Tomcatを使いやすくする: Tomcat 5のフィルター処理の技

保守の容易な高性能システムの設計

Sing Li
Author
Makawave

2003年 3月 25日

新しいTomcat 5サーバーは、フィルターを使って新しいレベルの配備の柔軟性を実現します。Tomcat 5は、近々決定されるServlet 2.4およびJSP 2.0の仕様をサポートしており、リクエスト・ディスパッチャーの処理と直接連携させる形で、そうした柔軟性の高いコンポーネントを統合し、配備するという新しいフィルター設計の方法を可能にしています。本稿では、Sing Liがこうした新しい機能拡張をひととおり解説した後、実際的な例題をいくつか紹介してくれます。Tomcat 5がWebアプリケーション・フレームワークにとって有効であり、保守の容易な高性能システムの設計すら可能にしてくれることが理解していただけることと思います。

2001年6月、私は (当時) 新しく発表されたばかりのServlet 2.3のフィルター処理 という機能を事前に紹介する記事を執筆しました (参考文献 参照)。Tomcat 4.xサーバーがこの機能をサポートし、Webアプリケーション開発者は、フィルターを利用することで、アプリケーション・サーバーのリクエスト・フローに (実際のリクエストが処理される前後に) 挿入したり、チェーンさせたりすることのできる柔軟性の高いアプリケーション・コンポーネントを作成できるようになりました。また、フィルターは、JSPページやサーブレットと同様、アプリケーション・レベルのコンポーネントとしてパッケージされるため、同じWARファイルにバンドルすることで、簡単にServlet 2.3準拠のサーバーに配備することができます。以来、大手のアプリケーション・サーバー・ベンダーは、Servlet 2.3標準を採用することで、フィルター処理をサポートするようになり、フィルターは、数多くのWebアプリケーションの基本的なコンポーネントとなりました。

フィルター処理のことをよく知らないという読者、あるいはフィルターの構築方法やフィルターの動作原理をおさらいしておきたいという方は、 1回目のTomcatのフィルター処理についての記事 をお読みいただくとよいでしょう。

Servlet 2.4でのフィルター機能の拡張

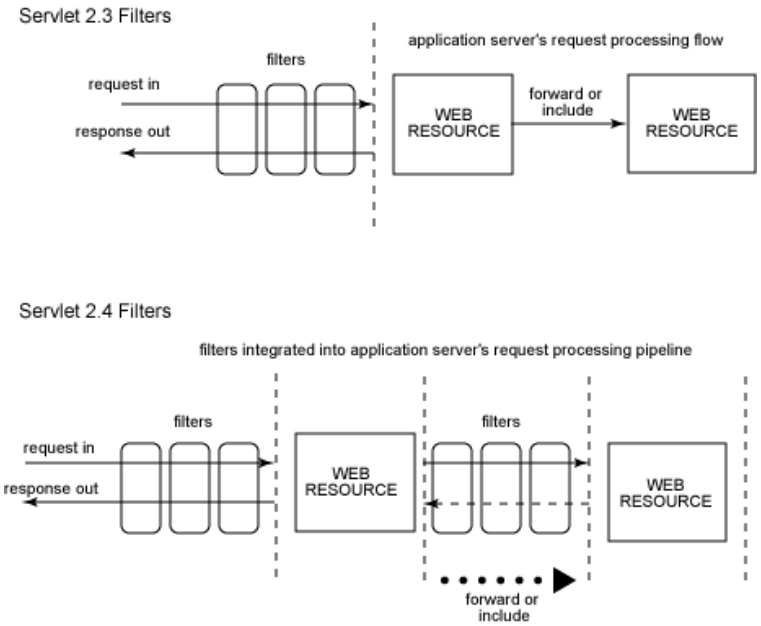
Web開発業界の「現場」からのフィードバックを基に、Servlet 2.4の仕様 (現在、詰めの作業が行われている) では、非常に重要なある新しい機能をフィルターに与えています。この新しい機能により、フィルターは、ついにサーブレットやJSPページとまったく対等な土俵で機能を果たせるようになり、Webアプリケーションを設計する際も、フィルターを使って重要なアプリケーション機能を追加できるようになります。この新しい機能は、また、JSPモデル2 とかモデル・ビュー・

コントローラー (MVC) 型Webアプリケーション設計と呼ばれる非常に人気の高い新しいスタイルのWebアプリケーション設計を推進することにもなります (JSPモデル2やMVCスタイルのWebアプリケーション設計の詳細については、 参考文献 を参照してください)。

リクエスト・ディスパッチャーの処理フローにおけるフィルターの位置付け

Servlet 2.4の新しいフィルター処理機能は、基本的に、アプリケーション・サーバーで実際にリクエスト処理を行う前 および後 のリクエスト・フローにおいてしかフィルターを働かせることができないという制約を緩和するものです。それに代わり、Servlet 2.4のフィルターは、すべてのディスパッチ・ポイントでリクエスト・ディスパッチャーとやりとりできるようになりました。このことは、あるWebリソースが別のリソースにリクエストを転送する場合 (たとえば、あるサーブレットが同じアプリケーション内のJSPページにリクエストを転送するような場合)、そのリクエストが転送先のリソースで処理される前に、フィルターを働かすことができるということを意味します。また、あるWebリソースが別のWebリソースからの出力や機能を含んでいる場合 (たとえば、あるJSPページが他の複数のJSPページからの出力を含んでいるような場合)、Servlet 2.4のフィルターは、それら関係する個々のリソースの前後で仕事をすることができる、ということにもなります。図1は、Servlet 2.3とServlet 2.4のディスパッチャーのやりとりの違いを示しています。

