平成22年度 修士論文

部分的再ロードによる Java プログラムの再起動の高速化手法

東京工業大学大学院 情報理工学研究科
数理・計算科学専攻

学籍番号 09M37264

別役 浩平

指導教員

千葉 滋 教授

平成23年2月28日

概要

Java 言語において，一部のクラスの定義を変えてプログラムを再起動するには，新たに作成したクラスローダを用いてアプリケーションを再ロードする必要がある．この再起動の手法は，アプリケーションサーバや DI コンテナ等で利用されている．一般に，その時点でアプリケーションをロードしているクラスローダを破棄し，新たに作成したクラスローダで全クラスをロードし直すことで実現される．ユーザからリクエストの度に再起動することで，対話的なアプリケーション開発を実現出来るが，全クラスの再ロードによる時間的コストが非常に大きく，サーバのパフォーマンスを低下させる．そこで本研究では，修正したアプリケーションを部分的に再ロードすることで，再起動を高速化する手法を提案する．再ロードすべきなのは変更されたクラスのみであり，全クラスを再ロードする必要はない．再起動時にどのクラスが変更されたかがわかれば，再ロードすべきクラスと再利用可能なクラスを分割することが出来る．しかし，変更されたクラスのみを新たに作成した子ローダで再ロードし，その他のクラスは親ローダでロード済みのものを再利用するというシンプルな実装では，クラスローダの仕様から上手く動作しない．変更したクラスに依存しているクラス群も新たに作成した子ローダで再ロードすれば，期待する動作が得られ，なおかつ親ローダでロード済みのクラスを再利用することが出来る．また，アプリケーションの各クラスに有限個の版があり，それらの組合せを変えて再起動する状況では，各クラスの各版毎にそれをロードするクラスローダを作成し，後の再起動時にそれらを部分的に親ローダとして活用することで，さらに多くのクラスを再利用することが出来る．しかし，この手法では一度に変更するクラス数が増えれば増えるほど，それに応じて作成するクラスローダインスタンスの数も増加するため，その時間的コストにより再起動がかえって遅くなる場合がある．そこで上記の 2 つの手法を我々の per-session AOP フレームワークにそれぞれ実装し，フレームワーク上で動作する Web アプリケーションにリクエストを送信する実験を行った．その結果，どちらの手法が優れているかが状況によって変わることが判明した．

To modify some definitions of classes of an application and restart it, the application should be reloaded by a new class loader. This approach is, for example, used for HOT deployment. However, it degrades the performance of a server application if reloading all the classes of an application. This paper proposes a technique to speed up restarting an application by partial class reloading. When an application is restarted, if we know which classes are modified, we can reuse classes that have been already loaded and are not changed at this restart. Furthermore, when each class of an application has some versions and the application is restarted repeatedly with combination of different versions of classes, we can reduce reloading costs by creating a class loader responsible for loading each version of a class. This allows us to reuse not only original classes but also various versions of each class. However, the latter approach may degrade the performance because a large number of class loader instances are created when we modify a number of classes. We applied above two approaches to our per-session AOP framework respectively and provided guidelines which approach you should use.

謝辞

本研究を進めるにあたり，研究の方針や論文の組み立て方について指導をしていただいた指導教員の千葉滋教授に感謝致します．

堀江倫大氏には，論文の書き方，プログラムの実装方法について親身になって指導をしていただきました．心より感謝致します．

森田悟氏には，研究室生活を送るにあたり，様々な助言をしていただきました．心より感謝致します．

戸部敦氏には，本研究の基盤となるソフトウェアの構築方法を指導していただきました．心より感謝致します．

そして、論文のスタイルファイルを作成して頂いた光来健一氏に感謝致します。最後に、本研究を行う上で励まして頂いた同研究室の皆様に心から感謝致します。

目 次

第 1 章 はじめに **9**
1.1 手法 1 11
1.2 手法 2 11
1.2.1 応用：Per-session AOP フレームワーク 11
1.3 実験 14
1.4 本稿の構成 15

第 2 章 関連研究 **16**
2.1 Dynamic Class Loading in the Java Virtual Machine . . . 16
2.2 HotSwap 16
2.2.1 Instrument API を用いた DAOP システムの実現 . 17
2.2.2 Wool 19
2.2.3 JAsCo, DJAsCo 19
2.3 Sister Namespace 19

第 3 章 新規クラスローダを用いたアプリケーションの再ロード **22**
3.1 クラスローダのアーキテクチャ 22
3.1.1 標準のクラスローダ 22
3.1.2 ClassLoader クラス 23
3.1.3 クラスのロード手順 24
3.1.4 開始ローダと定義ローダ 24
3.1.5 クラスローダの名前空間 25
3.1.6 アプリケーションのロード例 26
3.2 変更したクラスの再ロード 26

第 4 章 部分的再ロードによる再起動の高速化手法の提案 **29**
4.1 手法 1 29
4.1.1 Hot deploy への適用 30
4.2 手法 2 31
4.2.1 再起動時に使用するクラスローダの決定法 32
4.2.2 各クラスローダがロードするクラス群 33

第 5 章　応用 **35**
5.1　Per-session AOP フレームワーク 35
5.1.1　Per-session AOP フレームワークの実装 35
5.1.2　Per-session AOP フレームワークの問題点 39
5.2　フレームワークの効率化 40
5.2.1　提案手法の適用 40
5.2.2　キャッシュ 43

第 6 章　実験 **45**
6.1　実験概要 45
6.2　結果と考察 46
6.2.1　キャッシュ 0 46
6.2.2　キャッシュ 25-75 49

第 7 章　まとめと今後の課題 **57**
7.1　まとめ 57
7.2　今後の課題 57

図 目 次

1.1 アプリケーションの再ロード 9
1.2 DI コンテナの利用例 10
1.3 AspectJ のアスペクト 13
1.4 ホスティングサービス 14

2.1 エージェントクラス 18
2.2 クラスローダの切り替え 20
2.3 通常の式とリフレクションの比較 21

3.1 ClassLoader クラス 23
3.2 DirClassLoader クラス 25
3.3 サンプルアプリケーション 26
3.4 HOT deploy 28

4.1 子ローダの loadClass メソッド 30
4.2 手法 1 を用いた Hot deploy 31
4.3 手法 1 と 2 32
4.4 再起動時に使用するクラスローダの決定アルゴリズム . . . 33
4.5 クラス群の作成アルゴリズム 34

5.1 Tomcat のアーキテクチャ 36
5.2 Context.xml 36
5.3 Valve 37
5.4 SessionGetterValve クラス 37
5.5 WebappWrapper クラス 38
5.6 クラスローダの階層 38
5.7 掲示板アプリへのリクエスト送信実験 39
5.8 処理の内訳 40
5.9 GluonJ のアスペクト 41
5.10 変更対象クラスの取得 41
5.11 GlueWeaveLoader.findClass メソッド 42
5.12 再起動時に使用するクラスローダの決定アルゴリズム (改訂版) 43

5.13 クラス群の作成アルゴリズム (改定版) 44
5.14 CacheLRU クラス 44

6.1 実験用アスペクト 46
6.2 平均応答時間（手法 2） 48
6.3 アスペクトの使用数 49
6.4 クラスローダの平均活用率 50
6.5 平均応答時間（手法 0） 51
6.6 平均応答時間（手法 1） 51
6.7 平均応答時間の比較（手法 0，手法 2） 52
6.8 アスペクト使用数と平均応答時間の関係 52
6.9 クラスローダ活用率と平均応答時間の関係 53
6.10 アスペクト使用数と平均応答時間の関係（キャッシュ 25） 53
6.11 クラスローダ活用率と平均応答時間の関係（キャッシュ 25） 54
6.12 アスペクト使用数と平均応答時間の関係（キャッシュ 50） 54
6.13 クラスローダ活用率と平均応答時間の関係（キャッシュ 50） 55
6.14 アスペクト使用数と平均応答時間の関係（キャッシュ 75） 55
6.15 クラスローダ活用率と平均応答時間の関係（キャッシュ 75） 56

表 目 次

6.1 実験環境 45
6.2 コールグラフ上のクラス数 47
6.3 m/n とクラスローダ活用率 50

第1章　はじめに

今日，Web アプリケーションの開発には Java，PHP，Perl，C++ 等，様々なプログラミング言語が利用されている．次のような利点から Web アプリケーションの開発言語として特に Java（サーブレット）が広く普及している．

パフォーマンス サーブレットは初回リクエスト時に一度だけロードされ，その後はメモリ上に常駐するため，二回目以降のリクエストに対するレスポンスが高速である．

安定性 サーブレットでエラーが発生しても，Web サーバは影響を受けない．

移植性 C++ 等で書かれる CGI スクリプトと異なり，サーブレットは Java で書かれるため WORA（*Write Once, Run Anywhere*）という Java の利点がある．

セキュリティ サーバを介して実行されるため，ビジネスロジックがクライアントに直接参照されない．また，サーブレットは互いに独立しているため，1 つのサーブレットのエラーが他に影響しない．

一般にアプリケーションサーバは Java 言語により記述されるため，Java 言語と他の言語で記述されたアプリケーションとではリクエストの処理方法が異なる．CGI スクリプトでは，リクエストの度にプロセスを再起動するが，Java アプリケーションの場合，初回リクエスト時にサーバプロセスがクラスローダを用いてアプリケーションをロードし，インスタンスを作成する（図 1.1）．以後のリクエストはキャッシュされたインスタンスを用いてリクエストが処理される．サーバの稼働中にアプリケーションが

```
1 WebappClassLoader loader = new WebappClassLoader();
2 Class clazz = loader.loadClass(/* サーブレットのクラス名*/);
3 Servlet servlet = (Servlet)clazz.newInstance();
4 /* servlet のメンバの呼び出し*/
```

図 1.1: アプリケーションの再ロード

```
1 Interface PageLayout{
2  String getTitlePos(); String getbgColor(); .. }
3
4 Class HomePage extends HttpServlet{
5  PageLayout p;
6  void setLayout(){
7    setTitle(p.getTitlePos());
8    setColor(p.getbgColor()); .. }}
```

図 1.2: DI コンテナの利用例

修正された場合は，初回リクエスト時と同様に新たにクラスローダを作成し，ロードし直すことで反映される．以後，Java アプリケーションの再起動とはアプリケーションを再ロードし，再実行するという一連の動作を表すものとする．

この再起動の手法は，アプリケーションサーバだけでなく DI（*Dependency Injection*）コンテナ [6] でも利用されている．DI コンテナとは，DI というデザインに基づいて作られたコンポーネント群を管理するソフトウェアである．DI とはコンポーネント間の依存関係をソースコードから取り除くことで，プログラムの実行時までコンポーネント同士が依存関係を持たないようにするデザインパターンである．依存関係は， DI コンテナが設定ファイルを読み込み，指定されたクラスのインスタンスを自動的に作成し，依存元のオブジェクトのフィールドに代入することで実行時に実現する．図 1.2 に例を示す．設定ファイルには，HomePage クラスのインスタンスを作成するときは，フィールド p を MyLayout クラスのインスタンスで初期化する，などと記述することができる．ここで MyLayout クラスは PageLayout インタフェースを実装しているものとする．HomePage クラスは静的には MyLayout クラスに依存していないが，設定ファイルという外部モジュールを用いることで，実行時に MyLayout クラスへの依存性を注入することができる．

このように DI コンテナにおいてもアプリケーションサーバと同様に，その上で動作するアプリケーションを管理する必要がある．DI コンテナもまた Java 言語を用いて記述されているため，アプリケーションが修正された際には，上で述べたように新たにクラスローダ作成し，再ロードすることで再起動が実現される．

アプリケーションの再開発時には，リクエストの度に再起動されることが望ましい．こうすることでアプリケーションへの変更が即座に反映され，挙動を確認出来るため開発効率が向上するからである．しかし，Java アプリケーションの再起動は全クラスを再ロードすることにより実現されるため，その時間的コストによりサーバのパフォーマンスを低下させる．

再ロードすべきなのは変更されたクラスのみであり，全クラスを再ロードする必要はない．必要最低限のクラスのみを新たに作成したクラスローダでロードし，その他のクラスは既にロード済みのものを再利用すれば良い．Java クラスローダは親子関係を持っており，子ローダは親ローダに問い合わせることでロード済みのクラスを再利用することが出来る．つまり，新たに子ローダを作成する際には，再利用したいクラス群をロード済みのクラスローダを親ローダとして設定し，子ローダにアプリケーションをロードさせることで，再ロードが必要なクラス以外の無駄な再ロードを防ぐことが出来る．そこで本研究では，修正したアプリケーションを部分的に子ローダで再ロードし，その他のクラスは親ローダでロード済みのクラスを再利用することで再起動を高速化する 2 つの手法を提案する．

1.1　手法 1

再起動時にどのクラスが変更されたかがわかれば，再ロードすべきクラスと再利用可能なクラスを分割することが出来る．しかし，変更されたクラスのみを新たに作成した子ローダで再ロードし，その他のクラスは親ローダでロード済みのものを再利用するというシンプルな実装では，3.1 節で述べるクラスローダの仕様からクラスの変更をアプリケーションに反映できない．この問題は，変更したクラスに依存しているクラス群も新たに作成した子ローダで再ロードすることで解決出来る．こうすることでクラスの変更をアプリケーションに反映出来，なおかつ変更したクラスとそれに依存しているクラス以外の再ロードを省略することが出来る．

