Вестник Сыктывкарского университета.
Сер.1. Вып.12.2010

УДК 004.051

Повышение производительности веб-приложения

Е.А. Ключников

Рассматриваются проблемы производительности веб-приложений и способы их решения. Результатом исследований является создание производительной библиотеки шаблонов на основе динамичекой кодогенерации.

1. Введение

На текущий момент наиболее востребованным вариантом взаимодействия приложения с пользователем является вариант «веб-клиент»[1]. Существует несколько специализированных платформ для создания веб-приложений, таких как ASP, PHP, которые позволяют создавать относительно несложные приложения, однако для создания сложного приложения уровня предприятия оптимальным вариантом является использование полноценных платформ. На данный момент наиболее распространёнными и развитыми являются платформы «Sun Java» и «Microsoft .Net».

Веб-приложение, построенное на базе платформы J2EE, обладает, кроме прочего, таким качеством, как «горизонтальная масштабируемость»[2], что позволяет приложению обрабатывать очень большое количество запросов. Несмотря на это «время отклика»[3] может быть довольно значительным (некомфортным) в случае, если один или несколько

[1] «Веб-клиент» — взаимодействие посредством интернет браузера с сервером веб-приложения, см. `http://en.wikipedia.org/wiki/Web_application`

[2] «Горизонтальная масштабируемость» — horizontal scalability — характеристика системы, обозначающая линейную зависимость производительности системы от количества системных ресурсов, см. `http://en.wikipedia.org/wiki/Scalability`

[3] «Время отклика» — response time — период ожидания пользователем ответа системы на предпринятые действия (запросы)

уровней системы имеют неправильную или неэффективную реализацию.

Для построения представления в веб-приложениях, написанных на Java, чаще всего используется технология «Java Server Pages» (JSP). Благодаря JSP можно разделить непосредственно само представление и данные. Таким образом, можно решать вопросы, связанные с дизайном, без привлечения разработчиков, что заметно повышает производительность команды.

Хорошо структурированное приложение подразумевает использование тэгов — параметризуемых частей представления. Тэги могут быть заданы как JSP-файлом, так и Java-классом. Тэги могут быть объединены в библиотеку и, в дальнейшем, использованы в другом приложении. Сообществом разработчиков была разработана стандартная библиотека тэгов «JSP Standard Tag Library» (JSTL), включающая в себя тэги для общих нужд, таких как разбор XML данных, условная обработка, создание циклов и поддержка интернационализации.

Использование JSTL заметно сокращает цикл разработки. Однако оборотной стороной универсальности и простоты является уменьшение производительности. Иногда скорость генерации падает настолько заметно, что обработка большого количества запросов с использованием разумного количества оборудования становится невозможной.

В данной статье рассматриваются некоторые способы повышения производительности. На практике, эти способы позволили уменьшить время генерации представления в 400 раз.

2. Производительность XML тэгов JSTL

В рассматриваемом приложении пользовательское представление генерируется на основе двух XML. Первая XML описывает структуру данных и неизменные части представления (названия полей), вторая XML содержит актуальные данные пользователя. При таком подходе возможно создание нескольких независимых клиентов (не обязательно на платформе Java), отображающих актуальный набор данных. Веб-приложение может быть написано однократно, а использовано для представления различных моделей данных или моделей данных, изменяющихся во времени.

Приложение работало следующим образом:

1. XML в виде строки запрашивается у сервера;

2. строка при помощи разборщика, встроенного в платформу Java,

превращатеся в дерево объектов DOM[4];

3. два объекта DOM передаются JSP странице;

4. представление генерируется на основании просмотра объектов DOM.

