平成20年度 学士論文

ユーザ毎にカスタマイズ可能な Web アプリケーションの 効率の良い実装方法

東京工業大学 理学部 情報科学科
学籍番号 05-2162-1
別役 浩平

指導教員
千葉 滋 教授

平成21年2月13日

概要

今日、ユーザ毎にカスタマイズ可能な Web アプリケーションが多く普及している。例えば、iGoogle ではユーザ毎に Web ページの背景の変更、コンテンツの追加、タブの追加などが可能である。FC2 ブログでは、管理者ページから表示するプラグインを追加することができる。HTML を直接編集したり、ブログパーツや JavaScript を設置する事ができるプラグインも存在する。これらのカスタマイズは、ユーザの独自性を尊重するためである。Web アプリケーションがカスタマイズ可能であることは、ユーザの創造力をくすぐり、他のユーザと差をつけるべく競争心をあおる。Web アプリケーションをカスタマイズ可能にすることは、ユーザを引き付ける上で重要であり、Web サイトの集客を左右する要因になるといえる。

このような Web アプリケーションを開発するには、ユーザ毎に振る舞いを変更するための特殊な機構を設計段階で組み込む必要がある。しかし、ユーザにとってどのようなカスタマイズが必要であるかを事前に想像し、設計段階でこのような機構を組み込むことは容易ではない。それ故、Web アプリケーションの開発後にカスタマイズ機構の追加、削除が必要になる場合があるが、そのためには実装の大幅な変更が必要となり、コストが大きい。

そこで本研究では、このような Web アプリケーションの開発を支援するために、per-session AOP を用いたフレームワークを提案する。Per-session AOP とは、セッションから取得したユーザ ID からそのユーザに適用すべきアスペクトを特定し、そのアスペクトを Web アプリケーションに weave する技術である。この技術を用いるとユーザはサーバーにアスペクトを登録するだけで Web アプリケーションの任意の場所を自分専用にカスタマイズすることができる。

さらに本研究では、このようなフレームワークの実用性を高めるための per-session AOP の効率のよい実装方法を提案する。Per-session AOP ではユーザ毎に異なるクラスローダを作成する。異なるローダは異なる名前空間をもつのでユーザ毎に異なるアスペクトを weave した Web アプリケーションをロードすることができる。しかし、Web アプリケーション内で共通に使われているライブラリもこのローダでロードさせてしまうと、

ロードによるオーバーヘッドが大きくなってしまう。そこで、Web アプリケーション内で共通に使われているライブラリは、ユーザ共通のローダにロードさせるようにした。こうすることで、一度親ローダによってロードされたライブラリについては、Web アプリケーション実行中にロード要求があった場合、ロード済みのクラスを返すだけでよく、ロードによるオーバーヘッドを削減することができる。さらに、アスペクトの変更がないユーザのローダをキャッシュし、そのユーザから再度アクセスがあったとき、同一アスペクトを weave した Web アプリケーションを再ロードしないようにした。これらの工夫により、ClassLoader オブジェクトや Class オブジェクトの生成を抑えることができ、メモリ使用量を削減することができる。上で述べた実装に対して Web アプリケーションにリクエストを送信する実験を行い、フレームワークの使用・非使用とその効率化の有無による性能変化を測定した。その結果、フレームワークの実装の効率化によってロードによるオーバーヘッドとメモリ使用量を大幅に削減できることを確認した。

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。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、論文の内容について様々な助言をしてくださった堀江倫大氏に感謝致します。また、戸部敦氏には、論文の書き方、実装方法、実験方法など親身になって指導していただきました。心より感謝致します。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

図 目 次

表 目 次

第1章　はじめに

今日、ユーザ毎にカスタマイズ可能な Web アプリケーションが多く普及している。例えば、図 1.1 の iGoogle ではユーザ毎に Web ページの背景の変更、コンテンツの追加、タブの追加などが可能である。

図 1.1: iGoogle

その他にも My Yahoo! や my Rakuten、FC2 ブログなどがユーザ毎にカスタマイズ可能な Web アプリケーションの例としてあげられる。Web アプリケーションがユーザ毎にカスタマイズ可能であることは、ユーザの興味を引く要因となり、Web サイトの集客にも影響する。

このような Web アプリケーションを開発するには、ユーザ毎に振る舞いを変更するための特殊な機構を設計段階で組み込む必要がある。例えば、Java で実装された Web アプリケーションではプラグイン機構と呼ばれるユーザ毎に適用すべきクラスを切り替える機構を用いて実装されていることが多い。しかし、ユーザにとってどのようなカスタマイズが必要であるかを事前に想像し、設計段階でこのような機構を組み込むことは容易ではない。それ故、Web アプリケーションの開発後にカスタマイズ機構の追加、削除が必要になる場合があるが、そのためには実装の大幅な変

更が必要となり、コストが大きい。

そこで本研究では、このような Web アプリケーションのするために、per-session AOP を用いたフレームワークを提案する。per-session AOP とは、セッションから取得したユーザ ID からそのユーザに適用すべきアスペクトを特定し、そのアスペクトを Web アプリケーションに weave する技術である。AOP を用いることで、Web アプリケーションのソースコードを編集することなく、その挙動だけを変更することができるので、Web アプリケーションに特殊な機構を組み込むことなく、ユーザ毎に挙動の異なる Web アプリケーションを実現することができる。

per-session AOP ではユーザ毎に異なるクラスローダを用いることでユーザ毎に異なるアスペクトを weave した Web アプリケーションをロードすることを可能にしている。しかし、異なるクラスローダは異なる名前空間をもつため、新たにクラスローダを作成し、ロードする度に再ロードの必要のないクラスまでロードされてしまう。これによるオーバーヘッドを削減するために、本研究ではフレームワークのパフォーマンスの向上のための実装についても行った。

Web アプリケーション内で使われているライブラリはアスペクトによりカスタマイズされることはないので、ユーザ共通のローダにロードさせるようにした。こうすることで、一度親ローダによってロードされたライブラリは Web アプリケーション実行中にロード要求があった場合、ロード済みのクラスを返すだけでよく、ロードによるオーバーヘッドを削減することができる。さらに、クラスローダのキャッシュを用いる事で同一アスペクトを weave した Web アプリケーションを再ロードすることによるオーバーヘッドを削減できる。これらの工夫により、ClassLoader オブジェクトや Class オブジェクトの生成を抑えることができ、メモリ使用量を削減することができる。

以下、第 2 章では、既存の Web アプリケーションの実装と問題点、第 3 章では、提案するフレームワークについて、第 4 章では、フレームワークの実装、第 5 章では、フレームワークの性能測定のための実験について述べる。そして、第 6 章では、まとめと今後の課題について述べる。

第2章　既存の Web アプリケーションの実装とその問題点

今日、ユーザ毎にカスタマイズ可能な Web アプリケーションは多数存在し、Web アプリケーションがカスタマイズ可能であることはユーザを引き付ける上でも重要である。

しかし、このような Web アプリケーションには様々な問題点がある。本章では、既存 Web アプリケーションの内蔵と、それら問題点について述べ、それらの解決方法について議論する。

2.1　カスタマイズ可能な Web アプリケーション

今日のカスタマイズ可能な Web アプリケーションでは、ユーザ毎に Web ページの背景の変更、コンテンツの追加、削除、HTML の編集などが可能なものが多く存在する。このような機能はユーザを引き付ける大きな要因となり、Web サイトの集客にも影響する。しかし、このような Web アプリケーションには次のような問題点が存在する。

まず第一に通常の Web アプリケーション開発よりも多くの知識が必要であるという点である。このような Web アプリケーション開発には通常の Web アプリケーションの実装方法に関する知識とは別に、特定のポイントをカスタマイズ可能にするための機構に関する知識と実装方法について知る必要がある。

第二に、開発時に決めた拡張ポイント以外はカスタマイズ不可能であるという点である。拡張ポイントとは、カスタマイズ可能にするための機構を組み込んでおり、ユーザが拡張可能な Web アプリケーションの部分のことである。それゆえ、開発者は入念な Web アプリケーションの設計が要求される上、開発後に拡張ポイントを追加、削除することになった場合、通常の Web アプリケーションの再開発よりもコストが大きくなってしまう。

最後に、カスタマイズ可能な Web アプリケーションはソースコードが見づらくなってしまうという点である。通常の Web アプリケーションと同様のソースコードに加えて、ユーザ毎の設定を読み込むコード、それを処理するコード、エラーを処理するコードが必要になってくるからである。

以下、既存のカスタマイズ可能な Web アプリケーションに組み込まれている機構について紹介し、その問題点について述べ、解決方法について議論する。

2.1.1　プラグイン機構

ユーザ毎にカスタマイズ可能な Web アプリケーションを実現するには、カスタマイズ可能にする機構を設計段階より組み込まなければならない。カスタマイズを可能にする代表的な機構として、プラグイン機構が存在する。プラグイン機構を実装するには、まず Web アプリケーションにおいてカスタマイズ可能にする部分をインタフェースとして定義する。Web ページの背景色をユーザ毎にカスタマイズ可能にするには、図 2.1 のような Color インタフェースを実装する。

```
1  Interface Color{
2    String getName();
3  }
```

図 2.1: Color インタフェース

ここで、Color インタフェースに定義されている getName メソッドは white や#ffffff などのような HTML のソースコード内で使用できる色を表す文字列を返すメソッドである。

図 2.2 に色をカスタマイズ可能にした Web アプリケーションのプログラム例と Web アプリケーション実行時の流れを示す。

まず、Web アプリケーションでは変数 user に対応するユーザ ID をセッションやリクエストパラメータから取得する。次に、getColor メソッドによりデータベースなどに格納されているユーザの設定ファイルから Color インタフェースの実装クラスについて指定している部分を読み取り、そのクラスをロードし、インスタンスを作成する。最後に実装クラスの getName メソッドにより色を取得し、setBodyColor メソッドにより、Web ページの背景色を設定する。

ここで、getColor メソッドは設定ファイルの読み込みクラスのロード、インスタンスの作成を担うメソッドで、setBodyColor メソッドは引数に渡された文字列を HTML の色として設定し、HTML のソースをクライアントに送信するメソッドである。これらのメソッドは Web アプリケーション開発者が定義するであろうメソッドの一例である。

