Tomcatにおけるフィルタリング機能の活用

Servlet 2.3仕様へのフィルター機能の追加によりJ2EEアプリケーションのパフォーマンスが向上

Sing Li

2001年 6月 01日

新しいJava Servlet 2.3仕様における画期的な機能の1つとして、フィルタリング機能が挙げられます。一見したところ、Servlet2.3のフィルタリングは、Apache、IIS、Netscape Webサーバーその他の従来のフィルタリングと似ているように見えます。しかし実際は、Servlet2.3のフィルタリングは構造的にまったく異なったデザインを持っており、オブジェクト指向の性質を持つJavaプラットフォームによって新しいレベルのパフォーマンスを実現します。この記事では、Tomcat4におけるフィルタリングを紹介し、皆さんのプロジェクトにおけるフィルターの活用方法を示します。

フィルタリングは、Tomcat 4の新機能です (Tomcatに関する説明は「The Tomcat story」を参照してください)。フィルタリングはServlet 2.3仕様の一部であり、今後、この標準をサポートするすべてのJ2EEコンテナー・ベンダーがフィルターの実装を行うことになるでしょう。フィルターを使用することにより、開発者は、これまで実装にうまい方法が無かった、あるいは実装が難しかった次のような機能の実装が可能となります。

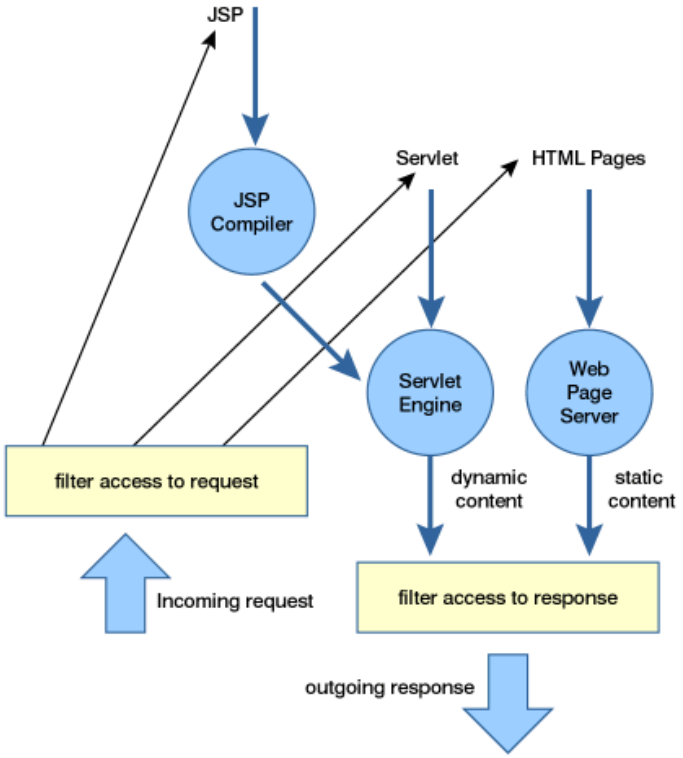
- リソース・アクセス (Webページ、JSPページ、サーブレットなど) のための認証のカスタマイズ
- アプリケーション・レベルでのリソース・アクセスの監査 / ロギング
- カスタマイズされた暗号化スキームに基く、アプリケーション全体におけるリソースへのアクセスの暗号化
- サーブレットやJSPからの動的アウトプットを含む、アクセスされるリソースのオンザフライの変換

当然、これがすべての機能ではありませんが、これらの機能から、フィルタリングがどのような付加価値を与えるかが大体分かるはずです。この記事では、Servlet2.3のフィルタリングを詳細に検討します。まず、J2EEのプロセッシング・モデルにフィルターがどのように組み込まれるかを見ます。Servlet2.3のフィルタリングはネストされた呼び出しに基づくものであり、従来の他のフィルタリング・スキームとは異なります。このような違いが、新しい高性能なTomCat4のデザインを構造的にどのように説明しているかを見ていきます。そして最後に、2つのServlet2.3フィルターのコード化 / テストを実際に行ってみましょう。これらのフィルター機能はきわめて簡単なものなので、フィルターのコーディング方法やWebアプリケーションへの組み込み方法に重点をおいてじっくりご覧いただけます。