図1. Servlet 2.3とServlet 2.4のフィルターの違い



フィルターとディスパッチャーのやりとりの制御

フィルターがコンテナーのリクエスト処理フローとの間で行うやりとりは、配備記述子 (web.xml ファイル) の `<filters-mapping>` エレメントの定義によって設定されます。これは、Servlet 2.3 仕様で確立された規則 (convention) であり、それはServlet 2.4でも同じです。一方、ディスパッチャーとの間でやりとりを行う新しい機能は、新しいオプションの `<dispatcher>` サブエレメントで制御されます。リスト1は、配備記述子中の `<filters-mapping>` エレメントで `<dispatcher>` サブエレメントを使って、インクルードされるかもしれないJSPページにBlue Filterというフィルター

をかけることを表したものです (アプリケーション中のJSPページは、すべて、 `/jsp/*` というURLパターンにマップされるものとします)。

リスト1. Servlet 2.4スタイルのフィルター・マッピング

```
<filter-mapping>
          <filter-name>Blue Filter</filter-name>
          <url-pattern>/jsp/*</url-pattern>
          <dispatcher>INCLUDE</dispatcher>
 </filter-mapping>
```

インクルードされるリソースだけでなく、ディスパッチャーから転送されるリソースにも同じフィルターをかけたい場合には、リスト2に示すように、もう1個 `<dispatcher>` サブエレメントを追加するだけで済みます。

リスト2. 転送とインクルードのマッピング

```
<filter-mapping>
          <filter-name>Blue Filter</filter-name>
          <url-pattern>/jsp/*</url-pattern>
          <dispatcher>INCLUDE</dispatcher>
          <dispatcher>FORWARD</dispatcher>
 </filter-mapping>
```

`<dispatcher>` サブエレメントに指定できる値は、表1のとおりです。

表1. <dispatcher> サブエレメントに指定できる値

値	意味
REQUEST	クライアントから発信されたリクエストにフィルターをかける (Servlet 2.3準拠のコンテナーのデフォルトの動作とまったく同じ)。
FORWARD	リクエストがWebリソース間で `Dispatcher.forward()` 呼び出しを使って転送される際に、そのリクエストにフィルターをかける。
INCLUDE	Webリソースが `Dispatcher.include()` 呼び出しを使ってインクルードされる際にリクエストにフィルターをかける
ERROR	エラー処理リソース (基本的には、エラーが発生したときにコンテナーが管理する転送メカニズム) にフィルターをかける。

もちろん、表1の値が、すべて、 `<filter-mapping>` の `<url-pattern>` サブエレメントで指定されるマッチに依存することに変わりはありません。基本的に、 `<url-pattern>` と `<dispatcher>` サブエレメントにマッチするURLだけにフィルターがかけられます。

新しいフィルター/ディスパッチャーのやりとりの動きをよく理解するために、いくつかのフィルターをTomcatサーバーで働かせてみたいと思います。

Tomcat 5のフィルターの設定

Tomcat 5とJSTLのダウンロードとインストール

コードは、すべて、本稿執筆時点での最新のリリース・ビルドであるTomcat 5.0アルファで充分にテストされたものです。Tomcatをインストールすることの他に、JSTL 1.0のバイナリーもダウンロードする必要があります。standard.jarとjstl.jarをwebapps/dvworks/web-inf/libディレクトリーに入れ、TLDファイルを、すべて、webapps/dvworks/web-infディレクト

リーに入れます (TomcatのインストールおよびJSTL 1.0のバイナリーのダウンロードの詳細については、 参考文献 を参照してください)。

Tomcat 5をインストールし、JSP Standard Tag Library 1.0といっしょに実行させているものとして(囲み記事「Tomcat 5とJSTLのダウンロードとインストール」参照)、以下では、実際のフィルターの例をいくつか紹介していきたいと思います。 ソース・コード をダウンロードして、サンプルのWebアプリケーションを入手してください。

まず、JSPページとフィルターのコーディングをいくつか見てみることにします。最初に以下のリソースを扱います。

- /jsp/filtercount.jsp
- /jsp/included.jsp
- AddAttributesFilter
- RedFilter

filtercount.jspは、ごく単純なもので、リスト3に示すコードからなっています。

リスト3. filtercount.jspのコード

```
<%@ taglib uri="http://java.sun.com/jstl/core_rt" prefix="c" %>
<HTML>
<HEAD>
</HEAD>
<BODY>
<H1>developerWorks Tomcat 5 Filters</H1>
<jsp:include page="/jsp/included.jsp">
  <jsp:param name="BoxColor" value="red"/>
</jsp:include>
<br/>
<FONT FACE="verdana,arial,sans serif" SIZE="4">
<c:if test="${(requestScope.BlueCounter != null) & (requestScope.BlueCounter > 0)}">
  <font color="blue">The BlueFilter has been activated ${requestScope.BlueCounter} times.</font></br>
</c:if>
<c:if test="${(requestScope.RedCounter != null) & (requestScope.RedCounter > 0)}">
  <font color="red">The RedFilter has been activated ${requestScope.RedCounter} times.</font></br>
</c:if>
</FONT>
</BODY>
</HTML>
```

このJSPページは、ディスパッチャーを呼び出して、 `/jsp/included.jsp` という別のJSPページをインクルードしています。また、JSTLの式言語 (Expression Language) を使って、送られてくるリクエストに属性として付加される2つのカウンター (`RedCounter` と `BlueCounter`) の値を、カウンターが存在し、かつカウントが1以上である場合に、表示しています。

具体的には、それぞれのカウンターを使って、リクエスト処理の際に `RedCounter` か `BlueCounter` が呼び出される回数がカウントされます。表2は、今回使用するフィルターと、それが何をするものなのかを示したものです。

表2. 今回の実験で使用されるフィルターの名前と機能

フィルターの名前	機能

AddAttributesFilter	送られてくるリクエストに BlueCounter 属性および RedCounter 属性を付加する。
RedFilter	リクエストに付加された RedCounter 属性の値を +1する。
BlueFilter	リクエストに付加された BlueCounter 属性の値を +1する。