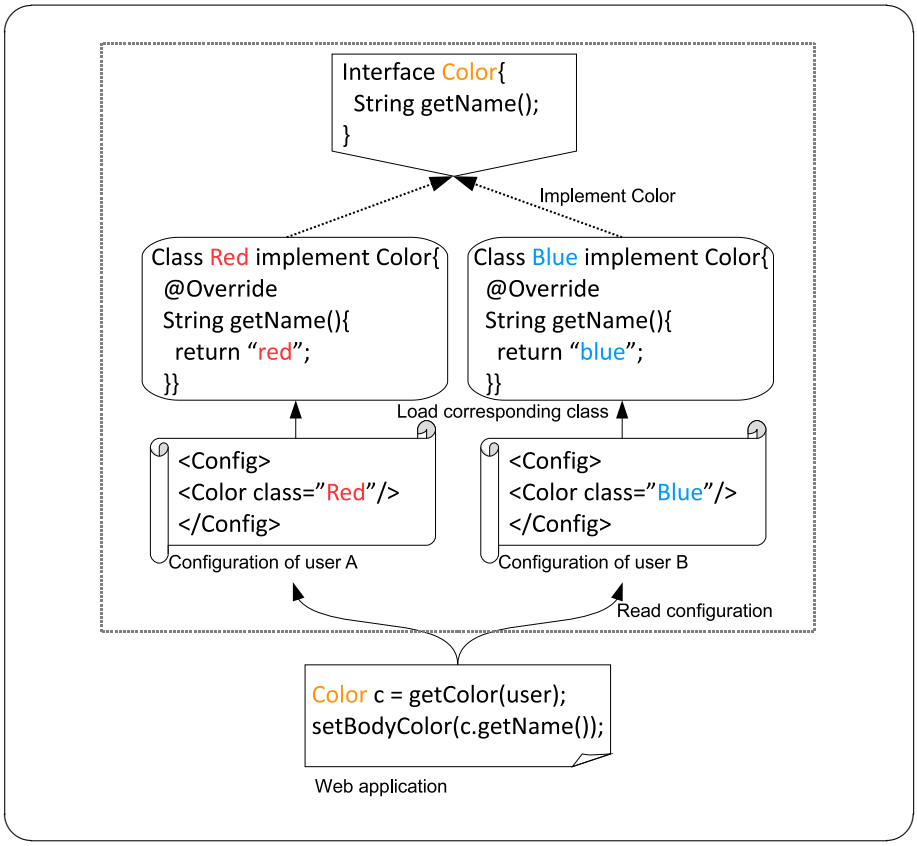

図 2.2: プラグイン機構の例

しかし、実際にはデータベースアクセスやクラスロード部分のコードは複雑であり、エラー処理も必要である。さらに、もっと高度なカスタマイズを可能にするためには、もっと複雑なインタフェースが必要になるだろうし、Web アプリケーション上での処理も複雑になる。

2.1.2　プラグイン機構の問題点

まず、前節を見てわかるように通常の Web アプリケーションの実装に比べて高度なプログラミングの知識が必要になるという点である。図 2.2 の getColor メソッドの内部では、次のような処理が必要になるだろう。

1. データベースにアクセスし、設定ファイルを取得
2. 設定ファイルを解析し、ユーザ情報を取得
3. 取得したユーザ情報をもとにクラスをロード

4. インスタンスの作成

getColor メソッドでは、これらの処理のほかにエラー処理なども必要になるだろう。

さらに、図 2.2 を見てもわかるように、プラグイン機構を実装するには、設定ファイルの管理、ユーザ毎に適用するクラスの管理、インタフェースの実装などが必要となり、通常の Web アプリケーションを実装する場合に比べてコストが大きいことが分かる。

次に、プラグイン機構の実装から明らかなように、機構を組み込んだ部分以外は Web アプリケーションのカスタマイズが不可能であるという点である。このため、Web アプリケーションの開発者は設計段階でどこをどのようにカスタマイズ可能にするかを考え、実装しなければならない。しかし、どの部分をカスタマイズ可能にするかという予測は非常に困難である。その上、多くの Web アプリケーションは小さな部分のみ実装されて作動し、その後徐々に機能を付加していくため、設計段階からプラグイン機構のような大きな機構を組み込むことは困難である。

最後に、プラグイン機構を組み込むことでソースコードが見づらくなってしまうという点である。例えば、図 2.2 のようなプラグイン機構を組み込まずに固定の色を使用する場合、色を設定する Web アプリケーション上のコードは図 2.3 のようになるだろう。

```
1  out.println(“<body bgcolor=#ffffff>”);
```

図 2.3: 背景色の設定

ここで、変数 out はクライアントに文字テキストを送信するための出力ストリームで、println メソッドは文字テキストを送信するメソッドである。

このように、プラグイン機構を組み込まない場合は単純に背景色を設定する HTML のコードをクライアントに送信するコードだけで済み、非常に単純である。しかし、プラグイン機構の場合には、図 2.2 のようなコードが必要になり、コードが煩雑になってしまう。

2.2　AOP(Aspect-Oriented Programming)

前節で述べたような Web アプリケーションの問題点を解決するには、AOP (Aspect-Oriented Programming) を用いた手法が有効である。本節では、AOP についての説明し、次節でその手法について議論する。

2.2.1　アスペクト指向とは

AOP とは OOP (Object-Oriented Programming) ではモジュール化できないソフトウェアの横断的関心事 (crosscutting concerns) をアスペクトとして分離し、モジュール化するためのプログラミング技法である。

まず最初に、AOP の概念について説明する。[9]

- アスペクト
  横断的関心事をまとめ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で記述される。
- 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実行中に存在するジョインポイントを限定し、集合を作る。
- ジョインポイント
  プログラム実行中のある「時」を表す。アドバイスを weave 可能なポイント。例として、メソッドの呼び出し、メソッドの実行、コンストラクタの実行、フィールドの参照、代入、オブジェクトの初期化などがある。ポイントカットを構成する要素。
- アドバイス
  ポイントカットを構成するジョインポイントで実行する処理を記述したもの。
- 織り込み (weave)
  ポイントカットを構成するジョインポイントでアドバイスで記述した処理が実行されるように、ポイントカットとアドバイスを結びつけること。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を埋め込むなどの方法がある。

上のように、AOP とはポイントカットとアドバイスの組で横断的関心事を記述し、それをアスペクトというモジュールにまとめる。そして、アスペクトの記述をもとにソフトウェアの振る舞いを変更する技術である。

2.2.2　AspectJ

AspectJ はアスペクト指向言語として最も有名であり、広くつかわれている言語である。AspectJ は Java 言語を拡張して、アスペクトの機能を使えるようにした言語である。[1]

以下に AspectJ の代表的なポイントカットを示す。

- call ポイントカット
  メソッド、コンストラクタを呼び出す時。
- execution ポイントカット
  メソッド、コンストラクタを実行する時。
- set ポイントカット
  フィールドに値を代入するとき。
- get ポイントカット
  フィールドの値を参照するとき。

また、これらのポイントカットを組み合わせることにより、新たなポイントカットを作成することができる。(図 2.4)

```
1 execution(void Test.foo()) || call(void Sample.bar())
```

図 2.4: ポイントカット

このポイントカットは Test クラスの foo メソッドが実行される時を表すポイントカットと Sample クラスの bar メソッドが実行されるときを表すポイントカットを論理演算子 || で結合したポイントカットである。図 2.4 のポイントカットは論理演算子の左辺のポイントカットと右辺のポイントカットの和集合に属するジョインポイントの集合である。

次に AspectJ の代表的なアドバイスを示す。

- before アドバイス
  ジョインポイントが実行される直前に実行されるアドバイス
- after アドバイス
  ジョインポイントが実行された直後に実行されるアドバイス
- around アドバイス
  ジョインポイントの期間全体の処理の代わりに実行されるアドバイス

図 2.5 のプログラムが実行されると、

```
foo
bar
```

```
1  public class Sample{
2    public void foo(){
3      System.out.println(“foo”);
4      bar();
5    }
6    public void bar(){
7      System.out.println(“bar”);
8    }
9    public static void main(String[] args){
10     new Sample().foo();
11   }
12 }
```

図 2.5: Sample クラス

```
1  public aspect HelloAspect{
2    pointcut printHello():call(void Sample.bar());
3    before():printHello(){
4      System.out.println(“hello”);
5    }
6  }
```

図 2.6: AspectJ のアスペクトの記述例

と出力される。

このプログラムの振る舞いを変更するために 図 2.6 のようなアスペクトを記述する。

図 2.6 のアスペクトの printHello ポイントカットは Sample クラスの bar メソッドが実行されるときを選択する。3 行目の before アドバイスは printHello ポイントカットが選択した時の直前に hello を出力するアドバイスである。

このアスペクトを適用して 図 2.5 のプログラムを実行すると、

```
foo
hello
bar
```

と出力される。図 2.5 10 行目で foo メソッドが実行されるとまず foo が出力される。そして、bar メソッドが実行され bar が出力される直前に

図 2.6 のアドバイスが実行されるのでこのような出力になる。

2.2.3 GluonJ

GluonJ は Java 用の AOP システムである。アノテーション (annotation) を用いることで、Java の文法内でアスペクトを記述することを可能にしている。[2]

アノテーションとは、クラスやメソッド、パッケージに対してメタデータとして注釈を記入するためのものである。

GluonJ ではアスペクトは @Glue によって注釈されたクラスによって表現される。GluonJ におけるポイントカットとアドバイスは @Glue クラス内で宣言された Pointcut 型のフィールドである。AspectJ と同じように call、set、get などのポイントカット、before、after、around などのアドバイスが実装されている。以下に、図 2.6 のアスペクトと同等の GluonJ におけるアスペクトの記述例を図 2.7 示す。

```
1  @Glue class HelloAspect{
2    @Before(“{System.out.println(‘hello‘);}”)
3    Pointcut printHello = Pcd.call(void Sample#bar());
4  }
```

図 2.7: GluonJ のアスペクトの記述例

Pcd クラスの call メソッドがポイントカットを表しており、@Before アノテーションによりアドバイスが表現されている。

さらに GluonJ ではクラスの改良 (Refinement) をサポートしている。Refinement は @Glue クラス内に含まれる static な入れ子クラスである。

```
1  @Glue class HelloAspect{
2    @Refine static class Hello extends Sample{
3      public void bar(){
4        System.out.println(“hello”);
5      }
6    }
7  }
```

図 2.8: Refinement の記述例

図 2.8 は Refinement の記述例である。@Glue クラス内で @Refine で注釈された static クラスを定義すること @Refine クラスのスーパークラスを改良することができる。図 2.8 は Sample クラスを改良する例である。

このアスペクトは Sample クラスの bar メソッドを hello を出力するように改良する。このアスペクトを適用した Sample クラスを実行すると以下のように出力される。

```
foo
hello
```

2.3　AOP を用いた解決策

アスペクトの織り込み技法として static weaving、dynamic weaving、load-time weaving などが研究されている。本節では、これらの織り込み技法についての説明をし、2.1 節で述べた Web アプリケーションの問題点の解決策について検討する。

Web アプリケーションの開発者は通常の Web アプリケーションを作成し、Web アプリケーションのユーザがアスペクトを記述し、そのアスペクトを Web アプリケーションに適用することができれば、どのような Web アプリケーションでもカスタマイズ可能にすることができる。それ故、Web アプリケーションの開発者はプラグイン機構のようなカスタマイズ可能にするための機構を組み込む必要がないし、Web アプリケーションのユーザは Web アプリケーションの任意の場所をカスタマイズする事が出来る。また、ソースコードを変更しないので、Web アプリケーションの可読性は下がらない。

2.3.1　static weaving

static-weaving はプログラムの実行前にアスペクトを weave しておく技術である。AspectJ では static weaving の中でも compile-time weaving を採用している。[5]

compile-time weaving とは、コンパイル時にアスペクトを weave する技術である。AspectJ のコンパイラは、java コンパイラを拡張されたものとして実装されている。AspectJ のコンパイラは、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 の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。AspectJ の実行例を 図 2.9 に示す。

```
$ajc Sample.java HelloAspect.java
$ls
HelloAspect.java HelloAspect.class
Sample.java Sample.class
```