Webアプリケーションの基本的要素としてのフィルター

物理的には、フィルターは、J2EE Webアプリケーションにおけるアプリケーション・レベルのJavaコード・コンポーネントです。サーブレットやJSPページの他、Servlet2.3仕様でのコーディングを行っている開発者は、Webアプリケーションにアクティブな振る舞いを追加するメカニズムとしてフィルターを使用することができます。特定のURLに限定されるサーブレットやJSPページとは異なり、フィルターは、J2EEコンテナーのプロセッシング・パイプラインに組み込まれ、Webアプリケーションで提供されるURLのサブセット(あるいはすべてのURL ）全体で使うことができます。図1は、J2EEのリクエスト処理においてフィルターがどのような位置に置かれるかを示しています。

図1. フィルターとJ2EEリクエスト処理



Servlet 2.3対応のコンテナーでは、フィルターは、Webリクエストの (Servletエンジンによって)処理「前」と処理「後」 (フィルターがレスポンスにアクセスできる) のどちらでもWebリクエストにアクセスできます。

この時、フィルターは以下のことを行うことができます。

- リクエスト処理の前に、リクエスト・ヘッダーを変更する
- 処理されるべき独自のリクエストを提供する
- リクエストの処理後、ユーザーに返す前にレスポンスを変更する
- コンテナーによるリクエスト処理をすべて無効にし、独自のレスポンスを作成する

フィルターが使用できるようになる以前は、J2EEのプロセッシング・パイプラインへの介入には、移植性がなく、コンテナー固有で、システム全体に対する拡張メカニズム(Tomcat 3インターセプターなど) が必要でした。

Tomcatにおけるフィルタリングの概念

Apache、IIS、Netscapeサーバーなどの一般的なフィルタリング・メカニズムと異なり、Servlet2.3フィルターはフック(hook)関数の呼び出しに基づくものではありません。実際、Tomcat4レベルのエンジン・アーキテクチャーは従来のTomcat 3.xとは一線を画するものです。新しいTomcat4エンジンは、1つの大きなエンジンによってリクエスト処理の各段階でフック・メソッドの呼び出しを行うのではなく、ネストされた呼び出しのフローとラップされたリクエストおよびレスポンスを内部で使用します。そして、別々のフィルターとリソース・プロセッサーによって一つのチェーンを形成します。

従来のアーキテクチャーでは

- 設定にかかわらず (設定していない場合でも)、フック・メソッドの呼び出しが、リクエストごとに行われます。
- メソッドの範囲設定や並行性の問題 (各メソッドが異なるスレッドで呼び出される場合もある)により、同じリクエストを処理する、異なるフック・メソッド間での変数や情報の共有を簡単あるいは効率的に行うことができません。

新しいアーキテクチャーでは

- ネストされたメソッド呼び出しは、フィルターのチェーンを通して行われますが、このチェーンは現在のリクエストに適用されるフィルターのみで構成されます。フック呼び出しに基づく従来の方法では、すべての処理段階(特定のフレーズに対するプロセッシング・ロジックが何も行わない場合でも)におけるフック・ルーチンへの呼び出しが必要です。
- ローカル変数は、実際のフィルタリング・メソッドが戻るまで保存され、利用することができます (アップストリームのフィルターへの呼び出しは常にスタックに置かれ、ダウンストリームの呼び出しが戻るのを待っているため) 。

この新しいアーキテクチャーは、将来におけるTomcatのパフォーマンス調整 /最適化のための、新しく、よりオブジェクト・フレンドリーな基盤を提供します。Servlet 2.3のフィルターは、この新たな内部アーキテクチャーを拡張したものです。このアーキテクチャーによって、Webアプリケーション・デザイナーは、フィルタリングの実装を移植性を持たせて行えるようになります。