表2のフィルターは、いずれも、 com.ibm.dvworks.filters パッケージに含まれています。

com.ibm.dvworks.filters.AddAttributesFilter のコードは、リスト4のとおりです。このコードは、単に、 java.lang.Integer 型のカウンター属性を2個作成し、その値を0にしているだけです。

リスト4. AddAttributesFilter.javaのコード

```
package com.ibm.devworks.filters;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public final class AddAttributesFilter implements Filter {
    public static final String RED_COUNTER_ATTRIB = "RedCounter";
    public static final String BLUE_COUNTER_ATTRIB = "BlueCounter";

    private FilterConfig filterConfig = null;
    public void doFilter(ServletRequest request, ServletResponse response,
                         FilterChain chain)
  throws IOException, ServletException {
        request.setAttribute(RED_COUNTER_ATTRIB, new Integer(0));
        request.setAttribute(BLUE_COUNTER_ATTRIB, new Integer(0));
        filterConfig.getServletContext().log("Inside AddAttributesFilter");
        chain.doFilter(request, response);
    }

    public void destroy() {
        this.filterConfig = null;
    }

    public void init(FilterConfig filterConfig) {
  this.filterConfig = filterConfig;
    }
}
```

このフィルターの構成は、基本的なコーディング・パターンにしたがっています。繰り返しますが、フィルターのコーディング方法の詳細については、 第1回目の記事 を参照してください。

同様に、 com.ibm.dvworks.filters.RedFilter のコードは、リスト5のとおりです。このコードは、単に RedCounter 属性を探し、あれば、それを +1するだけのものです。

リスト5. RedFilter.javaのコード

```
package com.ibm.devworks.filters;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public final class RedFilter implements Filter {
    public static final String COUNTER_ATTRIB = "RedCounter";
    private FilterConfig filterConfig = null;
    public void doFilter(ServletRequest request, ServletResponse response,
```

```
                        FilterChain chain)
  throws IOException, ServletException {
    filterConfig.getServletContext().log("Inside RedFilter");
    Integer curCount = (Integer) request.getAttribute(COUNTER_ATTRIB);
    if (curCount != null)
        request.setAttribute(COUNTER_ATTRIB, new Integer(curCount.intValue() + 1));
      chain.doFilter(request, response);
    }

    public void destroy() {
        this.filterConfig = null;
    }

    public void init(FilterConfig filterConfig) {
  this.filterConfig = filterConfig;
    }
}
```

com.ibm.dvworks.filters.BlueFilter クラスのコードも、リスト6に示すように、定数 COUNTER_ATTRIB を BlueFilter として定義している以外は、 RedFilter とまったく同じです。

リスト6. BlueFilterのコードの変更箇所

```
public static final String COUNTER_ATTRIB = "BlueCounter";
```

filtercount.jspファイルは、ディスパッチャーを使って、included.jspファイルをインクルードします。included.jspファイルは、 BoxColor という入力パラメーターで指定される色で塗られたテーブルを作成するだけのものです。リスト7に示すように、テーブルの色の指定には式言語 (EL) を使用しています。

リスト7. included.jspのコード

```
<table border="1" bgcolor="${param.BoxColor}" cellpadding="5">
      <tr> <td>this page is included in-line using dispatcher</td>
      </tr>
</table>
```

以上のフィルターとJSPページが用意できたところで、次に、フィルターの設定の5通りのシナリオについて、調べていくことにします。

シナリオ1: Servlet 2.3標準のフィルターでのやりとり -- REQUESTのみ

この最初のシナリオでは、旧来のServlet 2.3のコンテナーがサポートしているやり方でフィルターの動作を設定します。

ここで使用する <filter-mapping> エレメントは、リスト8のとおりです。

リスト8. Servlet 2.3標準互換のフィルター・マッピング

```
<filter-mapping>
  <filter-name>Add Attributes Filter</filter-name>
  <url-pattern>/jsp/*</url-pattern>
</filter-mapping>
<filter-mapping>
  <filter-name>Red Filter</filter-name>
  <url-pattern>/jsp/*</url-pattern>
</filter-mapping>
```

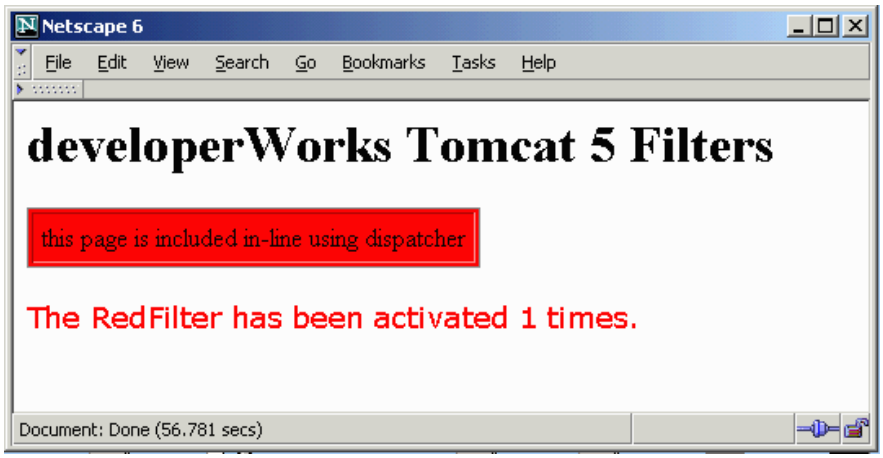
リスト8の設定に合わせてweb.xmlを編集します。このシナリオの場合、 `<dispatcher>` サブエレメントは指定しません。すなわち、Servlet 2.3のコンテナーでそうであったように、REQUESTだけを使ってフィルターがかけられます。リスト8の設定は、働きに関しては、リスト9のコードとまったく同じです。

