chtblyl0920052505的博客文章

作者: chtblyl0920052505
http://chtblyl0920052505.iteye.com

我的博客文章精选

目 录

1. jsp/servlet

1.1 j2ee 、j2se、jsp/servlet、jstl、tomcat version

发表时间: 2016-01-01 关键字: jsp, servlet

以下部分来自tomcat官网（地址->http://tomcat.apache.org/whichversion.html）：

servlet	jsp	el	websocket	tomcat	j2ee	j2se
4.0	TBD (2.4?)	TBD (3.1?)	TBD (1.2?)	9.0.x	8+	8+
3.1	2.3	3.0	1.1	8.0.x	7+	7+
3.0	2.2	2.2	1,1	7.0.x	6+	6+
2.5	2.1	2.1	N/A	6.0.x	5+	5+
2.4	2.0	N/A	N/A	5.5.x	1.4+	1.4+
2.3	1.2	N/A	N/A	4.1.x	1.3+	1.3+
2.2	1.1	N/A	N/A	3.3.x	1.2+	1.2+

注1：tomcat支持xxx版本，例如tomcat 6.0x支持servlet2.5和jsp2.1，当然它也支持低于此版本的servlet和jsp，这一点相比很好理解毕竟它仅仅是支持之前发布的版本！

注2：我们可从tomcat lib下的jsp servlet jar 一窥这两者之间的关系

注3：我们可从web.xml的<web-app version>标签声明一窥项目使用的jsp/servlet版本

注4：上面表格中提到了el版本（就是${xxx}），但没有提到jstl（通常需结合el来使用）版本。

注5：jstl标签库在1.1时包含standard.jar+jstl.jar，在1.2时仅需jstl-1.2.jar，原因是jstl-1.2.jar包含了standard.jar内容。

注6：j2ee、j2se每个版本名称中都带有一个数字“2”，这个“2”是指Java2：自从 Java1.2 发布后，Java 改名为 Java2；1998年12月，Sun发布了Java历史上最重要的JDK版本：JDK1.2，伴随JDK1.2一同发布的还有JSP/Servlet、EJB等规范，并将Java分成了J2EE、J2SE和J2ME三个版本。

注7：j2ee：j2ee是在j2se的基础上添加了一系列的企业级应用编程接口.j2ee包含有很多的技术！其中有你熟悉的jsp，servlet，jdbc，jsf等技术！j2ee主要是用来做B/S结构的应用程序！也就是说是基于浏览器和服务器的！

注8：j2ee中有一种技术叫jsf。

在myeclipse中构建web工程时，如果你选择javaee1.4，jstl support选项你可以选择，因为javee1.4jar文件中没有包含jstl jar文件！如果你仅仅使用el表达式，就不用拷贝这两个jar

在myeclipse中构建web工程时，如果你选择javaee5，jstl support选项变灰你不能进行选择，原因是javaee5的jar文件已包含有jstl jar！如果选的是javaEE5，它自带的jstl仅能在tomcat6.x下才能正常使用

什么情况下需要在web.xml中配置标签？

如果Web应用使用Servlet 2.4以上的规范，则无需在web.xml文件中配置标签库定义，因为Servlet 2.4规范会自动加载标签库定义文件。这也就可以解释在struts1遗留项目中为何他们的web.xml中配置有标签文！

1.2 jsp深入分析

发表时间: 2016-01-01 关键字: jsp

JSP核心技术——JSP引擎内幕

主题：

- 幕后
- 多线程和持久性
- 隐含对象
- JSP的生命期
- 编译JSP
- JSP的性能调整

太多的文章都讨论过JSP的背景、语法和元素。很少讨论JSP引擎如何工作的许多细节技术。开发优秀的JSP应用程序或者说要做一个优秀的JSP程序员，至少需要了解JSP引擎是如何工作应该是必备的基本知识。

1 幕后

当JSP引擎接收到对请求时，它将JSP页面的静态数据和动态元素转换成Java代码段。

JSP元素内包含的动态数据已经是Java代码，所以这些段不必修改就可以使用，而静态数据被包装进out.write()方法内——形成java代码；然后，这些Java代码段被顺序放进特殊的包装器类——即jsp被翻译成的类。

JSP包装器由JSP引擎自动创建，它处理支持JSP所设计的大多数工作，而不需开发者干预。包装器通常扩展javax.servlet.Servlet类，这意味着JSP实际上被转换为Java Servlet代码的特殊形式。在许多方面，JSP可以被看作为一种用于创建Java Servlet的宏语言；JSP页面实际上提供了一个到Java Servlet API的以页面为中心的接口。

然后，源代码被编译为功能完全的Java Servelt。这个由JSP引擎创建的新Servlet处理基本的异常处理、输入/输出、线程以及大量与网络和协议相关的其他任务。实际上，是由新生成的Servlet处理请求并生成返回给请求JSP页面的客户的输出。

1.1 重新编译

JSP引擎可以被设计为在接到新的请求时重新编译每个页面。每个请求产生它自己的Servlet来处理相应。幸运的是，JSP采用一种更高效的方式。

JSP页面和Java Servlet为每个页面创建一个实例，而不是为每个请求创建一个实例。当接到新的请求时，只是在已生成的Servlet内创建一个线程。这意味着对JSP页面的第一个请求将生成一个新的Servlet，但以后的请求只是重用第一个请求所创建的Servlet。

注意：第一个请求时的延迟

当一个JSP页面第一次通过JSP引擎运行时，在收到响应前可能有较长的延迟。出现延迟的原因是，JSP引擎需要将JSP转换为Java代码、进行编译以及将它初始化，然后才能响应第一个请求。

以后的请求会利用已编译的Servlet。第一个请求后的请求应该会更快地得到处理。

有些特殊地事件可以通过JSP引擎何时重新编译JSP页面。为了管理重新编译，JSP引擎保持JSP页面代码的记录，并在源代码改变时重新编译页面。JSP的不同实现对于何时重新编译有不同的规则，但所有的引擎必须在JSP源代码改变时重新编译页面。

记住，JSP页面的外部资源，例如JavaBean或者包含（include）的JSP页面可能不会造成页面重新编译。另外，不同的JSP引擎对于何时重新编译页面有不同的规则。

注意：预编译协议

从JSP 1.1起，规范里定义了一种预编译页面的方式。要想预编译特定的JSP，必须带着jsp_precomplie参数建立对此JSP的HTTP请求。

例如，键入如下URL：

http://www.javadesktop.com/core-jsp/catalog.jsp?jsp_precompile="true"

如果此JSP还未编译过或者JSP代码已经改变，那么就会编译它。

1.2 Servlet与JSP的关系

因为JSP页面被转换为Java Servlet，所以JSP表现出的许多行为与Java Servlet一样。JSP从Java Servlet继承了强大的功能和几个特点。

Java Servlet通过创建一个在JVM内运行的持久的应用程序进行工作。处理新的请求的方法实际上是，在这个持久的应用程序内运行一个新的线程。对JSP页面的每个请求在对应的Java Servlet内有一个新线程。

Java Servlet还为JSP开发人员提供几个内建方法和对象。它们提供一个到Servlet的行为和JSP引擎的直接（接口）通道。

2 多线程和持久性

JSP从Java Servlet继承了多线程和持久性。持久性允许对象在第一次创建Servlet时初始化，所以JSP Servlet的物理内存内容在请求之间保持不变。可以在持久空间内创建变量，这就允许Servlet执行缓存、会话跟踪以及在无状态环境内没有的其他功能。

JSP开发员接触不到线程开发所涉及的许多问题。JSP引擎处理创建、销毁和管理线程所涉及的大多数工作。这样JSP开发员就不必承担多线程开发的重担。但是JSP开发员需要了解几个影响JSP页面的多线程编程问题。

线程可能不经意地损害其他线程。在这种情况下是，JSP程序员所需要知道何时以及如何使页面不被线程化。

2.1 持久性

```
<%!
    int counter;//当所在jsp页面被转换为.java文件时，该语句将作为该类的属性！是可持久的
    int j;
    JspWriter out;
```

```
    public void a(){//当所在jsp页面被转换为.java文件时，该语句将被作为该
类的方法！是可持久的
        try{
            out.println("123465xxx");
        }catch(Exception e){
            e.printStackTrace();
        }
    }
%>
<%
    out.println("123465xiaocui");//当所在jsp页面被转换为.java文件时，该语
    句将被放在_jspService（...）方法内并输出！是不可持久的！
%>
```

因为Servlet只被创建一次，然后作为不变的实例一直运行，所以可以创建持久的变量。同一Servlet的所有线程共享持久的变量。对这些持久变量值的改变被反映到所有线程中。这里的持久变量在java代码里具体体现就是作为该类的属性！试想如果是某个方法所声明的变量，那么每次线程启动时用到得变量值都是最原始的值。

从JSP开发员的角度来看，所有在声明标记（<%! ... %>）内创建的对象和变量都是持久的。在线程内创建的对象和变量不是持久的。在Scriplet、Expression和Action标记内的代码将在一个新的请求线程内运行，因此不会创建持久的变量或对象。

拥有持久的对象允许开发者在页面请求之间跟踪数据。这就允许使用内存内的对象实现缓存、计数器、会话数据、数据库连接的缓冲以及许多其他任务。程序清单4-1显示了一个使用持久变量的计数器。页面首次装载时，创建变量counter。因为Servlet一直在内存中运行在，此变量将一直存在直到此Servlet重新启动。每次请求页面时，变量counter递增并显示请求者。每个用户应该看到一个页面计数器，其数字比上次访问此页面时所见到的数字大1。

程序清单4-1	count.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<%!
  int counter;
%>
<HTML>

