

Прозрачное кэширование XSL преобразований с использованием JAXP

Повысьте производительность и сохраните удобство использования, реализуя неявное кэширование внутри фабрик преобразователей.

XSLT Tutorial

Алексей Валиков

Когда XSLT (Extensible Stylesheet Language Transformations) применяется в Web контексте, где множество параллельных потоков неоднократно используют преобразования, реализация кэширования стилевых таблиц значительно увеличивает производительность. Однако, применение кэша стилевых таблиц на верхнем уровне стандартного JAXP (Java API for XML Parsing) часто неудобно или неприменимо для пользователей JAXP. Эта статья познакомит вас с идеей помещения функциональности кэша внутри реализации фабрики преобразователя, делающего использование кэша абсолютно прозрачным. *(В оригинальной версии на английском языке 1300 слов)*

Несомненно, XSLT мощная технология, которая может иметь множество применений в мире XML. В особенности, множество Web разработчиков могут извлечь пользу из XSLT, используемого на уровне представления, приобретая удобство и гибкость реализации. Однако, ценой таких преимуществ является более высокая нагрузка процессора и памяти, что делает разработчиков более внимательными к технологиям оптимизации и кэширования при использовании XSLT. Кэширование особенно важно в Web контексте, где множество потоков разделяют стилевые таблицы.

Прозрачное кэширование XSL преобразований с использованием JAXP

В таких случаях, правильное кэширование преобразований оказывается жизненно важным для производительности. Стандартной рекомендацией при использовании Java API for XML Processing (JAXP) является загрузка преобразований в объект `Templates` и последующее использование этого объекта для порождения `Transformer`, вместо создания объектов `Transformer` непосредственно из фабрики. В таком случае, объект `Templates` может быть повторно использован для дальнейшего создания преобразователей и экономит время, необходимое для разбора и компиляции стилевых таблиц. В "[Десятке советов по Java и XSLT](#)", Эрик Бюрке (Eric Burke) приводит в первом совете следующий код:

```
Source xsltSource = new StreamSource(xsltFile);

TransformerFactory transFact = TransformerFactory.newInstance();
Templates cachedXSLT = transFact.newTemplates(xsltSource);
Transformer trans = cachedXSLT.newTransformer();
```

В этом примере, преобразование из `xsltFile` сначала загружается в `cachedXSLT` — объект `Templates`, который потом используется для создания нового объекта преобразователя, `trans`. Преимущество данного метода состоит в том, что в дальнейшем, когда вам понадобится получить другой объект преобразователя, фазы разбора и компиляции стилевой таблицы могут быть пропущены:

```
Transformer anotherTrans = cachedXSLT.newTransformer();
```

Несмотря на то, что такая технология положительно сказывается на производительности (особенно, когда повторно используются одни и те же стилевые таблицы, как в Web приложениях), на самом деле, она не удобна для разработчиков. Причина этого заключается в том, что отдельно от основанного на `Templates` создания преобразователей, вы должны позаботиться о наблюдении за временем последнего изменения стилевых таблиц, повторной загрузке устаревших преобразований, предоставлении безопасного и эффективного многопоточного доступа к кэшу стилевых таблиц и множестве прочих мелких деталей. Даже кажущееся естественным решение — инкапсуляция всей необходимой функциональности в отдельную реализацию кэша преобразователей — не сможет оградить разработчика от модулей третьей стороны, которые используют стандартные операции JAXP без всякого кэширования. Хорошим примером такого модуля является

Прозрачное кэширование XSL преобразований с использованием JAXP

элемент `x:transform` из JSTL: его текущая реализация в классах `org.apache.taglibs.standard.tag.common.xml.TransformSupport` и `org.apache.taglibs.standard.tag.el.xml.TransformTag`, непосредственно использует метод `newTransformer(...)` класса `TransformerFactory`. Очевидно, `x:transform` не может воспользоваться преимуществами внешних реализаций кэширования.

Однако, существует простое и элегантное решение данной проблемы. До тех пор, пока JAXP позволяет нам замещать используемую реализацию `TransformerFactory`, почему бы нам просто не написать фабрику, которая будет иметь внутреннюю способность кэширования?

