィックを減らす.
3) Value Object Assembler パターン
   複数のバリューオブジェクトをまとめて, 複合バリューオブジェクトをつくる.
4) Service Locator パターン
   JNDI を使ったルックアップ処理を効率化する.

EJB 設計は考慮すべき点が多く, 経験の浅い開発者が優れた設計をすることは難しい. しかし, EJB パターンを利用して優れた設計の雛形を手に入れることで再利用性が高く, 高性能な EJB システムを構築することができる.

4. JBoss

JBoss はエンタープライズシステムを実現する環境を提供するオープンソースの J2EE サーバである. 無償で使用できるため, 本研究で EJB パターンを実装する際のサーバとして, この JBoss を採用する.

4.1. JBoss の機能

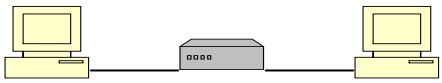
JBoss の特徴的な機能はホットデプロイである. ホットデプロイは, EJB アプリケーションとデプロイメント記述子をまとめた JAR (Java ARchive)ファイルをサーバ内のデプロイディレクトリにコピーするだけでデプロイメント(配備)を行えるものである. この機能により, 通常のデプロイメントツールに比べてデプロイメントに要する時間と手間を大幅に短縮することができる.

5. 性能評価

ビジネス層のアプリケーションは，J2EE システム全体の複雑なビジネスロジック実装の中核となるため，ボトルネックとなりやすい．このため，性能を考慮して EJB システムを設計することが重要である．しかし，EJB アプリケーションの設計は複雑で考慮点が多く，開発者の経験が大きく影響することが問題となる．そこで，我々は EJB 開発の初心者にも EJB パターンの効果を視覚的に実感できるように，その性能をグラフで表す．また，EJB パターンが性能を向上させる要因を検証する．本研究では各パターンの平均処理時間（ms）とスループット（件/秒）を測定して評価を行った．平均処理時間は 10 回測定し，その平均を測定値とした．スループットの算出には以下の式を用いた．

$$\text{スループット} = \frac{1000}{\text{平均処理時間}} \times \text{同時接続クライアント数}$$

また，ネットワークトラフィックの影響を受けないようにするため，図 1 のようにクライアントマシンと EJB サーバマシンを他のネットワークから切り離して接続した．テストプログラムを実行する際には，テストプログラム以外のアプリケーションは閉じてサーバマシンとクライアントマシンの動作に影響が出ないようにした．システム要件は表 2 に示す．



クライアントマシン　スイッチングハブ　EJB サーバマシ

図 1: システム構成

表 2: システム要件

マシン CPU	Pentium（R）4CPU 2.00GHz
マシンメモリ	632MB RAM
スイッチングハブ	100Mbps
OS	Windows XP Professional Ver.2002
J2SDK	Ver.1.4.01
JBoss	Ver.3.0.7

6. 各パターンの性能特性と実装例

パターンを用いることで，性能がどう変化するかを比較するために各パターン適用時と非適用時のスループットを測定した．結果を図 3 から図 6 に示す．

6.1. Session Façade パターン

Session Façade パターンは，セッションファサードと呼ばれる処理の仲介をおこなうセッション Bean をクライアントと EJB システムの間に置くことで，クライアントから EJB へのリモートメソッド呼び出しの回数を減らしネットワーク通信によるパフォーマンスコストを軽減するものである．図 3 より，パターンを適用したシステムでは同時接続クライアント数が増加してもスループットは上昇を続ける．またクライアント数が 50 のときには約 55 倍の開きがある．