呼び出しのチェーン

すべてのフィルターは、明確に定義されたインターフェースを通して実行される、フィルター呼び出しのチェーンに従います。フィルターを実装しているJavaクラスには、この`javax.servlet.Filter` インターフェースの実装が必要です。このインタフェースはフィルターが実装する必要のある３つのメソッドを含みます。

- **`doFilter(ServletRequest, ServletResponse, FilterChain)`**: このメソッドにおいてフィルター・アクションが実行されます。これはまた、アップストリームのフィルターが呼び出すメソッドでもあります。また送られる`FilterChain` オブジェクトは、呼び出しを行うべきダウンストリームのフィルターに関する情報を提供します。
- **`init(FilterConfig)`**: これはコンテナーが呼び出す初期化メソッドです。最初の`doFilter()` の呼び出しの前に、コンテナーによる呼び出しが必ず行われます。ここではweb.xmlファイルで指定された初期化パラメーターを得ることができます。

- **destroy()**: フィルター・インスタンスに対するdoFilter() 内のすべてのアクティビティーが停止した後、コンテナーは、このインスタンスを破壊する前にこのメソッドの呼び出しを行います。

注意 : 最近のベータ・サイクルでは、Filter インターフェースのメソッド名と意味が変わっています。Servlet 2.3仕様は、まだドラフトの最終段階に至っていません。Beta1では、インターフェースにinit() とdestroy() に代わってsetFilterConfig() とgetFilterConfig() のメソッドがあります。

ネストされた呼び出しの連鎖は、doFilter() メソッドのインプリメンテーションの中で行われます。(他のフィルターおよびリソース・プロセッサーによる) すべてのダウンストリームの処理を明示的にブロックするようにフィルターを設定する場合以外は、フィルターはdoFilter メソッドで以下のような呼び出しを行う必要があります。

```
FilterChain.doFilter(request, response);
```

フィルターのインストール : 定義およびマッピング

コンテナーは、配置ディスクリプターであるweb.xmlファイルからWebアプリケーションのフィルターに関する情報を得ます。<filter> と<filter-mapping> という、フィルターに関連した2つ新しいタグがあります。これら2つのタグは<web-app> タグの子としてweb.xmlファイル内で定義する必要があります。

フィルター定義のエレメント

<filter> タグはフィルターの定義に用いられ、<filter-name> および<filter-class> というサブ・エレメントを必要とします。サブ・エレメント<filter-name> は、このフィルターのインスタンスに関連したテキストで表現された名称です。<filter-class> はコンテナーによってロードされる実際のクラスを指定します。また必要に応じて、<init-param> サブ・エレメントを置き、フィルター・インスタンスに初期化パラメーターを与えることもできます。たとえば、以下はIE Filter と呼ばれるフィルターの定義です。

リスト1. フィルターを定義するタグ

```
 <web-app>
   <filter>
        <filter-name>IE Filter</filter-name>
        <filter-class>com.ibm.devworks.filters.IEFilter</filter-class>
   </filter>

</web-app>
```

コンテナーがweb.xmlファイルを処理する場合、通常、検索されたフィルター定義ごとにフィルターのインスタンスを1つ作成します。このインスタンスは、適用可能なあらゆるURLリクエストに対して使用されます。そのため、スレッド・セーフな方法によるフィルターのコード化が最も重要となります。

フィルター・マッピングとサブ・エレメント

<filter-mapping> タグはフィルター・マッピングを示すもので、フィルタリングの対象となるURLのサブセットを指定します。このサブセットには、マップされるフィルターの定義に対応し

た<filter-name> サブ・エレメントが必要です。次に、<servlet-name> か<url-pattern> というサブ・エレメントのいずれかを使用してマッピングの指定を行います。<servlet-name> は、このフィルターが適用されるサーブレット (web.xmlファイルのどこかで定義されている)を指定します。また、<url-pattern> によって、フィルターを適用するURLサブセットを指定することができます。たとえば、/*というパターンは、そのアプリケーションでサポートされるすべてのURLにフィルターのマッピングが適用されることを示します。また、/dept/humanresources/*というパターンは、フィルターのマッピングが人事部固有のURLのみに適用されることを示します。