Эта идея не сложна в реализации. Мы можем расширить любую подходящую реализацию `TransformerFactory` (в примере использовался [Саксон\(Saxon\)](#) Мишеля Кая(Michael Kay)) и переопределить родительский метод `newTransformer(...)` так, чтобы преобразования загружаемые из файловых потоков кэшировались и возвращались из кэша, если преобразования не были модифицированы со времени последней загрузки. Новая версия метода `newTransformer(...)` выглядит следующим образом:

```
public Transformer newTransformer(final Source source)
    throws TransformerConfigurationException
{
    // Проверяем, что источник имеет тип StreamSource
    if (source instanceof StreamSource)
        try
        {
            // Создаём URI (Uniform Resource Identifier
            // – универсальный идентификатор ресурса) источника

            final URI uri = new URI(source.getSystemId());
            // Если URI указывает на файл, загружаем преобразователь из файла
            // (или из кэша)

            if ("file".equalsIgnoreCase(uri.getScheme()))
                return newTransformer(new File(uri));
        }
}
```

Прозрачное кэширование XSL преобразований с использованием JAXP

```
catch (URISyntaxException urise)
{
    throw new TransformerConfigurationException(urise);
}

return super.newTransformer(source);
}
```

Как вы видите, если источник преобразователя не является поточным, или не указывает на файл, то будет использована родительская реализация `newTransformer(...)` для возврата преобразователя. Но если источник является файловым потоком, то он даёт нам возможность реализовать более интеллектуальную загрузку преобразований с помощью кэша.

Алгоритм кэширования для файловых стилевых таблиц довольно простой: для указанного файла мы проверяем, не сохранён ли ещё в кэше объект `Templates` преобразования с таким же абсолютным именем файла. Если нет, то мы создаём и кэшируем новый объект `Templates` для данного файла. Если в кэше уже что-то есть, то мы проверяем, был ли файл обновлён с момента последней загрузки `Templates`, сравнивая дату последней модификации файла с содержимым кэша. Если файл был обновлён, `Templates` должен быть загружен повторно, иначе он может быть взят из кэша. Наконец, используя объект `Templates` (загруженный из кэша или с диска, в зависимости от ситуации), мы просто создаём новый преобразователь. Реализацией этого алгоритма является следующий метод:

```
protected Transformer newTransformer(final File file)
    throws TransformerConfigurationException
{
    // Ищем шаблон в кэше
    TemplatesCacheEntry templatesCacheEntry = read(file.getAbsolutePath());

    // Если вхождение найдено
    if (templatesCacheEntry != null)
    {
        // Проверяем время модификации
        if (templatesCacheEntry.lastModified
            < templatesCacheEntry.templatesFile.lastModified())
            // Очищаем элемент, если он устарел
    }
}
```

Прозрачное кэширование XSL преобразований с использованием JAXP

```
        templatesCacheEntry = null;
    }

    // Если не найденно шаблона или он устарел
    if (templatesCacheEntry == null)
    {
        logger.debug("Загружаем преобразование [" +
            file.getAbsolutePath() + "].");

        // Если этот файл не существует, выбрасываем исключение
        if (!file.exists())
        {
            throw new TransformerConfigurationException(
                "Запрошенное преобразование [" +
                file.getAbsolutePath()
                + "] не существует.");
        }

        // создаём новый элемент кэша
        templatesCacheEntry =
            new TemplatesCacheEntry(new Templates(new StreamSource(file)), file);

        // Сохраняем этот элемент в кэше
        write(file.getAbsolutePath(), templatesCacheEntry);
    }
    else
    {
        logger.debug("Используется закэшированное преобразование [" +
            file.getAbsolutePath() + "].");
    }

    return templatesCacheEntry.templates.newTransformer();
}
```

Однако мы должны учитывать другую проблему: безопасность потоков. До тех пор, пока параллельные потоки совместно используют кэш, мы должны проявлять определённую осторожность, что бы сделать безопасными операции чтения (получения элементов из кэша) и записи (сохранения вновь загруженных стилевых таблиц в кэше). Если говорить о приведённом выше коде, то `read(...)` и `write(...)` должны не создавать конфликтов, даже при запуске множества

параллельных потоков.

Несмотря на то, что Java предлагает продвинутое средства синхронизации, проблема здесь не в синхронизации как таковой, а в балансе между синхронизацией и производительностью. Простейшим решением является полная синхронизация: мы объявляем весь метод `newTransformer(...)` синхронизированным и используем синхронизированный контейнер для сохранения элементов кэша или осуществляем доступ к кэшу из синхронизированного блока, но это неэффективно. Пока существует ограниченное число стилевых таблиц, и они изменяются не часто, кэш преобразований будет более часто читать информацию, чем записывать. Тогда полная синхронизация будет блокировать параллельные чтения, что, во-первых, не всегда необходимо, а во вторых, может создать узкое место в системе.