Для обхода узлов XML использовались вложенные конструкции следующего вида:

```
01| <xml:forEach select="$schemaDom/scope/zone" var="zone" >
02|   <xml:set select="string($zone/@id)" var="active" />
03|   <xml:set select="$data/scope/zone[@id=$active]" var="dtZone" />
04|   ...
```

Первая строка задаёт цикл по выборке объектов из XML, определяющей структуру данных. Вторая строка выбирает идентификатор текущего объекта. Третья строка выбирает объект, содержащий актуальные данные, соответствующие текущему объекту структуры. Во всех трёх случаях используется язык XPath[5] для задания выборки данных.

Первые испытания данной реализации на реальных данных показали, что производительность зависит от количества отображаемых объектов. Одно из полей модели данных содержит около 600 вариантов выбора. Для решения этой проблемы был использован технический приём — кэширование[6]. Тэг, представленный JSP-файлом, отвечающий за генерацию представления поля, был трансформирован в Java-класс, сохраняющий результаты в памяти для последующего их мгновенного возврата.

Время отклика приложения уменьшилось в два раза. Однако такой подход обладает и существенным минусом — становится невозможным изменение представления дизайнером.

При исследовании причин низкого быстродействия выяснился тот факт, что XML тэги JSTL каждый раз при своём выполнении запускают XPath-запрос в «сыром» строковом виде. Это общепризнанная «плохая практика» («bad practice»). Дело в том, что в случае запуска строкового запроса XPath, большинство реализаций не только осуществляют его разбор заново (несмотря на то, что запрос с таким содержимым уже встречался), но и выполняют обработку в режиме интерпретации.

---

[4]Document Object Model, см. `http://ru.wikipedia.org/wiki/DOM`

[5]XML Path Language, см. `http://ru.wikipedia.org/wiki/XPath`

[6]Кэш — cache, см. `http://en.wikipedia.org/wiki/Cache`

В противоположность этому — «хорошей практикой» («best practice») является компиляция запроса и запоминание результатов компиляции для последующего использования. В лучших реализациях XPath происходит не только разбор запроса, но и генерация байт-кода, выполняющего запрос, что позволяет добиться максимальной производительности.

В связи с этим было принято решение отказаться от использования JSTL.

3. Производительность встроенных разборщиков XML

Существует 3 способа разбора XML.

1. DOM — полный разбор и построение дерева объектов; приложение может осуществлять произвольный доступ к данным.

2. SAX[7] — приложение подписывается на события и получает данные в виде уведомлений о событиях во время разбора; данные «продавливаются» (push).

3. StAX[8] — приложение последовательно извлекает данные, запрашивая дальнейшие шаги разбора; данные «вытягиваются» (pull).

Каждый способ имеет свои недостатки и преимущества. DOM работает медленнее остальных способов, использует большее количество памяти, хранит избыточные структуры. SAX работает значительно быстрее, использует значительно меньше памяти, но гораздо труднее извлекать данные из потока событий. StAX предоставляет минимальное, но достаточное количество сервисов; скорость работы и затраты памяти сопоставимы с SAX, но модель управления разбором заметно удобнее.

Таким образом, для повышения производительности, наиболее подходящим вариантом является использование StAX вместо DOM.

Платформа Java предоставляет реализации для всех трёх способов. Это так называемые «эталонные» (reference) реализации. Для достижения максимальной производительности разработчики могут подключать и использовать «сторонние» (3^{rd}-party) библиотеки.

Наиболее интересной, с технологической точки зрения, библиотекой, работающей с XML (и не только), является Javolution[9]. Особенностью данной библиотеки является нацеленность на использование стандарта

[7]Simple API for XML, см. `http://en.wikipedia.org/wiki/Simple_API_for_XML`
[8]Streaming API for XML, см. `http://en.wikipedia.org/wiki/StAX`
[9]Javolution — A Java(TM) Revolution, см. `http://javolution.org/`

Real-Time Specification for Java (RTSJ[10]). Работа в режиме «реального времени» (real-time) означает, что выполнение любой операции завершается не позже предсказуемого момента времени (time-predictable).

