情報処理学会研究報告
IPSJ SIG Technical Report

ユーザ毎にカスタマイズ可能な Web アプリケーション用のフレームワークの実装

別役浩平[†1]　千葉滋[†1]

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の機構として我々は per-session AOP を用いたフレームワークを提案した．しかし，Web アプリケーションのロードによるオーバヘッドが大きく実用性に乏しかった．そこで本稿では，このフレームワークの効率的な実装について提案する．クラスローダのアーキテクチャを工夫することで，フレームワークの実行時性能を向上させることができた．

Implementation of A Framework for User-Customizable Web Apps

Kohei Betchaku[†1] and Shigeru Chiba[†1]

We have proposed a framework that uses a technique called per-session AOP for user-customizable web applications. However, because of serious overheads due to loading classes of web efficient implementation technique for per-session AOP . Our experiments revealed that our new implementation technique improved performance of this framework. It exploits the architecture of a class-loader.

1. はじめに

近年，AOP（*Aspect Oriented Programming*）[1] 用いてアプリケーションをカスタマイズする技術について多くの研究がなされている．AOP は本来モジュール間の横断的関心事をアスペクトとして分離することで，ソフトウェアのモジュール化を行うための技術であるが，独立したモジュール（アスペクト）を追加することでアプリケーションをカスタマイズできるようにする技術としても用いることができる．アスペクトはポイントカットとアドバイスの組から成る．ポイントカットとはアプリケーションの特定の実行点を指定するためのもので，アドバイスとはポイントカットで指定した実行点にアプリケーションが到達したときに実行されるコードである．ポイントカットにはメソッド呼び出しやフィールドアクセスなどのアプリケーションの特定の実行点を自由に指定できるため，アプリケーションを細かな粒度でカスタマイズすることができる．事前にポイントカットで指定可能な実行点を決めておく必要などはない．

Web アプリケーションのような複数のユーザによって共有されるアプリケーションにおいては，ユーザ毎にアプリケーションがカスタマイズ可能であることが望ましい．しかし，既存の AOP を用いた機構でこのような機能を提供するものはなかった．そこで我々は 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ため，*per-session AOP* を用いたフレームワークを提案した[2]．Per-session AOP とは，Web アプリケーションのロード時にユーザ毎に異なるアスペクトを weave できるようにする技術である．このフレームワークは，ユーザからの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，そのユーザ用のアスペクトでカスタマイズされた Web アプリケーションを作成し，そのリクエストを処理させる．我々は，このような技術を用いて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した．しかし，ユーザのリクエストを受信する度に新しいクラスローダを作り直し，Web アプリケーションを再ロードするため，ロードによるオーバヘッドが大きく実用性に乏しかった．

そこで本稿では，per-session AOP の効率的な実装について提案する．ユーザ毎に再ロードされる Web アプリケーションには類似性があり，リクエストの度にすべてのクラスを再ロードする必要はない．再ロードが必要なのはアスペクトが weave されたクラスのみであり，その他のクラスはあらかじめ親クラスローダでロードさせておけばよい．しかし，アスペクトが weave されたクラスのみを再ロードするという実装ではクラスローダの仕様からうまく動作しない場合がある．これを回避するために，あらかじめ Web アプリケーションのクラス間の依存関係を調べ weave 対象のクラス名が分かれば，再ロードすべきクラス群が分かるようにした．提案する実装を用いたフレームワーク上の 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，提案する実装方法によってロードによるオーバーヘッドを大幅に削減できることを確認した．

以下，2 章では，我々が過去に提案したフレームワークとその問題点について述べる．3

†1 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
Dept. of Mathematical and Compting Sciences. Graduate School of Information Science and Engineering. Tokyo Institute of Technology

章では，新しく提案する per-session AOP の実装方法について述べる．4 章では，我々が行った実験について伸べる．5 章で関連研究に触れ，6 章で本論文をまとめて，今後の課題を述べる．

2. Per-session AOP を用いたフレームワーク

本章では，まず我々のが提案した per-session AOP を用いたフレームワークの有用性について述べ，次にこれまでの実装方法における問題点について述べる．

2.1 Per-session AOP を用いたフレームワークの有用性

1 章で述べたように，per-session AOP ではアスペクトを Web アプリケーションのカスタマイズのために用いる．アスペクトを用いることで，アプリケーションの拡張者は Web アプリケーションを細かな粒度でカスタマイズできる．また，Web アプリケーション開発者は，後述のプラグイン機構のようなカスタマイズ機構を使う場合と異なり，カスタマイズ機構を強く意識した実装を行う必要がい．Web アプリケーション開発者が留意すべき点は，アスペクトの開発を容易にするようにできるだけ Web アプリケーションを細かなモジュールに分割することだけである．