<STYLE>
.pageFooter {
  position: absolute; top: 590px;
  font-family: Arial, Helvetica, sans-serif;
  font-size: 8pt; text-align: right;
}
</STYLE>

<BODY>
<DIV>
This page has been accessed
<%
  counter++;
  out.println(counter);
```

%>

times since last restarted.

</DIV>

</BODY>

</HTML>

2.2 线程的危险

然而，对象持久性也存在一些潜在的问题。为防止这些问题，JSP开发员需要理解并控制这些问题的出现。

那些使持久性有价值的因素也带来了一个严重问题——线程竞赛（Race Condition）。发生线程竞赛的情况是：一个线程正准备使用数据，而第二个线程在第一个线程使用数据前修改了此数据。

考虑以上的例子（程序清单4-1）有两个线程在运行的情况，仔细注意counter变量的值。

线程1——用户A请求此页面。

线程2——用户B请求此页面。

线程1——counter加1。

线程2——counter加1。

线程1——counter被显示给用户A。

线程2——counter被显示给用户B。

在此情况下，用户A所看到的数据实际是用户B应该看到的。这显然不是想要的结果。

以上例子所造成的问题是微不足道的，只不过是用户A看到了一个错误的页面计数。但是，线程竞赛可能就这样如此轻易造成非常严重的问题——设想一下，如果在给在线购物的用户发帐单时出现线程竞赛，那么会有什么后果。

不论线程竞赛所造成的问题是否重要，都要解决这个问题，这是个良好的编程习惯。线程执行的先后不能预先确定，所以线程竞赛的后果可能无规律地出现。线程竞赛可能难以定位。

2.3 线程安全

考虑线程安全时，最重要的是记住线程可以显著提高性能。线程安全几乎总是通过“禁止”代码的后些部分的线程化来实现的。

另一个重点是，线程竞赛只在使用持久变量时出现。如果一个应用程序所用的所有变量和对象都是由线程创建的，那么就不会发生线程竞赛。在这些情况下，不会发生线程化的问题。

1. SingleThreadModel

获取线程安全的最简单方法也带来了性能下降。这个方法就是关闭整个页面的线程化。关闭线程化是个糟糕的选择，在正常情况下应该避免，因为它是通过牺牲许多优势来避免潜在的问题。

JSP通过page指令属性isThreadSafe='false'提供了关闭线程化的方法。这使页面在SingleThreadModel下创建，SingleThreadMode只允许页面每次处理一个请求。

这里的siThreadSafe= ‘true’ 相当于你的servlet实现了接口

SingleThreadModel，这个接口的作用是这样的：当请求的实例数大于当前实例数的时候才会重新去创建个实例，然后把这个实例放到Stack中（实例池）中。换句话就是说当请求的实例数小于等于当前的实例数时它会从statck中取一个空闲的实例来完成请求！我们可以看出：servlet自身的非静态属性有安全隐患（因为你可能两次请求使用的是一个servlet对象），静态属性一定有安全隐患（所有的请求都公用这个属性），servlet之外的类也可能存在安全隐患

这个选项还不能百分之百有效。在session和application范围内创建的变量可能仍然受多个线程的影响。但是servlet类的属性

2. synchronized( )

一个保护变量不受线程竞赛影响的更实用而且高效的方法是，使用Java的同步接口。这个接口显露一个锁定机制，这个机制每次只允许一个线程处理某个代码块。

可以同步整个方法使其不受线程竞赛影响，办法是在方法的识别标志中使用synchronized关键字。这将保护在此方法内访问的所有持久变量。在此情况下，每次只有一个线程可以访问此方法。

也可以通过将代码包装在synchronized( )块内来使代码同步。在此情况下，需要有一个参数表示对象被锁定。

注意：同步的标志

派生自java.lang.Object的对象可以用作同步块的参数。每个对象有一个“块标志”，synchronized( )使用这个标志管理线程化（替代C程序中所用的互斥和信号量）。Java中的原始数据类型（8中基本类型）没有此标志，因此不能用作同步块的锁。

在创建同步块时，使用代表被同步数据的对象通常更高效。例如，如果一个写磁盘的同步块被编写成foo对象，最好使用synchronized(foo)。当然，也可以使用this或page对象，但同步块每次运行时都堵塞整个页面，这回造成瓶颈。

清单4-2是一个新的计数器页面例子，它使用同步块。在此例中，两行代码被放在同步块内。每次只有一个线程可以处理这个块。每个页面将锁定页面对象，递增变量，显示变量，然后解锁这个页面对象。

程序清单4-2	count2.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<%!
  int counter;
%>
<HTML>

<STYLE>
.pageFooter {
  position: absolute; top: 590px;
  font-family: Arial, Helvetica, sans-serif;
  font-size: 8pt; text-align: right;
```

```
}
</STYLE>

<BODY>
<DIV>
This page has been accessed
<%
  synchronized (page){
    counter++;
    out.println(counter);
  }
%>
times since last restarted.
</DIV>
</BODY>

</HTML>
```

3 隐含对象

Servlet还创建几个由JSP引擎使用的对象。这些对象中的大多数被显露给JSP开发员，并且可以直接调用而不必显式声明。

3.1 out对象

JSP的主要功能是描述发送到客户请求的响应输出流中的数据。这个输出流通过隐含的out对象显露给JSP开发员。

Out对象是javax.servlet.jsp.JspWriter对象的实例。这个对象可以代表输出流、经过滤的输出流或来自其他JSP页面的嵌套的JspWriter对象。但是输出应该不被直接发送到输出流，因为在JSP的生命期内可能有多个输出流。

根据页面是否被缓冲，初始的JspWriter对象的初始化有所不同。缺省情况下，每个JSP页面都打开了缓冲，这可以提高性能。缓冲功能很容易关闭，只要使用page指令的buffered='false'属性即可。

缓冲的out对象以块为单位收集和发送数据，这通常会提供最好的总体吞吐量。使用缓冲时，PrintWriter在第一个块被发送时创建，也就是在第一次调用flush()时。

如果不缓冲输出，将立即创建PrintWriter对象并引用out对象。在此情况下，发送到out对象的数据立即被发送到输出流。创建PrintWriter对象时将使用由服务器决定的缺省设置和头信息。

注意：HTTP头和缓冲

HTTP使用响应头描述服务器以及定义发往客户的数据的某些方面信息。这可能包括页面的MIME内容类型、新的cookie、转发URL或其他HTTP“动作”。

JSP允许开发者在创建OutputStream（即PrintWriter）前改变响应头的内容。一旦建立了OutputStream，头信息就不能改变了，因为它已被发送到客户。

在缓冲的out对象的情况下，直至缓冲区第一次刷新时才建立OutputStream。缓冲区被刷出很大程度上取决于page指令的autoFlush和bufferSize属性。通常，最好在有任何数据被发送到out对象前设置头信息。

对无缓冲的out对象，很难设置页面头。当无缓冲的页面建立时，几乎立即就建立了OutputStream。

在建立OutputStream后发送的头可能会造成大量不正常的结果。一些头被简单地忽略，其他的头可能产生异常，例如IllegalStateException。

JspWriter包含的方法大多数与java.io.PrintWriter类一样。但是JspWriter有另外几个用于处理缓冲的方法。与PrintWriter对象不同，JspWriter抛出IOExceptions。在JSP中，这些异常需要显式捕获和处理。

注意：autoFlush( )

在JSP中，缺省的缓冲行为是在缓冲区满了时自动刷出缓冲区。但是，有时JSP实际上是直接和另一个应用程序通信。在此情况下，不“何时”的行为可能是在超出缓冲区时抛出异常。

设置page指令的属性autoFlush='false'将造成缓冲区溢出而抛出异常。

3.2 request对象

客户每次请求页面时，JSP引擎创建一个代表此请求的新对象。这个对象是javax.servlet.http.HttpServletRequest的实例，并且具有描述此请求的参数。这个对象通过request对象显露给JSP开发员。

通过request对象，JSP页面可以对从客户机那儿接收的输入做出反应。请求参数被存储在特殊的键值对中，可以使用request.getParameter(name)方法获取请求参数。

request对象还提供了几个用于获取头信息和cookie数据的方法。它提供了认识客户和服务器的方式。

requst对象被限制在request范围内。不论page指令如何设置页面的范围，request对象总是为每个请求重新创建。对于来自一个客户的请求，都有一个request对象与之对应。

关于request的对象及其方法的可以自己再查找一些相关资料。

3.3 response对象

服务器要创建request对象，也要创建一个来代表对客户的响应。这个对象是javax.servlet.http.HttpServletResponse的实例，并且作为response对象显露给JSP开发员。

response对象处理返回给客户的数据流。out对象与response对象关系非常密切。response对象还定义了创建新的HTTP头的接口。通过response对象，JSP开发员可以添加新的cookie或数据标记，改变

页面的MIME内容类型，或者开始“服务器推”方法。JSP页面还包含关于HTTP的信息，从而能够返回HTTP状态码，例如使页面重定向。

关于response的对象及其方法的可以自己再查找一些相关资料。

3.4 pageContext对象

pageContext对象用于代表整个JSP页面。它适于作为访问关于页面的信息的方法，同时回避了大多数实现细节。

这个对象为每个请求存储request对象和response对象的引用。application、config、session和out对象通过访问此对象的属性派生出来。PageContext对象还包含关于发给JSP页面的指令的信息，这包括缓冲信息、errorPageURL和页面范围。

pageContext对象不只是作为数据资料库。它还管理嵌套的JSP页面，执行forward和include动作所涉及的大多数工作。pageContext对象还处理未被捕获的异常。

从JSP作者的角度来看，这个对象在获取关于当前JSP页面环境的信息方面很有用。如果你要创建组件，而组件的行为根据JSP page指令的不同而有所变化，那么pageContext对象特别有用。