図 2.9: AspectJ のコンパイル例

HelloAspect.class にはアドバイスがコンパイルされてできたメソッドが定義されており、Sample.class ではそのメソッド呼び出しが挿入されている。Sample.class は上で述べたように通常の VM で実行することができる。

この compile-time weaving を Web アプリケーションをカスタマイズするために用いる場合、ユーザからリクエストが来るたびに Web アプリケーションとユーザのアスペクトをコンパイルしロードする方法と、Web アプリケーションとユーザのアスペクトをコンパイルしたクラスファイルをユーザ毎に保持しておく方法が考えられるが、どちらの方法も現実的でない。

前者の場合はコンパイルによるオーバーヘッドが大きすぎるし、後者の場合はユーザが多い場合にディスクの領域を使いすぎてしまう。

2.3.2　dynamic weaving

Dynamic weaving はプログラムの実行中にアスペクトを weave し、プログラムの挙動を変更する技術である。

PROSE [7] はイベント通知を用いた Dynamic weaving システムである。ジョインポイントでイベントを発生させ、アドバイスを実行し、元のプログラムの実行を再開することができる。

しかし、この織り込み技法では、イベントの発生してから元のプログラムに復帰するまでの時間的コストが大きく、オーバーヘッドが大きくなってしまうという問題がある。

その他、java.lang.Instrument API や JPDA を用いることで HotSwap を用いた動的なアスペクトの weave を行うことができるが、通常の JVM の制限によりメソッドボディしか動的にアスペクトを weave する事ができないという問題がある。この問題によりアスペクトによる Web アプリケーションの拡張を制限されてしまう。この問題は VM を拡張することで回避することができる。

Streamloom [4] は RVM をアスペクト指向用に拡張することで動的な

織り込みを実現してる。しかし、VM の拡張は回避不可能はエラーを引き起こすこともあり、現実的ではない。

これらの理由により Dynamic weaving は Web アプリケーションの拡張には適当でない。

2.3.3　load-time weaving

load-time weaving はクラスのロード時にアスペクトを weave する技術である。GluonJ は static weaving と load-time weaving をサポートしているが、ここでは load-time weaving の例として示す。

GluonJ では java.lang.Instrument API を用いて load-time weaving を行っている。

java.lang.Instrument API を利用するには、観察対象となるクラスがロードされるのを監視するクラス (エージェント・クラス) が必要になる。図 2.10 にエージェント・クラスの記述例を示す。

```
1  public class Agent implements ClassFileTransformer{
2    public static void premain(String agentArgs,
         Instrumentation inst){
3                              ...
4      Agent agent = new Agent();
5      inst.addTransformer(agent);
6    }
7    public byte[] transform(ClassLoader loader, String cName,
8              Class<?> clazz, ProtectionDomain pd, byte[] b){
9      /*
10      * create bytecode
11      */
12   }
13 }
```

図 2.10: エージェント・クラス

ClassFileTransformer インタフェースの実装クラスの premain メソッドは観察対象のアプリケーションの main メソッドが呼ばれる前に実行される。この例では premain メソッド内で Agent クラスのオブジェクトをアプリケーションのロードの過程で呼び出される ClassFileTransformer オブジェクトとして登録している。こうすることで観察対象のアプリケーションのクラスがロードされるたびに登録された ClassFileTransformer オブジェクトの transform メソッドが呼び出されるようになる。transform

メソッドの引数 byte[] b にはロード前のクラスのバイトコードが渡されるので transform メソッド内でバイトコードを編集することができる。GluonJ はバイトコードの編集を Javassist [8] を用いて行っている。

Javassist とは Java のバイトコードを編集するためのライブラリである。Javassist を用いることにより、実行時に新たなクラスを定義したり、クラスのロード時にクラスファイルを修正したりすることができる。図 2.11 に Javassist の使用例を示す。

```
1  ClassPool classPool = ClassPool.getDefault();
2  CtClass ctClass = classPool.get(‘‘Sample’’);
3  CtMethod ctMethod = ctClass.getDeclaredMethod(‘‘bar’’);
4  ctMethod.insertBefore(‘‘System.out.println(‘‘hello’’)’’);
5  byte [] bytes = ctClass.toBytecode();
```

図 2.11: Javassist の使用例

ClassPool は CtClass オブジェクトのコンテナである。CtClass オブジェクトはクラスを表すオブジェクトであり、この例では Sample クラスを表すオブジェクトを ClassPool から取得している。CtMethod オブジェクトはメソッドを表すオブジェクトであり、この例では Sample クラスの bar メソッドを表すオブジェクトを取得している。4 行目では CtMethod クラスの insertBefore メソッドにより bar メソッドの先頭に hello という文字列を出力する分を挿入している。最後の行では編集した CtClass オブジェクトをバイトコードに変換している。

次に GluonJ を用いたロード時織り込みの実行方法を示す (図 2.12)。

```
java -javaagent:jarpath[=option]
java -javaagent:gluonj.jar=HelloAspect Sample
```

図 2.12: GluonJ の実行例

こうすることで Sample クラスの main メソッドが呼ばれる前に GluonJ のエージェント・クラスの premain メソッドが呼ばれる。premain メソッドの引数 agentArgs にはオプションで指定した HelloAspect という文字列が渡され、GluonJ は HelloAspect クラスをロードし、定義されたアスペクトの情報を得る。そして、Sample クラスのロード時に transform メソッドが呼ばれるので、そのアスペクトの情報をもとに Sample クラスのバイトコードを Javassist を用いて編集する。

static weaving のコンパイル時オーバーヘッドやディスク領域の問題、dynamic weaving の weave によるオーバーヘッドや VM のエラーの問題を考慮すると Web アプリケーションへのアスペクトの織り込み技法は load-time weaving が適当であると考えられる。load-time weaving を用いれば、Web アプリケーションのロード時にユーザのアスペクトを適用すれば良いので、サーバーサイドに登録しておくべき情報は、ユーザのアスペクトのみでよい。また、アスペクトの織り込みによるオーバーヘッドも小さい。しかし、load-time weaving にもロード時のオーバヘッドが存在する。それについては 4 章で議論する。

第3章　per-session AOP を用いたフレームワーク

本研究では、Web アプリケーションのユーザがアスペクトを登録することで、ユーザ毎に Web アプリケーションをカスタマイズすることができるフレームワークを開発した。Web アプリケーションの開発者はこのフレームワークが提供しているアスペクト登録用 API を用いてユーザのアスペクトを登録するだけで Web アプリケーションにユーザのアスペクトを weave することができる。つまり、Web アプリケーションの開発者は通常の Web アプリケーションに本フレームワークを適用することで Web アプリケーションをユーザ毎にカスタマイズ可能にする事が出来る。

3.1　本フレームワークの概要

本フレームワークを用いることで Web アプリケーションを容易にユーザ毎にカスタマイズ可能にすることができる。以下、本システムの概要を述べる。

3.1.1　本フレームワークの利点

ユーザ毎にカスタマイズ可能な Web アプリケーション開発には 2 章で述べたように以下のような問題点があった。

- 通常の Web アプリケーション開発よりも多くの前提知識が必要である。
- 開発時に決めた拡張ポイント以外はカスタマイズ不可能である。
- カスタマイズ可能にするための機構を組み込むことにより、ソースコードの可読性が下がる

しかし、本フレームワークを用いて Web アプリケーション開発を行えば、これらの問題点はすべて解決できる。AOP を用いて Web アプリケーションの挙動を変更するので、Web アプリケーションの開発者は通常の

Web アプリケーション開発をすればよい。それ故、カスタマイズ可能にするための前提知識も必要ないし、ソースコードの可読性も下がらない。また、AOP を用いることでユーザは Web アプリケーションの任意のポイントをカスタマイズすることができる。

3.1.2　フレームワークの使用例

まず最初に、図 3.1 のような掲示板アプリケーションを用いた本フレームワークの使用例を示す。

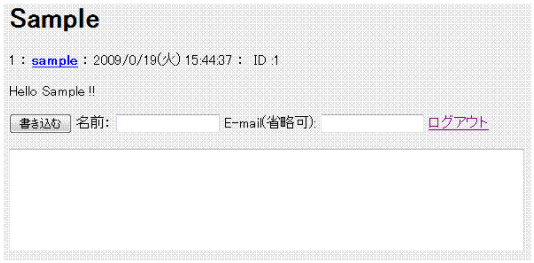

図 3.1: 掲示板アプリケーション

Web アプリケーションの開発者は本フレームワークを用いてアスペクトをユーザ毎に Web アプリケーションに適用するために、何らかの方法を用いてユーザがアスペクトをサーバサイドに送信できなければならない。この例では 図 3.2 のようなアスペクト登録フォームを掲示板アプリケーションに設けている。

図 3.2: アスペクト登録フォーム

図 3.2 のアスペクト登録フォームを用いて、Web アプリケーションのユーザはアスペクトを登録する。Web アプリケーション側では、登録フォームからのリクエストを処理するクラス内で本フレームワークが提供するアスペクト登録用 API を用いてフレームワークにユーザのアスペクトを登録する。

この例では、図 3.3 のようなアスペクトを登録する。

```
1  @Glue class ColorAspect{
2    @Refine static class Aspect extends BBS{
3      public void setColor(PrintWriter out){
4        out.println("<body bgcolor=#ffefcc>");
5      }}}
```

図 3.3: 背景色を変更するアスペクト

このアスペクトは Web アプリケーションの内のクラス BBS 内に定義されている背景色を設定するメソッド setColor メソッドを GluonJ の Refinement を用いて改良している。このアスペクトを適用することで、BBS の setColor メソッドの内容は以下のようになる。

```
out.println("<body bgcolor=#ffefcc>");
```

これによって、背景色は # ffcfcc に設定される。このアスペクトが適用されるのは、このアスペクトを登録したユーザがリクエストを Web アプリケーションに送信したときのみであり、他のユーザがリクエストを送信したときは適用されない。図 3.3 のアスペクトを登録したユーザがリクエストを送信したときは 図 3.4 のような掲示板アプリケーションが実行される。

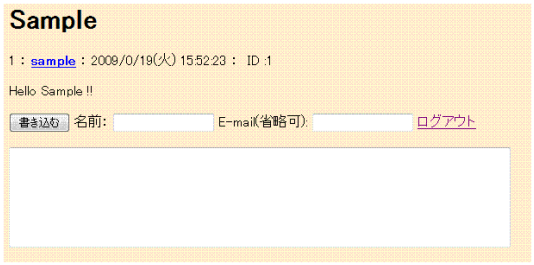

図 3.4: アスペクトを weave した Web アプリケーション

この例のように、プラグイン機構のような Web アプリケーションをカスタマイズ可能にするための機構を組み込まなくても AOP を用いれば、ジョインポイント全てが拡張ポイントとなり、さらにその拡張ポイントには任意のコードを挿入することができるので、ユーザは Web アプリケーションを自由にカスタマイズすることができる。

3.1.3　フレームワークの有効化

本フレームワークは Tomcat 5 サーバーの一部として実装されており、Tomcat 5 の設定ファイルでフレームワークを有効にするかどうかを設定することができる。