Per-session AOP はユーザ毎に異なるアスペクトを weave 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．論文[2] では，per-session AOP を用いた Web アプリケーション・フレームワークの実装法を示した．この実装法では，次に示す手順で per-session AOP を実現する．

(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ユーザ ID を取得する．
(2) ユーザ毎に異なるクラスローダを作成する．
(3) ユーザ ID からユーザのアスペクトを特定し，作成したクラスローダで，そのアスペクトが weave された Web アプリケーションをロードする．

このフレームワークは，tomcat[3] サーバの一部として実装されており，tomcat サーバのコンポーネントである valve と wrapper に独自のものを用いている．valve とはリクエストをインターセプトし，事前処理をするコンポーネントである．サーバ内では一連の valve がパイプラインを構成しており、invoke メソッド内でその処理を行う (図 1)． 本フレームワークが用意する独自の valve は，invoke メソッドの引数として渡されるリクエストからセッションを取得するために用いられる．

Wrapper とは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．Wrapper のインスタンスは，サーブレットのクラス毎に作られ，対応するサーブレットのインスタンス作成を行う allocate メソッドを備える．我々のフレームワークの wrapper がもつ allocate メソッド は，ユーザ毎に異なるクラスローダを作成し，valve で取得したセッションからユーザに適用すべきアスペクトを特定する．そして，そのアスペクトを Web アプリケーションに対して weave し，ロード及びインスタンスの作成を行う．JVM は親子関係を持たない複数のクラスローダがロードしたクラスを名前空間が異なるものとみなす．Per-session AOP ではリクエストを受信する度にユーザ毎に異なるクラスローダを作成することで，同一名のユーザ毎に異なる挙動をする Web アプリケーションのロードを可能にしている．このクラスローダについての詳細は 3.1.1 で述べる．

```
1 public void invoke(Request req, Response res){
2   //この valve における処理
3   getNext().invoke(req, res); //次の valve の呼び出し
4 }
```

図 1 valve の invoke メソッド

我々が作成したフレームワークでは AOP システムとして GluonJ[4] を使用している．GluonJ はは我々が開発した Java 言語用の AOP システムである．図 2 に GluonJ のアスペクトの例を示す． このアスペクトは Webapp クラスの table メソッドを書き換える．

```
1 @Glue class Customizer {
2   @Around(“{return (html コードの作成)}”)
3   Pointcut pc = Pcd.call(“Webapp#table(..)”);}
```

図 2 GluonJ によるカスタマイズ例

2.2 既存のカスタマイズ機構

Java 言語で実装された Web アプリケーションをカスタマイズ可能にする手法は既にいくつか知られているが，我々が提案する per-session AOP と異なり，これらの手法にはいくつかの問題点がある．

2.2.1 プラグイン機構

Java 言語で実装されたソフトウェアをカスタマイズ可能にする手法として，よく用いられるのがプラグイン機構である．例えば統合開発環境である Eclipse のプラグイン機構が

有名である．この手法では，ソフトウェア設計者が設計段階で，カスタマイズをおこなうための拡張ポイントを定めなければならない．ソフトウェアの実装者は，この拡張ポイントごとにインタフェースを定め，各拡張ポイントに組み込まれているプラグインをそのインタフェースを介して呼び出すようにプログラムを書かなければならない．プラグインは，そのインタフェースを実装した Java オブジェクトとして実装されることになる．ソフトウェアの拡張者は，新しくプラグインを作成して目的の拡張ポイントに組み込むことで，ソフトウェアのカスタマイズを実現する．

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

図 3 プラグイン機構における拡張ポイントの例

図 3 に拡張ポイントの実装例を示す．この例では，web ページのレイアウトを拡張ポイントとしている．ソフトウェアの設計者は PageLayout インタフェースを定め，プログラムの中でレイアウトを決めている箇所では，プラグインを取り出し，このインタフェースを介して呼び出すように実装者がプログラムを書かなければならない．例では，getLayout メソッドが，組み込まれているプラグインを探して取り出すメソッドである．引数の uid は現在処理中のリクエストを送信したユーザの識別子である．プラグインから返された戻り値に従って，web ページのレイアウトを決めていることがわかる．