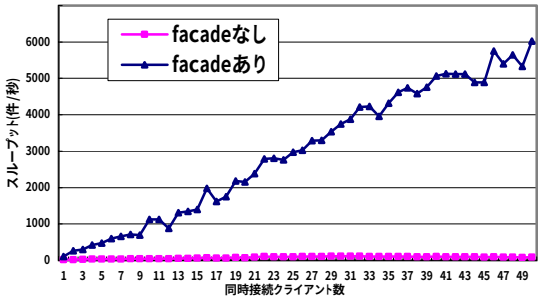


図 3:Session Façade パターン スループット比較

6.2. Value Object パターン

クライアントが EJB から取得する複数のデータをバリューオブジェクトというひとつのオブジェクトにまとめてデータ通信をすることでリモートメソッド呼び出しの回数を軽減する．以下にバリューオブジェクト実装の記述例を示す．

```
public abstract class BookBean implements
  EntityBean{
    public BookVO getBookVO(){
      BookVO bvo  = new BookVO(getBookId(),
          getTitle(),getAuthor(),getPublisher());
      return bvo;
    }
}
```

この記述例では，BookBean というエンティティ Baan が BookVO というバリューオブジェクトを生成し，その中に必要なデータを格納し，これをクライアントに返す．このときのスループットは図 4 のようであり，パターンを用いることでおよそ 2 倍から 3 倍の性能向上が見られる．

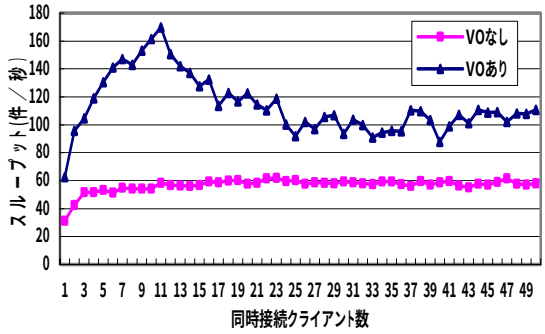


図 4:Value Object パターン スループット比較

6.3. Value Object Assembler パターン

扱うデータの種類が多いとき，複数のバリューオブジェクトを得るためにクライアントと EJB の間で何回ものリモートメソッド呼び出しが発生する可能性がある．Value Object Assembler パターンは複数のバリューオブジェクトを複合バリューオブジェクトとしてひとまとめにクライアントに送信することで，一度のリモートメソッド呼び出しで必要なすべてのバリューオブジェクトを手に入れることができるようにするものである．図 5 より，パターン非適用時にはクライアント数が 11 を超えた時点からスループットの低下が見られる．これはこの点がサーバの処理性能の限界であることを示す．これに対しパターン適用時にはほぼ右肩上がりの増加を続ける．これはサーバの処理性能がまだ限界に達していないことを示す．このことから本パターンを用いることでスループットだけでなくスケーラビリティも向上していると言える．バリューオブジェクトが複数個必要なシステムでは，Value Object Assembler パターンが有効であることがわかる．