Tomcat 5 サーバーのメインの設定ファイルである server.xml または、server.xml の context 要素のみを記述する context.xml に次のような設定をする事でフレームワークを有効化できる。

図 3.5 に Context.xml の記述例を示す。

```
<Context path="/SampleApp"
         reloadable="true"
         docBase="C:\Users\SampleApp"
         workDir="C:\Users\SampleApp\work"
         wrapperClass="WebappWrapper">
<Valve className="SessionGetterValve">
<Context/>
```

図 3.5: Context 要素

図 3.5 において通常の Web アプリケーションの設定と異なる部分は Context 要素の wrapperClass 属性と Valve 要素の className 属性である。この二つの属性に本フレームワークが提供するクラスを適用することでフレームワークが有効化される。

この例では SampleApp という名前の Web アプリケーションに対して本フレームワークを有効化している。Wrapper と Valve については 4 章で述べる。

3.1.4　アスペクト登録時の処理

図 3.6 に Web アプリケーションのユーザがアスペクトを登録した時の流れを示す。

1. クライアントが Web アプリケーション開発者が用意した登録フォームなどを利用することでアスペクトをリクエストに添付し、サーバーに送信する。

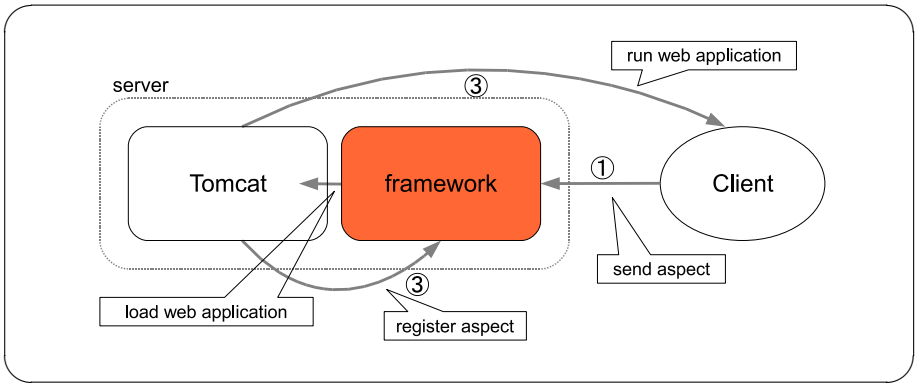

図 3.6: ユーザアスペクトの登録

2. クライアントが既にアスペクトを登録しているならばアスペクトを織り込んだ Web アプリケーションをロード、登録していないならば通常の Web アプリケーションをロードする。

3. Web アプリケーションを実行し、クライアントに応答する。また、リクエストからアスペクトを取得し、アスペクト登録用 API を用いてフレームワークにアスペクトを登録する。

ユーザのアスペクトを登録するために実行される Web アプリケーションは登録用 API による登録が完了されるまでカスタマイズされない。つまり、API 実行後にロードされる Web アプリケーション内のクラスはユーザのアスペクトによりカスタマイズされる。

3.1.5　Web アプリケーション実行

Web アプリケーションに本フレームワークを適用し、実行した際の流れを 図 3.7 に示す。

1. クライアント A がリクエストを送信する。

2. フレームワークがクライアントの ID を取得し、適用すべきアスペクトを特定する。

3. フレームワーク内でアスペクトを Web アプリケーションに weave、ロードし、インスタンスを作成する。

4. アスペクトが weave された Web アプリケーションを実行し、クライアントに応答する。

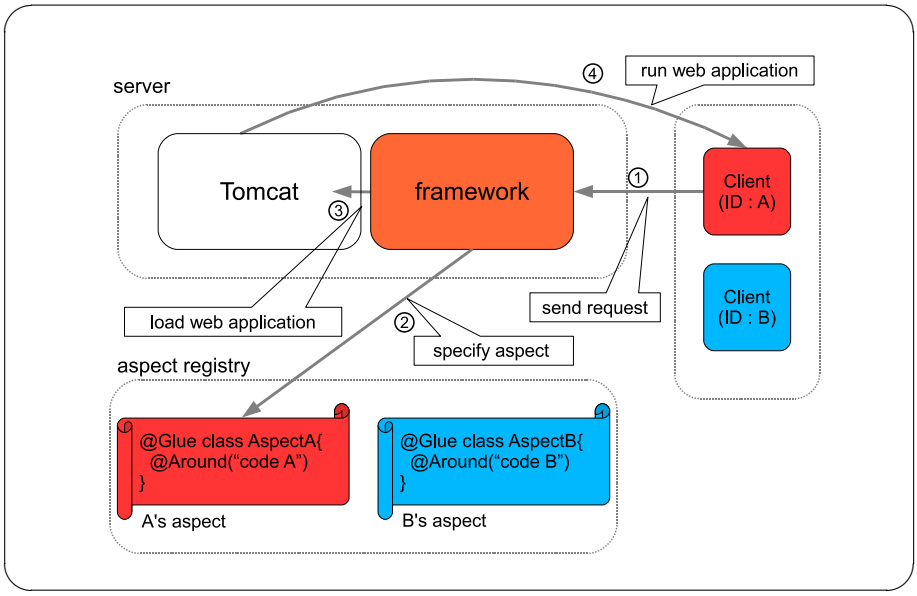

図 3.7: Web アプリケーションの実行

セッションが無い、セッションにユーザ ID がバインドされていない、アスペクトが登録されてない場合は通常の Web アプリケーションが実行される。

3.2　per-session AOP

本フレームワークを用いることで開発者は通常の Web アプリケーションを開発し、Web アプリケーションのユーザがアスペクトを登録するだけでユーザ毎に Web アプリケーションを振る舞いを変更することができる。

この機能を提供するためにはフレームワーク側でユーザ毎に異なるアスペクトを weave し、そのアスペクトは他のユーザのアスペクトに干渉してはならない。

これを解決するために本フレームワークでは per-session AOP という技法を用いている。本節ではまず、Java のクラスロードのアーキテクチャについて述べ、次に per-session AOP ついて述べる。

3.2.1　Java の ClassLoader

Java アプリケーションを実行する JVM は実行に必要になったクラスを実行中にロードする仕組みになっている。クラスのロードにはクラスローダが使用され、クラスローダは親子関係を持つ。JVM からクラスの

ロード要求があった場合、クラスローダは loadClass メソッドにより以下のような手順でクラスをロードする [6]。

1. findLoadedClass メソッドを呼び出して、クラスが既にロードされたかどうかを確認する。
2. 親クラスローダの loadClass メソッドを呼び出す。親が null の場合、仮想マシンに組み込まれたクラスローダが代わりに呼び出す。
3. 親クラスローダの loadClass メソッドが java.lang.ClassNotFoundException を投げた場合、findClass メソッドを呼び出して、クラスを検索する。

findLoadedClass メソッドは指定された名前を持つクラスがこのクラスローダによって既にロードされている場合、そのクラスを探すメソッドである。findClass メソッドは、クラスローダのクラス毎に異なるメソッドであり、このメソッドをオーバーライドすることで親クラスローダでクラスが見つからなかった場合のクラスの探索方法やその他の処理を独自に定義することができる。

異なるクラスローダは異なる名前空間をもっている。つまり、同一のクラスローダで、同じ名前の二つのクラスをロードすることはできないが、クラスローダが異なれば、同じ名前のクラスをロードすることができる (図 3.8)。

3.2.2　per-session AOP の仕組み

per-session AOP は load-time weaving を拡張した織り込み技法である。この per-session AOP を用いることでユーザ毎に異なるアスペクトを Web アプリケーションに weave することを可能にしている。いかに per-session AOP を用いた場合の処理の流れを示す。

1. ユーザのリクエストからセッションを取得し、そのセッションから ID を取得する。
2. ユーザのクラスローダに ID を渡し、ユーザの登録したアスペクトのパスをそのクラスローダに追加する。
3. そのクラスローダによる Web アプリケーションのロード前にアスペクトを weave し、ロードする。

per-session AOP では、先に述べたクラスローダのロードの仕組み利用して findClass メソッドを拡張し、そのメソッド内でアスペクトの weave を行う。独自のクラスローダをユーザ毎に作成することにより、ユーザ毎

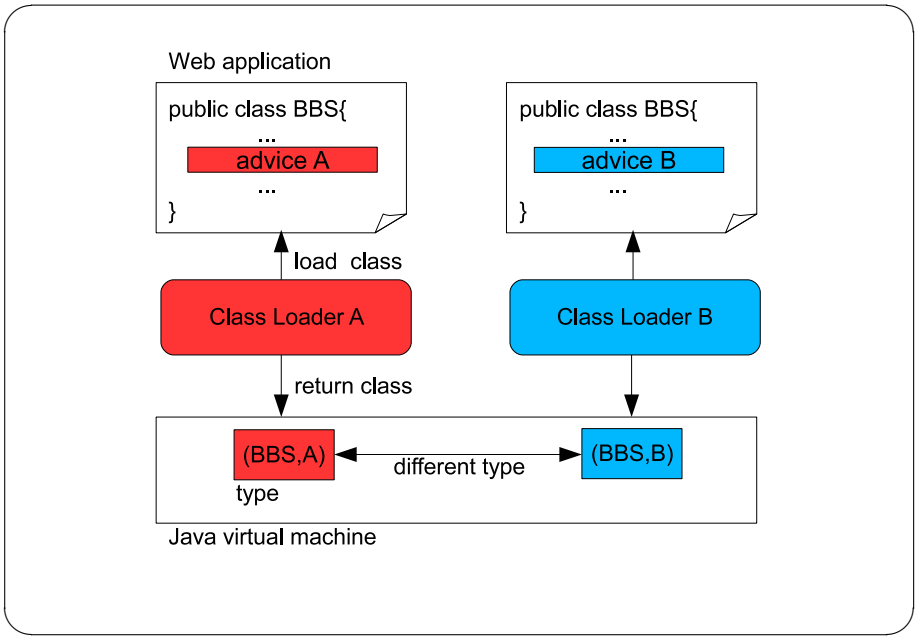

図 3.8: クラスのロード

に異なるアスペクトを weave した Web アプリケーションをロードすることができる。なぜなら、異なるクラスローダは異なる名前空間をもつので、ユーザ毎に異なるクラスローダを用いれば、異なるアスペクトが weave された Web アプリケーション名の衝突は起こらない。つまり、ユーザ毎に新たにクラスローダを作成すれば、そのクラスローダのロード前の名前空間には Web アプリケーションのクラスは存在しないので、ユーザ毎に新たに Web アプリケーションのクラスをロードすることができる。

3.3　アスペクト登録用 API

本フレームワークではユーザから送られてきたアスペクトを容易に登録することのできる API を提供している。

ソースコードによる登録

アスペクトのソースコードを API に渡すことでユーザのアスペクトを登録することができる。図 3.9 にこの API を使った登録を行う Web アプリケーションの例を示す。