コンテナーはこのようなフィルター・マッピングを使用して、特定のリクエストに対して特定のフィルターを用いるべきかどうかを判断します。以下は、リスト1で定義されたIE Filter をアプリケーションのすべてのURLに適用するフィルター・マッピングです。

リスト2. フィルター・マッピング・タグ

```
<filter-mapping>
        <filter-name>IE Filter</filter-name>
        <url-pattern>/*</url-pattern>
</filter-mapping>
```

簡単なフィルターの作成

それでは、最初のフィルターのコーディングを行いましょう。これは、リクエスト・ヘッダーを調べてInternetExplorerのブラウザーがURLの表示に使用されているかどうかを判断する簡単なフィルターです。InternetExplorerのブラウザーが使用されている場合、フィルターは「アクセス拒否」メッセージを表示します。簡単な機能ですが、この例は以下のことを示しています。

- フィルターの大まかな構造
- リソース・プロセッサーに届く前にヘッダー情報を調べるフィルター
- ランタイム検出条件 (認証パラメーター、IPの発信、日時など) に基づきダウンストリームの処理を停止させるフィルターのコーディング方法

このフィルターのソース・コードは、ソース・コード・ディストリビューション(参考文献を参照)内に、com.ibm.devworks.filters パッケージのIEFilter.java として置かれています。次に、このフィルターのコード全体を見てみましょう。

リスト3.Filter インターフェースの実装

```
public final class IEFilter implements Filter {
    private FilterConfig filterConfig = null;
```

すべてのフィルターには、Filter インターフェースの実装が必要です。フィルターの初期設定を行う際にコンテナーによって渡されるfilterConfig を組み込むローカル変数を作成します。これは、doFilter() に対する最初の呼び出しの前のある時点で行われます。

リスト4.`doFilter` メソッド

```
    public void doFilter(ServletRequest request, ServletResponse response,
                         FilterChain chain)
        throws IOException, ServletException {
      String browserDet = ((HttpServletRequest) request).getHeader("User-Agent").toLowerCase();
      if ( browserDet.indexOf("msie") != -1)  {
            PrintWriter out = response.getWriter();
            out.println("<html><head></head><body>");
            out.println("<h1>Sorry, page cannot be displayed!</h1>");
            out.println("</body></html>");
            out.flush();
            return;
          }
```

ほとんどの作業が`doFilter()` において行われます。「User-Agent」ヘッダーと呼ばれるリクエスト・ヘッダーを見てみましょう。このヘッダーはあらゆるブラウザーが供給します。このヘッダーを小文字に変換し、次に、目印となる「msie」という識別ストリングを探します。InternetExplorerのブラウザーが検出された場合、レスポンスを書き出すためにレスポンス・オブジェクトから`PrintWriter` を取得します。カスタム・レスポンスの書き出し後、メソッドは、別のフィルターに連鎖することなく、そのまま戻ります。このように、フィルターによってダウンストリームの処理をブロックすることができます。

ブラウザーがInternet Explorerではない場合には、そのまま通常のチェーンに従い、ダウンストリームのフィルターとプロセッサーはリクエストに応じることができます。

リスト5. 通常のチェーンを行う

```
      chain.doFilter(request, response);
    }
```

次いで、`init()` と`destroy()` のメソッドをそのままフィルターに組み込みます。

リスト6.`init()` メソッドと`destroy()` メソッド

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

`IEFilter` をテストする

Tomcat 4 beta 3 (またはそれ以降) がインストール済みで動作可能な状態である場合には、以下のステップに従って`IEFilter` を稼働させてください。

1. 以下のように、$TOMCAT_HOME/confディレクトリーのserver.xmlファイルに新たなアプリケーション・コンテキストを作成します。

```
      <!-- Tomcat Examples Context -->
       <Context path="/examples" docBase="examples" debug="0"
                reloadable="true">
      ...
        </Context>
       <Context path="/devworks" docBase="devworks" debug="0"
                reloadable="true">
         <Logger className="org.apache.catalina.logger.FileLogger"
                    prefix="localhost_devworks_log." suffix=".txt"
                timestamp="true"/>
        </Context>
```