1.2　手法 2

アプリケーションの各クラスに有限個の版があり，それらの組合せを変えることでアプリケーションを修正し，繰り返し再起動する状況を考える．このような状況下では，再起動時に使用するクラスの各版毎にそれをロードするクラスローダを作成し，後の再起動時にそれらのいくつかを部分的に親ローダとして活用することが出来る．この手法は，アプリケーションの修正後にいくつかのクラスの版を元の版に戻すという操作をする際に効果が大きい．

1.2.1　応用：Per-session AOP フレームワーク

この前提状況に合致するシステムに per-session AOP フレームワーク [21] がある．このフレームワークでは，ユーザからのリクエストの度にク

ラスローダを作成し，ユーザが選択したアスペクト群を Web アプリケーションに織り込み，再ロードする．こうすることでユーザ毎に見栄えや機能の異なる Web アプリケーションを実現可能にしている．なお，サーバにはあらかじめ有限個のアスペクトが登録されている．アスペクトをクラスの版とみなせば手法 2 をフレームワークに適用することが出来る．

AOP (Aspect Oriented Programming)

AOP（*Aspect-Oriented Programming*）とは，OOP（*Object-Oriented Programming*）では解決出来ないモジュール間にまたがる横断的関心事（*crosscutting concern*）をアスペクトと呼ばれる独立したモジュールに分離するプログラミング技法である．

- アスペクト
  ソフトウェアの横断的関心事を記述するためのモジュール．横断的関心事はポイントカットとアドバイスの組から成る．
- ジョインポイント
  プログラムのある実行点を表す．例えば，メソッドの呼び出し・実行，コンストラクタの呼び出し・実行，フィールドの参照・代入，オブジェクトの初期化などである．
- ポイントカット
  ジョインポイントの集合．条件を指定することでプログラム中のジョインポイントを限定し、集合を作る。
- アドバイス
  ポイントカットされたプログラムの実行点で行う処理．
- 織り込み（*weave*）
  アドバイスに記述された処理をプログラムの処理に反映させる処理．AOP の実装によって異なるが，アドバイス本体をジョインポイントに埋め込む，アドバイス本体を呼び出すコード（フック）を埋め込むなどの方法がある。

図 1.3 に AspectJ[9] のアスペクト例を示す．2-3 行目はポイントカットの定義である．クラス A の foo メソッドの実行と bar メソッドの実行（二つのジョインポイント）をポイントカットしている．5-7 行目はアドバイスの定義である．ログ出力を行うコードが foo メソッドと bar メソッドの冒頭に織り込まれる．

アスペクトの織り込み技法には，主に次の 3 つがある．

```
1 public aspect LoggerAspect {
2   pointcut loggedMethods() :
3     execution(void A.foo()) || execution(void A.bar());
4
5   before() : loggedMethods() {
6     System.err.println(“execute ...”);
7   }
8 }
```

図 1.3: AspectJ のアスペクト

- Compile-Time Weaving
  コンパイル時にアスペクトを織り込む技法であり，AspectJ で用いられている．AspectJ コンパイラは、コンパイル時にアドバイスをメソッドに変換し、その呼び出しをジョインポイントの前後、またはその場所に挿入する。AspectJ コンパイラで作成されたクラスファイルは通常の Java 仮想マシンで実行することができる。

- Load-Time Weaving
  クラスのロード時にアスペクトを織り込む技法であり，GluonJ[3] で用いられている．GluonJ では java.lang.Instrument API を用いることでクラスのロードをインターセプトし，ロード前のバイトコードにアスペクトの内容を反映させる．

- Dynamic Weaving
  プログラムの実行中にアスペクトを織り込む技法である．この手法を用いればアプリケーションを再起動することなく，特定のクラスの振る舞いを高速に変更することが出来る．

Per-session AOP フレームワークでは上記の織り込み技法の内，Load-Time Weaving を採用している．つまり，ユーザ（リクエスト）毎に作成されたクラスローダにロード要求があった際，ユーザが選択したアスペクトをクラスに織り込んだ後，ロードする．

見方を変えると，図 1.3 の LoggerAspect とクラス A の組は，クラス A の新たな版であるとみなすことが出来る．そこで，手法 2 を Per-session AOP フレームワークに適用する際には，アスペクト毎にクラスローダを作成すればよい．つまり，クラスローダはアスペクトにより変更された新版のクラスをロードする．ユーザからのリクエスト時には，そのユーザが選択したアスペクトを担当するクラスローダがシステム内に既に存在すれば，それを親ローダとして活用すれば，ロード済みのクラスを再利用することが出来る．

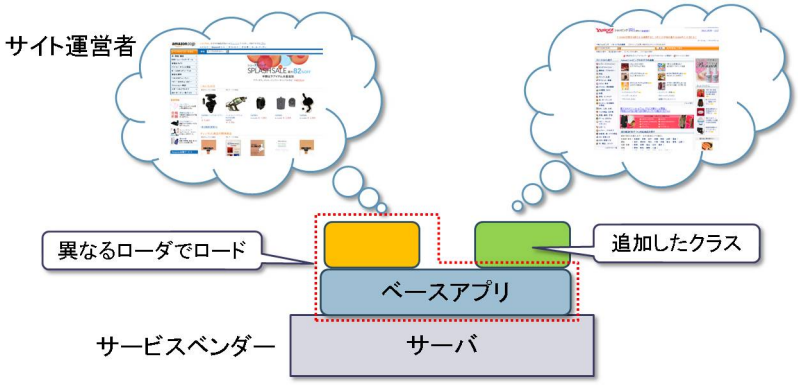

図 1.4: ホスティングサービス

ホスティングサービス

このフレームワークを用いることで，例えば図 1.4 のようなホスティングサービスを実現出来る．この例では，サービスベンダーがショッピングサイトの基本機能のみを備えた Web アプリケーションを提供し，各サイト運営者はクラスを追加したり，ベースアプリにアスペクトを織り込むなどしてアプリを拡張する．そしてサイトのユーザからリクエストが来た際には，URL 等の各サイト固有の識別子毎にクラスローダを作成し，アプリを再ロードする．この例では，左のサイトのユーザからリクエストが来た際には赤の点線で囲まれた部分を新たに作成したクラスローダでロードし，アプリを実行する．

1.3　実験

全クラスを再ロードする場合と手法 1 と 2 のパフォーマンスを比較するために 3 つのケースそれぞれを per-session AOP フレームワークに実装し，フレームワーク上で動作する Web アプリケーションにリクエストを送信する実験を行った．その結果，2 つの提案手法ともに全クラスを再ロードする場合に比べて高速であることが分かった．

また，再起動時に再ロードするクラス数は手法 2 の方が少なくなるが，一度に変更するクラス数が増えれば増えるほど，それに応じて作成するクラスローダインスタンスの数も増加するため，その時間的コストにより手法 1 よりも再起動が遅くなる場合がある．上記の実験の結果からどちらの手法が優れているかが状況によって変わることが判明した．

1.4　本稿の構成

次章から，本研究の背景と提案手法についてのべる．本稿の構成は以下の通りである．

2 章： 関連研究
クラスローダに関する関連研究と部分的再ロードに関する関連研究について述べ，アプリケーションの再起動への適用可能性と問題点について述べる．

3 章： 新規クラスローダを用いたアプリケーションの再ロード
クラスローダのアーキテクチャについて詳細に述べ，それを利用して，修正したアプリケーションの部分的再ロードを実現する方法について検討する．

4 章： 部分的再ロードによる再起動の高速化手法の提案
新しく提案する 2 つの再起動の手法について述べる．

5 章： 応用
提案する再起動の手法がどういった局面で適用できるかを per-session AOP フレームワークへの適用例を用いて示す．

6 章： 実験
per-session AOP フレームワーク上で動作する Web アプリケーションへのリクエスト送信実験の結果を示す．その結果から，提案手法が既存手法よりどの程度優れているか，また 2 つの提案手法どちらが優れているかを示す．

7 章： まとめと今後の課題
本研究をまとめて，今後の課題を述べる．

第2章　関連研究

本章では 2 つのクラスローダについての関連研究と DAOP（*Dynamic Aspect Oriented Programming*）システムについて述べる．DAOP を用いれば実行中のプログラムの再開発を容易に出来るが，クラスの拡張性の面で問題がある．

2.1 Dynamic Class Loading in the Java Virtual Machine

Liang らは論文 [10] 内で，Java のクラスローダの主要概念について述べている．

Lazy loading クラスのロードはオンデマンドに行われる．そのため，メモリの消費が抑えられ，システムのレスポンスが向上する．

Type-safe linkage 動的なクラスのロードにより型の安全性を犯してはならない．

User-definable class loading policy クラスローダはファーストクラスオブジェクトであり，プログラマーはクラスのロードポリシーを自由に定義することが出来る．

Multiple namespaces 異なるクラスローダは異なる名前空間を持つ．異なるクラスローダでロードされた同一名のクラスは JVM（*Java Virtual Machine*）上で異なる型であるとみなされる．

また，親子関係を用いたロードアルゴリズムや JVM 上での型の実装についても記述されており，我々はこれらを利用し，修正したアプリケーションの部分的再ロードを実現した．クラスローダのアーキテクチャの詳細は 3 章で述べる．

2.2 HotSwap

JVM 上にロード済みのクラスを新たな定義で置き換える技術に HotSwap がある．この技術は DAOP システムで頻繁に利用されている．DAOP シ

ステムでは 1.2.1 節で述べたアスペクトの織り込み技法の内，Dynamic Weaving が利用されている．まず一般的な DAOP システムの実現方法について述べ，DAOP に関する研究について述べる．

2.2.1　Instrument API を用いた DAOP システムの実現

Instrument API は DandyJ[8] などの DAOP システムで広く使われている．java.lang.Instrument パッケージでは，Java エージェントと呼ばれるプログラムの計測機構を提供しており，計測対象のアプリケーションのバイトコードをロード時・実行時に変換することが出来る．DAOP システムでは実行時のバイトコード変換機構を利用している．

Instrumentation インタフェース

このインタフェースではエージェントクラスがプログラムを計測するためのコアとなる機構を提供している．実行時のバイトコード変換をするための主な機能を以下に示す．

- redefineClasses メソッド
  引数に渡された ClassDefinition オブジェクトでクラスの定義を置換する．ClassDefinition オブジェクトとは java.lang.Class オブジェクトと新しいクラスファイルバイトをバインディングするオブジェクトである．なお，このメソッドを使った再定義では，メソッドの本文，定数プール，および属性の変更が可能である．ただし，フィールドまたはメソッドの追加，削除，あるいは名前の変更，メソッドのシグニチャーの変更，あるいは継承の変更は出来ない．

- isRedefineClassesSupported メソッド
  プログラムが実行されている現在の JVM が Instrument API によるクラスの再定義をサポートしているかどうかを返す．クラスの再定義機能は JVM のオプション機能であり，HotSwap を用いたシステムの移植性の低さを示している．

このインタフェースのインスタンスを取得するためには，エージェントクラスを指定してプログラムを実行する必要がある．

エージェントクラス

Instrumentation インタフェースのインスタンスはエージェントクラスの premain メソッドの引数として取得することが出来る．エージェントク

```
1  public class Agent {
2   private Instrumentation inst;
3
4   public static void premain(String args,Instrumentation inst) {
5     Agent agent = new Agent(args, inst);
6     /* 計測開始*/
7   }
8
9   public Agent(String agentArgs, Instrumentation inst) {
10    this.inst = inst;
11  }
12
13  public void weave(String className, String aspectName) {
14    byte[] woven = getWoven(className, aspectName);
15    ClassDefinition def
16      = new ClassDefinition(Class.forName(className), woven);
17    inst.redefineClasses(new ClassDefinition[] { def });
18  }
19
20  byte[] getWoven(String className, String aspectName) {
21    /* バイトコードの生成*/
22    return /* 新たなバイトコード */;
23  }
24 }
```

図 2.1: エージェントクラス

ラスには，このメソッドを実装しておかなければならない．javaagent オプションを指定してアプリケーションを実行することで，アプリケーションの main メソッドが呼ばれる前にエージェントクラスの premain メソッドが呼ばれ，計測をすることが出来る．

java -javaagent:*jarpath*[=*options*]

jarpath にエージェントクラスを含む jar ファイルを指定する．*options* で指定した文字列もまた premain メソッドの引数に渡される．図 2.1 に DAOP を実現するエージェントクラスを示す．アプリケーションの main メソッドが呼ばれる前に 4 行目の premain メソッドが呼ばれる．引数の Insturmentation オブジェクトをエージェントクラスに渡し，計測を開始する．13 行目の weave メソッドは，引数に指定されたクラスにアスペクトを織り込み，クラスを再定義する．21 行目の getWoven メソッドは，クラスバイトをアスペクトの内容で編集し，新たなバイト配列を返す．

例えば，premain メソッド内で GUI（*Graphical User Interface*）を作成し，その GUI オブジェクトにエージェントを渡すなどして，常にエージェントクラスのメンバを呼び出せるようにしておく．こうすることで動的にアプリケーションの振る舞いを変更することが出来る．

2.2.2 Wool

Wool[14] は Java Platform Debugger Architecture（JPDA）のクラス再定義機能を利用した DAOP システムである．既存の DAOP の実現方法には，全てのジョインポイントに対してフックを挿入する手法や JVM を改造する手法がある．しかし，前者では全てのジョインポイントでアドバイスを実行するかどうかの判定をしなければならないためプログラムの実効速度が低下するという問題がある．Java 言語では可搬性が重要であるため，後者の手法も現実的でない．PROSE[12] では，ジョインポイントに対応する場所でデバッガがアドバイスを実行することで実現しているが，メインプログラムとデバッガとのコンテキストスイッチによるオーバヘッドが非常に大きく実用的でない．