このようなプラグイン機構は，per-session AOP と比べて，カスタマイズ性についていくつか問題点がある．まず設計段階で決めた拡張ポイントを介したカスタマイズしかできないので，後から新しい種類のカスタマイズが必要になったとき，対応する拡張ポイントがなく，そのカスタマイズを実現できないことがある．設計段階で将来必要になるであろう拡張ポイントを全て抽出する必要があるが，現実的には非常に難しい．また全ての拡張ポイント各々について，プラグインが組み込まれているときは，図 3 の setLayout のように，それを明示的に呼び出すようにプログラムを書かなければならない．拡張ポイントの数が増えると，ソフトウェアの開発者の実装の負担が大きくなる．

これに対し AOP を利用すると，その obliviousness[5] により，事前に拡張ポイントを決めたり，そのインタフェースを定める必要がない．ソフトウェアの拡張者は，プログラム中の任意の実行点をポイントカットで指定し，カスタマイズ・コードをアドバイスとして組み込む (weave する) ことができる．したがって，開発者は通常のソフトウェア開発同様，後からカスタマイズしやすいように適切にモジュール分割することだけに留意すればよい．将来のカスタマイズに明示的に配慮したプログラムを書く必要はない．

2.2.2 DI コンテナ

カスタマイズ性を高めるために DI コンテナを用いる手法も一般的である．DI コンテナとは，DI (*Dependency Injection*) 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 4 に例を示す．

```
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 4 DI コンテナの利用例

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．プラグイン機構を用いた図 3 との違いは，setLayout メソッドの中で呼ばれる p の値を明示的に初期化

しなくてよい点である．

DI コンテナも，ある種のプラグイン機構であり，通常のプラグイン機構と同様の問題点をもつ．DI コンテナを用いた場合も，拡張ポイントとそのインタフェースは設計時に決めておかなければならず，拡張ポイントでない部分のカスタマイズを行うことはできない．またソフトウェア実装者の負担も小さくない．

DI コンテナによっては，AOP 機能として，指定された任意のクラスのインスタンスに対してインターセプタを設定でき，インスタンスの振る舞いを変更できるものもある．しかしながら，このような AOP 機能は限定的で，新しいメソッドの追加などはおこなえないため，カスタマイズに用いるには不十分である．また，我々が知る限り，per-session AOP のようにユーザ毎に異なるアスペクトを weave できるようにしたものは存在しない．

2.3 Per-session AOP 用いたフレームワークの問題点

Per-session AOP は柔軟なカスタマイズを容易に可能にするが，実行時性能に問題がある．ユーザ毎のカスタマイズの実現するために，ユーザからリクエストが来る度にクラスローダを作り直しているからである．もし，同一のクラスローダを用いてこれを実現しようとすると，ユーザ毎に異なるアスペクトを同一のクラスに weave しなければならないため，実装の異なる同一名のクラスを一つのクラスローダでロードすることになる．クラスローダの仕様から，これは不可能である．

しかし，リクエストの度にクラスローダを単に作り直すという実装では，Web アプリケーションの全クラスをその都度再ロードすることになるためオーバヘッドが大きい．我々のフレームワークを用いた掲示板アプリケーションに対して 100 スレッドで，100 ユーザが 10 リクエスト，計 1000 リクエストを並列に送信する実験を行った際の平均応答時間を図 5 示す．グラフからもわかるようにフレームワークを用いた場合の平均応答時間は用いない場合に比べて約 60 倍である．これでは頻繁にリクエストが来る Web サーバにおいて大きな負担となるため，実用的でない．

3. Per-session AOP の効率的実装の提案

2.3 で述べたように per-session AOP には実行時性能の面で問題がある．この問題点を解決するために，本章では per-session AOP の効率的実装について提案する．

3.1 Web アプリケーションの類似部分のローダの共通化

Per-session AOP の以前の実装では，ユーザからのリクエストが来る度にクラスローダを作り直し，Web アプリケーションの全クラスを再ロードしていた．しかし，実際に再ロードしなければならないのは，ユーザが選択したアスペクトが weave されて変更されたクラスのみである．そこで，ユーザ毎のクラスローダはアスペクトが weave されたクラスのみをロードし，その他のクラスは全ユーザ共通の親クラスローダにあらかじめロードさせておくことで実行時のオーバーヘッドを削減できる．

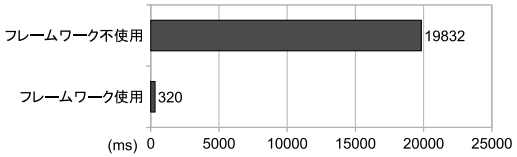

図 5 フレームワークの使用・不使用時の Web アプリケーションの平均応答時間の比較

まず，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る変更を不可としても実用上問題ないので，ユーザ毎のクラスローダに再ロードさせる必要はなく，あらかじめ共通の親クラスローダにロードさせるべきである．ライブラリから Web アプリケーションのその他のクラスへの参照はないので，後述する依存性の問題もなく，容易に共通の親ローダにロードさせることができる．

一方，その他のクラスは互いに依存し合っているため，アスペクトが weave されたクラスのみをユーザ毎のクラスローダでロードし，その他のクラスを親クラスローダにロードさせるという簡単な実装では上手く動作しない場合がある．クラスローダのアーキテクチャからクラス間の依存関係を解決できないからである．そのため，クラスローダのアーキテクチャを考慮した実装が必要となる．

3.1.1 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[6]．クラスのロードにはクラスローダが用いられ，java.lang.Class のインスタンスを生成する．全てのクラスローダは ClassLoader クラスのサブクラスである（図 6）．

クラスローダは親子関係を持ち，loadClass メソッドを用いて次のような手順でクラスのロードを行う．

(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，親クラスローダの loadClass メソッドを呼び出す
(3) 親クラスローダがロードできなかった場合，findClass メソッドを呼び出して，この