Написание приложений на платформе Java отличается от написания приложений на C++ тем, что нет необходимости контролировать удаление объектов. Система сама удаляет неиспользуемые объекты в результате «сбора мусора» (garbage collection). Поэтому программы на платформе Java работают стабильнее и меньше подвержены утечкам памяти.

Однако сбор мусора — это процесс, начало которого невозможно предсказать, а следовательно невозможно предсказать время завершения операций, прерываемых сбором мусора. Для создания быстродействующих систем и систем реального времени была разработана альтернативная модель работы с объектами в памяти, позволяющая выполнять операции, не прерываемые сбором мусора.

Одна из парадигм Java-систем реального времени — избегать создания объектов во время выполнения. Следование этому правилу позволяет повысить быстродействие систем даже в том случае, если альтернативная модель памяти не поддерживается платформой.

Пример. Эталонная реализация StAX возвращает значения в виде объектов типа `String`. Таким образом, каждый раз, когда есть необходимость сравнить, например, имя текущего XML-тэга с шаблоном, конструируется новый объект, который сразу после сравнения становится ненужным. Реализация StAX в библиотеке Javolution возвращает объект типа `CharSequence`, который конструируется один раз при инициализации библиотеки и используется многократно. Это классическое использование паттерна проектирования «приспособленец» («flyweight»). За счёт этого можно добиться производительности близкой к максимальной.

Использование StAX вместо DOM подразумевает, что данные будут храниться не в дереве, состоящем из универсальных узлов, а в структуре специфичной для приложения. Кроме того, размещение данных в этой структуре должно выполняться самим приложением, а не библиотекой. Это требует написания бóльшего количества кода, но при этом даёт дополнительные преимущества. В рассматриваемом случае при заполнении структуры можно построить индекс списков вложенных объектов. Тогда сопоставление структуры данных и актуальных данных по идентификатору можно выполнить за время O(1), а не за

[10]JSR 282: RTSJ version 1.1, см. `http://jcp.org/en/jsr/detail?id=282`

время O(n).

4. Библиотека генерации представления

Как было отмечено ранее, использование тэгов на основе Java-классов сводит на нет разделение представления и логики приложения. Решением данной проблемы служит создание библиотеки генерации представления на основании шаблонов.

Как показывает практика, в большинстве случаев JSP — это излишне универсальный инструмент не обладающий достаточной производительностью и при этом позволяющий написание идеологически некорректных страниц (где смешиваются логика и представление). Оптимальным вариантом, позволяющим добиться высокой производительности, является использование шаблонов. Шаблон — это текст с особой разметкой, разбивающей его на фрагменты двух типов:

- прямой текст (literal characters);
- дескриптор подстановки (substitution id).

С технической точки зрения шаблон — это класс, поочерёдно записывающий в поток данные, хранящиеся в нём, и данные, взятые из источника подстановок.

Первая пробная реализация принимала на вход объект потока и объект, реализующий интерфейс `Map<String, String>`. В моменты, когда необходимо было записать в поток значение подстановки, оно извлекалось как значение, ключом к которому являлся дескриптор подстановки.

Почти сразу выяснилось, что это неудобно в случае, когда значение подстановки генерируется на основании вложенного шаблона. Вторая реализация принимала на вход в качестве источника подстановок объект, реализующий интерфейс `IFastRenderer`. В этом интерфейсе объявлены два метода:

- метод `Writer getOut()`, возвращающий поток для вывода представления;
- метод `boolean render(String substituteId)`, выводящий в поток значение подстановки и возвращающий значение `false`, если дескриптор подстановки не опознан.

При таком подходе решаются три проблемы:

- упрощается генерация представления вложенных шаблонов;

- исчезает необходимость передавать поток вывода шаблону;
- добавляется централизованное оповещение об ошибках (о неопознанных дескрипторах подстановки).