このシステムではジャストインタイム・フック織り込み技術を用いることで，既存の DAOP システムよりも高い実行性能を実現した．この手法では，アスペクトの織り込み（プログラムの再定義）を可能な限りを遅延させ，オンデマンドに行う．つまり，再定義をする必要があるクラスが判明して初めて行われるため，無駄なフックをプログラムに織り込む必要が無い．

しかし，JVM のクラス再定義機能では，実行時にクラスにメンバを追加したり，クラス階層を変更出来ないため，アプリケーションの再開発には不向きである．

2.2.3 JAsCo, DJAsCo

JAsCo[15, 20] も Wool と同様に HotSwap を用いて DAOP システムのパフォーマンスを改善している．さらに JAsCo では，ジョインポイントに織り込まれるコード断片をキャッシュするシステム Jutta を提案し，アスペクトの実行系の性能も改善している．

DJAsCo[4] は JAsCo に Remote Pointcut[11] を実装することで分散環境に対応させた DAOP システムである．

しかし，JAsCo，DJAsCo にも Wool と同様にクラスの拡張性に制限ある．

2.3 Sister Namespace

オンラインサービスのように連続稼働する必要があるアプリケーションにおいては，停止させずに動的に修正出来ることが望ましい [13]．Java ではクラスローダ毎に名前空間が作られるため，新たに作成したクラスローダでアプリケーションをロードさせることで修正した同一名のクラスを

```
1 public Class loadClass(String name) {
2   if (/* name が変更対象クラス*/) {
3     ClassLoader loader = new CustomLoader();
4     return loader.loadClass(name);
5   }
6   else return super.loadClass(name);
7 }
```

図 2.2: クラスローダの切り替え

ロードすることが出来る．しかしながら，それを利用したアプリケーションのバージョンアップはクラスローダのバージョンバリアと呼ばれる制約によって強く制限されている．Sister Namespace ではこのバージョンバリアを緩めることで，あるクラスの版のインスタンスを異なるクラスの版の変数に代入可能にしている．

Web アプリケーションサーバで利用されている Hot deploy では　deploy 後の新コンポーネントが旧コンポーネントのインスタンス，例えばキャッシュやクッキー，セッションなどを扱えないという問題がある．本研究においても複数のクラスローダを作成し，アプリケーションをロードし直すため，提案手法を Web アプリケーションに適用する際には同様の問題が生じる．しかし，Sister Namespace を用いれば異なるクラスローダ（コンポーネント）間でインスタンスを共有できるため，この問題は解決できる．

次に，Sister Namespace を修正したアプリケーションの再起動に用いる方法を検討する．まず，アプリケーション内のあらゆる参照をにリフレクション API を用いた式に変形する必要がある．例えば，

```
App app = new App();
```

という式は，

```
Class clazz = mainLoader.loadClass("App");// 異なる版をロード
App app = (App) clazz.newInstance();
```

のように式変形を行う．これはアプリケーションの初回ロード時に行えばよい．なお，mainLoader にあらかじめ全てのクラスをロードさせておく．図 2.2 に，このクラスローダの loadClass メソッドを示す．ここで，CustomLoader とは変更したクラスをロードするクラスローダである．また，CustomLoader でロードされるクラス内の参照も全て mainLoader を使ったリフレクションに変更しておく．こうすることで，最小のロードコストで再起動を実現することが出来る．

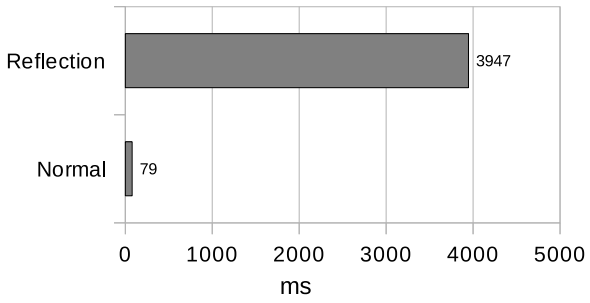

図 2.3: 通常の式とリフレクションの比較

しかし，この実現方法には 2 つの問題点がある．まず，1 つ目はリフレクション API を用いた呼び出しはオーバヘッドが大きいことである．以下のようなインスタンスの作成と空のメソッド呼び出しの実行時間を比較した．

```
// 通常の式
App app = new App();
app.func();

// リフレクション API を使った式
App app = (App) loader.loadClass("App").newInstance();
app.func();
```

図 2.3 はそれぞれを 1000 万回実行した際の測定結果である．このグラフから分かるように約 50 倍のオーバヘッドがある．2 つ目の問題点は，super() などの特殊なコンストラクタ呼び出しは，リフレクション API を用いて変形出来ないことである．これでは HotSwap のように拡張が制限されてしまう．

第3章　新規クラスローダを用いたアプリケーションの再ロード

本章ではまず，Java のクラスローダのアーキテクチャについて述べ，次に新たに作成したクラスローダを用いて，修正したアプリケーションを再ロードする方法について検討する．

3.1　クラスローダのアーキテクチャ

Java 言語ではクラスのロードはプログラムの実行中に動的に行われる [10]．クラスのロードは親子関係を持つ（木構造を構成する）複数のクラスローダにより実行され，java.lang.Class のインスタンスを生成する．

3.1.1　標準のクラスローダ

JVM が起動すると次の 3 つのクラスローダが作成される．

- ブートストラップクラスローダ
  Java の中核ライブラリ（JAVA_HOME/lib）をロードする．このクラスローダは JVM に組み込まれており，ネイティブコードで記述されている．ClassLoader クラスも java で書かれているため，このクラスをロードするクラスローダが必要である．その問題を解決するのがこのクラスローダである．

- 拡張クラスローダ
  拡張ディレクトリ（JAVA_HOME/lib/ext，java.ext.dirs プロパティで指定されたディレクトリ）内のクラスをロードする．

- システムクラスローダ
  CLASSPATH 変数内にあるクラスをロードする．通常，ユーザアプリケーションを起動するためのクラスローダである．

Java アプリケーションにおいて，ブートストラップクラスローダがクラスローダの木の根に当たる．その子ローダが拡張クラスローダ，さらにそ

```
1  class ClassLoader {
2    public Class loadClass(String name);
3    public final ClassLoader getParent();
4    protected Class findClass(String name);
5    protected final void resolveClass(Class c);
6    protected final Class defineClass(String name, byte[] buf, int
         off, int len);
7    protected final Class findLoadedClass(String name);
8    protected final Class findSystemClass(String name);
9    public static ClassLoader getSystemClassLoader();
10   ...
11 }
```

図 3.1: ClassLoader クラス

の子がシステムクラスローダである．アプリケーション内で新しく作成されたクラスローダのデフォルトの親クラスローダはシステムクラスローダである．

3.1.2　ClassLoader クラス

全てのクラスローダは ClassLoader クラスのサブクラスである（図 3.1）．

- loadClass メソッド
  JVM は未解決のクラス参照が見つかった際，クラスローダのこのメソッドを呼び出しで解決を試みる．クラスが見つからなければ ClassNotFoundException が投げられる．

- getParent メソッド
  親クラスローダを返す．親クラスローダがブートストラップクラスローダならば null を返す．セキュリティーマネージャが存在し，親クラスローダへのアクセスを許可しない場合に SecurityException が投げられる．

- findClass メソッド
  引数に指定されたクラス名を持つクラスを返す．このメソッドはオーバライドされ，独自アルゴリズムによってクラスを探索する．標準仕様ではクラスが見つからない場合，ClassNotFoundException が投げられる．

- resolveClass メソッド
  引数に指定されたクラスをリンクする。クラスが既にリンク済みならば何も行わずに復帰する．

- defineClass メソッド
  バイト配列から java.lang.Class のインスタンスを作成する．引数に渡されたバイト配列が有効なクラスでなかった場合，ClassFormatError が投げられる．

- findLoadedClass メソッド
  引数に指定されたクラスが既に JVM 上にロード済みであり，かつこのローダがそのクラスの開始ローダ（後述）として JVM に登録されていた場合は java.lang.Class のインスタンスを返す．それ以外の場合には null を返す．

- findSystemClass メソッド
  システムクラスローダ（getSystemClassLoader メソッドにより取得）を呼び出してクラスをロードする．クラスが見つからない場合は ClassNotFoundException が投げられる．

- getSystemClassLoader メソッド
  システムクラスローダを返す．

3.1.3　クラスのロード手順

JVM からロード要求があった際，loadClass メソッドを用いて次のような手順でクラスの探索を行う．

1. findLoadedClass メソッドを呼び出して，このクラスローダでクラスが既にロードされたかどうかを確認する
2. 未ロードの場合，親ローダの loadClass メソッドを呼び出す
3. 親ローダがロードできなかった（2 で ClassNotFoundException が投げられた）場合，findClass メソッドを呼び出して，このクラスローダ独自の方法でクラスを探し，defineClass メソッドを呼び出してクラスを定義する

3.1.4　開始ローダと定義ローダ

以下のような例を考える．

```
MyClassLoader loader = new MyClassLoader();
Class stringClass = loader.loadClass("java.lang.String");
```

```
1 private dir;
2 public DirClassLoader(String dir) {
3   this.dir = dir;
4 }
5 public Class loadClass(String name) {
6   ...
7   byte[] data = getData(dir, name);
8   return defineClass(name, data, 0, data.length());
9 }
```

図 3.2: DirClassLoader クラス

このコードでは loader の loadClass メソッドを呼び出すことで String クラスをロードしているが，実際には 3.1.3 で述べた手順から String クラスを定義するのは，ブートストラップクラスローダである．この例のように，あるクラス C がローダ L　の loadClass メソッドの結果として生成されるとき，L をクラス C の**開始ローダ**と呼ぶ．また，クラス C がローダ L の defineClass メソッドの結果として生成されるとき，L をクラス C の**定義ローダ**と呼ぶ．

以降，ロードと定義を同義として扱う．つまりクラス C ロードしたローダとは，C の定義ローダのことである．

3.1.5　クラスローダの名前空間

JVM において，クラスの型はそのクラスの名前と defineClass メソッドによりそのクラスを定義したクラスローダの対で一意に決定される．つまり，定義したクラスローダが異なれば同一名のクラスを JVM 上にロードすることができる．

図 3.2 のようなクラスローダを考える．このクラスローダはコンストラクタ引数に指定したディレクトリ内にあるクラスをロードする．ここで，getData メソッドとはディレクトリからクラスのデータを取得するメソッドである．まず，ディレクトリ oldDir 内にあるアプリケーションをロードし，実行させていたとする．次に，現在のアプリケーションを破棄し，newDir 内のアプリケーションを実行したいとする．このような時は新たに DirClassLoader インスタンスを作成（new DirClassLoader(“newDir”)）すればよい．もし，oldDir と newDir 内に同一名のクラスが存在してもそれを定義したクラスローダが異なるため，JVM 上にはそれぞれ異なる型として定義される．

```
1 class A {
2   void funcA() {
3     String s = "foo";
4     new B().funcB(s);
5   }
6   public static void main(String[] args) {
7     new A().funcA();
8   }
9 }
```

図 3.3: サンプルアプリケーション

3.1.6　アプリケーションのロード例

一般的な java アプリケーションがどのようにロードされるかを図 3.3 を用いて示す．

まず，main メソッドが定義されているクラスをシステムクラスローダがロード（defineClass メソッドにより定義）する．このクラスローダを L_S とすると，JVM 上にはクラスの型 <A,L_S > が定義される．次に main メソッドが呼び出され，クラス A の funcA メソッドが呼ばれる．funcA メソッド内では String クラスとクラス B を参照している．クラス C 内における他のクラスの参照は，そのクラスを定義したクラスローダ（ここでは L_S）によって解決される．つまり，funcA メソッドの実行中に

$$L_S\text{.loadClass(“String”);}$$
$$L_S\text{.loadClass(“B”);}$$

が呼び出される．String クラスなどの標準ライブラリは 3.1.3 で述べたロード手順により，ブートストラップクラスローダ（L_{Boot}）によってロードされ，JVM 上には型 <String,L_{Boot} > が定義される．一方，標準ライブラリでないクラス B のロードは，L_S 自身が実行する．つまり，L_S の findClass が呼び出され，JVM 上には型 <B,L_S > が定義される．

なお，一度解決されたクラス内の参照に対して，再びクラスローダが呼び出されることはない．例えば，アプリケーション内で再びクラス A の funcA メソッドが呼ばれたとしても，String クラスやクラス B の参照は，既に各々のクラスの定義にリンクされており，JVM がリンク先を参照することで解決される．

3.2　変更したクラスの再ロード

3.1 節で述べたクラスローダのアーキテクチャを考慮すると，変更されたクラスのみを新たに作成した子ローダで再ロードし，その他のクラスは

親ローダでロード済みのものを再利用するという再起動の手法が実現出来るように思われる．しかし，このような素朴な実装では上手く動作しない場合がある．

まず最初に以下のような例について考える．

```
class A {
  void funcA() { new B().funcB(); }
}
class B {
  void funcB() { new C().funcC(); }
}
class C {
  void funcC() {..}
}
```

p クラス A，B，C の全てが親ローダ L_P でロード済みであり，変更したのクラス B を新たに作成した子ローダ L_C にロードさせるとする．クラス A の funcA メソッドが呼ばれると JVM は L_P の loadClass メソッドを呼び出し，A クラス内の参照を解決しようとする．しかし，L_P によって解決可能なクラスは，L_P の親ローダ及び自身で解決可能なクラスなので，クラス A は L_C でロードさせたクラス B を参照できず，L_P でロード済みのクラス B を参照してしまう．これではクラス B への変更をアプリケーションに反映させることが出来ない．