リスト9. 同等のServlet 2.4のフィルター・マッピング

```
<filter-mapping>
  <filter-name>Add Attributes Filter</filter-name>
  <url-pattern>/jsp/*</url-pattern>
  <dispatcher>REQUEST</dispatcher>
</filter-mapping>
<filter-mapping>
  <filter-name>Red Filter</filter-name>
  <url-pattern>/jsp/*</url-pattern>
  <dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

Tomcat 5を起動し、 `http://<hostname>:8080/dvworks/jsp/filtercount.jsp` というURLでWebアプリケーションにアクセスします。図2は、filtercount.jspから返されてくる画面です。

図2. Servlet 2.3互換のREQUESTのフィルター処理



このfiltercount.jspの応答から、このシナリオでは、 `RedFilter` が1回だけ呼び出されていることがわかります。図3は、相互のやりとりを表した図で、いくつかのリクエスト処理器を通してリクエストが流れる様子を示しています。

図3. Servlet 2.3のフィルター処理におけるリクエスト・フロー

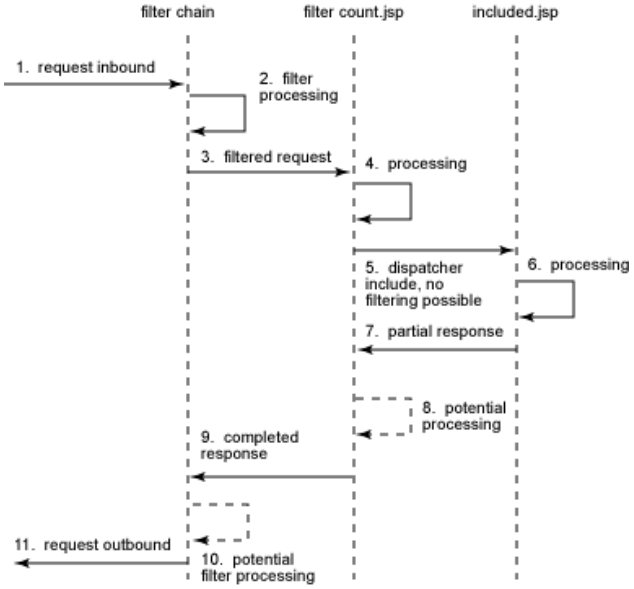
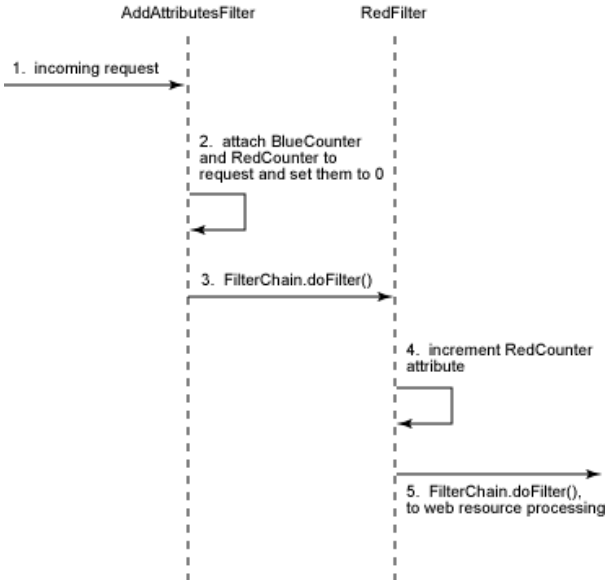


図3では、フィルター・チェーンは、クライアントからのリクエストがこのシステムに入ってきたときに1回だけ呼び出されています。もちろん、このフィルター・チェーンは、図4に示すように、 `AddAttributesFilter` と `RedFilter` からなっています。

図4. フィルター・チェーンの詳細



このServlet 2.3スタイルの設定では、クライアントから直接送られてくるリクエストに対して1回だけ `RedFilter` が適用されます。

シナリオ2: REQUESTとINCLUDEのやりとりの組み合わせ

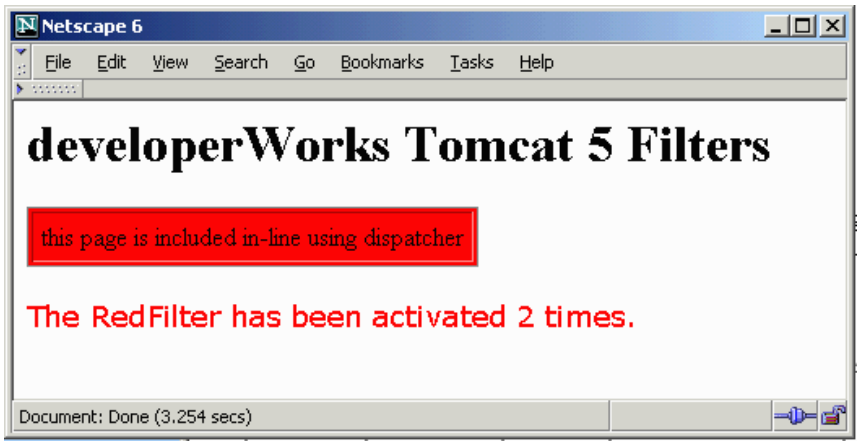
シナリオ2で使用するフィルター・マッピングは、リスト10のとおりです。Servlet 2.4の構文を使用し、 `RedFilter` にINCLUDEディスパッチャー・サブエレメントを追加している点に注意してください。

リスト10. REQUESTとINCLUDEのやりとりを組み合わせる場合のフィルター・マッピング