2. コード・ディストリビューションのdevworks/WEB-INFディレクトリーにあるweb.xmlファイルを編集し、以下のフィルター定義とマッピングを組み込みます。

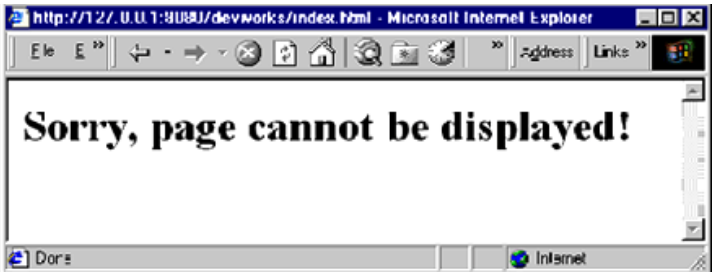
```
<web-app>
    <filter>
        <filter-name>IE Filter</filter-name>
        <filter-class>com.ibm.devworks.filters.IEFilter</filter-class>
    </filter>
     <filter-mapping>
        <filter-name>IE Filter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

3. $TOMCAT_HOME/webappsディレクトリーにdevworksという新しいディレクトリーを作成し、ソース・コード・ディストリビューションから、devworksディレクトリー(すべてのサブ・ディレクトリーを含む) にあるすべてのものをそこへコピーします。これでTomcat4の使用を開始する準備が整いました。
4. 以下のURLで、サンプルのindex.htmlページにアクセスしてください。

   `http://<hostname>/devworks/index.html`

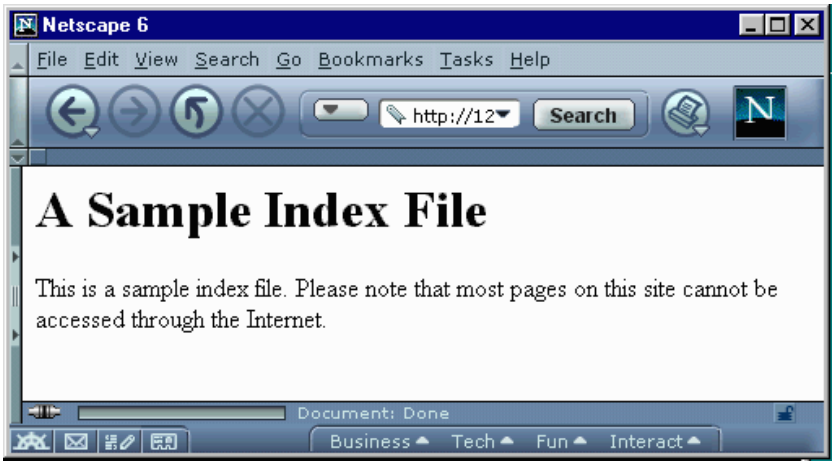
   Internet Explorerを使用している場合、図2に示されるような「アクセス拒否」のカスタム・メッセージが表示されます。

   図2. Internet Explorerでの`IEFilter`

   Netscapeを使用している場合、図3に示されているように、実際のHTMLページが表示されます。

図3. Netscapeブラウザーでの`IEFilter`のパススルー



リソース変換を行うフィルターを記述する

次は、より複雑なフィルターを試してみましょう。このフィルターは、以下のことを行います。

- フィルター定義の中のインスタンス初期化パラメーターから、一組の「検索」「置換」テキストを読み出す
- アクセスが行われているURLをフィルターにかけ、最初に検出された「検索」テキストを「置換」テキストに置きかえる

このフィルターの作業を行うにつれて、皆さんは、コンテンツ変換 / 置換フィルターの構造がよく分かるようになるでしょう。あらゆる暗号化、圧縮、変換(XSL変換 - XSLTによるXMLなど) のフィルターにおいても、同様の構造を使用することができます。

その秘密は、レスポンス・オブジェクトのカスタマイズしラップしたバージョンをチェーンのダウンストリームに渡すことです。このカスタム・ラッパーのレスポンス・オブジェクトは、オリジナルのレスポンス・オブジェクトを隠し(ラップし)、ダウンストリーム・プロセッサーが書き込みを行うカスタム・ストリームを提供しなければなりません。作業(テキストの置換、変換、圧縮、暗号化など) をオンザフライで行える場合、カスタム・ストリームが、ダウンストリームからの書き込みをインターセプトして必要な作業を行います。次いで、カスタム・ストリームは、変換されたデータをラップされたレスポンス・オブジェクトに書き込みます(つまり、簡単な文字置換による暗号化を行う)。作業をオンザフライで行えない場合、ダウンストリーム・プロセッサーがストリームへの書き込みを完了するまで(つまり、ストリームを閉じるかフラッシュするまで) カスタム・ストリームは待機しなければなりません。その後、カスタム・ストリームは変換を実行し、変換されたアウトプットを「実際の」レスポンスに書き込みます。

このフィルター (`ReplaceTextFilter`) では、カスタマイズされたラッパーのレスポンス・オブジェクトは`ReplaceTextWrapper`と呼ばれます。また、カスタム・ストリームのインプリメンテーションは`ReplaceTextStream`と呼ばれます。ソース・コードは、`com.ibm.devworks.filters`パッケージのReplaceTextFilter.javaにあります (参考文献を参照)。では、ソース・コードを見てみましょう。

リスト7.`ReplaceTextStream` クラス