Реализация метода `render` выглядела следующим образом:

```
01|  if (CONST1.equals(substituteId)) {
02|    ...
03|  } else if (CONST2.equals(substituteId)) {
04|    ...
05|  } ...
06|  } else {
07|    return false;
08|  }
09|  return true;
```

Сравнение строковых объектов является длительной операцией относительно операции сравнения двух целых чисел. Поэтому в следующей реализации дескриптор подстановки был заменён на значение его хэш-кода[11] (hash-code), которое предоставляется каждым объектом в Java и служит некоторым показателем схожести объектов. Вероятность совпадения хэш-кодов двух различных дескрипторов подстановки из одного шаблона исчезающе-мала (меньше чем 10^{-7}), соответственно, проводить более тщательную проверку не имеет смысла.

После того, как отладка библиотеки шаблонов была завершена и были написаны тэги, использующие эту библиотеку, были проведены тесты производительности. В результате были получены вполне предсказуемые, но значимые цифры. В приложении на основе XML тэгов JSTL на генерацию представления уходило, в среднем, 1450 миллисекунд (учитывая, что некоторые части представления извлекались из кэша). Приложение, использующее Javolution реализацию StAX и библиотеку шаблонов, ту же операцию выполняет, в среднем, за 3,6 миллисекунды, то есть примерно в 400 раз быстрее.

Логическим продолжением работы над библиотекой было создание тэгов, альтернативных используемым тэгам JSTL, для того, чтобы можно было полностью отказаться от последних. Созданные тэги обладают менее универсальной, но более востребованной функциональностью:

- реализованный тэг «if», в отличие от входящего в состав JSTL,

[11] Хэш — hash — результат свёртывания массива данных в целое число фиксированной разрядности, см. `http://en.wikipedia.org/wiki/Hash_function`

позволяет описывать альтернативный вариант генерации представления (при невыполнении условия);

- тэг «enlist» генерирует список на основании строки, в которой значения разделены специальными символами; шаблоном служит другая строка, в которой специальным символом отмечена позиция для вставки; за счёт удаления лишней логики и вызовов вложенных JSP-фрагментов, скорость генерации увеличена в несколько раз.

Отказ от библиотеки JSTL и удаление библиотек, от которых она зависит (реализация XPath), привели к уменьшению размеров артефакта веб-приложения на 4,5 мегабайта.

5. Динамическая генерация байт-кода

Построенная библиотека генерации представления по шаблону показала хорошие результаты в тестах на производительность, однако для её использования каждый раз необходимо реализовать метод `render`, который будет выглядеть не лучшим образом. К тому же, если подстановок в одном шаблоне много, то это может негативно отразиться на производительности.

В смысле удобства, идеальный вариант должен исключать написание разработчиком кода, не относящегося непосредственно к логике генерации представления. В данном случае, было бы удобно, если бы вместо одного метода `render` был набор методов `renderSomething`, у которых суффикс соответствует дескриптору подстановки.

Технически это может быть осуществлено двумя способами:

1. Использование механизмов интроспекции[12] и «отражения»[13] программной платформы Java.

2. Автоматическая генерация классов по заданным шаблонам.

Первый способ проще, к тому же, всё необходимое для его осуществления уже является частью технологической платформы Java. Однако его применение вносит в проект ряд отрицательных аспектов:

[12]Интроспекция — introspection — механизм, позволяющий проводить исследование полей, методов и констант Java-классов во время выполнения программы, см. `http://java.sun.com/docs/books/tutorial/javabeans/introspection`

[13]Отражение — reflection — механизм, позволяющий динамически загружать и создавать экземпляры класса, а также осуществлять доступ к полям и методам класса, см. `http://en.wikipedia.org/wiki/Reflective_programming#Java`

- производительность библиотеки шаблонов не увеличится, так как для генерации представления, так же как и в предыдущей реализации, будет осуществляться обход массива данных;
- на вызов методов через «отражение» затрачивается в 100–200 раз больше времени, чем на прямой вызов, что также отрицательно сказывается на производительности;
- среда выполнения Java не сможет выполнить оптимизацию исполнимого кода во время выполнения.