3.5 session对象

session对象用于在使用无状态连接协议（如HTTP）的情况下跟踪关于某个客户的信息。会话可以用于在客户请求之间存储任意信息。

每个会话应该只对应于一个客户，并且可以跨多个请求。会话通常通过URL重写或cookie来跟踪，但是跟踪进行请求的客户的方法对于session对象并不重要。

session对象是javax.servlet.http.HttpSession的实例，其行为方式与Java Servlet下的session对显完全相同。

关于session的对象及其方法的可以自己再查找一些相关资料。

3.6 application对象

application对象是产生的Servlet的ServletContext的直接包装器。它所具有的方法和接口与Java Servlet编程中的ServletContext对象一样。

这个对象在JSP页面的整个生命期内代表此页面。当JSP页面初始化时，创建这个对象；当使用jspDestory( )方法删除JSP页面时，或者JSP页面重新编译时，或者JVM崩溃时，将删除这个对象。JSP页面内所用的所有对象都可以使用此对象中存储的信息。

application对象还提供JSP与服务器进行通信的方法，此过程不涉及“请求”。这有助于寻找关于文件MIME类型的信息，直接向服务器发送日志信息，或者与其他服务器通信。

3.7 config对象

config对象是javax.servlet.ServletConfig的实例。这个对象是产生的Servlet的ServletConfig对象的直接包装器。它所具有的方法和接口与Java Servlet编程中的ServletConfig对象一样。

这个对象允许JSP开发员访问Servlet或JSP引擎的初始化参数。它有助于获取标准的全局信息，例如路径或文件位置。

3.8 page对象

这个对象是对页面对象的实例的实际引用。它可以看作是代表整个JSP页面的对象。

当JSP页面第一次被初始化时，通过获取对this对象的引用来创建page对象。所以，page对象实际上是this对象的直接同义词。

但是，在JSP的生命期内，this对象不能引用页面本身。在JSP页面的环境内，page对象将保持不变，并且总是代表整个JSP页面。

3.9 exception元素

exception对象是包含从前一个页面抛出的异常的包装器。它通常用于根据错误条件产生合适的响应。

前一个页面有未被捕获处理的异常而且使用了<%@ page errorPage=" ... " %>标记时，可以使用这个对象。

4 JSP的生命期

JSP引擎有三个方法用于管理JSP页面和它产生的Servlet的生命期。

JSP页面的核心通过使用产生的_jspService方法来处理。它由JSP引擎自身创建和管理_jspService不应该由JSP开发者管理；如果这么做，会造成灾难性后果。_jspService方法代表JSP页面处理所有请求和响应。实际上，所有线程调用_jspService方法。

注意：保留的名称

JSP规范特别保留了以jsp、_jsp、jspx和_jspx开头的方法和变量。JSP开发者可以访问具有这些名称的方法和变量，但是不能创建这样的新方法和新变量。JSP引擎期望自己控制这些方法和变量，所以改变它们或者创建新的方法和变量可能够了会造成奇怪的结果。

另两个方法， jspInit( )和jspDestroy( )，可由JSP开发员进行覆盖。实际上，如果JSP开发员不创建它们，它们并不存在。它们在管理JSP页面的生命期方面扮演特殊角色。另外你创建或者不创建jspInit( )和jspDestroy( )，jsp都会创建

_jspInit( )和_jspDestroy( )，这两个方法也是用于初始化和销毁的！~

4.1 jspInit( )

方法原型：void jspInit( )

jspInit( )方法只在JSP页面第一次被请求时运行一次。JspInit( )方法保证在任何请求被处理前被处理完。它与Java Servlet和Java Applet中的init()方法一样。

jspInit( )方法允许JSP作者为每个请求创建和装载所需的对象。这有助于装载状态信息、创建数据库连接缓冲池，以及执行只需在JSP页面首次启动时执行一次其他任务。

4.2 jspDestroy( )

方法原型：void jspDestroy( )

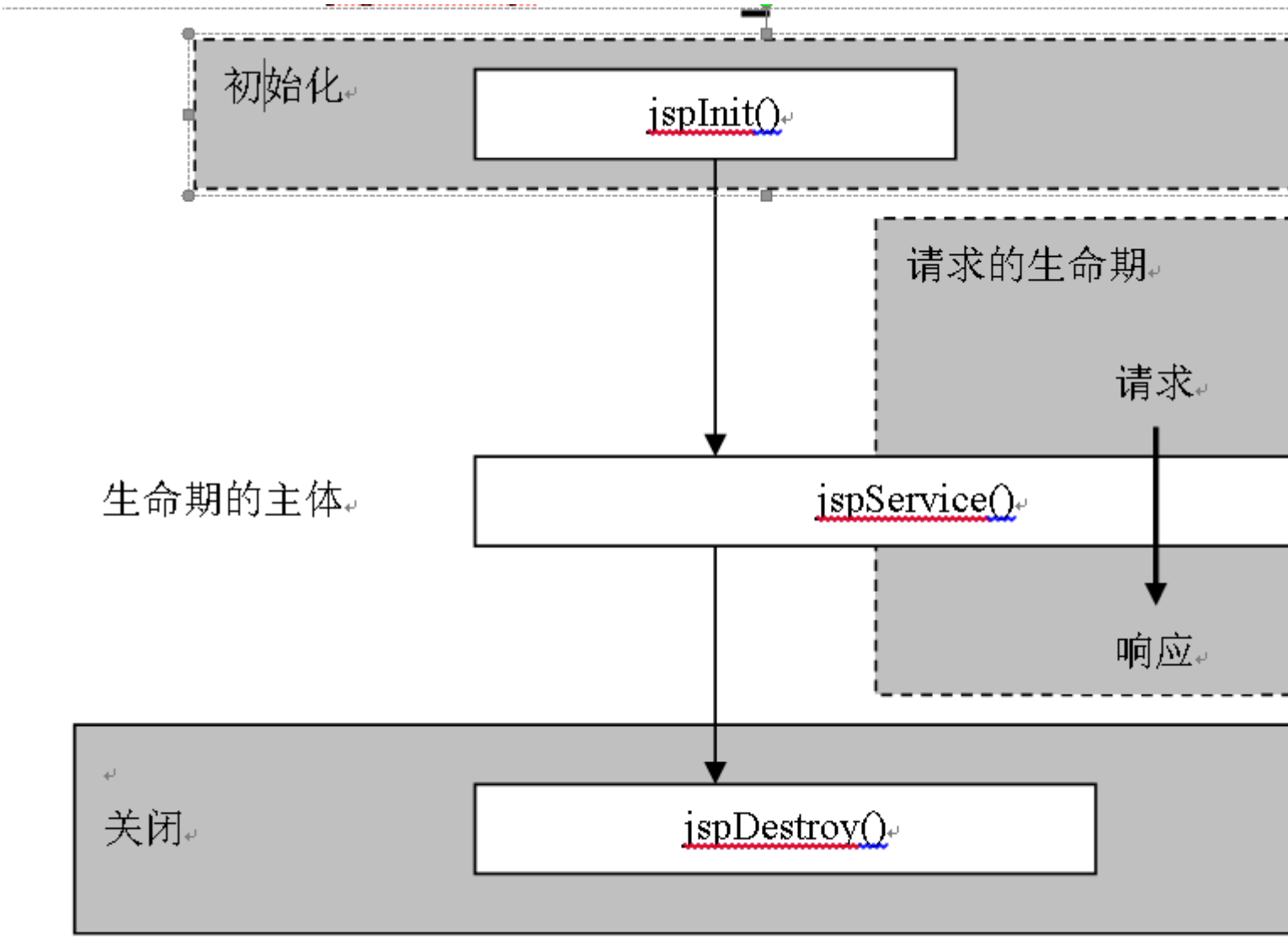
当Servlet从JVM卸载时，服务器调用jspDestroy( )方法。它与Java Servlet和Java Applet中的destroy( )方法一样。

与jspInit( )方法不同，不保证这个方法被执行。服务器在每个线程后尽力尝试运行这个方法。因为这个方法在处理完成后调用，有些情况可能使它不被执行。如服务器崩溃。

JspDestroy( )允许JSP开发员在Servlet完成前执行代码。它通常用于释放资源或关闭仍然打开着的连接，也可以用于释放存储状态信息或其他应该在实例之间存储的信息。

4.3 JSP的生命期概述

在第一次请求或预编译时，调用jspInit( )，此时页面开始运行，等待请求。现在，_jspService处理大多数事务、获取请求、运行线程并且产生响应。最后，当接到关闭信号时，调用jspDestroy( )方法。JSP页面的整个生命期见图4-1。



4.4 使用jspInit( )和jspDestroy( )的计数器

前面的页面计数器例子所用的变量只存储在运行的Servlet的内存中。它没有被写入磁盘，所以如果JSP页面重新启动，变量将被重置。

在程序清单4-3所示的例子中，jspInit( )方法在页面首次启动时装载变量的值以恢复此变量。此例子还使用jspDestroy()方法写此变量的值，以便JSP页面下一次启动时恢复。

程序清单4-3	count3.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<%@ import="java.util.*,java.sql.*" %>

<%!
 int counter;

 public void jspInit() {
  try {
   FileInputStream countFile =
                new FileInputStream ("counter.dat");
   DataInputStream countData =
                new DataInputStream (countFile);
   counter = countData.readInt();
  }
  catch (FileNotFoundException ignore) {
// No file indicates a new conter.
  }
 }

 public void jspDestroy() {
  try{
   FileOutputStream countFile =
```

```
                new FileOutputStream ("counter.dat");
      DataOutputStream countData =
                new DataOutputStream (countFile);
      countData.writeInt(counter);
    }
    catch(IOException e) {
      e.printStrackTrace();
    }
  }
%>
<HTML>