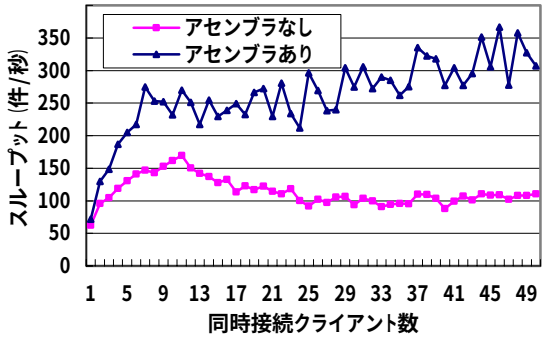


図 5:Value Object Assembler パターンスループット比較

6.4. EJB Home Factory パターン

JNDI サーバが JNDI 名から EJB への参照を検索するのはパフォーマンスコストの高い作業である．EJB Home Factory パターンでは，EJB 内にハッシュマップを用意し，JNDI 名をキーとして，参照をキャッシュする．一度取得した EJB 参照はハッシュマップにキャッシュされるため，検索にかかるパフォーマンスコストを軽減できる．以下にパターン実装例を示す．homeInterfaces はホームインタフェースクラスをキーとして，ホームインタフェースへの参照を格納するハッシュマップである．取得したい参照のクラスをキーとしてハッシュマップに get()を行うことで，格納されている参照を取得する．

```
public static EJBHome getHome(String jndiName, Class
        homeInterfaceClass) throws NamingException {
    EJBHome  homeInterface =
      (EJBHome)homeInterfaces.get(homeInterfaceClass);
    if  (homeInterface == null) {
        Object ref  = initContext.lookup(jndiName);
        homeInterface =
                (EJBHome)PortableRemoteObject.narrow
                                (ref, homeInterfaceClass);
        homeInterfaces.put(homeInterfaceClass,
                                        homeInterface);
    }
  return homeInterface;
}
```

図 6 から，パターン非適用時と適用時のスループットの値はほぼ同値である．この原因を探るため JNDI サーバに対する lookup()メソッドの呼び出しにかかる時間を測定したところ，ほぼ 0ms であった．これは JNDI サーバと EJB サーバがリモートでないことが原因の一つであると考えられる．このため本パターンでスループットの差は確認できなかった．しかし JNDI サーバが EJB サーバとは別のマシン上で動いている場合にはこのパターンは有効と考えられる．

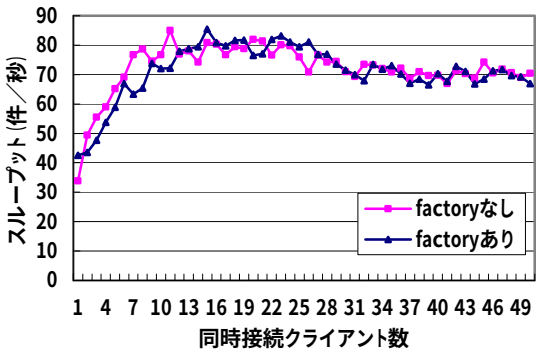


図 6:EJB Home Factory パターン スループット比較

7. パターンの組み合わせとボトルネックの検証

パターンの組み合わせによるパフォーマンスの変化について調査した．また，そのときのシステムを例にとり，ボトルネックを探った．

Session Façade パターンと Value Object パターンを組み合わせたと言える Value Object Assembler パターンと，Value Object パターンを単体で用いた場合のスループットを比較した．両方に同じ負荷をかけてテストした場合，組み合わせ時のほうが約 3 倍スループットが高かった．さらに組み合わせ時のみ負荷を上げていったときでも，単体で用いるよりも高いスループットを記録した．このことからパターンを適切に組み合わせることは単体で用いることよりも有効であることが確認できた．

また，この Value Object Assembler パターンについて，複合するバリューオブジェクトの数を 1 から 4 に変化させてそれぞれの場合の処理時間を比較した．このときバリューオブジェクトの数が 3 から 4 に移るとき処理時間が大きく上

昇した．これには何らかのボトルネックが存在すると考え，次の 4 点のいずれかが原因であると推測して調査した．

(1) サーバのメモリ不足
(2) サーバの CPU 不足
(3) 不適切なメソッド呼び出し
(4) キューの待ち時間

(1)について検証するためにサーバのメモリを 632MB から 0.99GB に増設して性能の測定を行ったが，性能の向上は認められなかった．次に(2)について検証するためパフォーマンス測定ツールを用いてサーバとして使用しているマシンの CPU 利用率を測定したところ，100%に達していなかった．このため CPU が不足していることが原因であるとは考えられない．次に(3)を検証するために，不適切なメソッド呼び出しに処理時間のロスが発生していないか，メソッド単位に処理時間の計測を行った．このときサーバ側で行われる getBookVO()メソッドの処理時間のみが，クライアント数の増加とともに大きく上昇していた（図 7）．

