# Servlet与Servlet容器

## Servlet

### Servlet是什么

Java Servlet是Java语言中编写Web服务器扩展功能的重要技术，同时它也是JSP技术的底层运行基础。它是运行在Web服务器或应用服务器上的程序，它是作为来自Web浏览器或其他HTTP客户端的请求和HTTP服务器上的数据库或应用程序之间的中间层。使用Servlet，可以收集来自网页表单的用户输入，呈现来自数据库或者其他源的记录，还可以动态创建网页。Servlet实际上就是按照Servlet规范编写的一个Java类。

Java Servlet通常情况下与使用CGI（公共网关接口）实现的程序可以达到异曲同工的效果。但是相比于CGI，Servlet有以下优势：

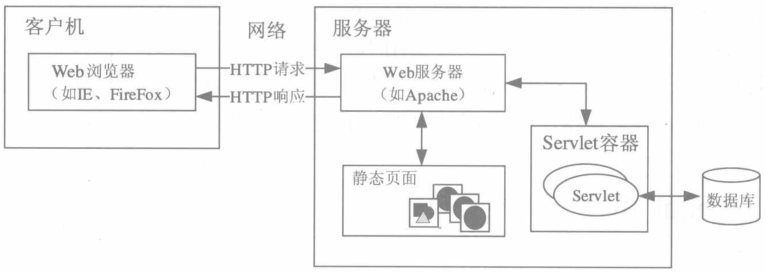
1. Servlet是单实例多线程的运行方式，每个请求在一个独立的线程中运行，而提供服务的Servlet实例只有一个。
2. Servlet具有可升级性，能响应更多的请求，因为Servlet容器使用一个线程而不是操作系统进程，而线程仅占用有限的系统资源。
3. Servlet使用标准的API，被更多的web服务器所支持。
4. Servlet使用Java语言编写，因此拥有Java程序语言的所有优点，包括容易开发和平台独立性。
5. Servlet可以访问Java平台丰富的类库，使得各种应用的开发更为容易。
6. Servlet容器给Servlet提供额外的功能，如错误处理和安全。

**Java Servlet的特点：**

1. Servlet不是一个独立的程序，没有main()方法。
2. Servlet不是由用户或程序员直接调用，而是生存在Servlet容器中，由Servlet容器管理。
3. Servlet包含init()和destroy()方法，在生命周期内有效。
4. Servlet没有图像界面。

### Servlet架构

Servlet在Web应用程序中的位置



### Servlet任务

Servlet执行以下主要任务：

1. 读取客户端（浏览器）发送的显式的数据。这包括网页上的HTML表单，或者也可以是来自applet或自定义的HTTP客户端程序的表单。
2. 读取客户端（浏览器）发送的隐式的HTTP请求数据。这包括cookies、媒体类型和浏览器能理解的压缩格式等等。
3. 处理数据并生成结果。这个过程可能需要访问数据库，执行RMI或CORBA调用，调用Web服务，或者直接计算得出对应的响应。
4. 发送显式的数据（即文档）到客户端（浏览器）。该文档的格式可以是多种多样的，包括文本（HTML或XML）、二进制文件（GIF图像）、Excel等
5. 发送隐式的HTTP相应到客户端（浏览器）。这包括告诉浏览器或其他客户端被返回的文档类型（例如HTML），设置cookies和缓存参数，以及其他类似的任务。

### Servlet包

Java Servlet是运行在带有支持Java Servlet规范的解释器的web服务器上的Java类。

Servlet可以使用java.servlet和javax.servlet.http包创建，它是Java企业版的标准组成部分，Java企业版是支持大型开发项目的Java类库的扩展版本。这些类实现Java Servlet和JSP规范。

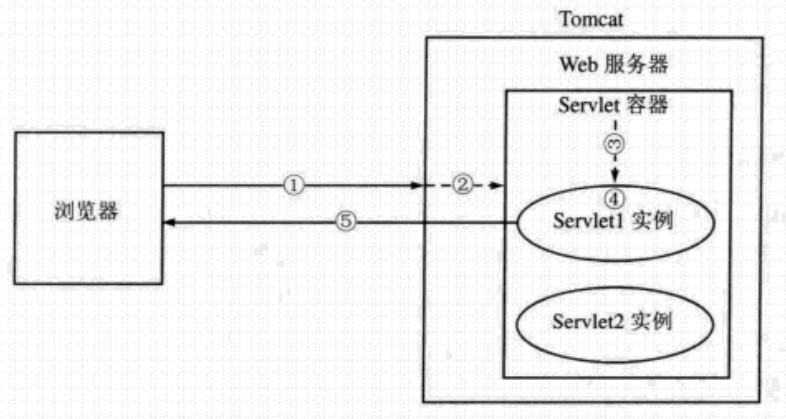
## Servlet容器

Servlet容器有时候也叫做Servlet引擎，是Web服务器或应用程序服务器的一部分，用于在发送的请求和响应之上提供网络服务，解码基于MIME的请求，格式化基于MIME的响应。Servlet容器在Servlet的生命周期内包容和管理Servlet。