<STYLE>
.pageFooter {
  position: absolute; top: 590px;
  font-family: Arial, Helvetica, sans-serif;
  font-size: 8pt; text-align: right;
}
</STYLE>

<BODY>
<DIV>
This page has been accessed
```

```
<%
  synchronized (page){
    counter++;
    out.println(counter);
  }
%>
times since last restarted.
</DIV>
</BODY>


</HTML>
```

5 编译JSP

大多数JSP引擎将它们创建的Servlet源代码放在一个工作目录中。在许多引擎中，这是一个必须显式地打开的选项，但是这个简单的任务。

通读产生的源代码有助于调试JSP页面中的问题。另外，这些源代码可以帮助有经验的Java开发员进一步了解JSP实现的内部原理。

程序清单4-4显示来自上一个计数器的编译的源代码。此清单中的源代码由Apache Jakarta Project的Tomcat 4.1.21产生；其他JSP引擎可能会产生略有差别的源代码。此源代码还被略微修改过，以便阅读方便。

程序清单4-4	count.jsp

6 JSP的性能调整

Java编程的几个方面会严重影响JSP页面的性能。

这些方面中的一些不是JSP特有的，而是Java编程的普遍问题。下面列出的是最常见的效率错误。

6.1 避免串联追加

在开发时，使用串联操作符（ + ，有的书称为“重载操作符” ）将String对象联结起来是很简单的。例如：

```
String output;
output += "Item: " + item + " ";
output += "Price: " + price + " ";
println (output);
```

然而，很容易忘记String对象是不可变的，并且不能包含可改变的数据。StringBuffer对象设计为用于操作字符串。载改变String对象时，实际上是创建一个新的StringBuffer对象，旧的字符串被StringBuffer.toString( )的结果替代。

在处理以上代码时，将会创建几个新的String对象和StringBuffer对象。如果将String对象转换为StringBuffer对象或一开始就使用StringBuffer对象，效率通常会高得多。例如：

```
StringBuffer output;
output.append(new String("Item: "));
output.append(item);
output.append(new String(" "));
output.append(new String("Price: "));
output.append(price);
output.append(new String(" "));
```

println (output.toString());

6.2 小心使用synchronized( )

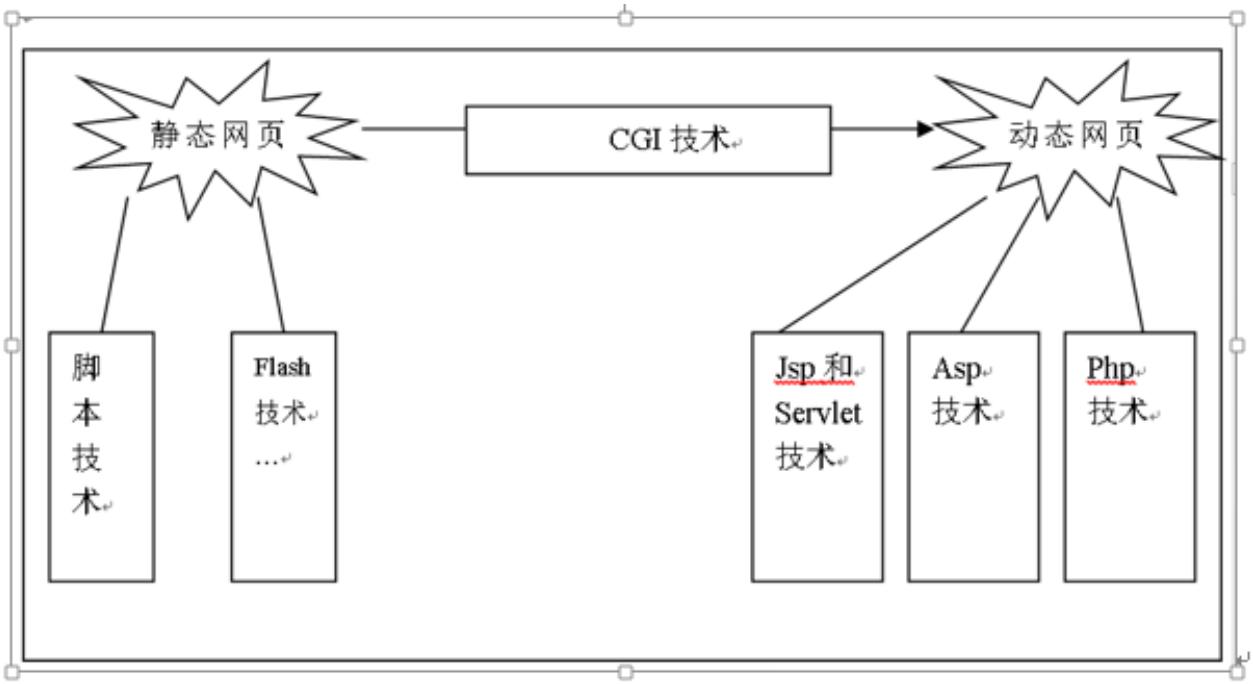
保护对象不受线程竞赛问题的影响是很重要的，但是同步过程容易造成性能瓶颈。

确保被同步的块包含尽可能少的代码行。在代码阻塞的线程中，哪怕减少一行代码，也会产生很大的不同。另外，尽可能同步大多数相应的锁定对象。通常，最好同步那些受到线程竞赛威胁的对象。尽可能避免使用this或page对象作为锁定对象。

总体上，JSP引擎你用了Java Servlet的非常强大的体系结构。线程化显著提高了性能，同时要注意防止线程竞赛。基于Java Servlet的其他调整技术可以用于调整JSP页面和底层Java Servlet的性能。

1.3 动态网页技术

发表时间: 2016-01-03



动态网页技术的鼻祖是cgi（common gateway interface, 即公用网管接口技术），但是它编程困难，效率低，修改复杂，现在已升级到了per动态网页技术的发展：

基于进程编程的动态网页技术:CGI--->perl：

基于线程编程的动态网页技术：

- ASP 与ASP.NET：asp是用vb脚本；asp.net 是用c#来编写。
- PHP：是草根，但近年来越来越火爆。
- JSP/SERVLET：。

1.4 j2se/j2ee/j2me

发表时间: 2016-01-03

j2se、j2ee、j2me：

j2se、j2ee、j2me，中的数字“2”是指Java2：自从 Java1.2 发布后，Java 改名为 Java2（我只能呵呵，可以说知道1.5后才有人将其改名为5,...）；

1998年12月，Sun发布了Java历史上最重要的JDK版本：伴随JDK1.2一同发布的还有JSP/Servlet、EJB等规范，并将Java分成了J2EE、J2SE和J2ME三个版本。

J2se（Java 2 Standard Edition）是Java的标准版,用于标准的应用开发，即针对普通PC应用开发；

J2ee（Java 2 Enterprise Edition）是Java的一种企业版用于企业级的应用服务开发，即针对企业网应用；

J2me（Java 2 Micro Edition）是Java的微型版,常用于手机上的开发，说白了针对嵌入式设备及消费类电器的。

J2se、j2ee、j2me之间的区别：

J2se是最基础的运行JAVA程序的工具包，包含了最基本的JAVA类库，例如java.lang.String就是基本的类库之一，j2se是 用来开发桌面应用程序（即所谓的C/S结构）足够了，例如eclipse就是它开发的，相当于微软的C#；

同理，手机应用程序比正常的桌面应用程序简单一点，因此不需要用到那么多类库，即所谓的JAVA服务（就是基本的类而已，楞是说 成服务，那以后C语言也就叫C服务得了），因此所谓的j2me，其实就是从j2se中去掉一些类库，就成为所谓的微型移动程序的开发平台了。

同理， j2ee是在j2se的基础上添加了一系列的企业级应用编程接口，其中就有我们熟悉的jsp，servlet，jdbc，jme，jsf等13中技术！ j2ee其实就相当于微软的ASP.NET， 是专门用于web开发的，它包含了j2se，另外还提供了一些与web开发相关的类库（说白了就是jsp、servlet等接口，例如 HttpServletRequest对象就是其类库中的一种），而j2se是开发桌面应用程序的，因此不需要用到这些HTTP对象（网络上将j2ee吹嘘 成什么企业级开发，晕！不就是开发个网站（或者是所谓的B/S结构）吗，但人们楞是冠之以一个强悍的吓人的名头，当真是将好大一部分初学者给吓住了（包括 我^-^）！）。

1.5 JSP内置对象

发表时间: 2016-01-03

application: 是ServletContext接口的实例，该类在javax.servlet包下！属于javaee

config:是ServletConfig接口的实例，该类在javax.servlet包下！属于javaee

exception:是Throwable类的实例，该类在java.lang包下！属于javase

out:是JspWriter类的实例，该类在javax.servlet.jsp包下。属于javaee

Page:是Object类的实例，该类在java.lang包下。属于javase

pageContext:是PageContext类的实例，该类在javax.servlet.jsp包下。属于javaee

request:是HttpServletRequest接口的实例，该类在javax.servlet.http包下，属于javaee

response:是HttpServletResponse接口的实例，该类在javax.servlet.http包下，属于javaee

session:是HttpSession接口的实例，该类在javax.servlet.http包下，属于javaee。

Cookie：javax.servlet.http包下的类！

从Servlet/Action中获取JSP的内置对象PageContext

从tomcate的work目录下随便找一个jsp被翻译成的java文件，在_jspService(..)方法里我们看到pageContext是这样获取的：

“

javax.servlet.jsp.PageContext pageContext =

JspFactory.getDefaultFactory().getPageContext(this, request,response, null, true, 8192, true);

”

因此我们在Servlet或Action中应这样获取JSP的内置对象PageContext：

//先获得JspFactory实例

JspFactory fac=JspFactory.getDefaultFactory();