新たに作成したクラスローダを用いて変更したクラスを含めた全クラスを再ロードすれば，クラスへの変更をアプリケーションに反映出来るが，その時間的コストがプログラムのパフォーマンスに大きく影響する場合がある．この手法の利用例として HOT deploy がある．HOT deploy とはサーバを再起動することなく，コンポーネントをデプロイ，アンデプロイする技術であり，JBoss Application Server[5]，Tomcat[17] などアプリケーションサーバや Seasar2[19] などの DI コンテナで利用されている．

HOT deploy の一種にユーザからのリクエストの度にアプリケーションの全クラスを再ロードするものがある．これは対話的な開発を行いたいときに利用されるが，リクエスト数が多いサーバにとっては，大きなオーバヘッドになりかねない．

図 3.4 はリクエストの度にアプリケーションの全クラスを再ロードする場合とそうでない場合に分けて，Health Watcher[7] アプリケーションにリクエストを送信した際の平均応答時間である．リクエストの度に再ロー

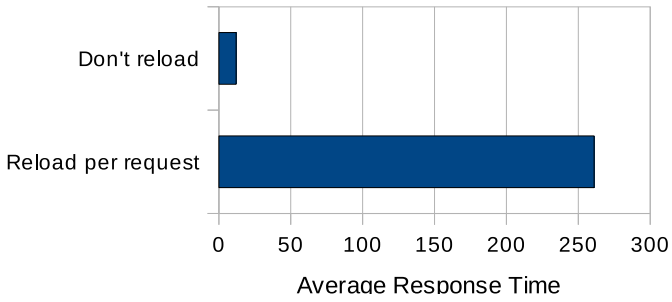

図 3.4: HOT deploy

ドした場合は，そうでない場合に比べて 20 倍以上遅く実用的でないことがわかる．

第4章　部分的再ロードによる再起動の高速化手法の提案

3.2 節で述べたように，変更されたクラスのみを子ローダでロードさせるという素朴な手法では，アプリケーションに変更を反映させることが出来ない．また，変更したクラスを含めた全クラスを再ロードするという手法はパフォーマンスの面で問題がある．そこで本章では，出来るだけロード済みのクラスを再利用しつつ修正したアプリケーションを再起動するための 2 つの手法について提案する．

4.1 節では，再起動時にクラスローダを一つしか作成しないが，比較的再ロードするクラス数が多い手法（手法 1），4.2 節では，クラスローダを多数作成するが，再ロードするクラス数が少ない手法（手法 2）について述べる．

4.1　手法 1

出来るだけ親ローダでロード済みのクラスを再利用しつつ，修正したアプリケーションを再ロードする方法について考える．3.2 節の例で問題なのは，L_P がロードしたクラス A から L_C がロードしたクラス B を参照できないことである．

この問題は，図 4.1 のようなクラスローダ L_C にアプリケーションを再ロードさせることで解決出来る．このクラスローダは変更対象クラスとそれに依存しているクラスならばロードし，その他のクラスは親ローダでロード済みのものを再利用する．つまり，変更対象クラスである B とそれに依存しているクラスである A はロードされ，その他のクラス（C）は親ローダ L_P でロード済みのものを再利用する．ここで，あるクラス X に依存しているクラスとは，クラス間の参照をたどることで X に到達可能なクラスのことである．これを用いることで，クラス A，B は L_P でロード済みのものを再利用することなく，L_C によってロードされる．このため，クラス A 内参照は L_C の loadClass メソッドにより解決され，変更された B を参照出来る．一方，クラス B が参照しているクラス C は L_P がロード済みのものを再利用するため，再ロードするクラス数を削減することが出来る．

```
1  public Class loadClass(String name) {
2    if (/* name が変更対象クラス，又はそれに依存しているクラス*/) {
3      Class clazz = findLoadedClass(name);
4      if (clazz == null) return findClass(name);
5      else return clazz;
6    }
7    else return getParent().loadClass(name);
8  }
9  public Class findClass(String name) {
10   byte[] b = null;
11   if(/* name が変更対象クラス*/) {
12     b = /* 新たなクラスの定義*/;
13   }
14   else {
15     b = /* クラスの原版*/;
16   }
17   return defineClass(name, b, 0, b.length());
18 }
```

図 4.1: 子ローダの loadClass メソッド

上で述べた手法を実現するためには，ロード時に変更対象クラスに依存しているクラス群を知る必要がある．クラス間の依存関係は Javassist[2] 等のバイトコードレベル API を用いれば探索可能である．

4.1.1　Hot deploy への適用

手法 1 は容易に Hot deploy に適用できる．どのクラスが変更されたかは，クラスファイルのタイムスタンプ等をチェックすることで知ることが出来る．タイムスタンプの変更を確認したら，変更されたクラスとそれに依存しているクラスをロードするクラスローダ（図 4.1）を作成し，アプリケーションのロードを行う．図 4.2 に 3 回 Hot deploy を行った際の例を示す．左から 1，2，3 回目とし，Original クラスローダは，変更されていないオリジナルのアプリケーションのクラスを全てロード済みであるとする．1 回目の Hot deploy ではクラス A，C を変更し，それらと依存しているクラスをロードするクラスローダを作成し再起動する．2 回目ではクラス B，E を変更し，それらと依存しているクラスをロードするクラスローダを作成し 1 回目に作成したクラスローダを親として再起動する．3 回目ではクラス A，D を変更し，それらと依存しているクラスをロードするクラスローダを作成し 1，2 回目に作成したクラスローダを親として再起動する．3 回目の再起動時には 1 回目と同様にクラス A を変更しているが，これについては問題ない．図 4.1 の 3 行目からわかるように自身でロード済みのクラスを先に探索するため，3 回目の再起動時にロード

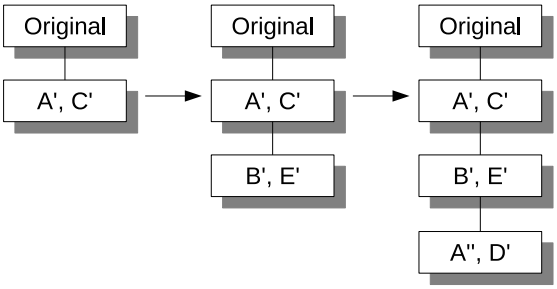

図 4.2: 手法 1 を用いた Hot deploy

されたアプリケーションでは必ず A'' が参照される．また，手法 1 を適用した Hot deploy を繰り返し行えば，クラスローダのインスタンスやロードされるクラスが増えつづけるため，メモリを圧迫する恐れがある．この点は，一定数以上のクラスローダが作られたら，全てのクラスローダを破棄し，新しく作成した一つのクラスローダでアプリケーションの全クラスを再ロードすれば問題ない．そして，作成したクラスローダをルートにして，また手法 1 を適用すれば良い．

4.2　手法 2

アプリケーションの各クラスに有限個の版があり，それらの組合せを変えて再起動する状況では，さらに再ロードするクラス数を削減することが出来る．例えば，アプリケーション内のクラス A，B をある版 A'，B' に変更して再起動し，次に A' のみ元の版 A に戻し，クラス C をある版 C' に変更して再起動するといった状況である．手法 1 では，初回の再起動時に新たに作成したクラスローダ L_1 が A'，B' をロードする．実際には A'，B' に依存しているクラス群もロードするが，この例では説明を簡単にするために省略する．二回目の再起動時には，新たに作成したクラスローダ L_2 が B'，C' をロードする．二回目の再起動時に L_1 を L_2 の親ローダとして利用すれば，L_2 で B' をロードする必要はないが，L_1 は A' をロード済みなので，期待する動作は得られない．そこで，初回の再起動時に A' をロードするクラスローダ L_A，B' をロードするクラスローダ L_B を個別に作成し，L_B を L_A の親ローダとする．二回目の再起動時には C' をロードするクラスローダ L_C を作成し，L_B をその親ローダとして活用すれば A' の影響を受けずに B' を再利用することが出来る．図 4.3 に，この時のクラスローダの木構造を示す．

このように，アプリケーションの各クラスの各版毎にそれをロードする

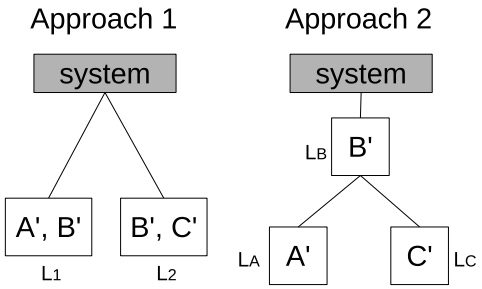

図 4.3: 手法 1 と 2

クラスローダを作成すれば，旧版のアプリケーションを部分的に再利用出来る．つまり，手法 2 は旧版のアプリケーションのバージョンを部分的に戻し，他のクラスに新たな変更を追加するといった複雑な再起動時に効果を発揮する手法であると言える．

4.2.1　再起動時に使用するクラスローダの決定法

図 4.4 に決定アルゴリズムの擬似コードを示す．

Input, Output S は再起動時に使用するクラスの版の集合（原版を除く）．P は S の要素の優先度の集合である．優先度は，そのクラスに依存しているクラス群のファイルサイズの合計等，ロードのコストが大きいものが高くなるようにあらかじめ設定，計算しておく．

1-10 行目 システム内でクラスローダの木を保持しておく．V はノード（クラスローダ）の集合で，初期値はシステムクラスローダ 1 個である．E はエッジ（クラスローダの親子関係）の集合である．例えば，(L_P, L_C) は L_P は L_C の親であることを表す．available(L) は L を親として活用することで，L の親も含めて再利用可能なクラスの版の集合を返す手続きである．3-7 行目で決定した L_P を親クラスローダとして活用することでロード済みのクラスの版を最大限再利用することが出来る．

11-16 行目 S の要素の内，再利用不可能なクラスの版について，それらのロードを担当するクラスローダをそれぞれ作成し，優先度が最も高い版をロードするクラスローダを L_P の子，その次に高い版をロードするものをその子,... と親子関係を設定する．こうすることで，ロードコストの大きい版をロードするクラスローダが木の根の近くに配置され，後に親として活用しやすくなる．

Input: クラスの版の集合 S とクラスの版それぞれの優先度の集合 P
Output: 再起動時に使用するクラスローダ $L_{restart}$

1: クラスローダの木 $T = (V, E)$ (初期値 $V = \{L_S\}$, $E = \phi$) について
2: $A \leftarrow \{L \in V \mid \text{available}(L) \subseteq S, \text{available}(L) \neq \phi\}$
3: **if** $A = \phi$? **then**
4: 　$L_P \leftarrow L_S$
5: **else**
6: 　$L_P \leftarrow A$ の中で $|\text{available}(L)|$ が最大となるもの
7: **end**
8: $S' \leftarrow S$ - available(L_P)
9: P を使って S' の要素を優先度が高い順にソート
10: $L_{restart} \leftarrow L_P$
11: **foreach** C S' {
12: 　$L_C \leftarrow C$ をロードする新規クラスローダ
13: 　$V \leftarrow V + L_C$
14: 　$E \leftarrow E + (L_{restart}, L_C)$
15: 　$L_{restart} \leftarrow L_C$
16: }

図 4.4: 再起動時に使用するクラスローダの決定アルゴリズム

このアルゴリズムによって決定された $L_{restart}$ にアプリケーションを再ロードさせる．

4.2.2　各クラスローダがロードするクラス群

図 4.4 で示したクラスローダの木において，各ノード（クラスローダ）がロードするアプリケーションはそれぞれ異なる．それらを親ローダとして活用し，新たにクラスローダを作成する際には，親ローダがロードするアプリケーションのクラス間の依存関係を考慮して，ロードすべきクラス群を決定する必要がある．

まず，アプリケーションの起動時にクラス間の依存関係を静的に探索し，各クラスに依存しているクラス群を決定しておく．クラス間の依存関係を表す有向グラフ $G = (V, E)$ の $\forall v \in V$ について，$S_v = \phi$ を作成し，$\forall v \in \{v | S_v = \phi\}$ について次のような操作を行う．

1. $S_v \leftarrow S_v + v$.
2. $\forall u \in \{u | (u, v) \in E\}$ について，

```
1: S ← S_C
2: foreach L_P (L_C の全ての親ローダ) {
3:   if L_P がロードを担当するクラスの版 D が S 内のクラスに参照を持つ ?
4:   then
5:     S ← S + S_D
6:   end
7: }
```

図 4.5: クラス群の作成アルゴリズム

- $S_u \neq \phi$ ならば，$S_v \leftarrow S_v + S_u$.
- $S_u = \phi$ ならば，u について（ 1 ）から同様の操作を行い，$S_v \leftarrow S_v + S_u$.

以上の操作により作成された S_v が v 及びそれに依存しているノード（クラス）の集合となる．

あるクラスの版 C のロードを担当するクラスローダ L_C がロードするクラス群 S は図 4.5 のように作成される．

1 行目 原版のアプリケーションのクラス間の依存関係を元に初期値を与える．

2-7 行目 L_C の親ローダ L_P がロードするクラスの版 D が S 内のクラスに参照を持つならば，S_D を S に追加する．

ここで，あるクラスの版 C と C の原版のクラス名は等しいことに注意されたい．

このように部分的にアプリケーションの依存関係を更新することで，小さなオーバヘッドでクラス群を決定することが出来る．

第5章　応用

4章で提案した2つの手法の応用例として，per-session AOP フレームワークの拡張を行った．まず最初に，per-session AOP フレームワークについて述べ，次に提案手法の実装について述べる．

5.1　Per-session AOP フレームワーク

AOP 言語の処理系は，プログラムのバイトコードを書き換えることでアスペクトの内容を反映させる．見方を変えると，アスペクトはソフトウェアの振る舞いを変更するモジュールであるとみなすことが出来る．

Per-session AOP フレームワークでは，リクエストに含まれるセッションからユーザ ID を判別し，ユーザの設定ファイルから，そのユーザが選択したアスペクトの情報を得る．そして，それらのアスペクトをウィーブすることで，ユーザ毎に異なる振る舞いをする Web アプリケーションを実現可能にしている．

5.1.1　Per-session AOP フレームワークの実装

Per-session AOP フレームワークは Tomcat サーバの一部として実装されている．図5.1 に Tomcat のアーキテクチャを示す [1].