### Tomcat介绍

Tomcat服务器是一个免费的开源Web应用服务器，它是Apache软件基金会的Jakarta项目中的一个核心项目，由Apache、Sun和其他一些公司及个人共同开发而成。由于有了Sun的参与和支持，最小的Servlet和JSP规范总是能在Tomcat中得到体现。因为Tomcat技术先进、性能稳定，而且免费，所以Tomcat成为目前比较流行的Web应用服务器。

Tomcat服务器接受客户请求并做出相应的图例：



1. 客户端（通常都是浏览器）访问Web服务器，发送HTTP请求。
2. Web服务器接收到请求后，传递给Servlet容器
3. Servlet容器加载Servlet，产生Servlet实例后，向其传递表示请求和响应的对象
4. Servlet实例使用请求对象得到客户端的请求信息，然后进行相应的处理
5. Servlet实例将处理结果通过响应对象发送回客户端，容器负责确保响应正确发出，同时将控制返回给Web服务器。

### Tomcat的目录结构及其用途

|  |  |
| --- | --- |
| 目录 | 用途 |
| /bin | 存放启动和关闭Tomcat的脚本文件（wendows下的启动文件为startup.bat） |
| /common/lib | 存放Tomcat服务器及所有Web应用程序都可以访问的JAR文件 |
| /conf | 存放Tomcat服务器的各种配置文件，其中包括server.xml（Tomcat的主要配置文件）、tomcat-users.xml和web.xml等配置文件 |
| /logs | 存放Tomcat的日志文件 |
| /server/lib | 存放Tomcat服务器运行所需要的各种JAR文件 |
| /server/webapps | 存放Tomcat的两个Web应用程序：admin应用程序和manager应用程序 |
| /shared/lib | 存放所有Web应用程序都可以访问的JAR文件 |
| /temp | 存放Tomcat运行是产生的临时文件 |
| /webapps | 当发布Web应用程序时，通常把Web应用程序的目录及文件放到这个目录下 |
| /work | Tomcat将JSP生成的Servlet源文件和字节码文件放到这个目录下 |

### Tomcat配置文件

server.xml是 Tomcat 的主配置文件，在 <CATALINA\_HOME>conf 目录下可以看到。重构后（删掉注释和格式化后）的默认”server.xml“文件如下：

<?xml version=*'1.0'* encoding=*'utf-8'*?>

<Server port=*"8005"* shutdown=*"SHUTDOWN"*>

<Listener className=*"org.apache.catalina.core.JasperListener"* />

<Listener className=*"org.apache.catalina.core.AprLifecycleListener"* SSLEngine=*"on"* />

<Listener className=*"org.apache.catalina.core.JreMemoryLeakPreventionListener"* />

<Listener className=*"org.apache.catalina.mbeans.GlobalResourcesLifecycleListener"* />

<Listener className=*"org.apache.catalina.core.ThreadLocalLeakPreventionListener"* />

<GlobalNamingResources>

<Resource name=*"UserDatabase"* auth=*"Container"* type=*"org.apache.catalina.UserDatabase"*

description=*"User database that can be updated and saved"*

factory=*"org.apache.catalina.users.MemoryUserDatabaseFactory"*

pathname=*"conf/tomcat-users.xml"* />

</GlobalNamingResources>

<Service name=*"Catalina"*>

<Connector port=*"8080"* protocol=*"HTTP/1.1"* connectionTimeout=*"20000"* redirectPort=*"8443"* />

<Connector port=*"8009"* protocol=*"AJP/1.3"* redirectPort=*"8443"* />

<Engine name=*"Catalina"* defaultHost=*"localhost"*>

<Realm className=*"org.apache.catalina.realm.LockOutRealm"*>

<Realm className=*"org.apache.catalina.realm.UserDatabaseRealm"*

resourceName=*"UserDatabase"*/>

</Realm>

<Host name=*"localhost"* appBase=*"webapps"* unpackWARs=*"true"* autoDeploy=*"true"*>