```
//然后调用JspFactory实例的getPagContext(..)方法。

PageContext pageContext=fac.getPageContext(this, request,response, null, false,
JspWriter.DEFAULT_BUFFER, true);
```

这里有必要再次说明一下pagecontext ，它用于方便存取各种范围的名字空间、servlet相关的对象的API——换句话就是说用它可以方便的获取其他内置对象！

1.6 JSP指令与标签

发表时间: 2016-01-03

1. JSP编译指令(3个)

JSP的编译指令是通知JSP引擎(Tomcat、WebLogic)的消息，它不直接生成输出。

JSP引擎的工作原理:

当一个JSP页面第一次被访问的时候，JSP引擎将执行以下步骤：

（1）将JSP页面翻译成一个Servlet，这个Servlet是一个java文件，同时也是一个完整的java程序

（2）JSP引擎调用java编译器对这个Servlet进行编译，得到可执行文件class

（3）JSP引擎调用java虚拟机来解释执行class文件，生成向客户端发送的应答，然后发送给客户端

以上三个步骤仅仅在JSP页面第一次被访问时才会执行，以后的访问速度会因为class文件已经生成而大大提高。当JSP引擎街道一个客户端的访问请求时，首先判断请求的JSP页面是否比对应的Servlet新，如果新，对应的JSP需要重新编译。

编译指令都有其默认值，因此无须为每个指令设置其值。

常见的编译指令有3个：

(1)、page：是针对当前页面的指令；

(2)、include：用于指定包含另一个页面；

(3)、taglib：用于定义和访问自定义标签。

使用编译指定的语法格式如下：

<%@ 编译指令名 属性1="属性值" 属性2="属性值" ...%>

1.1. page

针对当前页面的指令，通常位于JSP页面的顶端，一个JSP页面可以使用多page指令。

page指令格式如下：

<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>

<%@ page import="java.sql.*" %>

常用的page的属性如下：

(1)、contentType：用于设定生成网页的文件格式和编码字符集，即MIME类型和页面字符集类型，默认MIME类型是text/html，默认的字符集类型为:ISO-8859-1；

(2)、language：声明当前JSP页面使用的脚本语言的种类，因为页面是JSP页面，所以该属性的值通常都是java。

(3)、errorPage：指定错误处理页面，如果本页面产生了异常或者错误，而该页面没有对应的处理代码，则会自动调用该属性指定的JSP页面;

(4)、pageEncoding：指定生成网页的编码字符集；

(5)、import：用于导入包。下面几个包是默认自动导入的，不需要显示导入。默认导入的包有：java.lang.*、javax.servlet.*、javax.servlet.jsp.*、javax.servlet.http.*；

1.2. include

语法格式为：<%@ include file="要包含的页面路径" %>

等价于<jsp:include file=”” />?他们都是静态包含，

注意和<jsp:include page=”” />动态包含的区别。

1.3. taglib

用于定义和访问自定义标签。

2. JSP动作指令(7个)

动作指令和编译指令不同，编译指令是通知Servlet引擎如何处理消息，而动作指令只是运行时的动作。

编译指令在将JSP编译成Servlet时起作用；而处理指令通常可替换成JSP脚本，它只是JSP脚本的标准化写法。

- u jsp:forward：执行页面转向，将请求的处理转发到下一个页面。
- u jsp:param：用于传递参数，必须与其他支持参数的标签一起使用。

- u jsp:include：用于动态引入一个JSP页面。
- u jsp:plugin：用于下载JavaBean或Applet到客户端执行。
- u jsp:useBean：创建一个JavaBean的实例。
- u jsp:setProperty：设置JavaBean实例的属性值。
- u jsp:getProperty：输出JavaBean实例的属性值

2.1. jsp:forward指令

jsp:forward指令用于将页面响应转发到另外的页面。既可以转发到静态的HTML页面，也可以转发到动态的JSP页面，或者转发到容器中的Servlet中。

jsp:forward指令的语法格式：

```
<jsp:forward page="path<%expression%>">
<jsp:param name="" value="" />
</jsp:forward>
```

下面使用jsp:forward动作指令来转发用户请求：

jsp-forward.jsp:

```
<%@ page language="java" contentType="text/html; charset=utf-8"
  pageEncoding="utf-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>forward的原始页面</title>
</head>
```

```
<body>
<h2>这是jsp:forward的原始页面</h2>
<jsp:forward page="forward-result.jsp">
  <jsp:param value="21" name="age"/>
  <jsp:param value="evan" name="name"/>
</jsp:forward>
</body>
</html>
```

将客户端请求转发到forward-result.jsp页面，转发时增加了二个请求参数：一个参数名为age，参数值为21，另一个参数名为name，参数值为evan。

在forward-result.jsp页面中，可以通过request内置对象来获取增加的请求参数值。

forward-result.jsp页面代码如下：

```
<%@ page language="java" contentType="text/html; charset=utf-8"
  pageEncoding="utf-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>forward跳转之后的页面</title>
</head>
<body>
<h2>这是从jsp-forward页面跳转之后到得页面</h2>
```

```
<!-- 使用request内置队形获取age和name参数的值  -->
姓名：<%=request.getParameter("name") %>
年龄：<%=request.getParameter("age") %>
</body>
</html>
```

结果如下：

注意：JSP页面中有中文时，为防止出现中文乱码，使用UTF-8字符集。

2.2. jsp:include指令

jsp:include指令是一个动态include指令，也可以包含某个页面，它不会导入被include页面的编译指令，仅仅将被导入页面的body内容插入本页面。（只导入body的内容）

include指令的格式如下：

```
<jsp:include page="<url|expressions>" flush='true'/>
```

或者为：

```
<jsp:include page="<url|expressions>" flush='true'>
  <jsp:param name="请求参数名" value="请求参数值" />
</jsp:include>
```

flush属性用于指定输出缓存是否转移到被导入文件中。

实例：

jsp-include.jsp代码如下：

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
  pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>JSP:include指令测试</title>
</head>
<body>
<h2>JSP:include指令测试，这是jsp-include.jsp页面</h2>
<br>
<!-- 使用动态include指令导入页面 -->
<jsp:include page="jsp-include1.jsp" />
</body>
</html>
```