```
1 class ClassLoader{
2   public Class loadClass(String name);
3   protected Class findClass(String name);
4   protected final Class defineClass(String name, byte[] buf, int off, int len);
5   protected final Class findLoadedClass(String name);
6   ...
7 }
```

図 6 ClassLoader クラス

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

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 上のロードすることができる．Per-session AOP では，この性質を利用することでユーザ毎に異なるアスペクトを weave した同一名の異なる定義のクラスを同時にロードすることを可能にしている．

次に，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以下の例を用いて示す．

```
class A{
  void func(){
    String s; B b = new B();..
  }
  public static void main(String[] args){
    new A().func();
}}
```

まず，main メソッドが定義されているクラスをアプリケーション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_A とすると，JVM 上にはクラスの型 <A,L_A > が定義される．次に main メソッドが呼び出され，A の func メソッドが呼ばれる．func メソッド内では String クラスと B クラスを参照している．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_A）によって解決される．つまり，func メソッドの実行中に

L_A.loadClass(“String”);

が呼び出される．アプリケーションクラスローダは String クラスなどのシステムクラスをロードするシステムクラスローダを親に持つ．これを L_S とする．String クラスの参照は，先に述べたロードの手順から L_A の loadClass メソッドから呼ばれた L_S の loadClass で解決され，JVM 上には型 <String,L_S > が定義される．一方，システムクラスでない B クラスはシステムクラスローダがロードできないため，L_A 自身が実行する．つまり，L_A の findClass が呼び出され，JVM 上には型 <B,L_A > が定義される．

L_A.loadClass(“B”);

3.1.2 複数のクラスローダによるロードするクラス群の分割

Web アプリケーションの再ロードによるオーバヘッドを削減するために，アスペクトの weave 対象クラスのみを子クラスローダで再ロードするというシンプルな実装では上手く動作しない場合がある．weave 対象クラスだけでなく，そのクラスに依存しているクラスすべてを子クラスローダでロードしなければならない．ここで，クラス X に依存しているクラスとは，クラス X を直接または間接的に参照しているクラスである．また，参照しているクラス Y のロード済みの定義を発見することを，以下ではクラス Y への参照を解決するという．

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{..}
```

B クラスにユーザアスペクトが weave されるとし，単純に A クラス，C クラスを親クラスローダ L_P に，B クラスを子クラスローダ L_C にロードさせるとする．A クラスの 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 の親クラスローダ及び自身で解決可能なクラスなので，子クラスローダでロードさせた B クラスへの参照は解決できず，ClassNotFoundException が投げられる．仮に，L_C だけでなく L_P も B クラスを解決可能であるとすると，funcA の実行中に L_P によってロードされ，A クラスで使用される B クラスは <B,L_P > 型であり，L_C によってロードされ，アスペクトが weave された