<Valve className=*"org.apache.catalina.valves.AccessLogValve"* directory=*"logs"*

prefix=*"localhost\_access\_log."* suffix=*".txt"* pattern=*"%h %l %u %t "*%r" %s %b" />

</Host>

</Engine>

</Service>

</Server>

|  |  |  |
| --- | --- | --- |
| 元素名 | 属性 | 解释 |
| server | port | 指定一个端口，这个端口负责监听关闭tomcat的请求 |
| shutdown | 指定向端口发送的命令字符串 |
| service | name | 指定service的名字 |
| Connector(表示客户端和service之间的连接) | port | 指定服务器端要创建的端口号，并在这个断口监听来自客户端的请求 |
| minProcessors | 服务器启动时创建的处理请求的线程数 |
| maxProcessors | 最大可以创建的处理请求的线程数 |
| enableLookups | 如果为true，则可以通过调用request.getRemoteHost()进行DNS查询来得到远程客户端的实际主机名，若为false则不进行DNS查询，而是返回其ip地址 |
| redirectPort | 指定服务器正在处理http请求时收到了一个SSL传输请求后重定向的端口号 |
| acceptCount | 指定当所有可以使用的处理请求的线程数都被使用时，可以放到处理队列中的请求数，超过这个数的请求将不予处理 |
| connectionTimeout | 指定超时的时间数(以毫秒为单位) |
| Engine(表示指定service中的请求处理机，接收和处理来自Connector的请求) | defaultHost | 指定缺省的处理请求的主机名，它至少与其中的一个host元素的name属性值是一样的 |
| Context(表示一个web应用程序，通常为WAR文件，关于WAR的具体信息见servlet规范) | docBase | 应用程序的路径或者是WAR文件存放的路径 |
| path | 表示此web应用程序的url的前缀，这样请求的url为http://localhost:8080/path/\*\*\*\* |
| reloadable | 这个属性非常重要，如果为true，则tomcat会自动检测应用程序的/WEB-INF/lib 和/WEB-INF/classes目录的变化，自动装载新的应用程序，我们可以在不重起tomcat的情况下改变应用程序 |
| host(表示一个虚拟主机) | name | 指定主机名 |
| appBase | 应用程序基本目录，即存放应用程序的目录 |
| unpackWARs | 如果为true，则tomcat会自动将WAR文件解压，否则不解压，直接从WAR文件中运行应用程序 |
| Logger(表示日志，调试和错误信息) | className | 指定logger使用的类名，此类必须实现org.apache.catalina.Logger 接口 |
| prefix | 指定log文件的前缀 |
| suffix | 指定log文件的后缀 |
| timestamp | 如果为true，则log文件名中要加入时间，如下例:localhost\_log.001-10-04.txt |
| Realm(表示存放用户名，密码及role的数据库) | className | 指定Realm使用的类名，此类必须实现org.apache.catalina.Realm接口 |
| Valve(功能与Logger差不多，其prefix和suffix属性解释和Logger 中的一样) | className | 指定Valve使用的类名，如用org.apache.catalina.valves.AccessLogValve类可以记录应用程序的访问信息 |
| directory | 指定log文件存放的位置 |
| pattern | 有两个值，common方式记录远程主机名或ip地址，用户名，日期，第一行请求的字符串，HTTP响应代码，发送的字节数。combined方式比common方式记录的值更多 |

**<Server>元素**

它代表整个容器,是Tomcat实例的顶层元素.由org.apache.catalina.Server接口来定义.它包含一个<Service>元素.并且它不能做为任何元素的子元素.

<!--

一个“Server”是一个提供完整的JVM的独立组件，它可以包含一个或多个“Service”实例。服务器在指定的端口上监听shutdown命令。注意：一个“Server”自身不是一个“Container”（容器），因此在这里你不可以定义诸如“Valves”或者“Loggers”子组件

-->

<!-- 启动Server在端口8005处等待关闭命令如果接受到"SHUTDOWN"字符串则关闭服务器-->

测试：

telnet localhost 8005

输入：SHUTDOWN

结果：关闭tomcat

<Server port="8005" shutdown="SHUTDOWN" debug="0">

1. className指定实现org.apache.catalina.Server接口的类.默认值为org.apache.catalina.core.StandardServer
2. port指定Tomcat监听shutdown命令端口.终止服务器运行时,必须在Tomcat服务器所在的机器上发出shutdown命令.该属性是必须的.
3. shutdown指定终止Tomcat服务器运行时,发给Tomcat服务器的shutdown监听端口的字符串.该属性必须设置