jsp-include1.jsp代码如下：

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
  pageEncoding="UTF-8"%>
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>JSP:include指令测试</title>
</head>
<body>
<h2>JSP:include指令测试，这是jsp-include1.jsp页面</h2>
</body>
</html>
```

执行结果如下：

2.3. jsp:useBean、jsp:setProperty、jsp:getProperty指令

这三个指令都是和javaBean相关的指令，其中：

jsp:useBean指令用于在JSP页面中初始化一个JAVA实例；

jsp:setProperty指令用于为JavaBean实例的属性设置值；

sp:getProperty指令用于输出JavaBean实例的属性。

jsp:useBean指令的语法格式为：

<jsp:useBean id="name" class="classesname" scope="page | request | session | application" />

其中，id属性为JavaBean的实例名，class属性确定JavaBean的实现类。

scope属性用于指定JavaBean实例的作用范围，该范围有如下4个值：

(1)、page：该JavaBean实例仅在该页面中有效；

(2)、request：该JavaBean实例在本次请求中有效；

(3)、session：该JavaBean实例在本次session内有效；

(4)、application：该JavaBean实例在本应用内一直有效。

jsp:setProperty指令的语法格式如下：

<jsp:setProperty name="BeanName" proterty="propertyName" value="value" />

其中，name属性确定需要设定JavaBean的实例名；

proterty属性确定需要设置的属性名；

value属性则确定需要设置的属性值。

jsp:getProperty指令的语法格式如下：

<jsp:getProperty name="BeanName" proterty="propertyName" />

其中，name属性确定需要输出JavaBean的实例名；

proterty属性确定需要输出的属性名；

2.4. param指令

param用于设置参数值，这个指令本身不能单独使用，因此单独的param指令没有实际意义，param指令可以与以下指令结合使用：

<jsp：include ..

<jsp：forward..

<jsp：plugin ..

3. JSP标签

Jsp动作指令通常也被称之为标签...

3.1. jsp 注释

JSP注释：<%-- 注释内容 --%>

HTML注释：<!-- 注释内容 -->

在jsp声明和jsp脚步中你还可以使用java注释

3.2. jsp声明

<!% %>可以声明变量和方法

3.3. jsp脚本

<% %>

3.4. jsp表达式

<%=表达式%>输出表达式的值

3.5. 自定义标签

说起自定义标签我们先来说说javabean

JavaBean其实就是一个类，一个JavaBean实例叫做：Bean，它通过具体的“方法”实现相应的逻辑功能，并且为外部操作提供接口，外部JSP通过这些接口可以方便地使用它实现逻辑功能，这样就实现了代码的重用以及逻辑功能与页面显示层分离，它的好处就是解决了在JSP页面中实现逻辑功能会出现相同的代码在很多文件中出现导致不利于维护和更新的缺陷，并且可以逻辑功能程序员和页面编辑员的分工，让其各自在各自的领域里全心工作。

自定义标签：

自定义标签其实也是一个类，它封装了相应的逻辑功能，和JavaBean很类似，但是它们之间存在很大的区别：JavaBean通过提供接口供外部操作调用实现逻辑功能，而自定义标签是通过标签的形式为外部操作实现逻辑功能，例如JSP已有的标签：<jsp:forward>，它的好处就是调用十分方便，形式：<jsp:forward page="转向页面的url">就实现了其功能，其中的“JSP”代表的是标签库名，“forward”代表的是标签名，自定义标签的最终形式也很类似：<a:b>其中的a代表的是自定义库名，b代表的是自定义标签名，当然它是包含在a中的。

1.7 jsp/servlet 本质

发表时间: 2016-01-03

1. JSP与servlet

servlet是含有html的java代码，jsp是含有java代码的html页面！先出现的是servlet后出现jsp，jsp是为了解决servlet的输出问题而诞生！但是jsp却无法去除并替代servlet！他们的地位是平起平坐的！

可以这样说jsp 不过是编写servlet的一种方式而已，同时jsp最终还是要被翻译为servlet的。servlet会被编译（同java代码相同，其实它就是java代码），在请求期间，运行的是servlet，而非jsp！

jsp注重的是“简化html的创建和维护”，servlet注重的是“调用商业逻辑、执行复杂操作”。通常面向处理的任务servlet是最佳选择，而面向表示的任务jsp是最佳选择！通常一个项目中，最好的做法是将二者结合起来使用。可以这样讲：没有一个项目是全部使用jsp或者servlet来开发的！

2. Servlet生命周期

servlet版本2.3

servlet有良好的生存期的定义，包括如何加载、实例化、初始化、处理客户端请求以及如何被移除。这个生存期由javax.servlet.Servlet接口的init,service和destroy方法表达。
1、加载和实例化
容器负责加载和实例化一个servlet。实例化和加载可以发生在引擎启动的时候，也可以推迟到容器需要该servlet为客户请求服务的时候(由loadonstartup配置决定)。

1)load-on-startup元素标记容器是否在启动的时候就加载这个servlet(实例化并调用其init()方法)。

2)它的值必须是一个整数，表示servlet应该被载入的顺序

2)当值为0或者大于0时，表示容器在应用启动时就加载并初始化这个servlet；

3)当值小于0或者没有指定时，则表示容器在该servlet被选择时才会去加载。

4)正数的值越小，该servlet的优先级越高，应用启动时就越先加载。

5)当值相同时，容器就会自己选择顺序来加载。

所以，<load-on-startup>x</load-on-startup>，中x的取值1，2，3，4，5代表的是优先级，而非启动延迟时间。

首先容器必须先定位servlet类，在必要的情况下，容器使用通常的Java类加载工具加载该servlet，可能是从本机文件系统，也可以是从远程文件系统甚至其它的网络服务。容器加载servlet类以后，它会实例化该类的一个实例。需要注意的是可能会实例化多个实例，例如一个servlet类因为有不同的初始参数而有多个定义，或者servlet实现SingleThreadModel而导致容器为之生成一个实例池（所谓实例池，维护多个实例的池子！）。到目前为止，我小崔还一直是用一个实例完成开发！

2、初始化

servlet加载并实例化后，容器必须在它能够处理客户端请求前初始化它。初始化的过程主要是读取永久的配置信息，昂贵资源（例如JDBC连接）以及其它仅仅需要执行一次的任务。通过调用它的init方法并给它传递唯一的一个（每个servlet定义一个）ServletConfig对象完成这个过程。给它传递的这个配置对象允许servlet访问容器的配置信息中的名称 - 值对（name-value）初始化参数。这个配置对象同时给servlet提供了访问实现了ServletContext接口的具体对象的方法，该对象描述了servlet的运行环境。

2.1初始化的错误处理

在初始化期间，servlet实例可能通过抛出UnavailableException 或者 ServletException异常表明它不能进行有效服务。如果一个servlet抛出一个这样的异常，它将不会被置入有效服务并且应该被容器立即释放。在此情况下destroy方法不会被调用因为初始化没有成功完成。在失败的实例被释放后，容器可能在任何时候实例化一个新的实例，对这个规则的唯一例外是如果失败的servlet抛出的异常是UnavailableException并且该异常指出了最小的无效时间，那么容器就会至少等待该时间指明的时限才会重新试图创建一个新的实例。

2.2、工具因素

当工具（注：根据笔者的理解，这个工具可能是应用服务器的某些检查工具，通常是验证应用的合法性和完整性）加载和内省（introspect）一个web应用时，它可能加载和内省该应用中的类，这个行为将触发那些类的静态初始方法被执行，因此，开发者不能假定只要当servlet的init方法被调用后它才处于活动容器运行状态（active container runtime）。作为一个例子，这意味着servlet不能在它的静态（类）初始化方法被调用时试图建立数据库连接或者连接EJB容器。

3、处理请求

在servlet被适当地初始化后，容器就可以使用它去处理请求了。每一个请求由ServletRequest类型的对象代表，而servlet使用ServletResponse回应该请求。这些对象被作为service方法的参数传递给servlet。在HTTP请求的情况下，容器必须提供代表请求和回应的HttpServletRequest和HttpServletResponse的具体实现。需要注意的是容器可能会创建一个servlet实例并将之放入等待服务的状态，但是这个实例在它的生存期中可能根本没有处理过任何请求。

3.1、多线程问题

容器可能同时将多个客户端的请求发送给一个实例的service方法，这也就意味着开发者必须确保编写的servlet可以处理并发问题。如果开发者想防止这种缺省的行为，那么他可以让他编写的servlet实现SingleThreadModel。实现这个类可以保证一次只会有一个线程在执行service方法并且一次性执行完。容器可以通过将请求排队（使用锁机制）或者维护一个servlet实例池（singleThreadMode或者其他方式）满足这一点。如果servlet是分布式应用的一部分，那么，那么容器可能在该应用分布的每个JVM中都维护一个实例池。如果开发者使用synchronized关键字定义service方法(或者是doGet和doPost)，容器将排队处理请求，这是由底层的java运行时系统要求的。我们强烈推荐开发者不要同步service方法或者HTTPServlet的诸如doGet和doPost这样的服务方法。

3.2、处理请求中的异常

servlet在对请求进行服务的时候有可能抛出ServletException或者UnavailableException异常。ServletException表明在处理请求的过程中发生了错误容器应该使用合适的方法清除该请求。UnavailableException表明servlet不能对请求进行处理，可能是暂时的，也可能是永久的。如果UnavailableException指明是永久性的，那么容器必须将servlet从服务中移除，调用它的destroy方法并释放它的实例。如果指明是暂时的，那么容器可以选择在异常信息里面指明的这个暂时无法服务的时间段里面不向它发送任何请求。在这个时间段里面被被拒绝的请求必须使用SERVICE_UNAVAILABLE (503)返回状态进行响应并且应该携带稍后重试（Retry-After）的响应头表明不能服务只是暂时的。容器也可以选择不对暂时性和永久性的不可用进行区分而全部当作永久性的并移除抛出异常的servlet。

3.3线程安全

开发者应该注意容器实现的请求和响应对象（注：即容器实现的HttpServletRequest和HttpServletResponese）没有被保证是线程安全的，这就意味着他们只能在请求处理线程的范围内被使用，这些对象不能被其它执行线程所引用，因为引用的行为是不确定的。

4、服务结束

容器没有被要求将一个加载的servlet保存多长时间，因此一个servlet实例可能只在容器中存活了几毫秒，当然也可能是其它更长的任意时间（但是肯定会短于容器的生存期）

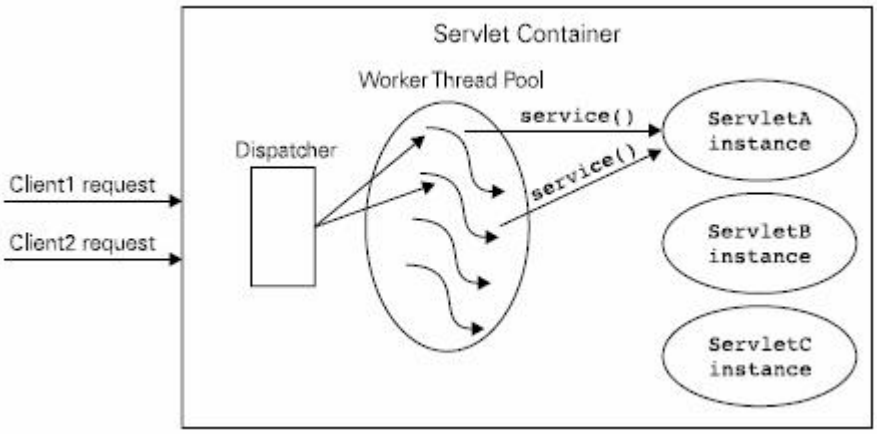
当容器决定将之移除时（原因可能是保存内存资源或者自己被关闭），那么它必须允许

servlet释放它正在使用的任何资源并保存任何永久状态（这个过程通过调用destroy方法达到）。容器在能够调用destroy方法前，它必须允许那些正在service方法中执行的线程执行完或者在服务器定义的一段时间内执行（这个时间段在容器调用destroy之前）。一旦destroy方法被调用，容器就不会再向该实例发送任何请求。如果容器需要再使用该servlet，它必须创建新的实例。destroy方法完成后，容器必须释放servlet实例以便它能够被垃圾回收。

3. 线程安全

很多情况下，可能会有很多并发的请求，为了提高运行效率，节省内存资源，容器采用Thread Pool.

一般情况下容器只对每个servlet生成一个实例，让他服务于所有的请求，对于客户端同时请求一个servlet，他们是被并发的处理的，并不是等上一个请求处理完成再处理下一个。如果两个请求同时到达，那么他们处理完成的时间也是差不多的。



当然有些时候处于并发的同步性考虑，servlet并不适合多线程，那么就可以通过继承标识接口 SingleThreadModel来实现。这样的话servlet就不会被多线程执行，而是生成他的多个实例来提高性能（注意不是一个）；单线程servlet仅仅表示他不会被多线程执行，并不能说明它是线程安全的。

Note:

单实例多线程和单线程servlet具体区别：

多线程情况下每个线程会保存实例中的局部变量的一个copy，对局部变量的修改只会影响自己的copy不会影响别的线程，所以局部变量是安全的。但是对于全局变量，是属于所有线程的。对他的修改会影响到其他的线程。

而对于单线程servlet，会有多个实例，这些实例的变量都是属于自己的，可能是线程安全的（原因是之前使用过的实例会被放入到实例池中，它可能会被下次请求使用），局部的变量是线程安全的；

但是在这两种情况下static变量都不是线程安的。

也可以通过Synchronizing servlet的service()方法来实现同步，但是这种方法不是很有效，比起SingleThreadModel方式来效率要低，所以很少用。原因是，这种方式的servlet在容器中只存在一个实例，而对于SingleThreadModel方式的会有多个实例同时存在，当然效率上会有差别。

对于Context,Session,Request的线程安全问题：

毫无疑问Context不是线程安全的，而对于Session呢？Session是被正在处理属于这个Session的线程来访问的，安理说他只会同时被一个线程访问，应该是线程安全的。但是也有特殊情况，如果一个客户同时开了多个窗口来同时提交请求的，还会出现多个线程同时访问一个Session的情况。Request是线程安全的。

小结：

1．设计servlet的时候要考虑线程安全的问题。

2．对于单线程servlet，要通过SingleThreadModel来实现，不要通过Synchronizing来实现。

3．SingleThreadModel并不能保证线程安全，他只保证不被多线程执行。

4．对于Context,Session的数据修改要考虑同步。或者其他方式来避免线程安全问题。

5．Servlet中可以把不变常量写成全局，或者静态。

6．对于不想被其他线程共同访问的变量要写成局部变量。保证安全。

1.8 url-pattern

发表时间: 2016-01-12

问题：为什么我使用“/”时 jsp没有被拦截，而其他包括css等静态资源都会被拦截而“/*”将会拦截所有资源？

准备知识：

一、url-pattern详解：

1、以”/’ 开头和以”/*”结尾的是用来做路径映射的。

2、以前缀”*.”开头的是用来做后缀（或称扩展）映射的。

3、 “/” 是用来定义default servlet映射的。

但是 你只能选择路径或者后缀映射，而不能同时用使用这两种映射，因为这样容器无法判断！例如这样的映射 “/*.action”

二、servlet 匹配规则：

当一个请求发送到servlet容器的时候，容器先会将请求的url减去当前应用上下文的路径作为servlet的映射url，

比如我访问的是 http://localhost/test/aaa.html，我的应用上下文是test，容器会将http://localhost/test去掉，

剩下的/aaa.html部分拿来做servlet的映射匹配。这个映射匹配过程是有顺序的，而且当有一个servlet匹配成功以后，

就不会去理会剩下 的servlet了.

其匹配规则和顺序如下：

1》精确路径匹配。例子：比如servletA 的url-pattern为 /test，servletB的url-pattern为 /* ，

这个时候，如果我访问的url为http://localhost/test ，

这个时候容器就会先 进行精确路径匹配，发现/test正好被servletA精确匹配，

那么就去调用servletA，也不会去理会其他的servlet了。

2》最长路径匹配。例子：servletA的url-pattern为/test/*，而servletB的url-pattern为/test/a/*，

此 时访问http://localhost/test/a时，容器会选择路径最长的servlet来匹配，也就是这里的servletB。

3》扩展匹配，如果url最后一段包含扩展，容器将会根据扩展选择合适的servlet。

例子：servletA的url-pattern：*.action

4》如果前面三条规则都没有找到一个servlet，容器会根据url选择对应的请求资源。

如果应用定义了一个default servlet，则容器会将请求丢给default servlet（什么是default servlet？就是以“default”配置的serverlet）。

总结以上规则可细化为以下便于记忆：

1：/abc/* ------1

2：/* ------2

3：/abc ------3

4：*.do ------4

5：/

三、tomcat的 “default servlet”和“jsp servlet” ，摘录如下：