<B,L_C > 型とは異なる．これでは，アスペクトによる Web アプリケーションのカスタマイズが実現できない．

そこで，A クラスも子クラスローダにロードさせなければならない．そうすれば A クラス内の B クラスへの参照は L_C 自身で解決できる．一方，B クラスが参照するクラスは L_P にロードさせてよい．なぜなら，B クラス内の参照は L_C が自身の loadClass メソッドを呼び出すことによって解決しようとするが，3.1.1 で述べたロードの手順から L_C の loadClass メソッドを実行する過程で L_P の loadClass メソッドを呼び出すため，L_P によって解決可能であるからである．

上で述べたクラス群の分割を実現するために，親クラスローダには Web アプリケーションの全クラスをあらかじめロードさせ，子クラスローダには図 7 のように loadClass メソッドを変更したものを用いる． 2 行目の isLoadTarget メソッドは className が weave 対象

```
1 public Class loadClass(String className){
2   if(isLoadTarget(className)){
3     Class clazz = findLoadedClass(className);
4     if(clazz == null) return findClass(className);
5     else return clazz;
6   }
7   else return getParent().loadClass(className);
8 }
```

図 7 子クラスローダの loadClass メソッド

クラス及びそれに依存しているクラスであれは true を返すメソッドである．図 7 からわかるように，あるクラスのロード要求があったとき，weave 対象クラス及びそれに依存しているクラスならば親クラスローダの loadClass メソッドを呼び出さず自身で探索を行い，それ以外のクラスならば親クラスローダの loadClass メソッドを呼び出すようにしている．こうすることで，weave 対象クラス及びそれに依存しているクラスは子クラスローダでロードされ，それ以外のクラスは親クラスローダでロードされることが保証される．これは，3.1.1 で示した通常のクラスローダによる手順と異なることに注意されたい．

3.1.3 異なるクラスローダでロードすべきクラス群の発見法

ユーザが選択したアスペクトを解析し，weave 対象クラスが判明した際，即座に子クラスローダでロードさせるべきクラスを決定する必要がある．さもなければ，クラスローダの初期化時にクラス間の依存関係を調べることになり，時間的コストが大きくなってしまう．そこで，すべてのクラスについて，そのクラスが依存しているクラス群をあらかじめ調べておく必要がある．

まず最初に，クラスの依存関係を表す有向グラフを作る．これは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名を取得することで実現できる．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，クラス内のすべてのメソッド呼び出しを調べ，その戻り値の型が java.lang.Class クラスであれば，そのクラスは Web アプリケーションのその他のクラスすべてに依存しているものとした．また，メソッド呼び出しによって java.lang.Class クラスのオブジェクトが Object クラスにキャストされて返される場合も考えられるので，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．また，クラス間の依存関係にサイクルがあれば，そのサイクル全体を一つのノードとみなした．これはグラフの探索速度を上げるためである．

こうしてできた有向グラフ $G = (V, E)$ 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，

- $S_u \neq \phi$ 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られた 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．

3.2 クラスローダのキャッシュ

Per-session AOP の以前の実装では同一のユーザからのリクエストに対しても毎回クラスローダを作り直している．weave するアスペクトを変更していないユーザのクラスローダは作り直す必要はなく，以前に作ったそのユーザのクラスローダを流用すれば再ロードのによるオーバヘッドを削減することができる．しかし，全ユーザのクラスローダを保持しておくのは，多数のユーザが利用する Web アプリケーションにおいては現実的でないので，クラスローダ用のキャッシュをフレームワークに組み込んだ．キャッシュのエントリ数はサーバ上の設定ファイルで決定できる．

4. 実　　験

本稿で提案した per-session AOP の効率的実装によって，どの程度実行時性能向上したかを検証した．Per-session AOP を組み込んだ Web アプリケーションフレームワークを作成し，その上で掲示板アプリケーションを動かした．なお，実験環境はサーバー側は CPU Xeon 2.83 GHz，メモリ 4GB，OS Linux 2.6.26 で，クライアント側は CPU Core2 Duo 3.00 GHz，メモリ 4GB，OS Windows Server 2003 である．Web サーバは tomcat 5.5，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 jakarta jmeter 2.3.2[7] を用いた．また，AOP システムとして GluonJ を用いた．