図のように Tomcat はネストされた複数コンポーネットにより構成されている．Per-session AOP フレームワークは，これらのコンポーネントの内 Valve と Wrapper に独自のものを使用することで実現されている．独自のコンポーネントを使用するためには，Tomcat の設定ファイル（Context.xml）でそれらを宣言しなければならない（図5.2）．

Valve

Valve はリクエストをインターセプトし，前処理を行うためのコンポーネントである（図5.3）．Per-session AOP フレームワークでは，リクエストからユーザのセッションを取得するために独自の Valve を使用している．図5.4 にフレームワークにおける Valve を示す．Tomcat 内の一連

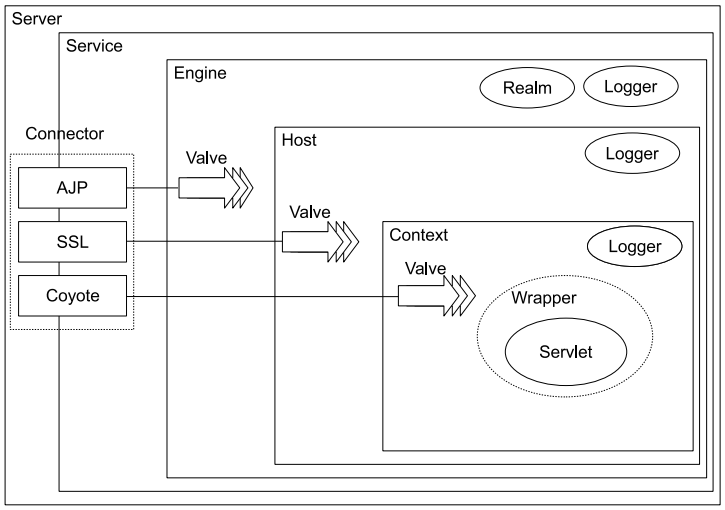

図 5.1: Tomcat のアーキテクチャ

```
<Context path="/App"
         ...
         wrapperClass="MyWrapper">
<Valve className="MyValve">
<Context/>
```

図 5.2: Context.xml

の Valve は，invoke メソッドを呼び出すことでリクエストを次のコンポーネント（Valve）へ渡す．図 5.4 の Valve では自身の invoke メソッドが呼ばれた際，リクエストからセッションを取得し，ThreadLocal 変数にセットする．Tomcat では各々のリクエストを異なるスレッドで処理するためセッションの格納には ThreadLocal 変数を用いなければならない．

getHttpSession メソッドは Valve で取得したセッションをフレームワーク内の他コンポーネントで使用するための getter メソッドである。

Wrapper

個々のサーブレットの定義を表すコンポーネントである．allocate メソッドでサーブレットの初期化処理･ロード処理が定義される．Per-session AOP フレームワークでは，図 5.5 の Wrapper を使用する．3 行目では，Valve からユーザのセッションを取得している．4-6 行目では，セッションが無

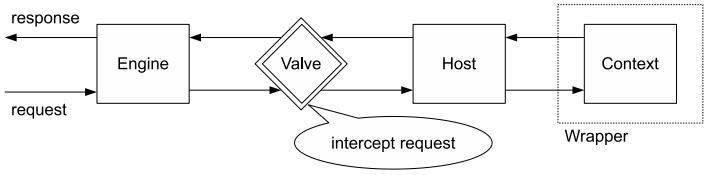

図 5.3: Valve

```
1  public class SessionGetterValve extends ValveBase{
2    private static ThreadLocal<HttpSession> sessionThreadLocal
3                              = new ThreadLocal<HttpSession>();
4
5    public void invoke(Request req, Response res)
6      throws IOException,ServletException {
7      setSession(req, res);
8      getNext().invoke(req, res);
9    }
10
11   private synchronized void setSession(Request req,
12     Response res) throws IOException, ServletException{
13     HttpSession session = req.getSession(false);
14     sessionThreadLocal.set(session);
15   }
16
17   public static HttpSession getHttpSession(){
18     return sessionThreadLocal.get();
19   }
20 }
```

図 5.4: SessionGetterValve クラス

い，又はユーザがアスペクトを選択していないとき，親クラスの allocate メソッドを呼び出している．こうすることで Web アプリケーションは Tomcat 内のデフォルトのクラスローダでロードされる．ユーザがアスペクトを選択している時は，サーバ内のレジストリからアスペクトを取得し，9 行目で GlueWeaveLoader のインスタンスに渡す．GlueWeaveLoader とは引数に渡されたアスペクトを Web アプリケーションに織り込み，ロードするクラスである．11 行目でアスペクトにより拡張された Web アプリケーションのインスタンスを返す．

```
1  public class WebappWrapper extends StandardWrapper{
2    public Servlet allocate() throws ServletException{
3      HttpSession session = SessionGetterValve.getHttpSession();
4      if(/* セッションが無い，又はユーザアスペクトが無い*/) {
5        return super.allocate();
6      }
7                    ...
8      HashSet set = getUserAspects(uid);
9      ClassLoader loader = new GlueWeaveLoader(set);
10     clazz = loader.loadClass(getServletClass());
11     return (Servlet) clazz.newInstance();
12   }
13                   ...
14 }
```

図 5.5: WebappWrapper クラス

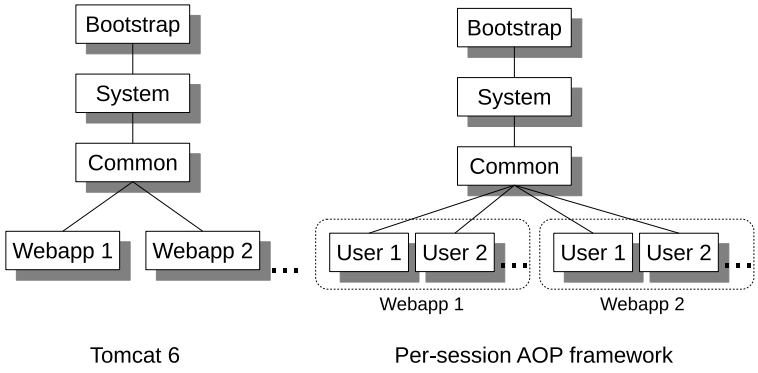

図 5.6: クラスローダの階層

クラスローダ

図 5.6 に Tomcat 6 におけるクラスローダと Per-session AOP フレームワークにおけるクラスローダの親子関係を示す．まず，Tomcat 6 標準のクラスローダについて解説する．

Bootstrap クラスローダは JVM に組み込まれているクラスローダで java 標準のライブラリや JVM の拡張ディレクトリ (jre/lib/ext) に置かれた jar ファイルなどのロードを行う．拡張ディレクトリに置かれたファイルのロードは拡張クラスローダが行う場合もある．これは JVM の実装に依存する．

System クラスローダは CLASSPATH 環境変数を参照してクラスのロードやリソースの検索を行う。Tomcat 6 において System クラスローダは CLASSPATH 以外に$CATALINA_HOME/bin/bootstrap.jar

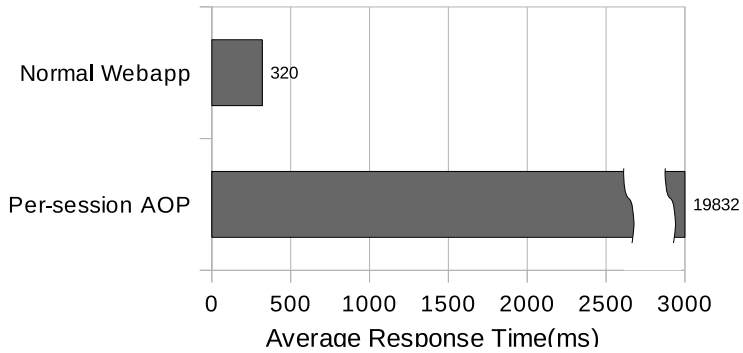

図 5.7: 掲示板アプリへのリクエスト送信実験

や $CATALINA_HOME/bin/tomcat-juli.jar 内のクラスのロードを行う。

Common クラスローダは $CATALINA_HOME/lib 内のあらゆるクラスや jar ファイルのロードを行う．このディレクトリにはサーバ内の全ての Web アプリケーション共通のライブラリ等を配置する．

Webapp X クラスローダは個々の Web アプリケーション（/WEB-INF/classes と /WEB-INF/lib 内のクラス）をロードする．

Per-session AOP フレームワークでは，Webapp X クラスローダの代わりにユーザ専用のクラスローダを作成する．

User X クラスローダは /WEB-INF/classes と /WEB-INF/lib 内のクラスに加えて，そのユーザ固有のアスペクトのロードも行う．また，そのアスペクトの織り込みも担当する．

5.1.2　Per-session AOP フレームワークの問題点

Web アプリケーションは複数のユーザによって共有されるため，あるユーザのアスペクトによる変更が，他のユーザに影響してはならない．そこで，当フレームワークでは，ユーザ毎に異なるクラスローダ（User X クラスローダ）を作成し，それぞれ異なるアスペクトをウィーブした Web アプリケーションをロード，再起動することで，これを回避している．しかし，再起動の度にアプリケーションの全クラスを再ロードするため，オーバーヘッドが非常に大きいという問題があった．

掲示板アプリケーションに Per-session AOP フレームワークを適用した場合とそうでない場合に分けてリクエストを送信する実験を行った．その結果を図 5.7 に示す．グラフを見ると Per-session AOP フレームワークを適用した場合は，そうでない場合に比べて約 60 倍遅いことが分かる．

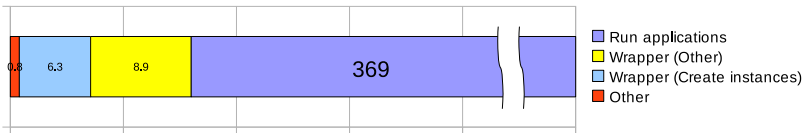

図 5.8: 処理の内訳

図 5.8 は，この実験におけるリクエストの処理の内訳を示したグラフである．このグラフからアプリの実行時間の割合が非常に大きいことが分かる．Java ではクラスのロードをアプリの実行中に動的に行う．Per-session AOP フレームワークではリクエストの度にアプリの全クラスを再ロードするため，その影響でアプリの実行速度が低下している．

この問題は 4 章で提案した手法をフレームワークに適用し，再ロードするクラス数を削減することで改善できる．図 4.1 でアスペクトの変更対象クラスとそれに依存しているクラス群をロードすれば手法 1 を適用出来る．また，クラスの版毎でなく，アスペクト毎にクラスローダを作成すれば，手法 2 を適用できる．次節でその方法を述べる．

5.2　フレームワークの効率化

4 章で提案した手法を Per-session AOP フレームワークに適用する方法について述べる．また，Web アプリケーションでは繰り返し使用されるインスタンスや複数のユーザに共有されるインスタンスもあるためキャッシュの実装も行った．

5.2.1　提案手法の適用

提案手法を実装するためには，再起動時に変更対象のクラスを知っておく必要がある．per-session AOP フレームワークでは，GluonJ[3] を使用している．GluonJ とは我々が開発した Java 言語用の AOP システムである．図 5.9 に GluonJ のアスペクトを示す．このアスペクトは，Web アプリケーション内のクラス Webapp の doGet メソッドを code の内容で書き換える．これは Java の通常のクラスであるため，Javassist を用いてバイトコードを調べることで変更対象クラスを知ることが出来る（図 5.10）．2 行目で CustomizerAspect クラスの内部クラス Customizer を取得している．3 行目でその親クラスを取得し，さらにそのクラス名を取得している．こうすることでユーザからの直接的なヒントを得ること無く，変更対象クラスを知ることが出来る．

```
1 @Glue
2 public class CustomizerAspect {
3   @Refine
4   static class Customizer extends Webapp {
5     public void doGet(HttpServletRequest req,
          HttpServletResponse res) {
6       /* code */
7     }
8   }
9 }
```

図 5.9: GluonJ のアスペクト

```
1 ClassPool pool = ...;
2 CtClass ctClass = pool.get(“CustomizerAspect$Customizer”);
3 String targetName = ctClass.getSuperclass().getName();
```

図 5.10: 変更対象クラスの取得

アプリケーションの各クラスに依存しているクラス群は，サーバの起動時にバイトコードを静的に調べ，クラス内で参照しているクラスを再帰的に調べることで決定できる．しかし，クラス内でリフレクション API を用いてクラスの定義を取得している場合，プログラムが実行されるまでそのクラスが依存しているクラスを決定することができない．そこで，クラス内で java.lang.Class クラスを参照していれば，そのクラスもその他のクラスすべてに依存しているものとした．これにより各クラスローダで再ロードするクラス数が増加してしまうが，アプリケーションの振る舞いに影響は無い．