```
<servlet>
    <servlet-name>default</servlet-name>
    <servlet-class>org.apache.catalina.servlets.DefaultServlet</servlet-class>
    <init-param>
      <param-name>debug</param-name>
      <param-value>0</param-value>
    </init-param>
```

```
        <init-param>
            <param-name>listings</param-name>
            <param-value>false</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
        <servlet-name>default</servlet-name>
        <url-pattern>/</url-pattern>
</servlet-mapping>



<servlet>
        <servlet-name>jsp</servlet-name>
        <servlet-class>org.apache.jasper.servlet.JspServlet</servlet-class>
        <init-param>
            <param-name>fork</param-name>
            <param-value>false</param-value>
        </init-param>
        <init-param>
            <param-name>xpoweredBy</param-name>
            <param-value>false</param-value>
        </init-param>
        <load-on-startup>3</load-on-startup>
```

```
</servlet>
<servlet-mapping>
        <servlet-name>jsp</servlet-name>
        <url-pattern>*.jsp</url-pattern>
    </servlet-mapping>

    <servlet-mapping>
        <servlet-name>jsp</servlet-name>
        <url-pattern>*.jspx</url-pattern>
</servlet-mapping>
```

四、对问题的解释

当你的spring mvc 的“dispatcher servlet”配置为：“/” 时，

它会替换掉“default servlet” ，此时“jsp servlet”先拦截，如果拦截失败 会才采用“default servlet”也即“dispatcher servlet” ！

显然：此时除了jsp及jspx外，dispatch servlet将拦截所有资源！

当你使用“/*”时，显然它会拦截所有资源，包括“jsp、jspx、js、css等”

当你使用“*.action”时，

显然对于“.jsp、.jspx”会被“jsp servlet”拦截。

对于“.js、.css”会被“的“dispatcher servlet”拦截。

对于“.action”会被“dispatcher servlet”拦截。

五、当spring mvc 使用“/”或者“/*”时如何放过像“js等静态资源”？

解决办法1：在项目的web.xml加入如下代码：

```
<servlet-mapping>
    <servlet-name>default</servlet-name>
    <url-pattern>*.htm</url-pattern>
</servlet-mapping>
```

这样的话 会优先匹配“*.htm”！

解决办法2：采用<mvc:default-servlet-handler />

在springMVC-servlet.xml中配置<mvc:default-servlet-handler />后，

会在Spring MVC上下文中定义一个 org.springframework.web.servlet.resource.DefaultServletHttpRequestHandler，

它会像一个检查员，对进入DispatcherServlet的URL进行筛查，如果发现是静态资源的请求，

就将该请求转由Web应用服务器默认的Servlet处理，如果不是静态资源的请求，才由DispatcherServlet继续处理。

一般Web应用服务器默认的Servlet名称是"default"，因此DefaultServletHttpRequestHandler可以找到它。

如果你所有的Web应用服务器的默认Servlet名称不是"default"，则需要通过default-servlet-name属性显示指定：

<mvc:default-servlet-handler default-servlet-name="所使用的Web服务器默认使用的Servlet名称" />

解决办法3：采用<mvc:resources />

<mvc:default-servlet-handler />将静态资源的处理经由Spring MVC框架交回Web应用服务器处理。而<mvc:resources />更进一步，

由Spring MVC框架自己处理静态资源，并添加一些有用的附加值功能。

首先，<mvc:resources />允许静态资源放在任何地方，如WEB-INF目录下、类路径下等，你甚至可以将JavaScript等静态文件打到JAR包中。

通过location属性指定静态资源的位置，由于location属性是Resources类型，因此可以使用诸如"classpath:"等的资源前缀指定资源位置。

传统Web容器的静态资源只能放在Web容器的根路径下，<mvc:resources />完全打破了这个限制。

其次，<mvc:resources />依据当前著名的Page Speed、YSlow等浏览器优化原则对静态资源提供优化。

你可以通过cacheSeconds属性指定静态资源在浏览器端的缓存时间，一般可将该时间设置为一年，以充分利用浏览器端的缓存。

在输出静态资源时，会根据配置设置好响应报文头的Expires 和 Cache-Control值。

在接收到静态资源的获取请求时，会检查请求头的Last-Modified值，如果静态资源没有发生变化，则直接返回303相应状态码，

提示客户端使用浏览器缓存的数据，而非将静态资源的内容输出到客户端，以充分节省带宽，提高程序性能。

在springMVC-servlet中添加如下配置：

<mvc:resources location="/,classpath:/META-INF/publicResources/" mapping="/resources/**"/>

以上配置将Web根路径"/"及类路径下 /META-INF/publicResources/ 的目录映射为/resources路径。

假设Web根路径下拥有images、js这两个资源目录,在images下面有bg.gif图片，在js下面有test.js文件，

则可以通过 /resources/images/bg.gif 和 /resources/js/test.js 访问这二个静态资源。

假设WebRoot还拥有images/bg1.gif 及 js/test1.js，则也可以在网页中通过 /resources/images/bg1.gif

及 /resources/js/test1.js 进行引用。

1.9 常见问题

发表时间: 2016-01-12

1、Jsp标签中动态INCLUDE与静态INCLUDE的区别？

答：动态INCLUDE用jsp:include动作实现
<jsp:include page="included.jsp"/>它总是会检查所含文件中的变化，适合用于包含动态页面，并且可以带参数。

静态INCLUDE用include伪码实现(也即伪指令实现),不会检查所含文件的变化，适用于包含静态页面
<%@ include file="included.htm" %>

静态包含适用于不经常变动的文件，而动态包含适用于变化比较大的文件！

我们查看这些jsp->java代码：静态包含包含文件和被包含文件最后形成的是一个java代

文件。而动态包含最终形成的是两个java代码文件！

我们深入研究一下“使用了这两个标签的jsp文件对应的java代码段”：

假设 用b.jsp分别进行一次静态的和动态的包含c.jsp。

发现

对于静态包含（<%@ include file="c.jsp" %>）：

```
out.write("<!-- c.jsp开始 --> ");
    out.print(request.getParameter("ano") );
    out.write('\r');
    out.write('\n');
    out.print(request.getParameter("bno") );
```