Поскольку целью создания библиотеки является достижение максимальной производительности, то выбор был сделан в пользу автоматической генерации классов по шаблонам.

Для генерации Java байт-кода существует несколько разработок, таких как ASM[14], BCEL[15], Javassist[16], Serp[17]. ASM на данный момент является наиболее проработанной и быстродействующей библиотекой. Проект ASM включает в себя так же удобные инструменты, интегрирующиеся в среду разработки Eclipse[18].

Разработка библиотеки на основе генерации байт-кода проходит обычно в три этапа. На **первом этапе** создаётся прототип класса, который соответствует будущему генерируемому байт-коду.

Прототип класса, соответствующего источнику подстановок:

```
01| import java.io.Writer;
02| public final class VictimImpl implements IFastRenderer {
03|   @Override public final Writer getOut() {
04|     return null;
05|   }
06|   public final void renderSomething() { }
07| }
```

Прототип класса, осуществляющего генерацию представления:

```
01| import java.io.IOException;
02| import java.io.Writer;
03| public final class TplStub implements ITemplate {
04|   final char[] literal;
```

[14] См. http://asm.objectweb.org/
[15] Byte Code Engineering Library, см. http://jakarta.apache.org/bcel/news.html
[16] Java Programming Assistant, см. http://www.csg.is.titech.ac.jp/~chiba
[17] См. http://serp.sourceforge.net/
[18] См. http://www.eclipse.org/

```
05|    public TplStub(char[] _literal) {
06|      literal = _literal;
07|    }
08|    @Override public final void process(IFastRenderer victim)
09|        throws IOException {
10|      Writer out = victim.getOut();
11|      VictimImpl _victim = (VictimImpl) victim;
12|      out.write(literal, 5, 6);
13|      _victim.renderSomething();
14|    }
15|  }
```

Для хранения прямого текста используется массив символов, в котором сохранены все отрывки. Это решение принято на основании анализа абстрактного класса `Writer` — все остальные способы вывода информации в поток используют абстрактный метод `write`, принимающий в качестве параметров массив символов, начальную позицию и длину последовательности. Вывод в поток без дополнительных преобразований позволяет достичь лучшей производительности.

Как видно из класса, метод process не является статическим. Объявить его статическим нельзя, так как нельзя объявлять статические методы в интерфейсе; как следствие, отсутствует возможность вызвать статический метод сгенерированного класса напрямую.

Затруднительным также является объявление статическим поля прямого текста. Дело том, что в виртуальной машине Java нет такого понятия, как константный массив символов; с точки зрения байт-кода это можно было бы обойти, добавив в класс статический инициализатор с параметрами, однако не существует способа передачи параметра в статический инициализатор.