```
<filter-mapping>
  <filter-name>Add Attributes Filter</filter-name>
  <url-pattern>/jsp/*</url-pattern>
  <dispatcher>REQUEST</dispatcher>
</filter-mapping>
<filter-mapping>
  <filter-name>Red Filter</filter-name>
  <url-pattern>/jsp/*</url-pattern>
  <dispatcher>REQUEST</dispatcher>
  <dispatcher>INCLUDE</dispatcher>
</filter-mapping>
```

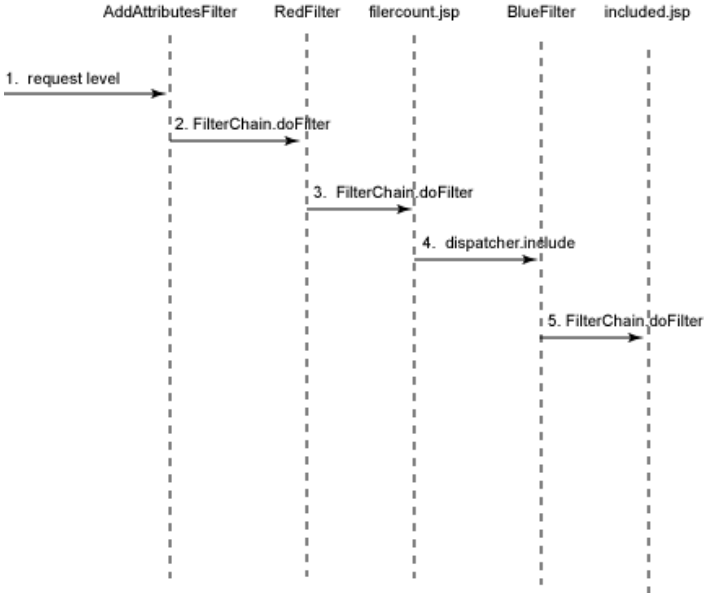
Tomcat 5を再起動し、今度も `http://<hostname>:8080/dvworks/jsp/filtercount.jsp` というURLでアプリケーションにアクセスします。そのときの応答が図5で、フィルター・カウントが表示されています。

図5. REQUESTとINCLUDEによるフィルター処理



今度は、 `RedFilter` が2回呼び出されています。リクエストが送られてきたときに1回、included.jspのインクルードの際にもう1回です。2回の呼び出しを相互のやりとりとして表したのが図6です。ここでは、システムに入るリクエストの流れだけを示すことで、図を簡略化してあります (先の図面と違い、戻りのフローは示していません)。

図6. REQUESTとINCLUDEのやりとりの組み合わせ



シナリオ3: REQUEST、INCLUDE、FORWARDのやりとりの組み合わせ

次に、リスト11に示すforwarder.jspという簡単なJSPページをもう1枚加えます。これは、ディスパッチャーを使って、送られてきたリクエストをfiltercount.jspに転送するというものです。

リスト11. forwarder.jspのコード

```
<jsp:forward page="/jsp/filtercount.jsp"/>
```

今度は、リスト12に示すように、 `BlueFilter` を配備記述子に追加することができます。

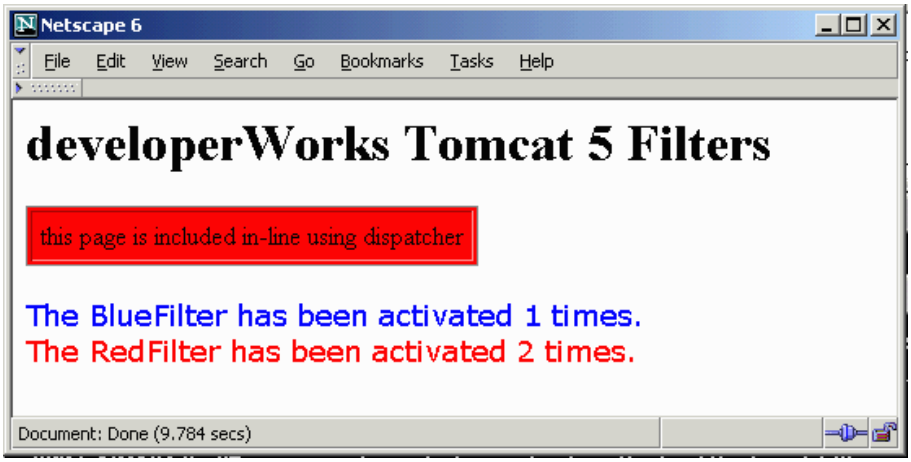
リスト12. REQUEST、INCLUDE、FORWARDのやりとりを組み合わせた場合のフィルター・マッピング

```
<filter-mapping>
  <filter-name>Add Attributes Filter</filter-name>
  <url-pattern>/jsp/*</url-pattern>
  <dispatcher>REQUEST</dispatcher>
</filter-mapping>
<filter-mapping>
  <filter-name>Red Filter</filter-name>
  <url-pattern>/jsp/*</url-pattern>
  <dispatcher>REQUEST</dispatcher>
  <dispatcher>INCLUDE</dispatcher>
</filter-mapping>
<filter-mapping>
  <filter-name>Blue Filter</filter-name>
  <url-pattern>/jsp/*</url-pattern>
  <dispatcher>FORWARD</dispatcher>
</filter-mapping>
```

`BlueFilter` は、転送されてきたリクエストだけを処理するように設定されており、それ以外のフィルター・マッピングに変更はありません。

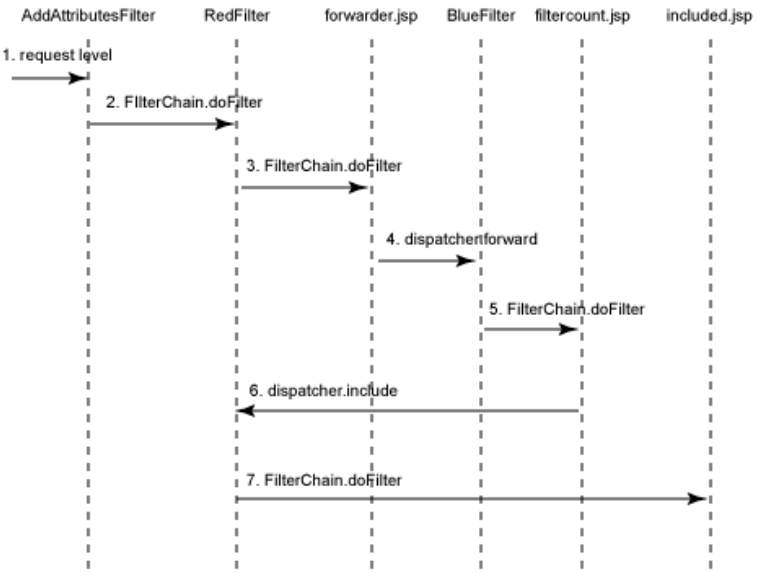
Tomcat 5を再起動して、 `http://<hostname>:8080/dvworks/jsp/forwarder.jsp` というURLでアプリケーションにアクセスします。そのときの応答が図7です。

図7. REQUEST、INCLUDE、FORWARDによるフィルター処理



このシナリオでは、リクエストをforwarder.jspからfiltercount.jspに転送する際に `BlueFilter` が1回呼び出されているのがわかります。図8の相互のやりとりの図が、この流れを示しています。

図8. REQUEST、INCLUDE、FORWARDのやりとりの組み合わせ



シナリオ4: REQUEST、INCLUDE、FORWARDと旧来のServlet 2.3のやりとり

このシナリオでは、前回のフィルター処理の記事で紹介したフィルターを追加します (参考文献 参照)。 `ReplaceTextFilter` を使って、応答の中に表示されるテキストを置き換えます。リスト13に示される `<filter>` の定義に含まれる最初のほうのパラメーターを何個か使って、そういう設定にします。

リスト13. ReplaceTextFilterのフィルター定義でのパラメーターの初期化