```
class ReplaceTextStream extends ServletOutputStream {
     private OutputStream intStream;
     private ByteArrayOutputStream baStream;
     private boolean closed = false;
     private String origText;
     private String newText;
     public ReplaceTextStream(OutputStream outStream, String searchText, String replaceText) {
          intStream = outStream;
          baStream = new ByteArrayOutputStream();
          origText = searchText;
          newText = replaceText;
     }
```

これは、カスタマイズされたアウトプット・ストリームのためのコードです。`intStream` 変数は、レスポンス・オブジェクトから実際のストリームへの参照を含んでいます。`baStream` は、ダウンストリーム・プロセッサーが書き込みを行うアウトプット・ストリームのバッファー化されたバージョンです。`closed`フラグは、そのストリーム・インスタンスにおいて`close()` がすでに呼び出されているかどうかを示しています。コンストラクターは、レスポンス・オブジェクトからのストリームへの参照を保存し、バッファー・ストリームを作成します。また、後の置換作業のためのテキスト・ストリングも保存します。

リスト8.`write()` メソッド

```
    public void write(int i) throws java.io.IOException {
         baStream.write(i);
     }
```

`ServletOutputStream` から (クラスを) 派生させるなかで、独自の`write()` メソッドを用意する必要があります。もちろんここでは、バッファーされたストリームに対して書き込みを行います。ダウンストリーム・プロセッサーからのより高レベルのアウトプット・メソッドは、このメソッドを最も低いレベルとして使用し、それにより、バッファーされたストリームに対してすべての書き込みが確実に行われるようにします。

リスト9.`close()` と`flush()` メソッド

```
    public void close() throws java.io.IOException {
          if (!closed) {
            processStream();
            intStream.close();
            closed = true;
          }
       }
    public void flush() throws java.io.IOException {
        if (baStream.size() != 0) {
             if (! closed) {
              processStream();           // need to synchronize the flush!
              baStream = new ByteArrayOutputStream();
              }
         }
       }
```

`close()` メソッドと`flush()` メソッドが、変換を行う場所です。使用されるダウンストリーム・プロセッサーによりますが、呼び出されるメソッド(どちらか、または両方) が決まります。こ

こでは異常な状況を避けるために、ブール値の`closed`フラグを使用します。実際の置換作業を`processStream()` メソッドに委譲したことに注意してください。

リスト10.`processStream()` メソッド

```
    public void processStream() throws java.io.IOException {
         intStream.write(replaceContent(baStream.toByteArray()));
         intStream.flush();
     }
```

`processStream()` メソッドは、`baStream` から、保存している`intStream` に対して、変換されたアウトプットを書き込みます。変換作業は`replaceContent()` メソッドとして分離されます。

リスト11.`replaceContent()` メソッド