**<Service>元素**

该元素由org.apache.catalina.Service接口定义,它包含一个<Engine>元素,以及一个或多个<Connector>,这些Connector元素共享用同一个Engine元素

<!--

一个“Service”是一个或多个共用一个单独“Container”(容器)的“Connectors”组合（因此，应用程序在容器中可见）。通常，这个容器是一个“Engine”（引擎），但这不是必须的。注意：一个“Service”自身不是一个容器，因此，在这个级别上你不可定义诸如“Valves”或“Loggers”子组件。

-->

<!-- Tomcat的Standalone Service Service是一组Connector的集合它们共用一个Engine来处理所有Connector收到的请求-->

<Service name="Catalina">

<Service name="Apache">

第一个<Service>处理所有直接由Tomcat服务器接收的web客户请求.

第二个<Service>处理所有由Apahce服务器转发过来的Web客户请求

1. className 指定实现org.apahce.catalina.Service接口的类.默认为org.apahce.catalina.core.StandardService
2. name定义Service的名字

**<Connector>元素**

由Connector接口定义.<Connector>元素代表与客户程序实际交互的给件,它负责接收客户请求,以及向客户返回响应结果.

<!--

一个“Connector”（连接器）代表一个请求被接收和应答所需要的端点。每个连接器通过相关的“Container”（容器）处理请求。默认情况下，一个非SSL的HTTP/1.1的连接器被绑定在端口8080。你也可以通过根据后面的使用说明并取消第二个连接器入口的注释，在端口8443上建立一个SSL HTTP/1.1的连接器。开放SSL支持需要下面几步（参见Tomcat 5文档中怎样配置SSL的说明以取得更多的详细信息）：

1. 如果你的JDK是1.3或1.3以前的版本，下载安装JSSE 1.0.2或以后版本，并放置JAR文件到“$JAVA\_HOME/jre/lib/ext”目录下。
2. 带一个“changeit”的口令值执行： %JAVA\_HOME%\bin\keytool -genkey -alias tomcat -keyalg RSA (Windows) $JAVA\_HOME/bin/keytool -genkey -alias tomcat -keyalg RSA (UNIX)来生成它自己的证书私钥。默认情况下，当一个web应用程序调用请求时，DNS查询是可行的。这将对性能造成一些不利的影响，因此，你可以将“enableLookups”设置为“false”来关闭DNS查询。当DNS查询被关闭时，request.getRemoteHost()将返回包含远程客户IP地址的字符串。

-->

<!-- Coyote HTTP/1.1 Connector

className : 该Connector的实现类是org.apache.coyote.tomcat4.CoyoteConnector

port : 在端口号8080处侦听来自客户browser的HTTP1.1请求.如果把8080改成80,则只要输入http://localhost/即可

protocol:设定Http协议,默认值为HTTP/1.1

minSpareThreads: 该Connector先创建5个线程等待客户请求，每个请求由一个线程负责

maxSpareThread:设定在监听端口的线程的最大数目,这个值也决定了服务器可以同时响应客户请求的最大数目.默认值为200

acceptCount : 当现有线程已经达到最大数75时，为客户请求排队.当队列中请求数超过100时，后来的请求返回Connection refused错误

redirectport : 当客户请求是https时，把该请求转发到端口8443去

enableLookups:如果设为true,表示支持域名解析,可以把IP地址解析为主机名.WEB应用中调用request.getRemoteHost方法返回客户机主机名.默认值为true

connectionTimeout:定义建立客户连接超时的时间.如果为-1,表示不限制建立客户连接的时间

allowTrace：是否允许HTTP的TRACE方法，默认为false

emptySessionPath：如果设置为true，用户的所有路径都将设置为/，默认为false。

enableLookups：调用request、getRemoteHost()执行DNS查询，以返回远程主机的主机名，如果设置为false，则直接返回IP地址。

maxPostSize：指定POST方式请求的最大量，没有指定默认为2097152。

protocol：值必须为HTTP1.1，如果使用AJP处理器，该值必须为AJP/1.3

proxyName：如这个连接器正在一个代理配置中被使用，指定这个属性，在request.getServerName()时返回

redirectPort：如连接器不支持SSL请求，如收到SSL请求，Catalina容器将会自动重定向指定的端口号，让其进行处理。

scheme：设置协议的名字，在request.getScheme()时返回，SSL连接器设为”https”，默认为”http”