Второй этап — анализ и оптимизация байт-кода. Далее приведено мнемоническое представление байт-кода прототипа класса, осуществляющего генерацию представления:

```
01|// class version 50.0 (50)
02|// access flags 49
03|public final class pkg/TplStub implements pkg/ITemplate {
04|  // access flags 16
05|  final [C literal
07|  // access flags 1
08|  public <init>([C)V
09|    L0
10|      LINENUMBER 9 L0
11|      ALOAD 0
12|      INVOKESPECIAL java/lang/Object.<init>()V
13|    L1
14|      LINENUMBER 10 L1
```

```
15|      ALOAD 0
16|      ALOAD 1
17|      PUTFIELD pkg/TplStub.literal : [C
18|    L2
19|      LINENUMBER 11 L2
20|      RETURN
21|    L3
22|      LOCALVARIABLE this Lpkg/TplStub; L0 L3 0
23|      LOCALVARIABLE _literal [C L0 L3 1
24|      MAXSTACK = 2
25|      MAXLOCALS = 2
27|  // access flags 17
28|  public final process(Lpkg/IFastRenderer;)V throws java/io/IOException
29|    L0
30|      LINENUMBER 15 L0
31|      ALOAD 1
32|      INVOKEINTERFACE pkg/IFastRenderer.getOut()Ljava/io/Writer;
33|      ASTORE 2
34|    L1
35|      LINENUMBER 16 L1
36|      ALOAD 1
37|      CHECKCAST pkg/VictimImpl
38|      ASTORE 3
39|    L2
40|      LINENUMBER 17 L2
41|      ALOAD 2
42|      ALOAD 0
43|      GETFIELD pkg/TplStub.literal : [C
44|      ICONST_5
45|      BIPUSH 6
46|      INVOKEVIRTUAL java/io/Writer.write([CII)V
47|    L3
48|      LINENUMBER 18 L3
49|      ALOAD 3
50|      INVOKEVIRTUAL pkg/VictimImpl.renderSomething()V
51|    L4
52|      LINENUMBER 19 L4
53|      RETURN
54|    L5
55|      LOCALVARIABLE this Lpkg/TplStub; L0 L5 0
56|      LOCALVARIABLE victim Lpkg/IFastRenderer; L0 L5 1
57|      LOCALVARIABLE out Ljava/io/Writer; L1 L5 2
58|      LOCALVARIABLE   victim Lpkg/VictimImpl; L2 L5 3
59|      MAXSTACK = 4
60|      MAXLOCALS = 4
61|}
```

Поскольку генерация байт-кода будет осуществляться без генерации исходного кода, то добавление данных о номере строки избыточно. Соответственно можно отбросить строки 10, 14, 19, 30, 35, 40, 48, 52.

В конструкторе (строки 7–25) метки L1 и L2 становятся неиспользуемыми, их также можно отбросить (строки 13, 18).

Строки 42–43 соответствуют обращению к члену экземпляра и выполняются для каждого участка прямого текста. Это неэффективно, поскольку значение неизменно. Эффективнее было бы хранить ссылку в локальной переменной.

В строках 44–45 хорошо прослеживается, что для того чтобы выполнить операцию сохранения на стеке целого числа существуют различные сокращения. Так, например, для чисел от 0 до 5 существуют байт-коды, не принимающие параметров. Для чисел в диапазоне от 6 до 127 можно воспользоваться командой `BIPUSH`, операнд которого имеет длину 1 байт. Такая команда выполняется быстрее и занимает меньше места, чем команда `LDC`, принимающая операнд длиной 4 байта (двойное слово). То же самое можно сказать про команду `SIPUSH`, операнд которой имеет длину 2 байта. Очевидным становится, что для удобства генерации байт-кода желательно создать метод, который сохраняет числовые константы на стеке.

Известно, что локальные переменные при выполнении кода неравнозначны. Так, чаще всего, первые три из них имеют особое значение и хранятся в регистрах процессора. Как видно из дампа, в основной процедуре объявлено использование четырёх локальных переменных, и ещё одна будет использована для хранения локальной ссылки на массив символов прямого текста. Однако, для работы центрального блока метода используются только три переменные: ссылка на источник данных, ссылка на поток вывода, ссылка на прямой текст.

Так, предметом оптимизации может служить использование локальных переменных. Действительно, если в строке 38 изменить номер локальной переменной для хранения приведённого источника подстановок с 3 на 1, то высвобождается одна переменная. Переменная 0 (`this`) не используется после получения ссылки на прямой текст. Соответственно переменную 0 можно использовать для хранения ссылки на прямой текст.

Данные шаги оптимизации неосуществимы для компилятора Java, поскольку для верного определения действия типов переменных (строки 55–58) пришлось бы ставить метки между началом операции (запись переменных в стек, вызов метода) и окончанием операции (перенесение результатов из стека в локальную переменную). Но поскольку генерация байт-кода осуществляется сторонними средствами, такое становится возможным. Таким образом, достигается производительность, превосходящая максимальную для стандартного языка Java.