この API を用いれば文字列での登録が可能なので 図 3.9 のようなアスペクトのソースを直接書き込むフォームを作成し、Web アプリケーショ

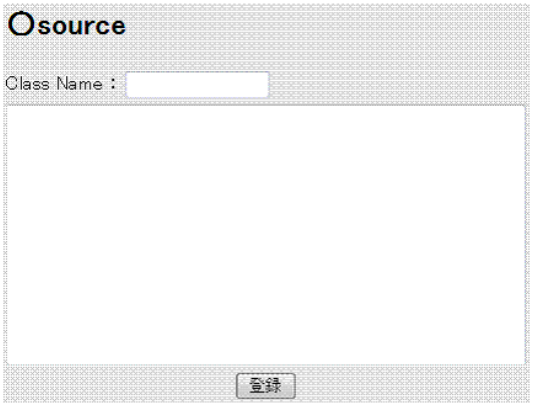

図 3.9: 文字列によるアスペクトの登録

ン内でその文字列をリクエストから取得すれば、ユーザのアスペクトを登録することができる。

ファイルによる登録

ファイルによる登録を行う API ではアスペクトを記述した java ファイル、アスペクトをコンパイルした class ファイル、アスペクトの java ファイルやその java ファイル内で使用するクラスを記述した java ファイルを圧縮した zip ファイルを登録することができる。ユーザのファイルを取得するには図 3.2 のようなフォームを Web アプリケーションに作成すればよい。

第4章　実装

この章では、本フレームワークの実装に必要な技術についての詳細と、実際の実装方法について述べる。

4.1　Tomcat

本フレームワークは Tomcat 5 サーバーの一部として実装されている。Tomcat は java で実装された Web サーバーであり、また、サーブレットコンテナでもある。サーブレットコンテナとはサーブレットを実行させることのできる Web サーバーのことである。[3]

以下、Tomcat のアーキテクチャについて述べる。

4.1.1　Tomcat の構造

図 4.1 に Tomcat の構造を示す。

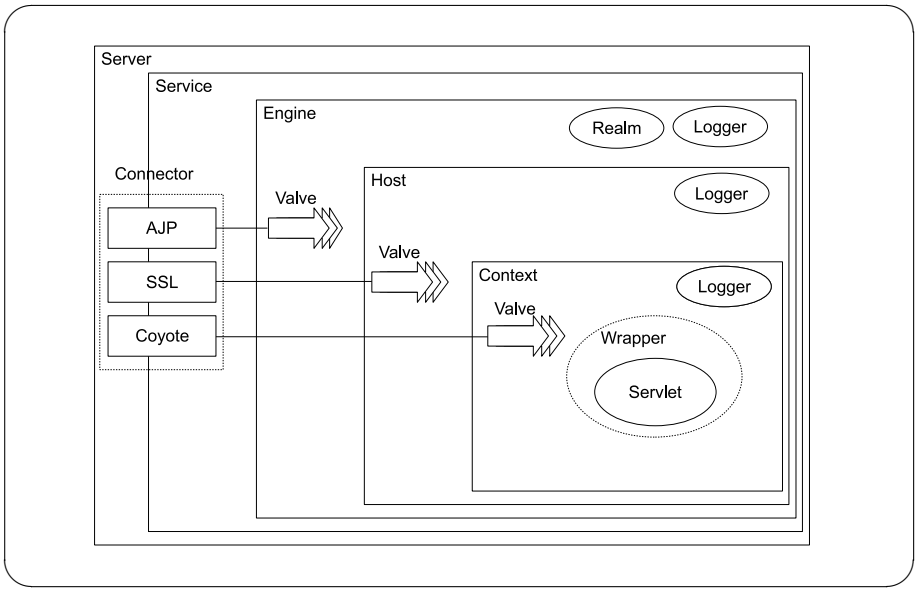

図 4.1: Tomcat の構造

Tomcat はネストされた階層のコンポーネントから構成されている。これらのコンポーネントのいくつかはトップレベルコンポーネントと呼ばれ、階層構造の一番上に位置することを意味している。

以下、それぞれのコンポーネントについて説明する。

Server

Server は Web サーバーのインスタンスとしての Tomcat 自身のことであり、トップレベルコンポーネントである。個々の JVM 上に作成できる Tomcat Server インスタンスは一つのみである。

Service

Service はトップレベルコンポーネントであり、Engine と Connector をグループ化する。

Connector

Connector はクライアントからのリクエストを受け取り、そのリクエストをサーバーのポートに送る。デフォルトの Connector は Coyote であるが、その他 AJP Connector、SSL Connector などが用意されている。

Engine

Engine はリクエストを処理するトップレベルコンポーネントである。Engine は HTTP ヘッダをチェックして、どの仮想ホスト、または Context にリクエストを送るかを決定する。

Realm

Engine の Realm はユーザ認証を管理するためのコンポーネントである。Realm は Engine 全体で共有される。

Logger

Logger は各コンポーネントの状態をロギングするためのコンポーネントである。

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。Tomcat で提供されている Valve の例を以下に示す。

- org.apache.catalina.valves.AccessLogValve
  リクエストのロギング
- org.apache.catalina.valves.RequestFilterValve
  リクエストのフィルタリング
- org.apache.catalina.valves.RequestDumperValve
  特定のリクエストとレスポンスについての情報をダンプ
- org.apache.catalina.valves.ErrorReportValve
  HTML のエラーページの出力

Host

Host は Tomcat における仮想サーバーのことである。Tomcat 上では Host 名の異なる複数の仮想サーバーを立てることが可能であり、Engine で HTTP ヘッダを見て Host 名を特定し、どの Host にリクエストを送信するかを決める。

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 メソッドにより初期化されたサーブレットインスタンスを割り当てる。

Context

Web アプリケーション自体を表すコンポーネントである。

4.1.2　Tomcat 5 の ClassLoader

Tomcat 5 では、独自の親子関係に編成されたクラスローダを用いて Web アプリケーションのロードを行っている。

図 4.2 に Tomcat 5 におけるクラスローダの親子関係を示す。

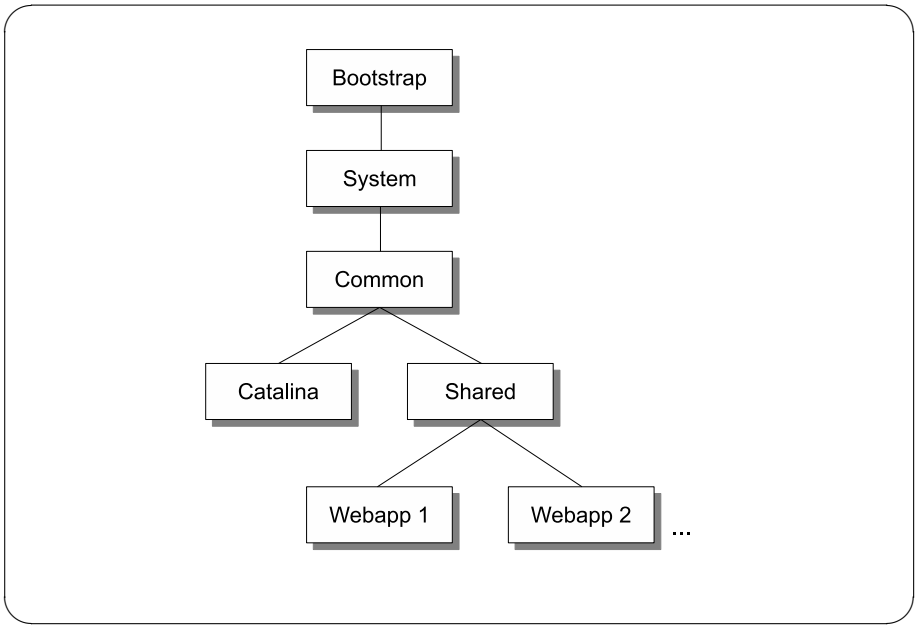

図 4.2: Tomcat 5 のクラスローダ

Bootstrap

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。

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組み込みローダである Bootstrap クラスローダである。

System

このクラスローダは 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 5 において System クラスローダは CLASSPATH 以外に$CATALINA_HOME/bin/bootstrap.jar や $CATALINA_HOME/lib/tools.jar 内のクラスのロードを行う。

bootstrap.jar にはサーバーの main メソッドを含むクラスや Tomcat 5 のクラスローダの実装クラスなどが含まれている。

tools.jar には JSP をサーブレットに変換するためのコンパイラなどが含まれている。

Common

このクラスローダは tomcat 内部からも Web アプリケーションからも参照可能なクラスのロードを行う。$CATALINA_HOME/common/classes、$CATALINA_HOME/common/lib、$CATALINA_HOME/common/endorsed に含まれるクラスや jar ファイルに含まれるクラス、リソースの検索を行う。

Catalina

このクラスローダは tomcat 5 そのものを実装するのに必要なクラスのロードを行う。$CATALINA_HOME/server/lib、$CATALINA_HOME/server/classes に含まれるクラスや jar ファイルに含まれるクラス、リソースの検索を行う。Tomcat 5 のクラスローダの親子関係からもわかるように、これらのクラスやリソースは Web アプリケーションからは参照できない。

Shared

このクラスローダは全ての Web アプリケーションに共通して共有したいライブラリやクラスのロードを行う。$CATALINA_HOME/shared/lib、$CATALINA_HOME/shared/classes に含まれるクラスや jar ファイル内のクラス、リソースの検索を行う。

Webapp x

このクラスローダは各 Web アプリケーションに一つ作られるクラスローダで Web アプリケーションと Web アプリケーション内で使われるライブラリのロードを行う。Web アプリケーションの配置場所の /WEB-INF/lib、/WEB-INF/classes に含まれるクラスや jar ファイル内のクラス、リソースの検索を行う。

4.2　本フレームワークの実装

本フレームワークは独自の Valve と Wrapper を tomcat 5 サーバー内で使用することにより実現されている。さらに、独自のクラスローダの親子関係を用いることでユーザ毎に異なるアスペクトを weave することを可能にしている。以下、本フレームワークの実装について述べる。

4.2.1　Valve

tomcat では設定ファイル内の Valve 要素でクラス名を指定することにより、独自の Valve を追加することができる (図 4.3)。

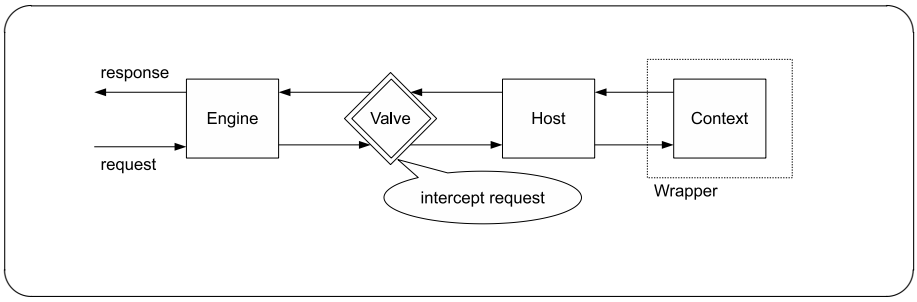

図 4.3: Valve の追加

本フレームワークではリクエストをインターセプトしセッションを取得するために Valve を追加している。図 4.4 に本フレームワークにおける Valve を示す。

Tomcat 内の一連の Valve は、invoke メソッドを呼び出すことでクライアントからのリクエストを次の Valve へ渡す。invoke メソッド内ではリクエストを取得できるので図 4.4 の Valve では setSession メソッドで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。