secure：在SSL连接器可将其设置为true，默认为false

URIEncoding：用于解码URL的字符编码，没有指定默认值为ISO-8859-1

useBodyEncodingForURI：主要用于Tomcat4.1.x中，指示是否使用在contentType中指定的编码来取代URIEncoding，用于解码URI查询参数，默认为false

xpoweredBy：为true时，Tomcat使用规范建议的报头表明支持Servlet的规范版本，默认为false

acceptCount：当所有的可能处理的线程都正在使用时，在队列中排队请求的最大数目。当队列已满，任何接收到的请求都会被拒绝，默认值为10

bufferSize：设由连接器创建输入流缓冲区的大小，以字节为单位。默认情况下，缓存区大的大小为2048字节

compressableMimeType：MIME的列表，默认以逗号分隔。默认值是text/html，text/xml，text/plain

compression：指定是否对响应的数据进行压缩。off：表示禁止压缩、on：表示允许压缩（文本将被压缩）、force：表示所有情况下都进行压缩，默认值为off

connectionTimeout：设置连接的超时值，以毫秒为单位。默认值为60000=60秒

disableUploadTimeOut：允许Servlet容器，正在执行使用一个较长的连接超时值，以使Servlet有较长的时间来完成它的执行，默认值为false

maxHttpHeaderSize：HTTP请求和响应头的最大量，以字节为单位，默认值为4096字节

maxKeepAliveRequest：服务器关闭之前，客户端发送的流水线最大数目。默认值为100

maxSpareThreads：允许存在空闲线程的最大数目，默认值为50

minSpareThreads：设当连接器第一次启协创建线程的数目，确保至少有这么多的空闲线程可用。默认值为4

port：服务端套接字监听的TCP端口号，默认值为8080（必须）

socketBuffer：设Socket输出缓冲区的大小（以字节为单位），-1表示禁止缓冲，默认值为9000字节

toNoDelay：为true时，可以提高性能。默认值为true

threadPriority：设JVM中请求处理线程优先级。默认值为NORMAL-PRIORITY

例：

<Connector

port="8080" maxHttpHeaderSize="8192" maxThreads="150" minSpareThreads="25" maxSpareThreads="75" enableLookups="false" redirectPort="8443" acceptCount="100" connectionTimeout="20000" disableUploadTimeout="true" />

AJP连接器：

用于将Apache与Tomcat集成在一起，当Apache接收到动态内容请求时，通过在配置中指定的端口号将请求发送给在此端口号上监听的AJP连接器组件。

属性：

backlog：当所有可能的请求处理线程都在使用时，队列中排队的请求最大数目。默认为10，当队列已满，任何请求都将被拒绝

maxSpareThread：允许存在空闲线程的最大数目，默认值为50

maxThread：最大线程数，默认值为200

minSpareThreads：设当连接器第一次启动时创建线程的数目，确保至少有这么多的空闲线程可用，默认值为4

port：服务端套接字的TCP端口号，默认值为8089（必须）

topNoDelay：为true时，可以提高性能，默认值为true

soTimeout：超时值

例：

<!--Define an AJP1.3 Connector on port 8089-->

<Connector port=”8089” enableLookups=”false” redirectPort=”8443” protocol=”AJP/1.3” />

-->

<Connector port="8080" maxThread="50" minSpareThreads="25" maxSpareThread="75" enableLookups="false" redirectPort="8443" acceptCount="100" debug="0" connectionTimeout="20000" disableUploadTimeout="true" />

<Connection port="8009" enableLookups="false" redirectPort="8443" debug="0" protocol="AJP/1.3" />

第一个Connector元素定义了一个HTTP Connector,它通过8080端口接收HTTP请求;第二个Connector元素定义了一个JD Connector,它通过8009端口接收由其它服务器转发过来的请求.

**<Engine>元素**

每个Service元素只能有一个Engine元素.处理在同一个<Service>中所有<Connector>元素接收到的客户请求.由org.apahce.catalina.Engine接口定义.