手法 1 を適用した場合，リクエストの処理の流れは次のようになる．

1. Valve でリクエストを取得する．
2. Valve で取得したリクエストからセッションを取得し，セッションに含まれるユーザ ID からサーバに登録されたユーザ用のアスペクトを取得する．
3. アスペクトを解析し，ウィーブ（変更）対象クラスの名前を取得する．
4. 図 4.1 のクラスローダを作成し,（2）クラス名とそれに依存しているクラスをこのクラスローダがロードするクラス群とする．
5. （3）で作成したクラスローダで Web アプリケーションを再ロードし，再実行する．

```
1  protected Class findClass(String name) {
2    CtClass cc;
3    if (/* name が変更対象クラス*/) {
4      Weaver weaver = /* name のクラスを変更するアスペクト*/
5      cc = weaver.transform(name);
6    }
7    else {
8      cc = pool.get(name);
9    }
10   byte[] b = cc.toByteCode();
11   return defineClass(name, b, 0, b.length);
12 }
```

図 5.11: GlueWeaveLoader.findClass メソッド

なお，アスペクトのウィーブは，クラスローダに対するクラスのロード要求時に，findClass メソッド（図 5.11）内で動的に行われる．Weaver クラスとは GluonJ が提供しているクラスでアスペクトのウィーブを担当するクラスである．その transform メソッドにより引数のクラスがアスペクトにより書き換えられる．toByteCode メソッドは CtClass オブジェクトをバイト配列に変換するメソッドである．

手法 2 を実現するためには，各クラスに依存しているクラス群に加えて，アスペクトの優先度をあらかじめ設定，計算しておく必要がある．優先度は以下のように設定する．

> 優先度 = (アスペクト + アスペクトのウィーブ対象クラス + それに依存しているクラス) のファイルサイズの合計 × アスペクトの利用者数

このように設定することで，ロードするバイトコード量が多いクラスローダほど，また頻繁に利用されるクラスローダほど木の上位に配置されるため，よりロードコストが大きいクラスローダを親ローダとして活用出来る．

リクエストの処理は図 4.4 のアルゴリズムを変更した図 5.12 のアルゴリズムに沿って行われる．Input の集合 S はアスペクトの集合，2 行目の available2(L) は L がその親も含めてウィーブ可能なアスペクトの集合を返す手続き，12 行目の L_C は C のウィーブにより変更されたクラスとそれに依存しているクラス群をロードするクラスローダとなる．

また，図 4.5 のアルゴリズムは図 5.13 のように変更される．S_C はアスペクト C のウィーブにより変更されたクラスとその原版に依存しているクラスの集合を表す．

Input: アスペクトの集合 S とアスペクトそれぞれの優先度の集合 P
Output: 再起動時に使用するクラスローダ $L_{restart}$

1: クラスローダの木 $T = (V, E)$ (初期値 $V = \{L_S\}$, $E = \phi$) について
2: $A \leftarrow \{L \in V \mid \text{available2}(L) \subseteq S, \text{available2}(L) \neq \phi\}$
3: **if** $A = \phi$? **then**
4: 　$L_P \leftarrow L_S$
5: **else**
6: 　$L_P \leftarrow A$ の中で |available2(L)| が最大となるもの
7: **end**
8: $S' \leftarrow S$ - available2(L_P)
9: P を使って S' の要素を優先度が高い順にソート
10: $L_{restart} \leftarrow L_P$
11: **foreach** C S' {
12: 　$L_C \leftarrow C$ のウィーブ対象クラスをロードする新規クラスローダ
13: 　$V \leftarrow V + L_C$
14: 　$E \leftarrow E + (L_{restart}, L_C)$
15: 　$L_{restart} \leftarrow L_C$
16: }

図 5.12: 再起動時に使用するクラスローダの決定アルゴリズム (改訂版)

5.2.2　キャッシュ

実装を行ったキャッシュの内，主だったものについて述べる．なお，キャッシュ（図 5.14[18]）には LRU（*Least Recently Used*）アルゴリズムを用いている．

クラスローダ

Per-session AOP フレームワークではユーザからリクエストの度にクラスローダを作成していたが，ユーザが登録しているアスペクトに変更がなければ，新たにクラスローダを作成する必要はない．

Weaver

Weaver はアスペクト毎に作成されるインスタンスである．Weaver は同じアスペクトを選択している複数ユーザによって共有されるため，これを

```
1: S ← S_C
2: foreach L_P (L_C の全ての親ローダ) {
3:   if L_P が担当するアスペクト D のウィーブにより変更されたクラスが
       S 内のクラスに参照を持つ？
4:   then
5:     S ← S + S_D
6:   end
7: }
```

図 5.13: クラス群の作成アルゴリズム (改定版)

```
1 public final class CacheLRU {
2   public CacheLRU(int capacity);
3   public Object getElement(Object key);
4   public final void addElement(Object key, Object value);
5   public final Iterator keys();
6   public final int size();
7   public final int capacity();
8   public final boolean isFull();
9 }
```

図 5.14: CacheLRU クラス

キャッシュすることでインスタンス作成による時間的コストを削減することが出来る．

ユーザアスペクト

図 5.12 の 9 行目ではユーザの選択したアスペクトをソートしている．ユーザが非常に多くのアスペクトを選択していた場合，このオーバーヘッドは無視できなくなる．そこで，ソート後のアスペクト（名）のリストをキャッシュするようにした．

第6章　実験

各々の実装方法から，リクエストの度に全クラスを再ロードする場合（手法 0 とする），手法 1 を用いる場合，手法 2 を用いる場合の順に再ロードするクラス数が少なくなるのは明らかである．しかし，手法 2 では，一度に多くのクラスを変更する際，最悪そのクラス数分のクラスローダを作成しなければならない．そのため，かえって再起動が遅くなる場合がある．そこで，各手法を用いた場合の再起動時間を測定し，それらの比較・分析を行った．

6.1　実験概要

Per-session AOP フレームワークに 3 つの手法それぞれを実装し，その上で Health Watcher アプリケーションを動作させた．Health Watcher とは，健康管理のための中規模 Web アプリケーションであり，コード行数はおよそ 9 KLOC である． Web サーバは Tomcat 6.0，クライアント側で使用するサーバ負荷テスト用ソフトには Apache JMeter[16] 2.3.2 を用いた．なお，マシン環境は表 6.1 のとおりである．

アプリケーションの各クラス毎に，そのクラスにフィールドを追加するだけのシンプルなアスペクト（図 6.1）を 1 個作成し，あらかじめサーバに登録した．つまり，アプリケーションのクラス数と等しい数のアスペクトがサーバ内にあり，1 個のアスペクトは 1 個のクラスを変更する．それぞれのフレームワークの実装別に分け，さらにユーザのアスペクトの選択パターン別に分けて平均応答時間を測定した．アスペクトの選択パターン (n, m) とは，Servlet クラス用のアスペクトを n 個ランダムに選び，そのコールグラフ上のクラス用のアスペクトを m 個ランダムに選ぶというものである．Servlet クラスとは，Web アプリケーションにおいて，リクエ

表 6.1: 実験環境

	CPU	Memory	OS
Server	Xeon 2.83GHz	4GB	Linux 2.6.26
Client	Core2Duo 3.00GHz	4GB	Windows Server 2003

```
1  @Glue
2  public class AddressAspect {
3    @Refine
4    public static class AddressRefine extends Address {
5      public static Object field = null;
6      public AddressRefine(){
7        super();
8      }
9    }
10 }
```

図 6.1: 実験用アスペクト

スト処理の開始ノードとなるクラスである．例えば，(n, m) = (2, 5) の実験では，全ユーザが Servlet クラス用のアスペクトから 2 個，そのコールグラフ上のクラス用のアスペクトから 5 個選択する．そして，全ユーザ（100 ユーザ）がアプリケーション内の Servlet クラスそれぞれに対しランダムにリクエストを送り，その平均応答時間を測定する．選択パターンをこのように定義した理由は，Web アプリケーションにおいて，メインロジックを担う Servlet クラスが最も変更されやすく，次いで，そのコールグラフ上のクラスも変更されやすいだろうと予想したからである．図 6.2 に HealthWathcer アプリケーション内の Servlet クラスとそのコールグラフ上にあるクラス数を示す．

上記の実験を行った結果，多数のアスペクトが選択され，多くのクラスが変更される場合には手法 1 が，比較的少数のアスペクトが選択された場合には手法 2 を使った方が高速である．但し，多数のアスペクトが選択されても，同じアスペクトが繰り返し選ばれる場合があり，その場合は手法 2 の方が高速であるとわかった．

6.2　結果と考察

6.1 節で説明した実験を Per-session AOP フレームワークの（ユーザ用の）クラスローダのキャッシュエントリ数別に行った．6.2.1 ではフレームワークの全ての実装においてキャッシュを使用せずに実験を行った結果を示し，その分析結果を示す．6.2.2 では，キャッシュを実装して実験を行った結果を示す．

6.2.1　キャッシュ 0

図 6.2 に手法 2 を用いた場合の平均応答時間を示す．このグラフは，アスペクトの選択パターン別の実験結果を平均応答時間の昇順にソートした

表 6.2: コールグラフ上のクラス数

Servlet クラス名	クラス数
ServletInsertAnimalComplaint	47
ServletSearchHealthUnitsBySpecialty	49
ServletInsertSpecialComplaint	48
ServletUpdateEmployeeSearch	47
ServletSearchSpecialtiesByHealthUnit	49
ServletInsertFoodComplaint	47
ServletInsertEmployee	46
ServletGetDataForSearchByHealthUnit	49
ServletWebServer	46
ServletLogin	47
ServletUpdateEmployeeData	46
ServletUpdateHealthUnitData	46
ServletGetDataForSearchByDiseaseType	49
ServletSearchDiseaseData	45
ServletGetDataForSearchBySpeciality	49
ServletUpdateComplaintData	46
ServletUpdateComplaintSearch	52
ServletUpdateHealthUnitSearch	50

 もので，横軸がアスペクトの選択パターン，縦軸が平均応答時間である．以下，図 6.4-6.9 は全て，このグラフの横軸に対応するようにソートされている．

図 6.3 は各アスペクトの選択パターン毎のユーザのアスペクトの使用数とその線形回帰直線である．アスペクトの使用数とは選択パターン (n, m) における n + m のことである．図 6.4 は，手法 2 における再起動時の親ローダの平均活用率とその線形回帰直線である．クラスローダの活用率とは，図 4.4 の $|A|/|S|$ に相当し，再起動時に変更するクラス（再ロードが必要なクラス）の内，どれだけロード済みのクラスを再利用出来ているかを表している．これらのグラフから，再起動の速度はアスペクトの使用数に反比例し，親ローダの活用率に比例すると考えられる．

図 6.5 と図 6.6 はそれぞれ手法 0 を用いた場合と手法 1 を用いた場合の平均応答時間とその線形回帰直線である．ほとんどのアスペクトの選択パターンおいて手法 1 を用いた再起動のほうが高速であった．

図 6.7 は手法 0 と手法 2 の平均応答時間を比較したグラフである．ほとんどのアスペクトの選択パターンおいて手法 2 を用いた再起動のほう

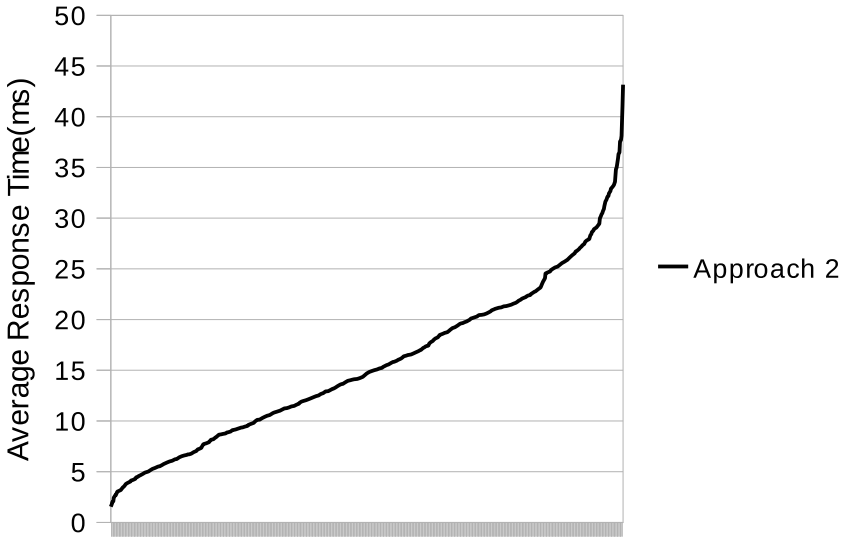

図 6.2: 平均応答時間（手法 2）

が高速であった．

図 6.8 は手法 1，2 の平均応答時間とアスペクトの使用数との関係，図 6.9 は親ローダの活用率との関係を示したグラフである．図 6.8 を見ると，アスペクトの使用数が 56 個（Health Watcher 内の全クラスの 76 %）以上の場合は手法 1 を用いた再起動の方が高速である．このグラフから，非常に多くのクラスを変更するときは手法 1 を使ったほうがよいとわかる．また，そのときの親ローダの活用率は 55 %であった．図 6.3 の左上に着目すると，アスペクトの使用数が多い，つまり多数のクラスを変更しているにも関わらず平均応答時間が短いケースが見られる．これらのほとんどはアスペクトの選択パターン (n, m) の n に対して m が非常に大きい場合であった．各ユーザが選択した Servlet クラスに対するアスペクトが異なったとしても，そのコールグラフには多くの共通部分があるため，必然的に各ユーザが選択するコールグラフ上のクラスに対するアスペクトも類似する．このため，あるユーザが再起動の際使用したクラスローダ群を他のユーザが親ローダとして活用しやすくなったからである．各ユーザが使用するアスペクト数が 56 個以上で，手法 1 よりも手法 2 の方が早くなるアスペクトの選択パターン (n, m) の m/n の平均値は 5.3 で，親ローダの活用率の平均値は 69 %であった．