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で使用するための getter メソッドである。

tomcat では各々のリクエストに対してスレッドを作成し処理を行うのでセッションの格納には ThreadLocal 変数を用いなければならない。

例えば、Valve でセッション A を取得し、フレームワーク内で Valve の getHttpSession メソッドが呼ばれ、セッション A を取得する前に次のリクエストにより Valve 内でセッション B を取得した場合、getHttpSession メソッドにより取得されるセッションはセッション A ではなく、セッション B になってしまう。本フレームワークではセッションを見てユーザのアスペクトを特定するので、セッションはスレッド内で一意でなければならない。

4.2.2　Wrapper

Wrapper も Valve と同様に設定ファイル内で指定したものを Servlet の Wrapper として使用する事ができる。本フレームワークで使用する Wrapper は図 4.5 次のようなものである。

```
1  public class SessionGetterValve extends ValveBase{
2
3    private static ThreadLocal<HttpSession> threadLocal
4          = new ThreadLocal<HttpSession>();
5
6    @Override
7    public void invoke(Request req, Response res)
8      throws IOException,ServletException {
9
10     setSession(req, res);
11     getNext().invoke(req, res);
12   }
13
14   public synchronized void setSession(Request req,
15     Response res) throws IOException, ServletException{
16
17     HttpSession session = req.getSession(false);
18     threadLocal.set(session);
19   }
20
21   public static HttpSession getHttpSession(){
22     return threadLocal.get();
23   }
24 }
```

図 4.4: フレームワークで使用する Valve

Wrapper の allocate メソッドは、この Wrapper がラップしている Servlet クラスのインスタンスを作成し、返すメソッドである。

本フレームワークの Wrapper では、Valve からセッションを取得し、ユーザ ID を取得する。そして、そのユーザ ID をもとにユーザ専用のクラスローダを作成し、アスペクトを weave し、Web アプリケーションのロード、インスタンスの作成を行う。以下、詳細を述べる。

まず 9 行目で Valve からセッションを取得する。変数 uid にはセッションから取得したユーザ ID が格納される。セッションが無い、またはこのユーザがアスペクトを登録していない場合、アスペクトを weave する必要が無いので 12 行目ではスーパークラスの allocate メソッドによりデフォルトのインスタンスの割り当てを行う。

アスペクトが登録されている場合には、28 行目が実行される。createGlueWeaveLoader メソッドは、この Wrapper クラスに定義されているユーザ専用のローダを作成するメソッドである。29 行目では getServletClass メソッドにより、この wrapper がラップしている Servlet ク

ラスのクラス名を取得し、ロードする。loadClass メソッド内でこのユーザのアスペクトが weave される。

4.2.3　フレームワークで使用するクラスローダの構造

ユーザ毎に異なるアスペクトを weave したクラスをロードするためには、ユーザ毎に異なるクラスローダが必要である。全てのユーザに対して同一のクラスローダを用いた場合、そのクラスローダの名前空間には最初にロードした Web アプリケーションのクラスが存在し、クラスローダにロード要求があった時、常にそのクラスを返してしまう。これを避けるためにはユーザ毎に異なるクラスローダを作成する必要がある。そこで、本フレームワークでは独自のクラスローダの親子関係を用いている。

図 4.6 に本フレームワークのクラスローダの構造を示す。

GlueWeaveLoader

GlueWeaveLoader は 図 4.6 における User x ローダのことである。このクラスローダはユーザ毎に作成され、ユーザが登録したアスペクトを weave した Web アプリケーションをロードする。

GlueWeaveLoader は 図 4.7 にのようなクラスである。

コンストラクタ内の 8 行目では親ローダを WebappCommonLoader クラスのローダに設定している。10 行目では ClassPool オブジェクト作成し、そのオブジェクトに Web アプリケーションのパスとユーザアスペクトのパスを通している。12 行目では setAspect メソッドを呼び出し、Weaver クラスのオブジェクトを作成している。Weaver クラスとは、GluonJ を実装しているクラスの一つで、引数に渡されたアスペクトの内容を特定のクラスに weave するためのクラスである。

loadClass メソッドでは、23 行目でコンテキストクラスローダをこのローダ自身に設定している。Class クラスの forName メソッドではコンテキストクラスローダを用いてロードを行うので、Web アプリケーション内で forName メソッドの呼び出しがあった場合、Web アプリケーションをロードしたローダ自身に設定しなければならない。そうしなければ次のようなコードがあった場合に型の不一致が起きてしまう。

```
Class clazz = Class.forName("BBS");
BBS bbs = (BBS)clazz.newInstance();
```

Tomcat ではリクエストが来るたびにコンテキストクラスローダを 図 4.2 の Webapp x ローダに設定するので、上の forName で BBS クラス

のロードをするのは Webapp x ローダである。しかし、ユーザアスペクトによりカスタマイズされたクラスは User x ローダによりロードされるので 2 行目の BBS クラスは User x ローダにロードされてしまう。JVM ではクラスローダが異なれば、クラス名が同じでも異なる型とみなされるので 2 行目で ClassCastException が投げられてしまう。

findClass メソッドではアスペクトの weave とクラスの定義を行う。setAspect メソッドにより作成した Weaver クラスのオブジェクトの trasform メソッドによりアスペクトの weave を行う。アスペクトを weave した CtClass オブジェクトをバイト列に変換し、defineClass メソッドにより Class オブジェクトを返す。

WebappCommonLoader

WebappCommonLoader は図 4.6 における WebappCommon x ローダのことである。WebappCommon x は図 4.2 における Shared ローダを親ローダに持つクラスローダである。このクラスローダは Web アプリケーション毎に作成され、Web アプリケーション内で使用されているライブラリのみロードする。すなわち、Web アプリケーションの配置場所の /WEB-INF/lib に含まれる jar ファイルのロードを行う。

WebappCommon x ローダは CommonLoaderFactory クラスで管理されている。CommonLoaderFactory クラスでは、WebappCommon x ローダと Web アプリケーションの配置場所とのマップを保持している。それぞれの WebappCommon x ローダは作成時に Shared ローダを親ローダに設定される。

Wrapper で User x ローダを作成するときに、Web アプリケーションの配置場所を CommonLoaderFactory の getCommonLoader メソッドの引数に渡すことで、この Web アプリケーションで使われているライブラリのロードを行う WebappCommon x ローダを取得することができる。このローダを GlueWeaveLoader のコンストラクタの引数に渡すことで、図 4.6 のようなクラスローダの親子関係を構築することができる。

本フレームワークではユーザ毎に異なるアスペクトを weave するためにユーザ毎に異なるクラスローダを作成している。そのため、ローダを新たに作成するたびに Web アプリケーションを再定義しなければならないのでロード時のオーバーヘッドが大きい。WebappCommon x ローダはそのオーバーヘッドを軽減するための対策である。WebappCommon x ローダでロードされるクラスはユーザがカスタマイズすることの無いライブラリなので、ユーザ毎に作成する必要はなく、Web アプリケーション毎に一つ作成しておき、JVM からライブラリのロード要求があった場合は WebappCommon x がロード済みのライブラリを返せばよい。こうす

ることで、User x ローダが Web アプリケーション内のすべてのクラスをロードする場合に比べてオーバーヘッドを軽減することができる。

4.2.4　ClassLoader のキャッシュ

WebappCommonLoader と同様に、ロード時のオーバーヘッド軽減のための対策である。

User x ローダはユーザからリクエストが送信されるたびに作成されるが、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。そこで、本フレームワークでは Web アプリケーションごとにクラスローダ用のキャッシュを設けるようにした。キャッシュには LRU (Least Recently Used) アルゴリズムを用いている。

キャッシュ内に格納されているクラスローダはユーザのアスペクトを weave した Web アプリケーションをロード済みなのでロード要求があればそのクラスを返すだけでよい。それ故、findClass によるクラスの定義は必要なく、ロード時のオーバーヘッドを軽減することができる。

図 4.5 の 16 行目ではユーザのアスペクトが変更されたら新たにローダを作成し、それをキャッシュに格納するようにしている。新たにローダを作成せずにキャッシュから取得したローダを用いた場合、新たに登録したアスペクトは反映されず、以前にロードした Web アプリケーションがローダにより返されてしまうからである。

21 行目では、キャッシュからローダを取得している。キャッシュ内にローダがなかった場合は、ユーザが初めてアスペクトを登録した場合か、キャッシュからそのユーザのローダが破棄されてしまった場合なので、新たにローダを作成し、キャッシュに登録する必要がある。

```
1  public class WebappWrapper extends StandardWrapper{
2    private static LoaderCache loaderCache;
3                    ...
4
5    @Override
6    public Servlet allocate() throws ServletException{
7      ClassLoader loader;
8      Class<?> clazz;
9      String uid;
10     HttpSession s = SessionGetterValve.getHttpSession();
11                   ...
12     if(session==null || !AspectRegistry.hasUserAspect(uid)){
13       return super.allocate();
14     }
15                   ...
16     if(AspectRegistry.aspectChaged(uid)){
17       AspectRegistry.removeChangedUser(uid);
18       loader = createGlueWeaveLoader(uid);
19       loaderCache.addElement(servletDir,uid, loader);
20     }
21     else {
22       loader = loaderCache.getElement(servletDir, uid);
23       if(loader == null){
24         loader = createGlueWeaveLoader(uid);
25         loaderCache.addElement(servletDir, uid, loader);
26       }
27     }
28                   ...
29     loader = createGlueWeaveLoader(uid);
30     clazz = loader.loadClass(getServletClass());
31     return (Servlet) clazz.newInstance();
32   }
33                   ...
34 }
```

図 4.5: 本フレームワークで使用する Wrapper

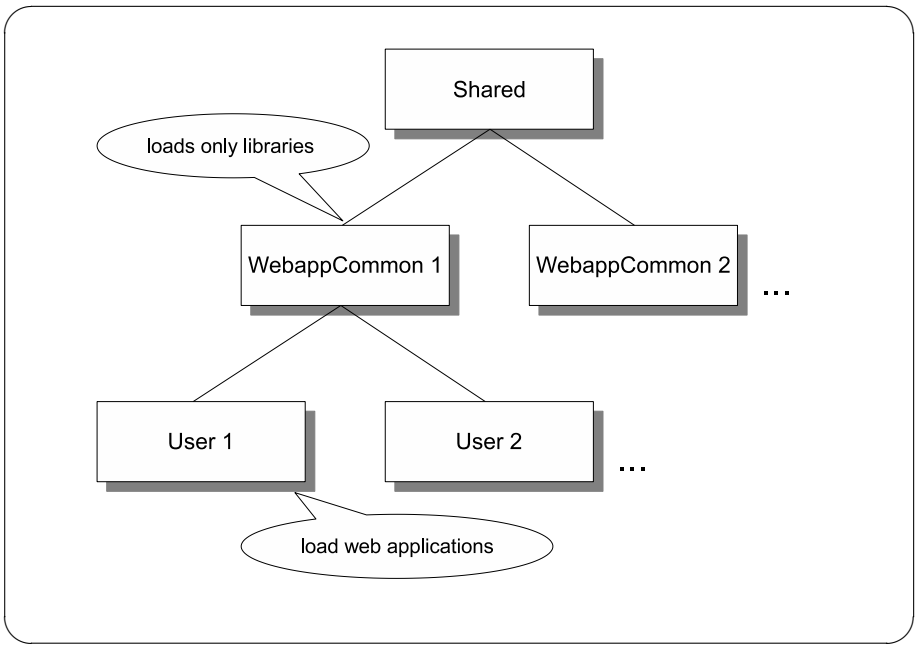

図 4.6: フレームワーク内のクラスローダの構造

```
1  public class GlueWeaveLoader extends ClassLoader{
2    private Weaver weaver;
3    private ClassPool pool;
4            ...
5    public GlueWeaveLoder(String uid, DirContext servletDir,
6                 WebappCommonLoader commonLoader)throws ...{
7
8      super(commonLoader);
9            ...
10     pool = new ClassPool();
11     // insert classpath into pool
12     setAspect(AspectRegistry.getAspectClassName(uid));
13   }
14           ...
15   private void setAspect(String aspect)throws ...{
16           ...
17     weaver = new Weaver(aspect,pool,this.getParent());
18   }
19
20   public Class<?> loadClass(String name)
21          throws ClaasNotFoundException{
22
23     Thread.currentThread().setContextClassLoader(this);
24     return super.loadClass(name);
25   }
26
27   protected Class<?> findClass(String name)
28             throws ClassNotFoundException{
29           ...
30     CtClass cc = weaver.transform(name);
31     byte[] b = cc.toByteCode();
32     return defineClass(name, b, 0, b.length);
33   }
34 }
```