```
    <filter>
        <filter-name>RFilter</filter-name>
        <filter-class>com.ibm.devworks.filters.ReplaceTextFilter
        </filter-class>
         <init-param>
      <param-name>search</param-name>
      <param-value>color="red"</param-value>
    </init-param>
        <init-param>
      <param-name>replace</param-name>
      <param-value>color="green"</param-value>
    </init-param>
    </filter>
```

さらに、リスト14に示すように、前のシナリオで使われていたものに `<filter-mapping>` を追加します。

リスト14. REQUEST、INCLUDE、FORWARDと旧来のServlet 2.3のやりとりの組み合わせの場合のフィルター・マッピング

```
<filter-mapping>
  <filter-name>Replace Text Filter</filter-name>
  <url-pattern>/jsp/*</url-pattern>
</filter-mapping>
...
```

このシナリオでは、 `<dispatcher>` サブエレメントを指定していません。したがって、このフィルターは、REQUESTのみの処理を行うものになっています。

Tomcat 5を再起動して、 `http://<hostname>:8080/dvworks/jsp/forwarder.jsp` というURLをアクセスします。すると、図9のような応答が表示されるはずです。これまでの設定と異なり、インクルードされているボックスは緑色で表示されています。

図9. REQUEST、INCLUDE、FORWARD、および旧来のフィルター処理の組み合わせ

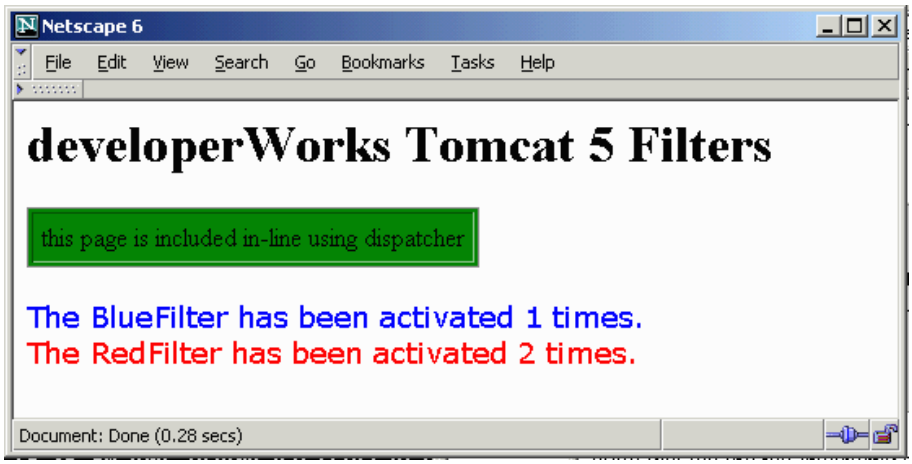
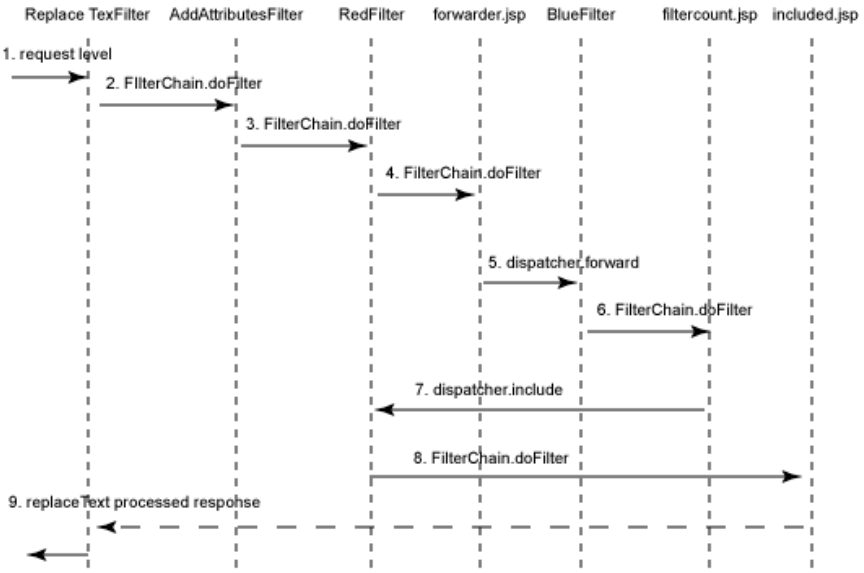


図10の相互のやりとりを表す図には、応答をクライアントに返す前に、最終的な処理を行う目的で、応答が `ReplaceTextFilter` に渡されている様子が示されています。テーブルの色を赤から緑色に変更しているのは、この部分です。

図10. REQUEST、INCLUDE、FORWARD、および旧来のやりとりの組み合わせ



Servlet 2.4のコンテナーは、Servlet 2.3のフィルターや定義やマッピングとの完全な下位互換性を備えています。

シナリオ5: ERRORディスパッチによるフィルター処理

最後のシナリオとして、エラー処理ページにディスパッチする場合のフィルターの使い方を簡単にみておきます。ここで使用するフィルター・マッピングは、リスト15のとおりです。web.xmlから他のすべてのフィルター・マッピングを削除するのを忘れないでください。

リスト15. ERRORを使ったフィルター処理