```
    out.write("\r\n");
out.write("<!-- c.jsp结束 -->");
```

对于动态包含（ <jsp:include page="included.jsp"/> ）的代码片段：

```
if (true) {
    _jspx_page_context.forward("c.jsp");
    return;
}
```

说明静态包含是把c.jsp文件先翻译为java代码，然后把这些java代码加入到

b.jsp翻译后的java代码中，位置不变（在jsp中的位置和翻译为java代码后的位置）

这里需要指出的是c.jsp中不能使用<html>、</html>、<body>、</body>标记，因为这将会影响在原JSP文件中同样的标记，有时会导致错误，如果去掉了这些标记还出错，那么你要检测被包含文件和包含文件中使用的<%%>是否有相同的变量或方法！为什么？我们之前说过这个标签内部的代码都要放到jsp_service方法中，这两个文件如果有相同的变量，在使用静态包含后，就相当于你在这个标签中进行了两次相同变量的声明！

对于动态包含，我们发现它其实它是服务器端跳转！

2、Jsp标签两种跳转方式分别是什么?有什么区别?

答：有两种，分别为：
<jsp:include page="included.jsp" flush="true">
<jsp:forward page= "nextpage.jsp"/>
前者页面不会转向include所指的页面，只是显示该页的结果，主页面还是原来的页面。执行完后还会回来，相当于函数调用。并且可以带参数.

后者完全转向新页面，不会再回来。相当于go to 语句。

您可能已注意到 jsp:include 代码示例中的 flush 属性。顾名思义， flush 指示在读入包含内容之前是否清空任何现有的缓冲区。JSP 1.1 中需要 flush 属性“=true” ，因此，如果代码中不用它，会得到一个错误。但是，在 JSP 1.2 中， flush 属性缺省为 false。由于清空大多数时候不是一个重要的问题，因此，我的建议是：对于 JSP 1.1，将 flush 设置为 true；而对于 JSP 1.2 及更高版本，将其设置为关闭。

清空现有缓冲区,是清空那里?tomcat目录下的work目录就是所谓的缓冲区!

个人理解，仅存在两种跳转（跳转）：“服务器转向”和“客户端转向”。

假设：在视图a进行表单提交，请求url-A，url-A在服务端请求向url-B，....，url-N，

服务器器转向的特点：地址栏不变，请求信息不丢失；而客户端跳转却恰恰相反，它仅仅是向客户端发送一个状态码和地址（这个地址可以携带参数），让客户端那这个地址去重新访问服务器端。

以上述“假设”为基础，说这些url之间的转向是服务器转向，意思是说他们之间的跳转发生在服务器端，它给客户一个错觉让客户感觉到它访问的是url-A，其实不是，它访问的是url-N，但客户不知道！根据上面的“假设”我们并不能判断出视图a到url-N之间他们究竟进行的是什么跳转！这要看这种跳转是由谁引导的，

服务器跳转发生在下列两种情况下：

* <jsp:forward page=”” />

* request.getRequestDispatcher("url").forward(request, response);

而客户端跳转发生在下列几种情况

* 表单提交

* 超链接

* response.sendRedirect("url?参数");

3、JspWriter和PrintWriter

JspWriter是jsp 9大内置对象之一，是javaEE范畴，在servlet中只能通过PageContext内置对象获得，而PageContext只能通过JspFactory获得。而且JspWriter是一个继承自j2se抽象类Writer的抽象类。

PrintWriter属于j2se范畴，继承自j2se抽象类Writer的类，而且PrintWriter不是抽象类。而且在servlet中可以通过response获得。

JspWriter和PrintWriter这两个类的方法在没有牵扯到缓冲的时候方法可以通用。所以我们在servlet类中经常使用PrintWriter类。

4、<%%>和<!%%>

<%%>与<!%%>的区别，后者声明全局变量，后者可以声明方法，前者不能声明方法，其他都相同！可以这样说，前者在jsp翻译为servlet后位于一个方法内（_jspService方法），后者将成为servlet类的一个字段或者该类的一个方法！

5、异常

异常名字：java.lang.IllegalStateException: getOutputStream() has already been called for this response

在jsp或者在servlet中有时要用到response.getOutputStream()，但是此时会在后台报这个错误java.lang.IllegalStateException: getOutputStream() has already been called for this respons，这问题困扰了我好久都没解决，最近这个项目中我又遇到了，下定决心一定要解决掉，最后终于让我给找到解决的方法了，这个异常时因为response.getOutputStream()跟response.getWriter()相冲突早场的，呵呵！现在记录下，发出来和大家共享下，希望能帮到遇到同样问题的朋友们，解决方法如下：

```
out.clearBuffer();

out = pageContext.pushBody();
```

在调用response.getOutputStream()之前加上上面两代码，就ok了！

为什么？

JSP的主要功能是描述发送到客户请求的响应输出流中的数据。这个输出流通过out对象显露给JSP开发员。

Out对象是javax.servlet.jsp.JspWriter对象的实例。这个对象可以代表输出流、经过滤的输出流或来自其他JSP页面的嵌套的JspWriter对象。但是输出应该不会被直接发送到输出流，因为在JSP的生命期内可能有多个输出流。

根据页面是否被缓冲，初始的JspWriter对象的初始化有所不同。缺省情况下，每个JSP页面都打开了缓冲，这可以提高性能。缓冲功能很容易关闭，只要使用page指令的buffered='false'属性即可。

缓冲的out对象以块为单位收集和发送数据，这通常会提供最好的总体吞吐量。使用缓冲时，PrintWriter在第一个块被发送时创建，也就是在out第一次调用flush()时。

如果不缓冲输出，将立即创建PrintWriter对象并引用out对象。在此情况下，发送到out对象的数据立即被发送到输出流。创建PrintWriter对象时将使用由服务器决定的缺省设置和头信息。

注意：HTTP头和缓冲

HTTP使用响应头描述服务器以及定义发往客户的数据的某些方面信息。这可能包括页面的MIME内容类型、新的cookie、转发URL或其他HTTP“动作”。

JSP允许开发者在创建OutputStream（即PrintWriter）前改变响应头的内容。一旦建立了OutputStream，头信息就不能改变了，因为它已被发送到客户。

在缓冲的out对象的情况下，直至缓冲区第一次刷新时才建立OutputStream。缓冲区被刷出很大程度上取决于page指令的autoFlush和bufferSize属性。通常，最好在有任何数据被发送到out对象前设置头信息。

对无缓冲的out对象，很难设置页面头。当无缓冲的页面建立时，几乎立即就建立了OutputStream。

在建立OutputStream后发送的头可能会造成大量不正常的结果。一些头被简单地忽略，其他的头可能产生异常，例如IllegalStateException。

JspWriter包含的方法大多数与java.io.PrintWriter类一样。但是JspWriter有另外几个用于处理缓冲的方法。与PrintWriter对象不同，JspWriter抛出IOExceptions。在JSP中，这些异常需要显式捕获和处理。

注意：autoFlush( )

在JSP中，缺省的缓冲行为是在缓冲区满了时自动刷出缓冲区。但是，有时JSP实际上是直接和另一个应用程序通信。在此情况下，不“何时”的行为可能是在超出缓冲区时抛出异常。

设置page指令的属性autoFlush='false'将造成缓冲区溢出而抛出异常。

5、pageEncoding与contentType

pageEncoding的charset是jsp文件本身的编码

contentType的charset是指服务器发送给客户端时的内容编码

JSP要经过两次的“编码”，第一阶段会用pageEncoding，第二阶段会用utf-8至utf-8，第三阶段就是由Tomcat出来的网页， 用的是contentType。

第一阶段是jsp编译成.java，它会根据pageEncoding的设定读取jsp，结果是由指定的编码方案翻译成统一的UTF-8 JAVA源码（即.java），如果pageEncoding设定错了，或没有设定，出来的可能就是中文乱码。

第二阶段是由JAVAC的JAVA源码至java byteCode的编译，不论JSP编写时候用的是什么编码方案，经过这个阶段的结果全部是UTF-8的encoding的java源码。

JAVAC用UTF-8的encoding读取java源码，编译成UTF-8 encoding的二进制码（即.class），这是JVM对常数字串在二进制码（java encoding）内表达的规范。

第三阶段是Tomcat（或其的application container）载入和执行阶段二的来的JAVA二进制码，输出的结果，也就是在客户端见到的，这时隐藏在阶段一和阶段二的参数 contentType就发挥了功效

contentType的设定.

pageEncoding 和contentType的预设都是 ISO8859-1. 而随便设定了其中一个, 另一个就跟着一样了(TOMCAT4.1.27是如此). 但这不是绝对的, 这要看各自JSPC的处理方式. 而pageEncoding不等于contentType, 更有利亚洲区的文字CJKV系JSP网页的开发和展示, (例pageEncoding=GB2312 不等于 contentType=utf-8)。

jsp文件不像.java，.java 在被编译器读入的时候默认采用的是操作系统所设定的locale所对应的编码，比如中国大陆就是GBK，台湾就是BIG5或者MS950。而一般我们不管 是在记事本还是在ue中写代码，如果没有经过特别转码的话，写出来的都是本地编码格式的内容。所以编译器采用的方法刚好可以让虚拟机得到正确的资料。

但是jsp文件不是这样，它没有这个默认转码过程，但是指定了pageEncoding就可以实现正确转码了。

举个例子:

<%@ page contentType="text/html;charset=utf-8" %>

大都会打印出乱码，因为输入的“你好”是gbk的，但是服务器是否正确抓到“你好”不得而知。

但是如果更改为

<%@ page contentType="text/html;charset=utf-8" pageEncoding="GBK"%>

这样就服务器一定会是正确抓到“你好”了。