掲示板アプリケーションの中で，ライブラリ及び Web アプリケーションの他のクラスを参照するクラス（BBS クラス）と，他のクラスを参照しないクラス（FrontPage クラス）に対して 100 ユーザが 100 スレッドで，各ユーザが 10 リクエスト，計 1000 リクエストを並列に送信する実験を行った．

4.1 BBS クラス

他のクラスを参照するクラス（BBS クラス）にアスペクトを用いてプリント文を weave し，掲示板にリクエストを送信する実験を行った．実験結果を図 8 に示す．

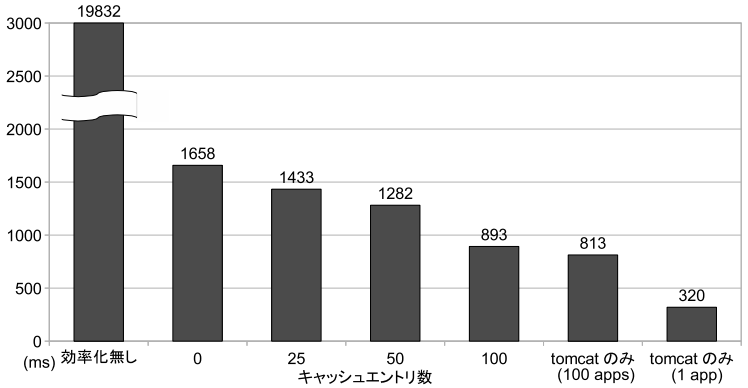

図 8　BBS クラスへリクエストを送信した時の平均応答時間

縦軸は平均応答時間（ms），横軸の効率化無しという項目は以前の実装方法に基づいたフレームワークを用いた場合，キャッシュエントリ数という項目は効率的実装をしたフレームワークをキャッシュエントリ数別にわけて実験を行った結果である．tomcat のみ（100 apps）は，我々のフレームワークを用いずに同じ掲示板アプリケーションを 100 個別々のプログラムとして作成し，それぞれに 100 ユーザが 10 リクエスト，計 1000 リクエストを並列に送信した結果である．tomcat のみ（1 app）は 1 個の掲示板アプリケーションに対して同様のリクエストを送信した結果である．tomcat のみの項目を 100 apps と 1 app と分けているのは，キャッシュエントリ数 100 の場合と条件をそろえるためである．キャッシュエントリ数 100 の場合，リクエストの送信中常に 100 個のクラスローダインスタンスが存在することになる．このメモリ圧迫によりサーバの処理が遅くなると考え，これを検証するために 100 個の掲示板アプリケーションを作成し実験を行った．tomcat では一つの Web アプリケーション毎にクラスローダを作成するため，tomcat のみ（100 apps）ではクラスローダが 100 個存在することになる．これは，100 人のユーザが AOP を使わずに，それぞれ直接カスタマイズした掲示板アプリケーションを，通常の tomcat サーバ上で動かす状態に対応する．

グラフからわかるように以前の実装法に比べて，今回提案する方法では 11 〜22 倍の速度向上が見られた．また，キャッシュエントリ数 100 の合でも tomcat のみ（1 app）よりも 3 倍程度遅いのはクラスローダインスタンスによるメモリ圧迫が原因であるとわかった．

次に，クラス間の依存関係の探索による効果を検証するために，クラスの依存関係の探索を行わず，ライブラリは親クラスローダに，その他のクラスは全て子クラスローダにロードさせるフレームワークを作成し，今回提案する実装方法を施したフレームワークと比較する実験を行った．図 9 の左側はクラスの依存関係の探索を行わないフレームワーク，右側は提案する実装法を施したフレームワークである． BBS クラスは ResDate クラス，DBAccesser クラス，BBSLogger クラスを参照している．このため，表 1 からわかるようにクラス間の依存関係の探索をしない場合，計 12600 byte のクラスファイルを再ロードすることになるが，探索をした場合は再ロードすべきクラスは BBS クラスのみになるため 5627 byte である．これは，この実験では BBS クラスにリクエストを送信するため，アプリケーションの実行が BBS クラスから始まるためである．もし，リクエストの処理が BBS クラスに依存しているクラスから始まる場合，3.1.2 で述べたように，BBS クラス及びそれに依存しているクラスも再ロードする必要がある．