```
  public byte []  replaceContent(byte [] inBytes) {
          String retVal ="";
          String firstPart="";
        String tpString = new String(inBytes);
          String srchString = (new String(inBytes)).toLowerCase();
          int endBody = srchString.indexOf(origText);
 if (endBody != -1) {
              firstPart = tpString.substring(0, endBody);
           retVal = firstPart + newText + tpString.substring(endBody + origText.length());
            } else {
              retVal=tpString;
            }
                       return retVal.getBytes();
    }
}
```

`replaceContent()` で、検索および置換が行われます。それは、1つのバイト配列をインプットとし、また、1つのバイト配列を戻します。これにより、きわめて整った概念的インターフェースを形成します。実際、このメソッドのロジックを換えることによって、どのようなタイプの変換でも実行できます。ここで、非常に簡単なテキスト置換を実行してみましょう。

リスト12.`ReplaceTextWrapper` クラス

```
class ReplaceTextWrapper extends HttpServletResponseWrapper {
    private PrintWriter tpWriter; private ReplaceTextStream tpStream;
    public ReplaceTextWrapper(ServletResponse inResp, String searchText,
                              String replaceText) throws java.io.IOException { super((HttpServletResponse)
 inResp);
         tpStream = new ReplaceTextStream(inResp.getOutputStream(), searchText, replaceText);
         tpWriter = new PrintWriter(tpStream);
    }
    public ServletOutputStream getOutputStream() throws java.io.IOException {
            return tpStream;
     }
    public PrintWriter getWriter() throws java.io.IOException {
            return tpWriter;
     }
}
```

カスタム・ラッパーのレスポンス (オブジェクト) は、便利なことにヘルパー・クラスの`HttpServletResponseWrapper` から定義することができます。このクラスは、メソッドの簡単な実装をすでに数多く行っているため、使う側としては、単に、カスタム・アウトプット・スト

リームのインスタンスを用意して、`getOutputStream()` メソッドおよび`getWriter()` メソッドの再定義をするだけでよいわけです。

リスト13.`ReplaceTextWrapper()` メソッド

```
public final class ReplaceTextFilter implements Filter {
    private FilterConfig filterConfig = null;
    private String searchText = ".";
    private String replaceText = ".";
    public void doFilter(ServletRequest request, ServletResponse response,
                         FilterChain chain)
        throws IOException, ServletException {
     ReplaceTextWrapper myWrappedResp = new ReplaceTextWrapper( response, searchText, replaceText);
         chain.doFilter(request,  myWrappedResp);
         myWrappedResp.getOutputStream().close();  }
    public void destroy() {
    }
```

では最後に、フィルターそのものを見てみましょう。ここで示されるように、フィルターは`FilterChain` を使用してレスポンスのダウンストリームを扱うためのカスタム・ラッパーのレスポンス・インスタンスを生成しているだけです。

リスト14. カスタム・ラッパーのレスポンス・インスタンスを生成する

```
    public void init(FilterConfig filterConfig) {
      String tpString;
      if (( tpString = filterConfig.getInitParameter("search") ) != null)
           searchText = tpString;
      if (( tpString = filterConfig.getInitParameter("replace") ) != null)
           replaceText = tpString;
        this.filterConfig = filterConfig;
    }
}
```

この`init` メソッドでは、フィルター定義で指定された初期パラメーターを検索します。その実行には、`filterConfig` オブジェクトの`getInitParameter()` メソッドを使用すると便利です。

`ReplaceTextFilter`

前述の手順に従って`IEFilter` のテストを行い、すべてのファイルを $TOMCAT/webapps/devworks にコピーしたと仮定すると、次のような手順で`ReplaceTextFilter` をテストすることができます。

1. $TOMCAT/wepapps/devworks/WEB-INFディレクトリーにあるweb.xmlファイルを編集し、以下のフィルター定義とマッピングを組み込んでください。