Третий этап представляет чисто техническую процедуру написания генератора байт-кода, загрузчика классов, разборщика шаблонов. В процессе тестирования выявляются и исправляются мелкие недочёты.

Замер производительности показал, что на генерацию представления среднего по сложности шаблона без вложенных шаблонов, затрачивается около 2600 наносекунд.

6. Дальнейшая оптимизация

Для исследования возможности дальнейшей оптимизации приложения была проведена серия «синтетических тестов»[19]. Один из тестов показал, что вызов метода с указанием абстрактного класса как цели вызова длится в 4,5 раза дольше, чем вызов метода с указанием класса, которому известна реализация метода.

В нашем случае, генерируемый класс при выводе данных из шаблона в поток указывал на метод абстрактного класса `java.io.Writer`. Следовательно, подстановка конечного класса увеличивает производительность. После реализации расширенного варианта кодогенерации был проведён сравнительный тест производительности. Результаты синтетического теста нашли отражение в новом тесте — скорость выполения оказалась в 4,5 раза выше. Однако, на практике это не оказывает влияния на производительность поскольку:

- время выполнения характерной реализации метода `write` класса `java.io.Writer` на два порядка выше времени выполнения вызова;
- в большинстве случаев нет возможности определить класс реализующий поток вывода до первого вызова генерации представления в силу того, что построение предположений о контейнере приложения является недопустимым с точки зрения переносимости приложения.

Второе ограничение можно обойти, если генерация всего представления осуществляется непосредственно приложением, без использования технологии JSP. При этом можно увеличить производительность генерации представления: поскольку длина исходящего потока имеет предсказуемый размер, то возможно создание реализации `java.io.Writer`, которая не затрачивает время на выделение блоков памяти и увеличение буфера записи.

7. Заключение

В процессе работы по уменьшению времени отклика веб-приложения была создана высокопроизводительная библиотека генерации представления на основе шаблонов. Данная библиотека осуществляет динамическую генерацию исполнимого кода, что позволяет достигать време-

[19] «Синтетический тест» — тест изучающий определённую часть системы и не учётывающий влияние остальных частей. Результаты таких тесвов подлежат тщательной интерпретации.

ни выполнения, близкого к минимальному. Созданная библиотека является продуктом с открытым исходным кодом и доступна по адресу `http://mist4j.googlecode.com`. В статье также рассмотрены приёмы, позволяющие решить проблемы скорости в веб-приложениях как на финальной стадии разработки веб-приложений, так и на более ранних стадиях.

Литература

1. **Alur Deepak, Crupi John, Malks Dan** Core J2EE Patterns: Best Practices and Design Strategies — 2^{nd} edition. Prentice Hall Ptr, 2003. 650 с., ил.

2. **Bruce Eckel.** Thinking in Java — 4^{th} ed. Prentice Hall Ptr, 2006. 1408 с., ил.

3. **Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.** Приемы объектно-ориентированного программирования. Паттерны проектирования. СПб: Питер, 2001. 368 с., ил.

4. **Тейт Б.** Горький вкус Java. СПб: Питер, 2003. 336 с.

5. **Фаулер М.** Рефакторинг. Улучшение существующего кода. Символ-Плюс, 2007. 432 с.

6. **Шпильман С.** JSTL. Практическое руководство для JSP-программистов. КУДИЦ-Образ, 2004. 272 с.

7. **Kuleshov E.** Using the ASM framework to implement common Java bytecode transformation patterns.
`http://asm.objectweb.org/current/asm-transformations.pdf`

8. **Bruneton E., Lenglet R., Coupaye T.** ASM: a code manipulation tool to implement adaptable systems.
`http://asm.objectweb.org/current/asm-eng.pdf`

Summary

Kluchnikov E.A. Web-application performance tuning

The author consider a typical Web2.0 application performance issues. As a solution a new high-performance template-processor dynamic code-generation library is developed.

Сыктывкарский университет *Поступила 30.10.2008*