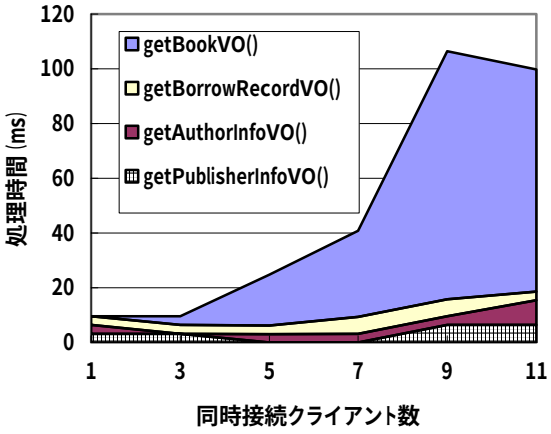


図 7 : Value Object Assembler パターン（バリューオブジェクト 4 個） のサーバ側各メソッド処理時間

上のグラフで示す 4 つの get()メソッドは，サーバ内で getBookVO()から順に処理される．一番最初に呼び出される getBookVO()メソッドのみ処理時間が急激に増加していることから，クライアントが同時にこのメソッドを呼び出したことでサーバに対するメソッドリクエストのキューがたまってしまい，それがボトルネックになったと考えた．このメソッドは同一サーバ内の EJB から EJB へのローカルメソッド呼び出しである．このことからリモートメソッド呼び出しだけでなくローカルメソッド呼び出しもボトルネックとなりうることがわかる．エンタープライズシステムのように多数のクライアントからのアクセスが考えられるシステムでは，ローカルメソッド呼び出しがボトルネックとなることも考慮しなくてはならないことがわかる．

8. 考察

EJB パターンを用いることでサーバの処理速度が大幅に向上する．これはリモートメソッド呼び出しの回数を減少することが主な要因である．つまりリモートメソッド呼び出しのパフォーマンスコストは高く，これを考慮することは EJB システムの性能設計において重要である．また，パターンを組み合わせることでさらにパフォーマンスが向上することが確認できた．パターン単体ではボトルネックを排除しきれないためである．よってパターンを適切に組み合わせることは性能向上に有効であると言える．

ボトルネックの検証から，リモートメソッド呼び出しだけでなくローカルメソッド呼び出しのリクエストキューの待ち時間もボトルネックになりうることがわかった．しかし多層システムではボトルネック要因が飛躍的に増えるため，その原因を探るのは非常に時間的コストの高い作業である．多量のクライアントを扱い，またシステムダウンの許されないエンタープライズシステムにおいてパフォーマンスを上げるにはボトルネックの検証が重要である．その際まず着目すべきなのはリモートメソッド呼び出しの行われる各層間のネットワーク部分である．

9. まとめ

実際にシステムを構築するというプロセスを通して，EJB 初心者の観点から EJB パターンの実装の方法やパフォーマンス向上の結果，システムを構築する際の注意点などを探った．われわれの経験から作成したこの文書から EJB に初めて触れる人が EJB パターンの効果や実装方法を感じ取ってくれれば幸いである．

今後，J2EE によるシステム構築は現在よりも多くなると考えられる．しかし開発者が J2EE 技術開発に追いつけていないという実情もあるため J2EE パターンの普及と開発者の育成が課題となるだろう．

今後の課題として，パターンの普及や様々なパターン組み合わせ例の検証が必要である．

参考文献

[1] R. Adatia, et al.:“プロフェッショナル EJB”, インプレス (2002).
[2] ディーパック・アラー, ほか:“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).
[3] F. Marinescu :“EJB デザインパターン”, 日経ＢＰ社 (2003).
[4] Sun Java Center : Sun Blueprints, http://jp.sun.com/blueprints/
[5] 皆本 房幸：“JBoss 入門”, 技術評論社(2003).
[6] 日立ソフトウェアエンジニアリング(株)インターネットビジネス部:“J2EE パフォーマンスチューニング徹底解説”, 技術評論社(2003).