С другой стороны, использование не синхронизированных контейнеров, таких как `HashMap` для хранения элементов кэша опасно. Если мы не предпримем каких-либо мер, одновременные чтение и запись могут (с некоторой вероятностью) стать причиной конфликта, приводящего к нестабильности системы.

Таким образом, мы имеем классическую проблему чтения/записи: для заданного ресурса, может быть только одна запись или несколько чтений в любой момент времени. Эта классическая проблема имеет классическое решение, которое мы можем взять из книги [Параллельное программирование в Java \(Concurrent Programming in Java\)](#) Дуга Леа (Doug Lea). Идея состоит в отслеживании состояния выполнения методом подсчёта активных или ожидающих чтения и записи потоков, и разрешать чтение только, когда не существует активных пишущих потоков, и запись только тогда, когда нет, не читающих, не пишущих активных потоков.

Чтобы сделать это, мы выделим доступ к кэшу в два метода, `read()` и `write()` :

```
protected TemplatesCacheEntry read(final String absolutePath)
{
    beforeRead();

    final TemplatesCacheEntry templatesCacheEntry =
        (TemplatesCacheEntry) templatesCache.get(absolutePath);

    afterRead();
}
```

Прозрачное кэширование XSL преобразований с использованием JAXP

```
    return templatesCacheEntry;
}

protected void write(final String absolutePath, final TemplatesCacheEntry
    templatesCacheEntry)
{
    beforeWrite();

    templatesCache.put(absolutePath, templatesCacheEntry);

    afterWrite();
}
```

Две пары до/после (*before / after*), методов чтения/записи (*read / write*) выполняют синхронизацию потоков, обеспечивая безопасный, но при этом эффективный доступ к кэшу:

```
protected synchronized void beforeRead()
{
    while (activeWriters > 0)
        try
        {
            wait();
        }
        catch (InterruptedException iex)
        {
        }

    ++activeReaders;
}

protected synchronized void afterRead()
{
    --activeReaders;
    notifyAll();
}

protected synchronized void beforeWrite()
{
```

Прозрачное кэширование XSL преобразований с использованием JAXP

```
while (activeReaders > 0 || activeWriters > 0)
    try
    {
        wait();
    }

    catch (InterruptedException iex)
    {
    }

    ++activeWriters;
}

protected synchronized void afterWrite()
{
    --activeWriters;
    notifyAll();
}
```

Имея доступ к кэшу, реализованный как показано выше, мы наконец получаем фабрику преобразователей, которая прозрачно реализует эффективное кэширование основанных на файлах стилевых таблиц (вы можете скачать полный исходный код из [Ресурсов](#)). Единственное, что осталось — это сделать нашу фабрику доступной через стандартные процедуры JAXP.

Различными способами можно сделать так, чтобы метод `TransformerFactory.newInstance()` возвращал экземпляр реализованной нами фабрики преобразователей. Наиболее простой и правильный метод заключается в определении имени класса фабрики в системном свойстве `javax.xml.transform.TransformerFactory`:

```
System.setProperty("javax.xml.transform.TransformerFactory",
    "de.fzi.dbs.transform.CachingTransformerFactory");
```

Это метод имеет преимущество наибольшей приоритетности, но его недостатком является необходимость ручного кодирования.

Другой подход состоит в использовании конфигурационного файл JRE (Java Runtime Environment) `${JRE_HOME}/lib/jaxp.properties` для определения вашего

Прозрачное кэширование XSL преобразований с использованием JAXP

собственного имени класса:

```
...  
# Определяет реализацию фабрики преобразователей  
  javax.xml.transform.TransformerFactory=de.fzi.dbs.transform.CachingTransformerFactory  
...
```

Последней возможностью является использование Services API для предоставления имени фабрики преобразователей в библиотеке метаинформации. Просто создайте файл с именем `javax.xml.transform.TransformerFactory` в директории `META-INF/services` вашего `jar` файла. Содержимым этого файла должна быть единственная строка, определяющая имя класса реализованной нами фабрики преобразователей. Этот метод, однако, опасен: другой JAR может также попытаться установить класс фабрики через Services API. Например, если вы положите ваш JAR и JAR Саксона (Saxon) в директорию `WEB-INF/lib` своего Web приложения, то используемая фабрика будет зависеть от порядка, в котором эти JAR файлы загружаются. Для того, чтобы избежать этой неизвестности в Web приложениях, просто конфигурируйте свою фабрику в файле `WEB-INF/classes/META-INF/services/javax.xml.transform.TransformerFactory`. В нашем случае, он будет содержать единственную строку `de.fzi.dbs.transform.CachingTransformerFactory`.