図 4.7: GlueWeaveLoader クラス

第5章　実験

フレームワークの実装の効率化による性能変化を測るための実験を行った。この章では、実験結果を示し、その考察を行う。

5.0.5　実験の概要と実験環境

実験 1、実験 2 ともに、

- **ケース 1** 効率化をしていないフレームワークを使用し Web アプリケーションにリクエストを送る場合
- **ケース 2** 効率化をしたフレームワークを使用し Web アプリケーションにリクエストを送る場合
- **ケース 3** フレームワークを使用せずに Web アプリケーションにリクエストを送る場合

の 3 つの場合に分けて実験を行った。
実験環境は以下のとおりである。

Web アプリケーション　掲示板アプリケーション

コンパイラ　JDK1.6.0

負荷テスト用ソフト　jakarta jmeter 2.3.2

Web Server　Tomcat 5.5

Client マシン　OS：Windows Vista 、CPU：Core2 Duo 3.00 GHz、Memory：4 GB

Server マシン　OS：Linux 2.6.26、CPU：Xeon 2.83 GHz、Memory：4 GB

5.0.6　実験 1

掲示板アプリケーションの中で、ライブラリを多用しているクラスに対してリクエストを送信する実験を行った。このクラス内で参照されるライブラリは次のようなものである。

- DbUtils … DB を扱うための補助ライブラリ
- Lang … java.lang の拡張ライブラリ
- Logging … ロギングライブラリ
- MySQL Connector … DB へのコネクションライブラリ

ケース 1 については 100 ユーザ 100 スレッドで、各ユーザが 1 回ずつ、計 100 回リクエストを送信し、 1 リクエストあたりの平均値を測定した。その他のケースでは 各ユーザが 10 回ずつ、計 1000 回リクエストを送信し、1 リクエストあたりの平均値を測定した。これは、ケース 1 の場合だと、メモリ使用量が大きすぎ 100 リクエストを超えたあたりで OutOfMemoryError により停止してしまうためである。

グラフ内横軸の ＊1 はケース 1 を、＊2 はケース 3 を表している。キャッシュエントリ数についてはケース 2 についてエントリ数を変化させた場合である。図 5.1 は各ケースについてリクエストの送信を行い、1 リクエストあたりの平均応答時間を測定した結果である。

平均応答時間については、ケース 2 がケース 1 に比べて、7 倍から 20 倍程度の速度向上が見られた。しかし、キャッシュエントリ数 100 の場合でも、ケース 3 よりも 3 倍遅い。これは、通常 Tomcat では Servlet インスタンスのキャッシュを行っているが、本フレームワークを用いた場合は、そのキャッシュが無効になってしまうためであると考えられる。

図 5.2 は 1 リクエストを処理し終えるまでにロードに費やされる時間の平均値を測定した結果である。

効率化の実装によってライブラリを親ローダにロードさせているため、効率化をしていない場合に比べてロード時間をかなり短縮できていることが分かる。クラスローダがキャッシュされていれば、クラスの再ロードの必要がなくなるので、キャッシュエントリ数が多ければ多いほど平均ロード時間が短くなっていることが分かる。キャッシュエントリ 100 の場合にはすべてのユーザのローダがキャッシュされており、さらにすべてのクラスはロード済みの状態なので、ロードにかかる時間はほとんどない。

図 5.3 は 1 リクエストの処理かかる時間の内訳の平均値を測定した結果である。これに関しては 1 スレッドによる測定を行った。複数スレッドだと各ブロックの正確な時間が測定できないためである。

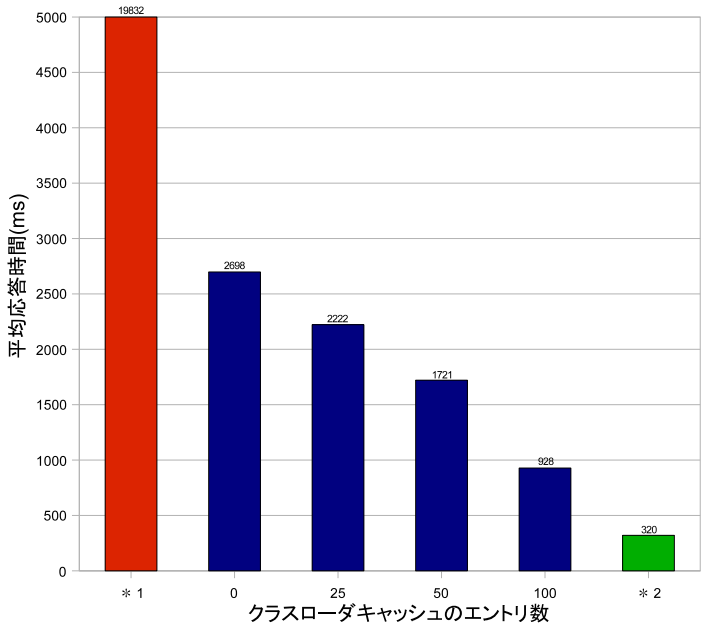

図 5.1: 平均応答時間 (実験 1)

ケース 1 の Web アプリケーションの実行時間が他と比べて非常に遅いのは、Java はアプリケーション実行中に必要となったクラスをロードするため、ロードに時間のかかるケース 1 のアプリケーションの実行に影響が出てしまっているためである。ケース 2 でキャッシュエントリ数を増やせば増やすほど、wrapper での処理時間が短くなっている。これは、クラスがロード済みであればインスタンスの作成も速いし、ローダがキャッシュされていれば、クラスローダの作成や初期化にかかる時間を短縮できるためである。

最後に、各ケース別のリクエスト送信開始から終了までのメモリ使用量を表 5.1 に示す。Java には GC(Garbage Collection) があるため、正確なメモリ使用量は測定できない。表はおおよその値を示したものである。

ケース 2、3 については 1000 リクエスト送り終えるまでに使用したメモリ量であるが、ケース 1 については 100 リクエスト送信後のものである。つまり、ケース 1 で 1000 リクエスト送信する場合は、さらにメモリが必要となる。この表からメモリ使用量についても大幅に改善できたことが分かる。

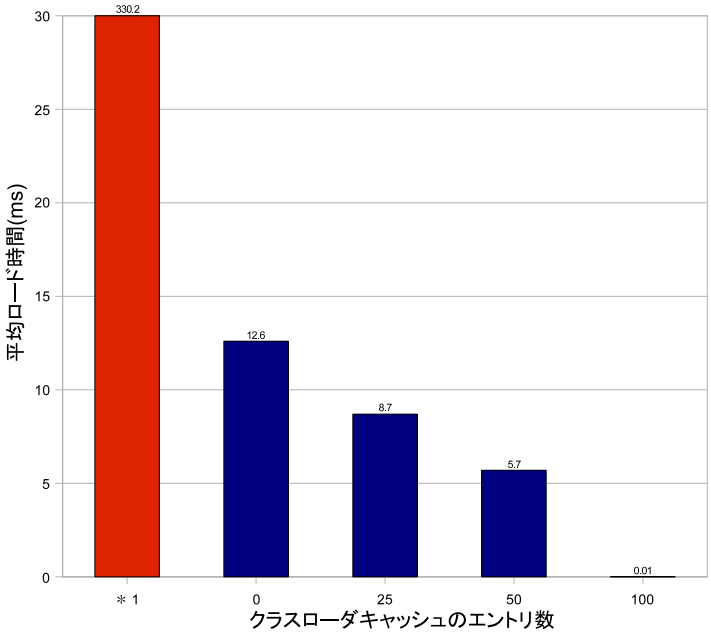

図 5.2: 平均ロード時間 (実験 1)

5.0.7　実験 2

この実験では、実験 1 とは対照的にライブラリを全く使用しないクラスに対してリクエストの送信を行った。実験内容は実験 1 と同様である。図 5.4 は平均応答時間、図 5.5 は平均ロード時間、図 5.6 はリクエスト処理の内訳を測定した結果である。

図 5.4 を見るとケース 1 のほうがケース 2 のキャッシュ 0 の場合よりもはやくなっている。これは、ライブラリを使用していないクラスであるにも関わらず、効率化を行ったフレームワークではローダ作成時に、親ローダの設定などのより多くの処理が必要であるためだと考えられる。しかし、キャッシュを使用すればケース 1 よりも速くなるので、さほど大きな問題ではない。

図 5.5 を見てわかるように、ライブラリを使用しなければ当然ケース 1 においても飛躍的ロードにかかる時間は短くなるので、ケース 2 との差は小さくなるが、キャッシュエントリ数を増やすことで、段階的にロード時間を短縮できていることが分かる。

図 5.6 を見てみると、Web アプリケーションの実行割合が全体的低くなっていることが分かる。wrapper での処理時間が段階的に小さくなって

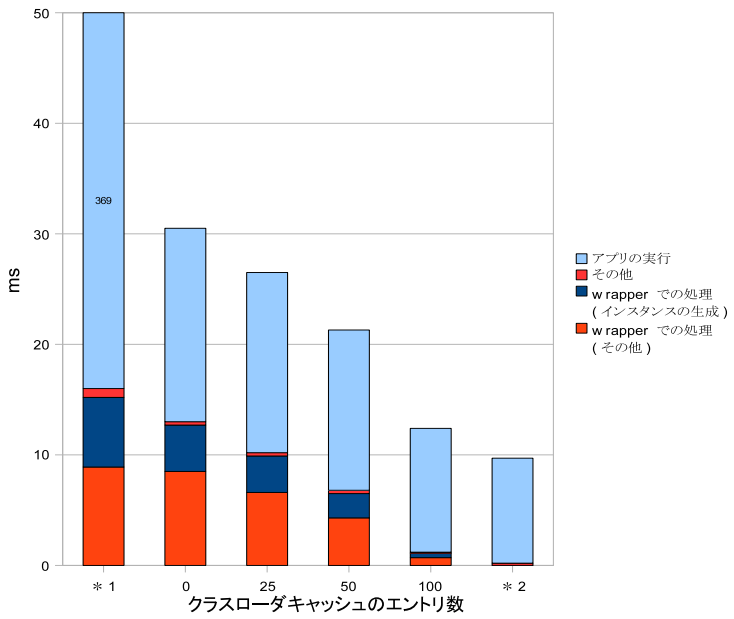

図 5.3: リクエスト処理の内訳 (実験 1)