<!--

一个“Engine”（引擎）代表处理每个请求的入口点（在Catalina内）。这个Tomcat的标准独立引擎实现分析包含在请求中的HTTP头信息，并将请求传送到适当的主机或虚拟主机上。

-->

<!-- Engine用来处理Connector收到的Http请求它将匹配请求和自己的虚拟主机，并把请求转交给对应的Host来处理默认虚拟主机是localhost-->

<Engine name="Catalina" defaultHost="localhost" debug="0">

1. className指定实现Engine接口的类,默认值为StandardEngine
2. defaultHost指定处理客户的默认主机名,在<Engine>中的<Host>子元素中必须定义这一主机
3. name定义Engine的名字

在<Engine>可以包含如下元素<Logger>, <Realm>, <Value>, <Host>

**<Host>元素**

它由Host接口定义.一个Engine元素可以包含多个<Host>元素.每个<Host>的元素定义了一个虚拟主机.它包含了一个或多个Web应用.

<!-- 定义默认的虚拟主机注意：XML模式确认将不能与Xerces 2.2同工作。-->

<!-- 虚拟主机localhost

appBase : 指 定虚拟主机的目录,可以指定绝对目录,也可以指定相对于<CATALINA\_HOME>的相对目录.如果没有此项,默认 为<CATALINA\_HOME>/webapps. 它将匹配请求和自己的Context的路径，并把请求转交给对应的Context来处

autoDeploy:如果此项设为true,表示Tomcat服务处于运行状态时,能够监测appBase下的文件,如果有新有web应用加入进来,会自运发布这个WEB应用unpackWARs:如果此项设置为true,表示把WEB应用的WAR文件先展开为开放目录结构后再运行.如果设为false将直接运行为WAR文件

alias:指定主机别名,可以指定多个别名

deployOnStartup:如果此项设为true,表示Tomcat服务器启动时会自动发布appBase目录下所有的Web应用.如果Web应用 中的server.xml没有相应的<Context>元素,将采用Tomcat默认的Context

-->

<Host name="localhost" debug="0" appBase="webapps" unpackWARs="true" autoDeploy="true">

在<Host>元素中可以包含如下子元素

<Logger>, <Realm>, <Value>, <Context>

**<Context>元素**

它由Context接口定义.是使用最频繁的元素.每个<Context元素代表了运行在虚拟主机上的单个Web应用.一个<Host>可以包含多个<Context>元素.每个web应用有唯一的一个相对应的Context代表web应用自身.servlet容器为第一个web应用创建一个

<!-- Context，对应于一个Web App

path : 该Context的路径名是""，故该Context是该Host的默认Context

docBase : 该Context的根目录是webapps/mycontext/

reloadable:如果这个属性设为true, Tomcat服务器在运行状态下会监视在WEB-INF/classes和Web-INF/lib目录CLASS文件的改运.如果监视到有class文件 被更新,服务器自重新加载Web应用

useNaming:指定是否支持JNDI,默认值为了true

cookies指定是否通过Cookies来支持Session,默认值为true

-->

<Context path="" docBase="mycontext" debug="0"/>

**Tomcat Server处理一个http请求的过程：**

假设来自客户的请求为：http://localhost:8080/wsota/wsota\_index.jsp

1) 请求被发送到本机端口8080，被在那里侦听的Coyote HTTP/1.1 Connector获得

2) Connector把该请求交给它所在的Service的Engine来处理，并等待来自Engine的回应

3) Engine获得请求localhost/wsota/wsota\_index.jsp，匹配它所拥有的所有虚拟主机Host

4) Engine匹配到名为localhost的Host（即使匹配不到也把请求交给该Host处理，因为该Host被定义为该Engine的默认主机）

5) localhost Host获得请求/wsota/wsota\_index.jsp，匹配它所拥有的所有Context

6) Host匹配到路径为/wsota的Context（如果匹配不到就把该请求交给路径名为""的Context去处理）

7) path="/wsota"的Context获得请求/wsota\_index.jsp，在它的mapping table中寻找对应的servlet

8) Context匹配到URL PATTERN为\*.jsp的servlet，对应于JspServlet类

9) 构造HttpServletRequest对象和HttpServletResponse对象，作为参数调用JspServlet的doGet或doPost方法

10)Context把执行完了之后的HttpServletResponse对象返回给Host

11)Host把HttpServletResponse对象返回给Engine

12)Engine把HttpServletResponse对象返回给Connector

13)Connector把HttpServletResponse对象返回给客户browser

# Servlet技术

## Servlet生命周期

Servlet运行在Servlet容器中，其生命周期由容器来管理。Servlet的生命周期通过javax.servlet.Servlet接口中的init()、service()和destroy()方法来表示。