1. Сделаем использование кэша прозрачным

Теперь, когда всё закончено, вы имеете ещё одной головной болью меньше. Вы можете больше не беспокоиться о загрузке, кэшировании и повторной загрузке стилевых таблиц. Вы имеете гарантию, что библиотеки сторонних производителей, которые используют стандартный JAXP, будут успешно использовать кэширование. Вы можете быть уверены в отсутствии конфликтов параллельного доступа, и том, что кэш не станет узким местом в системе.

Есть, однако, несколько недостатков в использовании данной реализации. Во-первых, эта фабрика кэширует только стилевые таблицы, загруженные из файлов. Причиной этого является то, что в то время, как мы можем легко проверить время последней модификации файла, это не всегда возможно для других источников. Другая проблема остаётся со стилевыми таблицами, которые импортируют или включают другие

стилевые таблицы. Изменение импортированной или включённой стилевой таблицы не вызовет повторной загрузки основной стилевой таблицы. Наконец, расширение существующей реализации фабрики привязывает вас к определённому XSLT процессору (до тех пор, пока вы не напишите кэширующее расширение для любой фабрики, которую вы можете использовать). К счастью, в большинстве случаев, эти вопросы не критичны и мы можем получить преимущества размещённого на фабрике кэширования: прозрачность использования, удобство и производительность.

2. Об авторе

Алексей Валиков (Alexey Valikov) компьютерный специалист с хорошей программистской подготовкой, особенно в области технологий Java и XML. Его текущие исследования в FZI (Research Center for Computer Science, Karlsruhe/Germany) сосредоточены на вопросах эффективности в разработке Web приложений. Работая в центре компетенции по XML в FZI, он также консультантом и преподавателем по XML технологиям, и принимает участие в европейской комиссии исследовательских проектов. Алексей является автором популярного практического руководства по XSLT, *The Technology of XSLT*, изданного в России.

3. Ресурсы

- Скачайте исходный код к данной статье:
<http://www.javaworld.com/javaworld/jw-05-2003/xsl/jw-0502-xsl.zip>
- "Десятка советов по Java и XSLT", Эрика Бюрке (Eric M. Burke) (*java.oreilly.com*, Август 2001):
http://java.oreilly.com/news/javaxslt_0801.html
- *Параллельное программирование в Java, второе издание: Принципы и шаблоны проектирования*, Дуг Леа (Doug Lea) (Addison-Wesley Pub Co., 1999; ISBN: 0201310090):
<http://www.amazon.com/exec/obidos/ASIN/0201310090/javaworld>
- *Параллельное программирование в Java online приложение*:
<http://gee.cs.oswego.edu/dl/cpj/>
- Saxon, XSLT процессор написанный Michael Kay:
<http://saxon.sourceforge.net/>

Прозрачное кэширование XSL преобразований с использованием JAXP

- JAXP документация:
<http://java.sun.com/xml/jaxp/index.html>
 - Спецификация файлов Jar, документация по поставщикам услуг:
<http://java.sun.com/j2se/1.4/docs/guide/jar/jar.html#Service%20Provider>
 - log4J, пакет для работы с логами на Java (используется в предлагаемой реализации фабрики):
<http://jakarta.apache.org/log4j>
 - Apache Ant, основанное на Java средство сборки (вы можете использовать его для сборки предложенной реализации фабрики):
<http://ant.apache.org/>
- Другие статьи по XSLT на *JavaWorld* :
- " [Динамическая генерация классов JavaBean с XSLT](#) ," Victor Okunev (February 2002)
 - " [Поддержка Struts с XSLT и XML](#) ," Julien Mercay и Gilbert Bouzeid (Февраль 2002)
 - " [XSLT Blooms with Java](#) ," Taylor Cowan (Декабрь 2001)
- Посмотрите раздел **Java и XML** тематического индекса *JavaWorld*
http://www.javaworld.com/channel_content/jw-xml-index.shtml
 - Обсудите XSLT в нашем **XML & Java** форуме:
<http://forums.devworld.com/webx?50@@.ee6b78f>
 - Подпишитесь на еженедельную рассылку *JavaWorld* по *Enterprise Java* :
<http://www.javaworld.com/subscribe>

Reprinted with permission from the May 2003 edition of JavaWorld magazine. Copyright © ITworld.com, Inc., an IDG Communications company.

View the original article at:
<http://www.javaworld.com/javaworld/jw-05-2003/jw-0502-xsl.html>

[Перевод на русский © Олег Лапшин, 2003](#)