```
<filter-mapping>
  <filter-name>Red Filter</filter-name>
  <url-pattern>/jsp/*</url-pattern>
  <dispatcher>REQUEST</dispatcher>
  <dispatcher>INCLUDE</dispatcher>
  <dispatcher>ERROR</dispatcher>
</filter-mapping>
```

意図的にエラーを発生させるために、リスト16に示すerrorgen.jspというJSPページを作成します。このページは、エラーを処理するためのerrorhdlr.jspというページをセットアップした後、例外をスローするようになっています。

リスト16. errorgen.jspのコード

```
<%@page errorPage="/jsp/errorhdlr.jsp" %>
<% if (true)
      throw (new Exception("Purposely generated exception!"));
%>
```

このページがエラーを捕捉すると、コンテナーは `@page` 指令の指定にしたがって `/jsp/errorhdlr.jsp` ページにリクエストを転送します。errorhdlr.jspページには、リスト17のようなコードが記述されています。

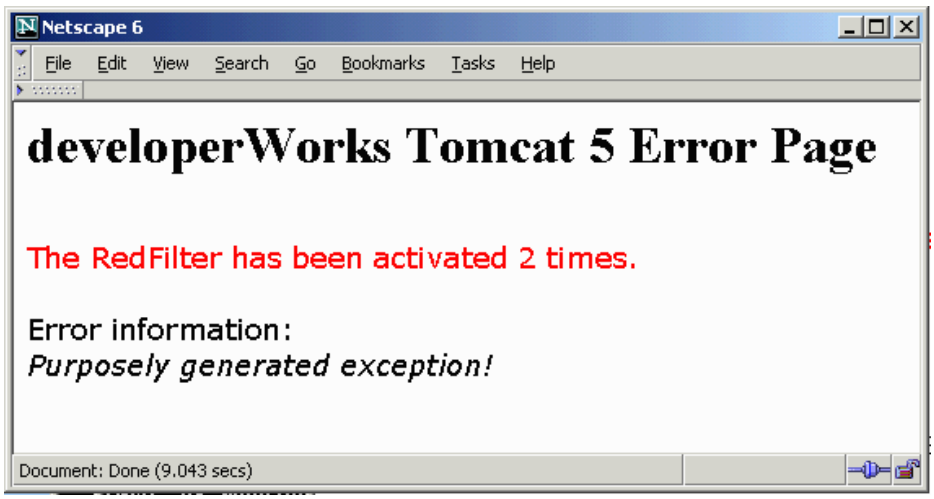
リスト17. errorhdlr.jspのコード