Servlet的生命周期包含以下4个阶段：

* 加载和实例化

Servlet容器负责加载和实例化Servlet。当Servlet容器启动时，或者在容器检测到需要这个Servlet来响应第一个请求时，创建Servlet实例。

* 初始化

在Servlet实例化之后，容器将调用Servlet的init()方法初始化这个对象。初始化的，目的是为了让Servlet对象在处理客户端请求前完成一些初始化的工作，如建立数据库的链接，获取配置信息等。在初始化期间，Servlet实例可以使用容器为它准备的ServletConfig对象从Web应用程序的配置信息（web.xml中配置）中获取初始化的参数信息。

* 请求处理

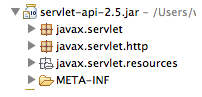
Servlet容器调用Servlet的service()方法对请求进行处理。在service()方法调用之前，init()方法必须成功执行。

* 服务终止

当容器检测到一个Servlet实例应该从服务中被移除的时候，容器就会调用实例的destroy()方法，以便让该实例可以释放它所使用的资源，保存数据到持久存储设备中。

## Servlet API

Servlet API主要由两个Java包组成：javax.servlet和javax.servlet.http



1. 在javax.servlet包中定义了Servlet接口及相关的通用接口和类；
2. 在javax.servlet.http包中主要定义了与HTTP协议相关的HttpServlet类，HttpServletRequest接口和HttpServletResponse接口；

### Servlet接口

javax.servlet.Servlet接口主要作用是提供Servlet生命周期的init()、service()和destroy()方法。这3个方法都是由Servlet容器来调用的，容器会在Servlet的生命周期的不同阶段调用对应的方法。

**public void init(ServletConfig config)**

该方法负责初始化Servlet对象，在Servlet实例化之后，Servlet容器会调用init()方法来初始化该对象，主要是为了让Servlet对象在处理客户请求前可以完成一些初始化的工作。例如：建立数据库的连接，获取配置信息等。对于每一个Servlet实例，init()方法只能被调用一次。init()方法有一个类型为ServletConfig的参数，Servlet容器通过这个参数向Servlet传递配置信息。Servlet使用ServletConfig对象从Web应用程序的配置信息中获取以名-值对形式提供的初始化参数。另外，在Servlet中，还可以通过ServletConfig对象获取描述Servlet运行环境的ServletContext对象，使用该对象，Servlet可以和它的Servlet容器进行通信。

**public void service(ServletRequest req, ServletResponse res)**

容器调用service()方法来处理客户端的请求。要注意的是，在service()方法被容器调用之前，必须确保init()方法正确完成。容器会构造一个表示客户端请求信息的请求对象（类型为ServletRequest）和一个用于对客户端进行响应的响应对象（类型为ServletResponse）作为参数传递给service()方法。在service()方法中，Servlet对象通过ServletRequest对象得到客户端的相关信息和请求信息，在对请求进行处理后，调用ServletResponse对象的方法设置响应信息。

**public void destroy()**

当容器检测到一个Servlet对象应该从服务中被移除的时候，容器会调用该对象的destroy()方法，以便让Servlet对象可以释放它所使用的资源，保存数据到持久存储设备中。在Servlet容器调用destroy()方法前，如果还有其他的线程正在service()方法中执行，容器会等待这些线程执行完毕或等待服务器设定的超时值到达。一旦Servlet对象的destroy()方法被调用，容器不会再把其他的请求发送给该对象。如果需要改Servlet再次为客户端服务，容器将会重写产生一个Servlet对象来处理客户端的请求。

在Servlet接口中还定义了两个方法：

**public ServletConfig getServletConfig()**

该方法返回容器调用init()方法时传递给Servlet对象的ServletConfig对象，ServletConfig对象包含了Servlet的初始化参数。

**public String getServletInfo()**

该方法返回一个String类型的字符串，其中包括了关于Servlet信息，例如：作者、版本和版权。

### ServletRequest接口

javax.servlet.ServletRequest接口，用于作为service()方法的其中一个参数传递给Servlet实例，来向Servlet提供客户端的请求信息。当Servlet容器接收到客户端要求访问特定Servlet的请求时，容器先解析客户端的原始请求数据，把它包装成一个ServletRequest对象。