図 9 見ると依存関係の探索を行ったほうがキャッシュエントリ数が 0 〜50 の場合，1300 ms 〜 500 ms 速い．キャッシュエントリ数 0 の場合，リクエストの度に再ロードを行うため，純

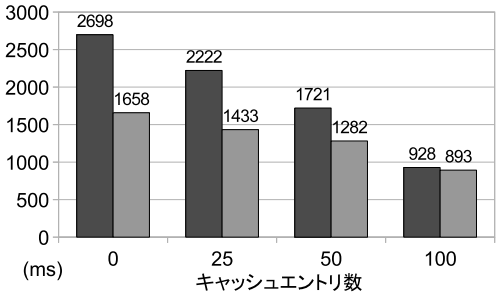

図 9 依存関係の探索を行わないフレームワークと効率的な実装方法を施したフレームワークのの平均応答時間の比較 (BBS クラス)

クラス名	フィールド	メンバ	サイズ (byte)
BBS	5	5	5627
DBAccesser	3	8	5285
FrontPage	1	4	4322
ResDate	1	1	1088
BBSLogger	0	1	600

表 1 クラスファイルのサイズ (抜粋)

粋に再ロード一回当たりの比較をすることができる．つまり，ResDate クラス，DBAccesser クラス，BBSLogger クラスを合わせた 6973 byte の再ロードの削減により，平均応答時間が 1300 ms 短縮できたことが分かる．

4.2 FrontPage クラス

次に他のクラスを参照しない FrontPage クラスにリクエストを送信する実験を行った．実験結果を図 10 に示す．他のクラスを参照しない場合，クラスローダのキャッシュの効果しかないため，さほど大きな効果は見られなかった．

4.1 と同様にクラスの依存関係の探索を行わないフレームワークと提案する実装法を施したフレームワークを比較する実験を行った．実験結果を図 11 に示す． 参照しているクラスがない場合，図 10 の効率化無しの時とキャッシュエントリ数 0 の時，図 11 のそれぞれのフレームワークのキャッシュエントリ数 0 〜50 の時の平均応答時間はそれぞれ同じになるはずである．しかし，提案する実装法を施したフレームワークのキャッシュエントリ数 0 〜50 の時の方が速くなっている．これは，提案する実装法を施したフレームワークで使用されるクラスローダは，再ロードの対象クラスのロード要求があった際，まず自身でそのクラスを探索する仕様になっているが，以前の実装法及びクラスの依存関係の探索を行わないフレームワークで用いられるクラスローダは，まず親クラスローダに委譲し，見つからなかった場合にのみ自身で検索するという通常のクラスローダの検索アルゴリズムに則っているため，その分遅延があると考えられる．

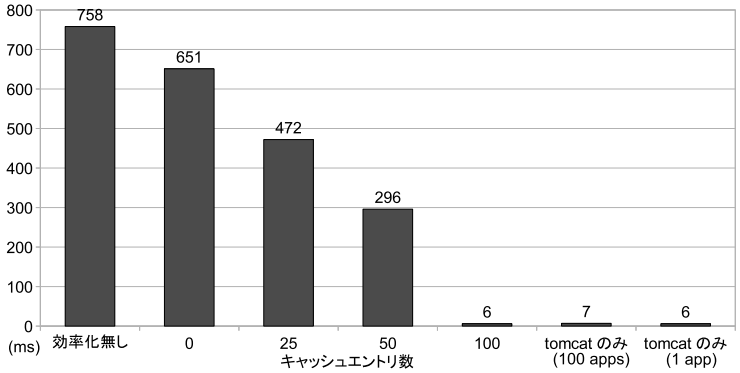

図 10 FrontPage クラスへリクエストを送信した時の平均応答時間

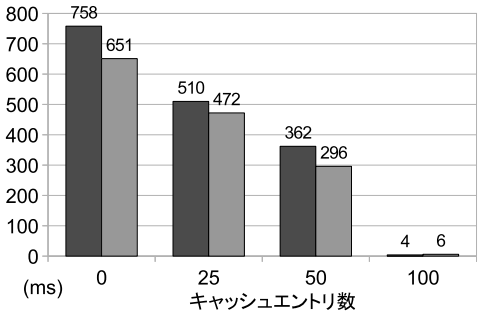

図 11 依存関係の探索を行わないフレームワークと提案する実装方法を施したフレームワークのの平均応答時間の比較 (FrontPage クラス)