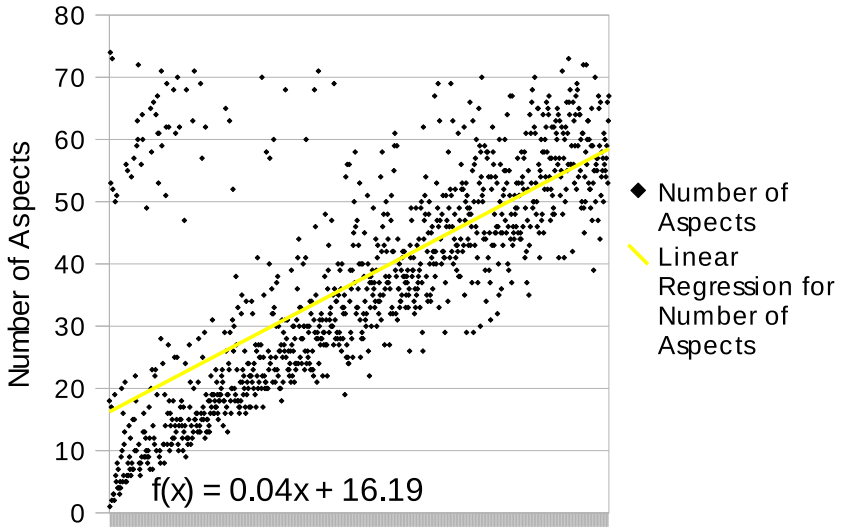

図 6.3: アスペクトの使用数

6.2.2　キャッシュ 25-75

キャッシュエントリ数 25-75 の場合についても同様に実験を行った．それぞれの実験結果のうち，アスペクト使用数と平均応答時間の関係，クラスローダ活用率と平均応答時間の関係のグラフのみを示す．

キャッシュ 25

図 6.10 を見ると，アスペクトの使用数が 53 個（Health Watcher 内の全クラスの 72 %）以上の場合は手法 1 を用いた再起動の方が高速である．また，図 6.11 からそのときの親ローダの活用率は 62 %であった．

キャッシュ 50

図 6.12 を見ると，アスペクトの使用数が 57 個（Health Watcher 内の全クラスの 77 %）以上の場合は手法 1 を用いた再起動の方が高速である．また，図 6.13 からそのときの親ローダの活用率は 81 %であった．

キャッシュ 75

図 6.14 を見ると，アスペクトの使用数が 56 個（Health Watcher 内の全クラスの 76 %）以上の場合は手法 1 を用いた再起動の方が高速である．

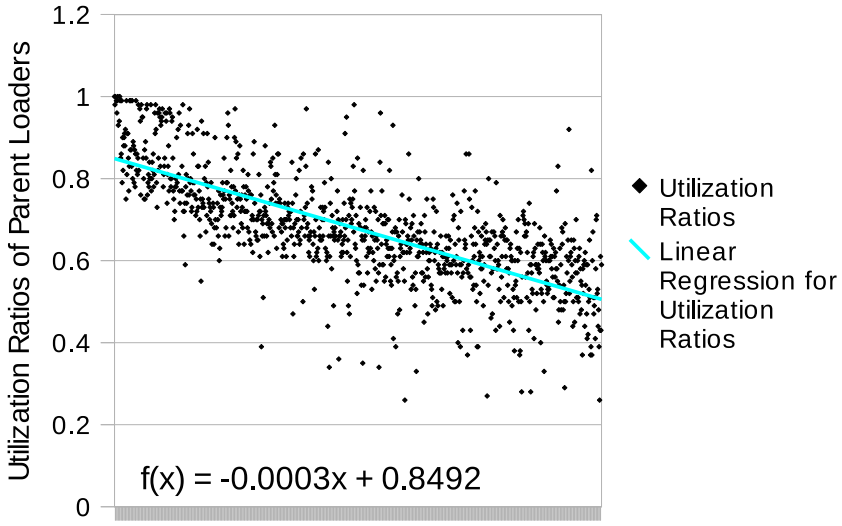

図 6.4: クラスローダの平均活用率

表 6.3: m/n とクラスローダ活用率

キャッシュ	m/n	ローダ活用率
25	6.1	75
50	5.0	84
75	5.3	92

また，図 6.15 からそのときの親ローダの活用率は 87 %であった．

考察（キャッシュ 25-75）

キャッシュ 25-75 の手法 2 実験においても，キャッシュ 0 の実験と同様に多数のクラスを変更しているにも関わらず平均応答時間が短いケースが見られた．また，これらのケースもアスペクトの選択パターン (n, m) の n に対して m が非常に大きい場合であった．表 6.3 にそれらのケースの m/n の平均値と親ローダ活用率の平均値を示す．