```
<web-app>
<filter>
 <filter-name>Replace Text Filter</filter-name>
 <filter-class>com.ibm.devworks.filters.ReplaceTextFilter</filter-class>
 <init-param>
    <param-name>search</param-name>
    <param-value>cannot</param-value>
 </init-param>
 <init-param>
    <param-name>replace</param-name>
    <param-value>must not</param-value>
 </init-param>
</filter>

<filter-mapping>
    <filter-name>Replace Text Filter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
</web-app>
```

2. Tomcatを再スタートしてください。
3. 次に、以下のURLでindex.htmlページにアクセスしてください。
   `http://<host name>:8080/devworks/index.html`

`ReplaceTextFilter` によって「cannot」という言葉がオンザフライでどのように「must not」に変換されたかに注目してください。「cannot」という文字列を含むアウトプットを持つJSPページあるいはサーブレットの記述を行い、フィルタリングがすべてのリソースに関して可能であることを確認することもできます。

フィルター・チェーンの順序付けの重要性

フィルター・チェーンの順序はweb.xmlディスクリプター内の`<filter-mapping>` ステートメントの順序によって決定されます。ほとんどの場合、フィルター・チェーンの順序は重要です。つまり、フィルターBの前にフィルターAを適用する場合と、フィルターAの前にフィルターBを適用する場合とでは、まったく異なった結果が得られるということです。1つのアプリケーションで複数のフィルターを使用する場合、`<filter-mapping>` ステートメントの入力は慎重に行う必要があります。

web.xmlファイル内の`<filter-mapping>` を以下のようにアレンジすることによって、簡単にその効果を確認することができます。

リスト15. フィルタリングの順序 -`IE Filter` から開始する

```
<web-app>
<filter-mapping>
        <filter-name>IE Filter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
<filter-mapping>
        <filter-name>Replace Text Filter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

次に、index.htmlページをInternet Explorerにロードしてください。 `IE Filter` がチェーンの最初のフィルターであるため、`Replace Text Filter` は実行されません。したがって、「Sorry, page cannot be displayed!」というメッセージが出力されます。

次に、`<filter-mapping>` タグの順序を以下のように逆にしてください。

リスト16. フィルタリングの順序 -`Replace Text Filter` を最初に置換する

```
<web-app>
<filter-mapping>
        <filter-name>Replace Text Filter</filter-name>
        <url-pattern>/*</url-pattern>
</filter-mapping>
<filter-mapping>
        <filter-name>IE Filter</filter-name>
        <url-pattern>/*</url-pattern>
</filter-mapping>
</web-app>
```

index.htmlページをInternet Explorerに再度ロードします。今回は、`Replace Text Filter` は最初の入り口として実行し、ラップされたレスポンス・オブジェクトを`IE Filter` に与えます。`IE Filter` によるカスタム・レスポンスの書き込みの後、エンド・ユーザーに届く前に、特化されたレスポンス・オブジェクトによってアウトプットの変換が行われます。したがって、今回は「Sorry,page must not be displayed!」というメッセージが表示されます。

アプリケーションにフィルターを組み込む

本稿の執筆にとりかかる際、Tomcat 4はベータ・サイクルの最終段階にあり、公式なリリースを間近に控えていました。J2EEコンテナーの大手ベンダーはすべて、自社製品にServlet2.3仕様を組み込む準備を整えています。Servlet 2.3フィルターの仕組みの基礎を理解することにより、J2EEベースのアプリケーション設計/ コーディングに使うことのできる汎用ツールがもう 1 つ増えることになるでしょう。

関連トピック

- 本記事で使用したソース・コードをダウンロードすることができます。
- Tomcatのホームページでは、入手可能なTomcatのすべてのバージョンに関する情報を提供しています。フィルタリングに関する最新のドキュメンテーションについては、このページのドキュメンテーション・リンクを定期的にチェックしてください。
- 開発者のニーズに応えるためのJSPのテクノロジーの変化について学びましょう (developerWorks、2001年6月)。
- developerWorksのJavaゾーンでは、Javaに関する参考文献が多数提供されています。