5. 関 連 研 究

AOP を用いたカスタマイズのための機構として，DAOP（*Dynamic Aspect Oriented*

Programming）システムがしばしば使われてきた．DAOP はアプリケーションの実行中にアスペクトを weave するプログラミング技法である．例えば，PROSE[8] はイベント通知を用いた DAOP 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．しかし，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あまり実用的でない．この問題を改善したシステムも提案されてきたが[9]，記述可能なアスペクトに制限があり，カスタマイズには不向きである．

AOP を用いて実装された Web アプリケーションに Health Watcher[10] がある．Greenwood らは，OOP（*Object Oriented Programming*）を用いて実装したバージョンと AOP を用いて実装したバージョンの両方を作成し，それらを開発の時系列に沿って比較，実験することでアプリケーション開発において AOP が有用であることを述べている．

今回我々はクラスローダのアーキテクチャを工夫することで Web アプリケーションの実行時性能を向上させた．Web アプリケーションの実行時性能の研究は数多い．QoSWeaver[11] は，アスペクトを用いて Web アプリケーションに直接スケジューリングコードを weave することで，実行時性能を向上させている．杉木ら[12] は，クライアントからのリクエストの待機間隔を監視しながら，データ送受信の頻度に合わせて keep-alive 時間を動的に設定する手法について提案している．

6. まとめと今後の課題

Per-session AOP を用いたフレームワークの効率化のための実装について提案した．以前に提案した実装方法では，ユーザからのリクエストの度にクラスローダを作り直し，Web アプリケーションを再ロードするため，オーバヘッドが大きく実用性に問題があった．本稿では，クラスローダのアーキテクチャを利用した Web アプリケーションのクラスの類似部分の共通化を行い，実行時性能を向上させた．

今後の課題は次の通りである．

- 詳細な実験
- セキュリティ対策

クラスの依存関係は，Web アプリケーションによって様々である．また，アスペクトを weave するクラスを変えることで子クラスローダにロードさせるクラスの数も変わるため weave 対象のクラスを変えて実験する必要がある．現在の実装ではセキュリティ対策がないため，サーバをダウンさせたり，不正にファイルを読み込むなど，悪意のあるアスペクトが Web アプリケーションに weave されてしまう．これについては，Java 言語のセキュリティマネージャーを利用する，weave される前にアスペクトのバイトコードを走査するなどの方法を検討中である．

参 考 文 献

1) Kiczales, G., Irwin, J., Lamping, J., Loingtier, J.-M., Lopes, C.V., Maeda, C. and Mendhekar, A.: Aspect-Oriented Programming, *ECOOP*, LNCS 1241, pp.220–242 (1997).
2) 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).
3) : Apache Tomcat. http://tomcat.apache.org/.
4) : GluonJ Home Page. http://www.csg.is.titech.ac.jp/projects/gluonj/.
5) Filman, R.E. and Friedman, D.P.: Aspect-Oriented Programming is Quantification and Obliviousness, *Aspect-Oriented Software Development* (Filman, R.E., Elrad, T., Clarke, S. and Aksit, M., eds.), Addison-Wesley, pp.21–35 (2005).
6) 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).
7) : Apache JMeter. http://jakarta.apache.org/jmeter/.
8) Popovici, A., Gross, T. and Alonso, G.: Dynamic weaving for aspect-oriented programming, *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, New York, NY, USA, ACM, pp.141–147 (2002).
9) Suvée, D., Vanderperren, W. and Jonckers, V.: JAsCo: an aspect-oriented approach tailored for component based software development, *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, New York, NY, USA, ACM, pp.21–29 (2003).
10) 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, *ECOOP*, pp. 176–200 (2007).
11) Kourai, K., Hibino, H. and Chiba, S.: Aspect-oriented application-level scheduling for J2EE servers, *AOSD '07: Proceedings of the 6st international conference on Aspect-oriented software development*, New York, NY, USA, ACM, pp.1–13 (2007).
12) 杉木章義，河野健二，岩崎英哉：リクエスト待機間隔を考慮したウェブサーバの keep-alive 時間の自動設定，コンピュータソフトウェア， Vol.24, No.2, pp.68–78 (20070424).