```
<%@ taglib uri="http://java.sun.com/jstl/core_rt" prefix="c" %>
<%@page isErrorPage="true" %>
<HTML>
<HEAD>
</HEAD>
<BODY>
<H1>developerWorks Tomcat 5 Error Page</H1>
<br/>
<FONT FACE="verdana,arial,sans serif" SIZE="4">
<c:if test="${(requestScope.BlueCounter != null) & (requestScope.BlueCounter > 0)}">
  <font color="blue">The BlueFilter has been activated ${requestScope.BlueCounter} times.</font><br/>
</c:if>
<c:if test="${(requestScope.RedCounter != null) & (requestScope.RedCounter > 0)}">
  <font color="red">The RedFilter has been activated ${requestScope.RedCounter} times.</font><br/>
</c:if>
<br/>
Error information:<br/>
<i>${pageContext.errorData.throwable.message}</i>
</FONT>
</BODY>
```

errorhdlr.jspは、フィルターのカウンター値を表示し、捕捉された例外のメッセージをELを使って表示します。

Tomcat 5を再起動し、 `http://<hostname>:8080/dvworks/jsp/errorgen.jsp` というURLをアクセスします。errorgen.jspにアクセスしようとしても、表示されるのはerrorhdlr.jspです (例外が明示的にスローされるため)。表示されるエラー・ページを図11に示します。

図11. REQUESTとERRORによるフィルター処理



`RedFilter` は2回呼び出されています。REQUESTレベルで1回、コンテナーがエラー・ページに転送する際にもう1回です。

Servlet 2.4のフィルターは、エラー処理用のカスタムWebリソースの処理の直前に働かすことができます。

フィルターとモデル2のWebアプリケーション・フレームワーク

Tomcat 5で拡張されたフィルター処理機能のおかげで、一般によく使われているアプリケーション・フレームワークを使用する場合、配備に利用できる新たな機会が得られるようになりました。StrutsやTurbineなどのフレームワーク (参考文献 参照) は、JSPモデル2アーキテクチャーを採用することで、データ処理 (MVCパターンのModel) をビジネス・ロジックの処理やプレゼンテーションの処理 (View) から分離しています。そうしたフレームワークは、静的なXMLの指令ファイルによるか、実行時イントロスペクションを使用するかして、送られてきたリクエストをユーザー定義のさまざまなActionコンポーネントに転送するスイッチング・サーブレット (Controller サーブレット) を中心にして機能を構成しています。これらのリクエストは、通常、応答が作成されるまで、コンポーネント間でさらに転送が繰り返されます。モデル2アーキテクチャーでは、サーバーサイドのコーディングをコンポーネント化することで、変化に対応したり、長期的な保守を容易なものにすることが可能になります。

アプリケーション・コンポーネント間だけでなく、コントローラー・サーブレットとアプリケーション・コンポーネントの間でリクエストを転送する場合にも、ディスパッチャーは幅広く利用できますので、フィルターを処理フローに組み入れるのは容易です。Servlet 2.4の機能拡張とともに、フィルターは、今では、モデル2のアプリケーション・フレームワークを使う場合、Webアプリケーションのコンポーネント・ミックスに不可欠な部品になりつつあります。

アプリケーションをパイプライン化して高い性能を目指す

ハードウェア領域から捉えたアーキテクチャー・パターン

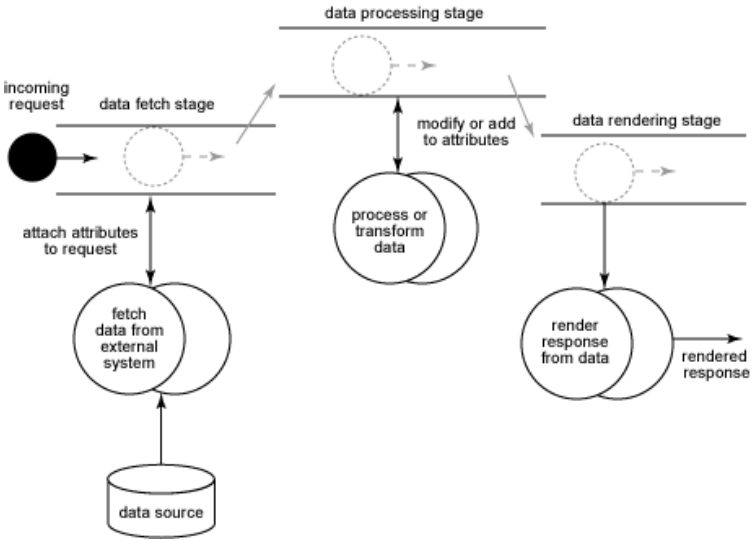
サーバーをクラスター化して目いっぱいの規模のWebアプリケーションを実行させるという新しい考え方は、マイクロプロセッサーの考え方からきています。送られてくるリクエストによって起こり得る処理の集合は、CPUの命令セットと同様、有限です。サーバーでのリクエストの処理は、ハードウェア・コアでの命令の実行に対比させることができます。

パイプライン化というのは、コンポーネント化されたリクエスト処理をリクエスト・フローを中心にして捉えることを意味します。パイプライン化する場合、リクエストがアプリケーション・サーバーの中を流れる際、その1個のリクエストを何段階かの処理に分けて捉えます。

パイプライン・モデルでは、リクエストが生産ライン (パイプライン) を通って流れていき、最終的に応答が作成され、クライアントに返送されるまでの各段階で、リクエストが一連の処理器によって修飾/変換/加工されていくものと考えます。Jakarta StrutsやTurbineなどのよく使われているWebアプリケーション・フレームワークは、それぞれJSPモデル2アーキテクチャーの利点を活かして、すでに、パイプライン方式でサーバーサイドのロジックをコンポーネント化しています。

図12は、一般的なパイプラインの各段階を示したもので、Webアプリケーションやサービス・リクエストは、これらの段階を経ることで、データのページを表示したりします。

図12. パイプライン・モデルの一般的な段階



要するに、リクエストは、表3に示すパイプラインの3つの段階を経るわけです。

表3. パイプライン・モデルの各段階

パイプラインの段階の名前	通常実行される処理
データ・フェッチ (Data fetch)	送られてきたリクエストの情報にしたがって、外部システム (JDBCデータ・ソースなど) からデータを読み出してくる
加工 (Processing)	データ・エレメントの値を使って内部または外部のソースなどから関連するデータを読み出してきて、データ・エレメントの妥当性チェックあるいは変換を行う
出力 (Rendering)	加工されたデータを基に、通常、表示用フォーマット (HTMLなど) または交換用フォーマット (XMLなど) の形で、出力応答を作成する

設計の主眼をリクエスト処理のパイプラインに置くことで、Webアプリケーションの設計は、性能面での調整や最適化の可能性を追求する新しい時代に入りつつあります。最近のマイクロプロセッサーのハードウェア設計に利用されている (パイプラインの最適化に関する) 技法の多くは、近い将来、Webアプリケーション・サーバーのソフトウェアやハードウェアの最適化にも活用されるようになるはずです。

たとえば、データ・フェッチの段階を通るすべてのリクエストに対する処理仕様が同じであるWebアプリケーションの場合、最適化されたアプリケーション・サーバー/ハードウェア・プラットフォームなら、非常によくアクセスされるデータについてはプリフェッチしたり、ローカルにキャッシュしておくことで、送られてくるほとんどのリクエストについて、データ・フェッチのオーバーヘッドを完全に無くしてしまうことができます。また、XSLTの変換やXMLの構文解析など、出力段階において相当なCPUの処理資源を要求するリクエストの場合、パイプラインの出力段階に、ハードウェアで速度向上を図ったXML処理器のパラレル・バンクを設けることも考えられます。こうした最適化は、今実現できるかどうかと言えば、少し先走りしている観はありますが、そうしたものを現実のものとするためには、パイプライン・モデルを考慮してWebアプリケーションを設計することが必要な要素となっています。

まとめ

Tomcat 5の新しいフィルター処理機能を利用すれば、Webアプリケーションのリクエスト処理フローのすべての段階に介入できるようになります。この機能によって、アプリケーション・フレームワークでのフィルターの利用が容易になり、性能を最適化したり調整するという新しい可能性が開かれることになります。

著者について

Sing Li

Sing LiはWrox Pressから出版されている多数の本の著者で、Professional Apache Tomcat 、Early Adopter JXTA 、Professional Jini などを執筆しています。技術雑誌に頻繁に寄稿しており、P2P発展に関する熱心なエバンジェリスト（伝道者）でもあります。