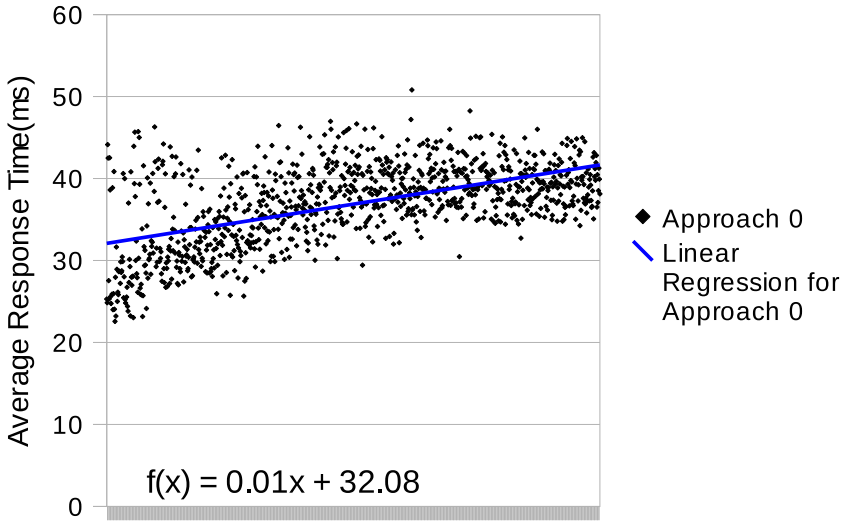

図 6.5: 平均応答時間（手法 0）

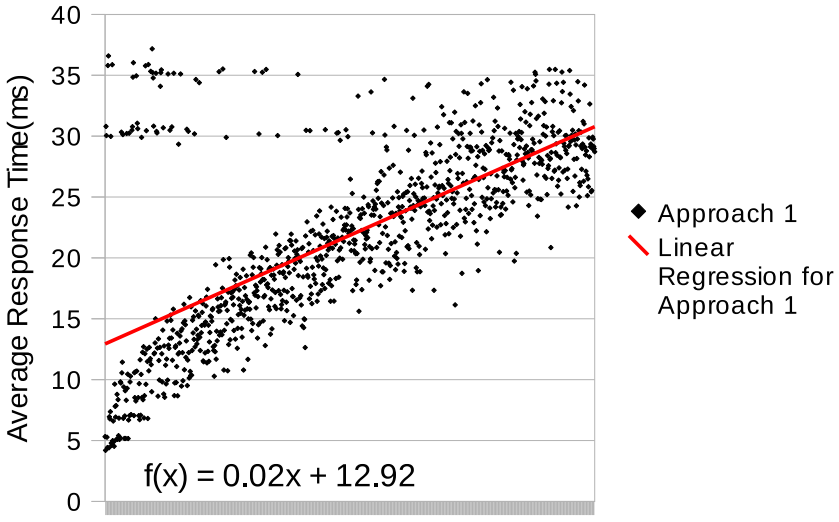

図 6.6: 平均応答時間（手法 1）

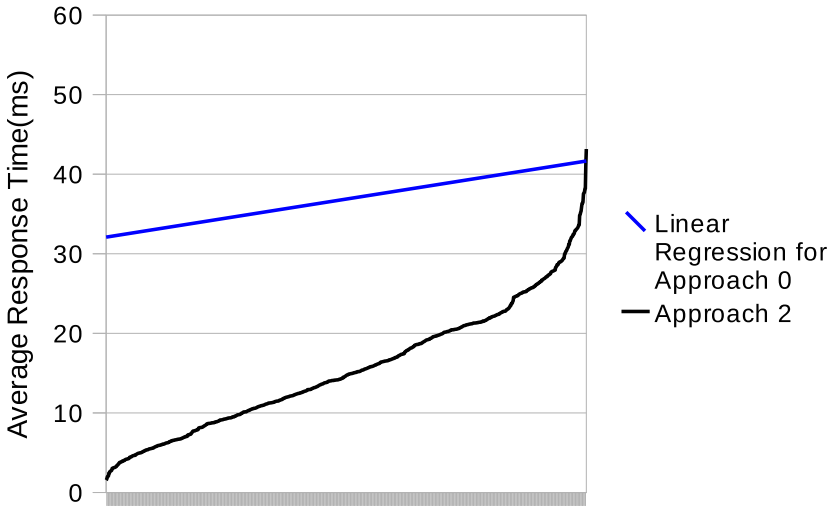

図 6.7: 平均応答時間の比較（手法 0，手法 2）

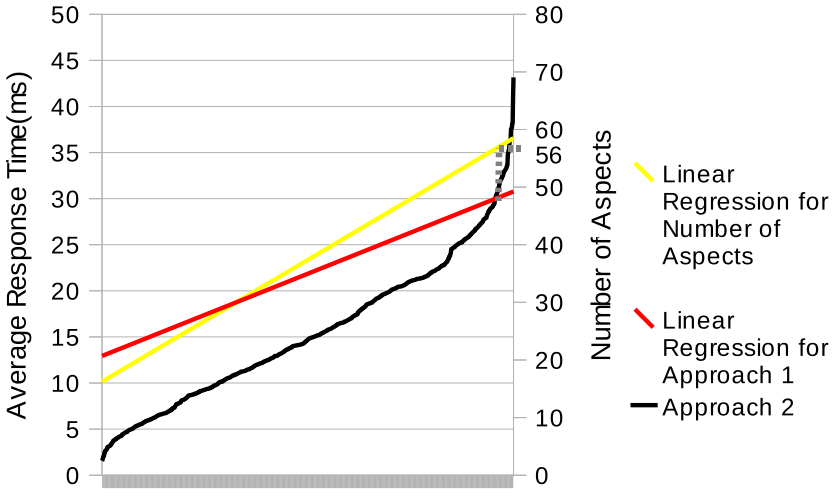

図 6.8: アスペクト使用数と平均応答時間の関係

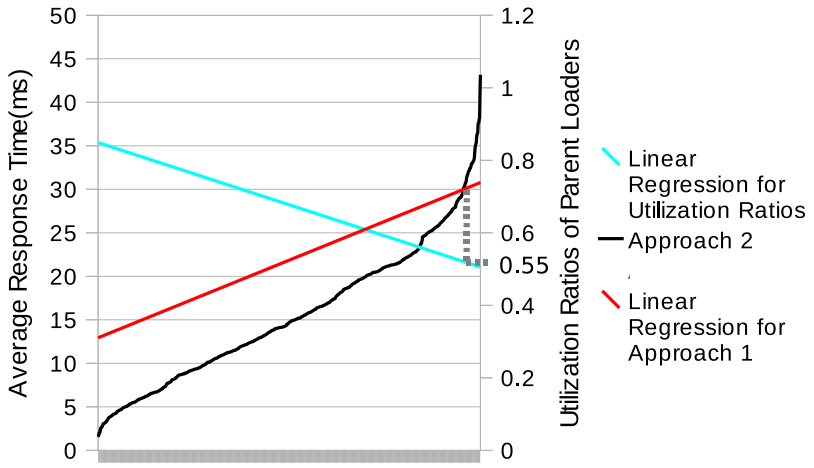

図 6.9: クラスローダ活用率と平均応答時間の関係

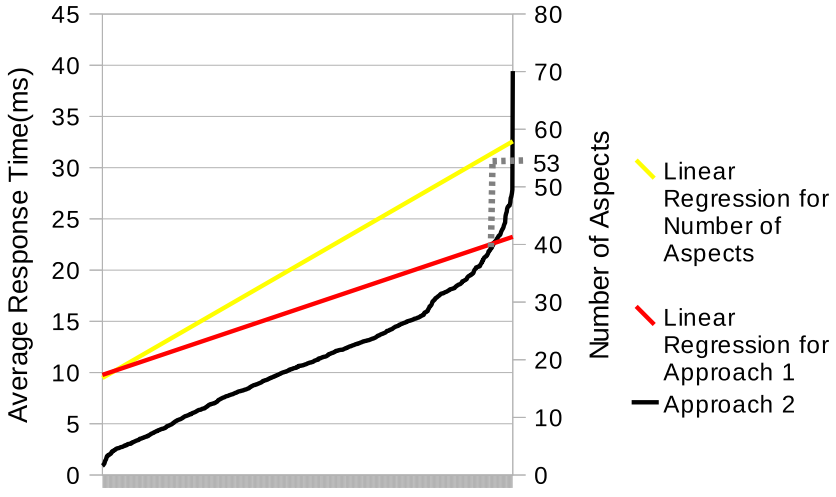

図 6.10: アスペクト使用数と平均応答時間の関係（キャッシュ 25）

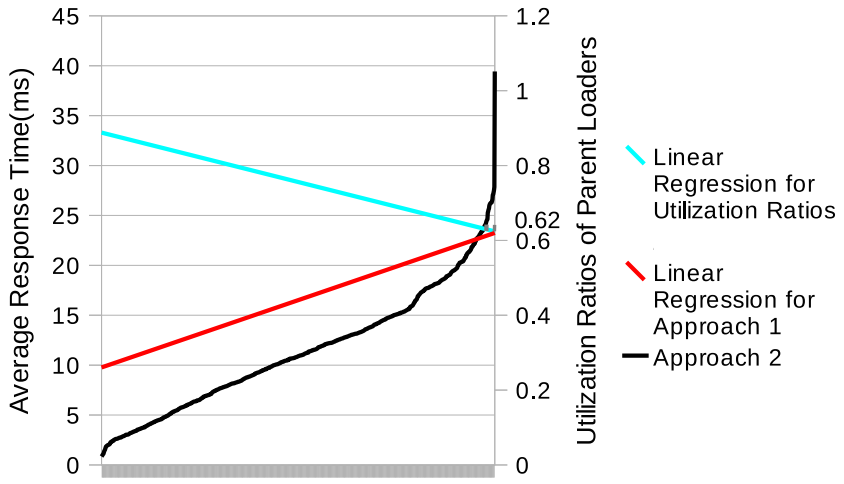

図 6.11: クラスローダ活用率と平均応答時間の関係（キャッシュ 25）

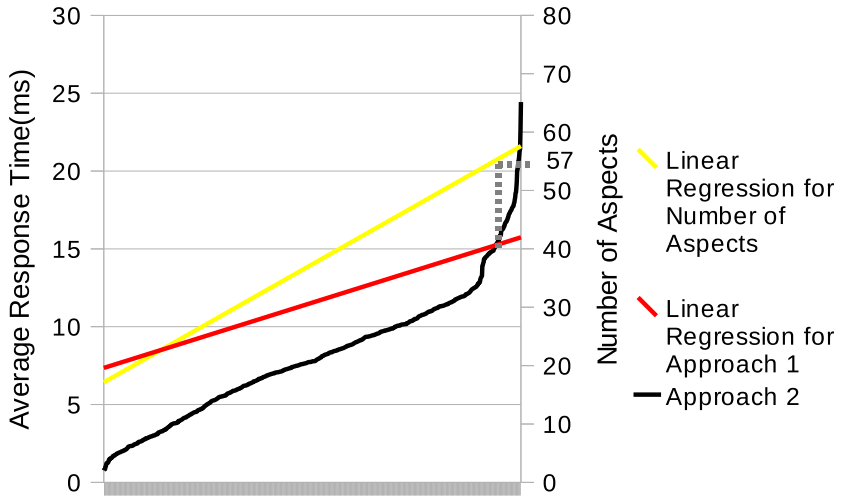

図 6.12: アスペクト使用数と平均応答時間の関係（キャッシュ 50）

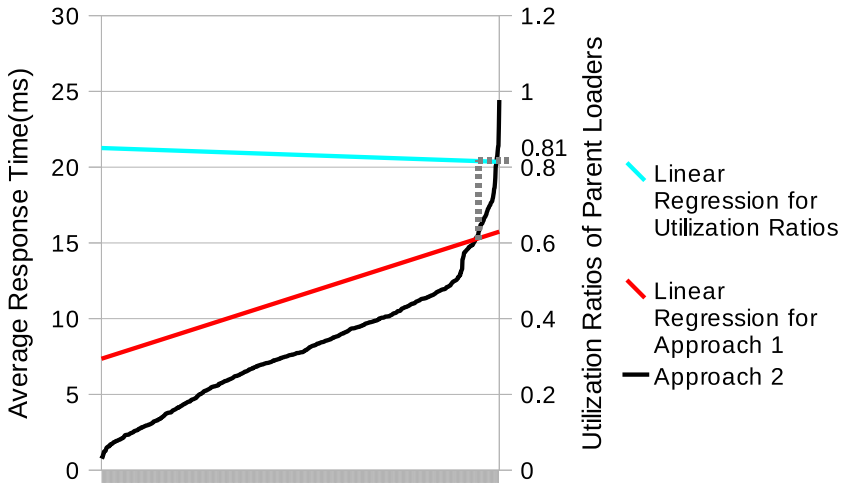

図 6.13: クラスローダ活用率と平均応答時間の関係（キャッシュ 50）

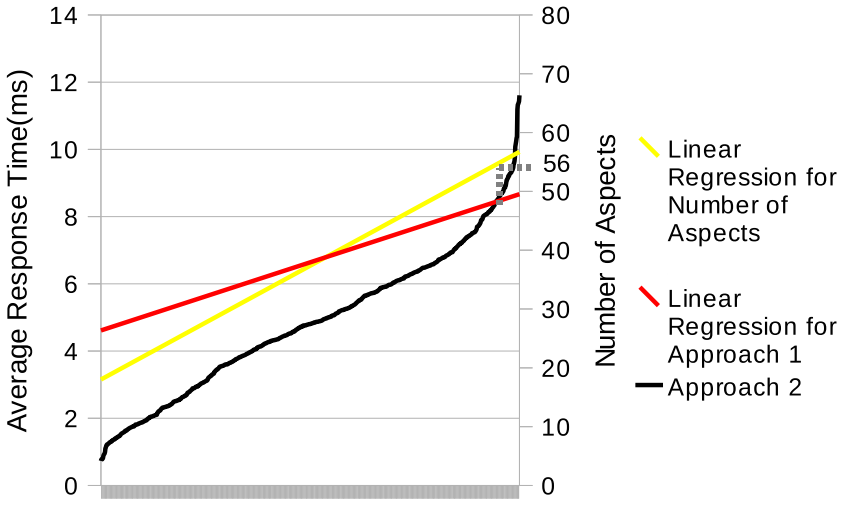

図 6.14: アスペクト使用数と平均応答時間の関係（キャッシュ 75）

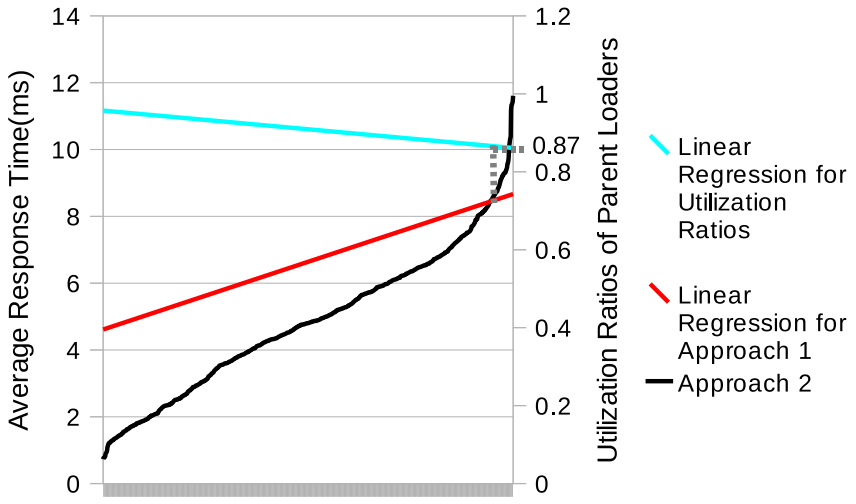

図 6.15: クラスローダ活用率と平均応答時間の関係（キャッシュ 75）

第7章　まとめと今後の課題

7.1　まとめ

部分的再ロードによる Java プログラムの再起動の高速化手法について提案した．リクエストの度にアプリケーションの全クラスの再ロードし直すという再起動は，対話的な開発が可能であるという点で有用であるが，その時間的コストによりサーバのパフォーマンスを低下させるという問題があった．再ロードが必要なのは変更されたクラスのみであり，全クラスを再ロードする必要はない．そこで本研究では，

1. 変更対象のクラスとそれに依存しているクラスを子ローダ再ロードし，その他のクラスは親ローダでロード済みのものを再利用する手法

2. アプリケーションの各クラスの各版毎にクラスローダを作成し，それを親ローダとして活用することで，旧版のアプリケーションを部分的に再利用する手法

の 2 つを提案した．これらの手法を用いることで再ロードするクラス数を削減出来ることは明らかであるが，手法 1 にはクラスローダを 1 個しか作らなくて良いという利点，手法 2 には手法 1 よりも再ロードするクラス数を削減できるという利点があったため実験により比較する必要があった．そこで 2 つの手法を per-session AOP フレームワークに実装し，その上で動作する Health Wather アプリケーションにリクエストを送信する実験を行った．その結果から，多数のアスペクトが選択され，多くのクラスが変更される場合には手法　1 がが高速である．但し，多数のアスペクトが選択されても，同じアスペクトが繰り返し選ばれる場合があり，その場合は手法　2 の方が高速であるとわかった．

7.2　今後の課題

ケーススタディ

手法 2 はアプリケーションの各クラスに複数の版があり，それらを組み合わせて何度も再起動をするという特定条件下で再ロードするクラス数

を削減する手法である．本研究では Per-session AOP フレームワークという 1 つの応用例しか示していないため，この状況に合致するシステムをさらに調査する必要がある．

さらなる実験

本研究ではフィールドを追加するだけのアスペクトが各クラス用に一つずつサーバに登録されているという条件下で実験を行っている．アプリケーションに複雑な参照を追加するアスペクトを，各クラス用に複数登録した状況下でも実験を行い，パフォーマンスにどう影響が出るかを検証する必要がある．また，Health Watcher アプリケーションのみならず，小規模・大規模問わず様々なアプリケーションで実験を行い，手法 1・2 のどちらを使うべきかの指針をさらに明確にする必要がある．

参考文献

[1] Bakore, A., Bhattacharjee, D., Bhattacharjee, S., Chopra, V., Fowler, C., Galbraith, B., Irani, R., Li, S. and Wiggers, C.: *Professional Apache Tomcat*, Wiley Publishing (2003).

[2] Chiba, S.: Load-time Structural Reflection in Java, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2001)*, Springer-Verlag, pp. 313–336 (2000).

[3] Chiba, S., Igarashi, A. and Zakirov, S.: Mostly modular compilation of crosscutting concerns by contextual predicate dispatch, *The ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2010)*, ACM Press, pp. 539–554 (2010).

[4] Daniel, L., Navarro, B., Sdholt, M., Vanderperren, W., Fraine, B. D. and Suve, D.: Explicitly distributed AOP using AWED, *Proceedings of the 5st international conference on Aspect-oriented software development (AOSD 2006)*, ACM Press, pp. 51–62 (2006).

[5] Fleury, M. and Reverbel, F.: The JBoss Extensible Server, *In Proceedings of the ACM/IFIP/USENIX International Middleware Conference (Middleware 2003)*, Springer-Verlag, pp. 344–373 (2003).

[6] Fowler, M.: Inversion of control containers and the dependency injection pattern, `http://www.martinfowler.com/articles/injection.html`.

[7] Greenwood, P., Bartolomei, T. T., Figueiredo, E., Dósea, M., Garcia, A. F., Cacho, N., Sant'Anna, C., Soares, S., Borba, P., Kulesza, U. and Rashid, A.: On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2007)*, Springer-Verlag, pp. 176–200 (2007).

[8] Horie, M., Morita, S. and Chiba, S.: Distributed Dynamic Weaving is a Crosscutting Concern, *Proceedings of the 26th Annual ACM Symposium on Applied Computing (SAC 2011)* (2011).

[9] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. G.: An Overview of AspectJ, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2001)*, Springer-Verlag, pp. 327–353 (2001).

[10] Liang, S. and Bracha, G.: Dynamic Class Loading in the Java Virtual Machine, *In Proc. 13th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98), volume 33, number 10 of ACM SIGPLAN Notices*, ACM Press, pp. 36–44 (1998).

[11] Nishizawa, M., Chiba, S. and Tatsubori, M.: Remote pointcut: a language construct for distributed aop, *Proceedings of the 3rd international conference on Aspect-oriented software development (AOSD 2004)*, ACM Press, pp. 7–15 (2004).

[12] Popovici, A., Gross, T. and Alonso, G.: Dynamic weaving for aspect-oriented programming, *Proceedings of the 1st international conference on Aspect-oriented software development (AOSD 2002)*, ACM Press, pp. 141–147 (2002).

[13] Sato, Y. and Chiba, S.: Loosely-separated "Sister" Namespaces in Java, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2005)*, Springer-Verlag, pp. 49–70 (2005).

[14] Sato, Y., Chiba, S. and Tatsubori, M.: A Selective, Just-in-Time Aspect Weaver, *Proceedings of the 2nd international conference on Generative programming and component engineering (GPCE 2003)*, Springer-Verlag, pp. 189–208 (2003).

[15] Suvée, D., Vanderperren, W. and Jonckers, V.: JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development, *Proceedings of the 2nd international conference on Aspect-oriented software development (AOSD 2003)*, ACM Press, pp. 21–29 (2003).

[16] The Apache Software Foundation: Apache JMeter, `http://jakarta.apache.org/jmeter/`.

[17] The Apache Software Foundation: Apache Tomcat, http://tomcat.apache.org.

[18] The Apache Software Foundation: Jakarta ORO, http://jakarta.apache.org/oro/.

[19] The Seasar Project: Seasar2, http://s2container.seasar.org/2.4/ja/.

[20] Vanderperren, W. and Suvee, D.: Optimizing JAsCo dynamic AOP through HotSwap and Jutta, *Dynamic Aspects Workshop*, pp. 120–134 (2004).

[21] 戸部敦, 千葉滋: Web アプリをユーザ毎にカスタマイズ可能にする AOP フレームワーク, 楽天研究開発シンポジウム (2008).