いるが、これはクラスローダの作成時間や Servlet インスタンスの生成時間に起因していると考えられる。

表 5.2 はメモリ使用量の測定結果である。

この測定に関してはケース 1 についても、1000 リクエストの送信を行った。実験 1 程ではないが、メモリ使用量を削減できている。実験 1 よりも差が小さいのは、ロードされるクラスが少ないため、必然的に生成される Class オブジェクトが少なくためである。

表 5.1: ケース別メモリ使用量 (実験 1)

	MB
case 1	800
case 2 (cache 0)	300
case 2 (cache 25)	300
case 2 (cache 50)	200
case 2 (cache 100)	200
case 3	150

表 5.2: ケース別メモリ使用量 (実験 2)

	MB
case 1	200
case 2 (cache 0)	150
case 2 (cache 25)	80
case 2 (cache 50)	50
case 2 (cache 100)	20
case 3	5

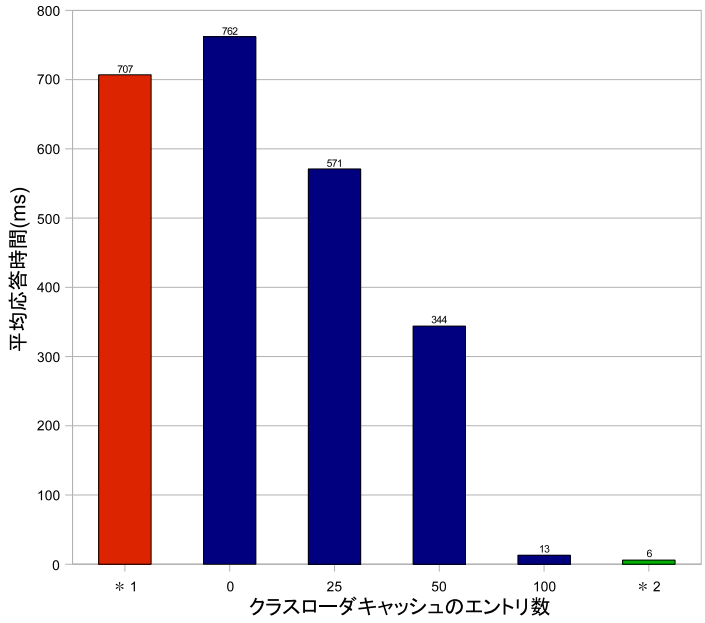

図 5.4: 平均応答時間 (実験 2)

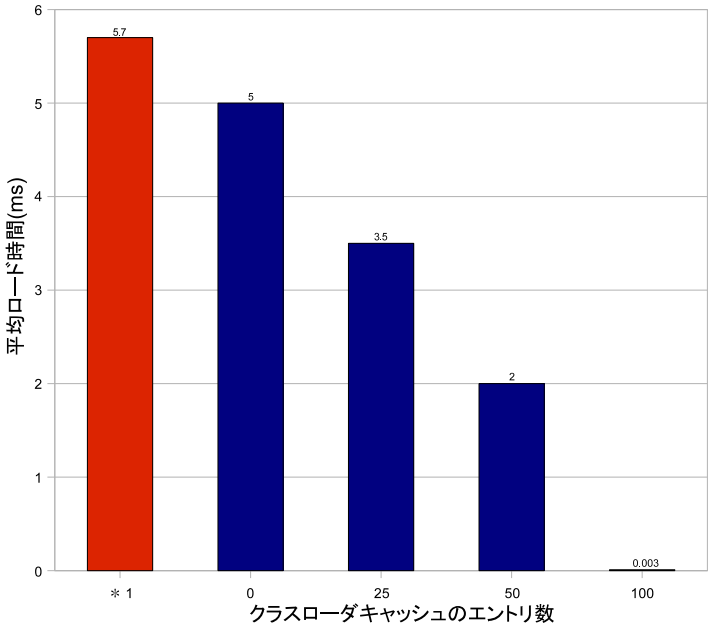

図 5.5: 平均ロード時間 (実験 2)

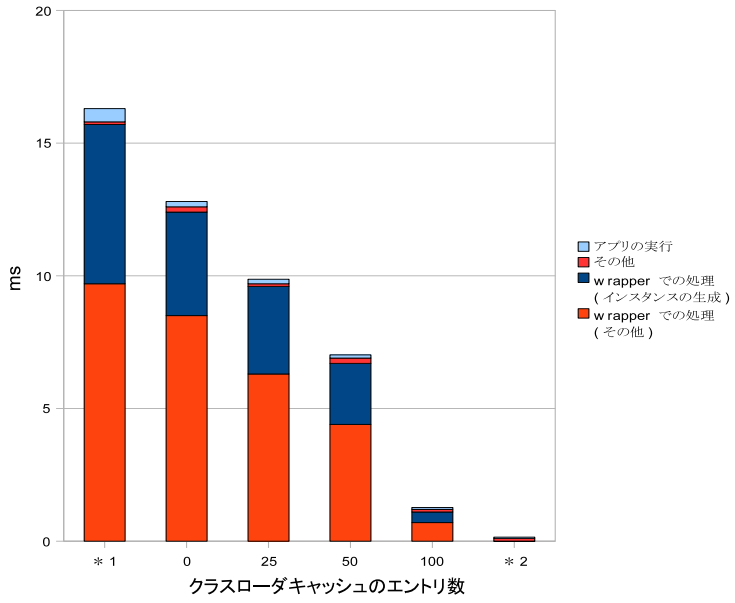

図 5.6: リクエスト処理の内訳 (実験 2)

第6章　まとめと今後の課題

6.1　まとめ

ユーザ毎にカスタマイズ可能な Web アプリケーションを実現するためにはプラグイン機構のような特殊な機構を Web アプリケーションに組み込む必要がある。しかし、設計段階でこのような機構を組み込むことは困難であり、開発後に拡張ポイントを追加、削除をするためには大幅な実装の変更が必要となる。

そこで本研究では、ユーザ毎にカスタマイズ可能な Web アプリケーション開発を支援するための per-session AOP を用いたフレームワークを提案した。

per-session AOP を用いることで、ユーザ毎に異なるアスペクトを Web アプリケーションに weave する事ができる。そして、per-session AOP を用いたフレームワークをサーバー上で動かすことで、ユーザがアスペクトをサーバーに送信し、それを Web アプリケーション内で本フレームワークが提供する API を用いて登録するだけで容易にユーザ毎にカスタマイズ可能な Web アプリケーションを実現することができた。

さらに本研究では、per-session AOP のパフォーマンスを向上させるために、その実装の効率化についても提案した。単純にユーザからのリクエストが到着する度にクラスローダを作成し、アスペクトを weave した Web アプリケーションをロードするという実装では、クラスのロードによるオーバーヘッドが大きすぎる。そのオーバーヘッドについては、クラスローダの親子関係とキャッシュを用いることで大幅に削減できた。さらに、この実装によりクラスの再ロードの回数を減らすことができ、結果的にメモリ使用量も削減することができた。

6.2　今後の課題

実験

5 章で、ライブラリを多用しているクラスと、ライブラリを使っていないクラスに分けて平均応答時間の測定を行った。その結果、キャッシュをエントリが 100 にも関わらず、フレームワークを使用しない場合に比べ

て 2 倍から 3 倍遅かった。その原因として、Tomcat の Servlet インスタンスのキャッシュをあげたが、その予測が本当に正しいかどうかの検証をする必要がある。そのためには、Tomcat の実装を調べ、キャッシュを実装から除外し、再度実験を行う必要がある。

セキュリティ対策

本フレームワークでは AOP を用いることでユーザが Web アプリケーションの任意の場所を自由にカスタマイズすることを可能にしている。

しかし、ユーザが自由にアスペクトを記述できるということは、サーバーをシャットダウンさせるコードや不正なファイルアクセスを行うコードがサーバー上で実行できてしまうことを意味する。このようなコードを実行させないためには、Web アプリケーションでアドバイスが実行されている間、何らかの方法で監視する必要がある。そこで、アドバイスの実行中のみ SecurityManager を切り替えるという方法を検討中である。独自の SecurityManager クラスを作成し、その SecurityManager はアスペクト専用のポリシーファイルを読み込み、アドバイスに特定のコードの実行権が与えられているかどうかをチェックする。

しかし、単純にアドバイスの前後に System クラスの setSecurityManager メソッドの呼び出しを挿入するだけでは問題がある。なぜなら、複数のスレッドが同時に実行されていた場合、あるスレッドでアドバイスの実行が始まると SecurityManager がセットされ、他スレッドに対しても、その SecurityManager が適用されてしまう。

上の問題点の解決策として、現在アドバイスを実行中のスレッドを管理し、SecurityManager クラスの checkPermission メソッドで処理を分けるようにすればよい。checkPermission メソッドとは SecurityManager クラスの現在実行されているコードがポリシーファイルで許可されているかどうかを確かめるメソッドである。checkPermission が呼ばれたら、まずそのスレッドの ID を調べ、そのスレッドがアドバイス実行中であるかを確かめる。もし、そのスレッドがアドバイス実行中であれば、独自の SecurityManager の checkPermission を呼び出し、そうでなければ、対比させておいた Web アプリケーションの SecurityManager の checkPermission を呼び出す。

参考文献

[1] The aspectj project. `http://www.eclipse.org/aspectj/`.

[2] Gluonj home page. `http://www.csg.is.titech.ac.jp/projects/gluonj/`.

[3] Sandip Bhattacharjee Vivek Chopra Chad Fowler Ben Galbraith Romin Irani Sing Li Chanoch Wiggers Amit Bakore, Debashish Bhattacharjee. *jakarta Tomcat* **エキスパートガイド**. ソフトバンクパブリッシング株式会社, 9 2003.

[4] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual machine support for dynamic join points. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 83–92, New York, NY, USA, 2004. ACM.

[5] Erik Hilsdale and Jim Hugunin. Advice weaving in aspectj. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35, New York, NY, USA, 2004. ACM.

[6] Sheng Liang and Gilad Bracha. Dynamic class loading in the java tm virtual machine. In *In Proc. 13th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98), volume 33, number 10 of ACM SIGPLAN Notices*, pages 36–44. ACM Press, 1998.

[7] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147, New York, NY, USA, 2002. ACM.

[8] 千葉 滋. Javassist home page. `http://www.csg.is.titech.ac.jp/~chiba/javassist/index.html`.

[9] 千葉 滋. **アスペクト指向入門　*-Java*・オブジェクト指向から *AspectJ* プログラミングへ**. 技術評論社, 11 2005.