**ServletRequest接口中的常用方法:**

**public Object getAttribute(String name)**

返回请求中所以name为名字属性的Object对象值。如果该属性不存在，这个方法将返回null。

**public Enumeration getAttributeNames()**

返回请求中所有可用的属性的名字。如果在请求中没有属性，这个方法将返回一个空的枚举集合。

**public void removeAttribute(String name)**

移除请求中名字为name的属性

**public void setAttribute(String name, Object o)**

在请求中保存名字为name的属性。如果第二个参数o为null，那么相当于调用removeAttribute(name)。

该方法用于servlet转发过程中将需要传递的值以键值对的形式保存到ServletRequest对象中。传递给其他servlet。可以可以通过getAttribute()方法获取到传递的值。

**public String getCharacterEncoding()**

返回请求正文使用的字符编码的名字。如果请求没有指定在字符编码，这个方法将返回null。

**public int getContentLength()**

以字节为单位，返回请求正文的长度。如果长度不可知，这个方法将返回-1

**public String getContentType()**

返回请求正文的MIME类型。如果类型不可知，这个方法将返回null

**public ServletInputStream getInputStream()**

返回一个输入流，使用该输入流以二进制方法读取请求正文的内容。javax.servlet.ServletInputStream是一个抽象类，继承自java.io.InputStream。

**public String getLocalAddr()**

返回接收到请求的网络接口的IP地址。

**public String getLocalName()**

返回接收到请求的IP接口的主机名。

**public int getLocalPort()**

返回接收到请求的网络接口的IP端口号。

**public String getParameter(String name)**

返回请求中name参数的值。如果name参数有多个值，那么这个方法将返回值列表中的第一个值。如果在请求中没有找到这个参数，这个方法将返回null。

该方法用于获取客户端表单中提交的参数值。获取的值类型都是String类型。

**public Enumeration getParameterNames()**

返回请求中包含的所有的参数的名字。如果请求中没有参数，这个方法将返回一个空的枚举集合。

**public String[] getParameterValues(String name)**

返回请求中name参数所有的值。如果这个参数在请求中并不存在，这个方法将返回null。

**public String getProtocol()**

返回请求使用的协议的名字和版本

**public BufferedReader getReader()**

返回BufferedReader对象，以字符数据方式读取请求正文。

**public String getRemoteAddr()**

返回发送请求的客户端或者最后一个代理服务器的IP地址

**public String getRemoteHost()**

返回发送请求的客户端或者最后一个代理服务器的完整限定名

**public int getRemotePort()**

返回发送请求的客户端或者最后一个代理服务器的IP源端口。

**public RequestDispatcher getRequestDispatcher(String path)**

返回RequestDispatcher对象，作为path所定位的资源的封装。（请求转发）request.getRequestDispatcher("/test.jsp").forward(request,response); //将客户端的请求转发给“/test.jsp”文件进行处理

**public String getServerName()**

返回请求发送到的服务器的主机名

**public int getServerPort()**

返回请求发送到的服务器的端口号

**public void setCharacterEncoding(String env)**

覆盖在请求正文中所有使用的字符编码的名字

### ServletResponse接口

javax.servlet.ServletResponse接口，用于作为service()方法的其中一个参数传递给Servlet实例，来向Servlet提供服务端响应的信息。Servlet容器会将响应信息封装成一个ServletResponse对象，在以参数的形式传递给Servlet实例。

**ServletResponse类中常用方法：**

**public void flushBuffer()**

强制把任何在缓存中的内容发送到客户端。

**public int getBufferSize()**

返回实际用于响应的缓存的大小，如果没有使用缓存，这个方法将返回0。

**public String getCharacterEncoding()**

返回在响应中发送的正文所使用的字符编码（MIME字符集）

**public String getContentType()**

返回在响应中发送的正文所使用的MIME类型。

**public ServletOutputStream getOutputStream()**

返回ServletOutputStream对象，用于在响应中写入二进制数据。javax.servlet.ServletOutputStream是一个抽象类，继承自java.io.OutputStream。

**public PrintWriter getWriter()**

返回一个PrintWriter对象，用于发送字符文本到客户端。PrintWriter对象使用getCharacterEncoding()方法返回的字符编码。如果没有指定响应的字符编码方式，默认将使用ISO-8859-1。

**public boolean isCommitted()**

返回一个布尔值，指示是否已经提交了响应。

**public void reset()**

清除在缓存中的任何数据，包括状态代码和消息报头。如果响应已经被提交，这个方法将抛出IllegalStateException异常

**public void resetBuffer()**

清除在缓存中的响应内容，保留状态代码和消息报头。如果响应已经被提交，这个方法将抛出IllegalStateException异常

**public void setBufferSize(int size)**

设置响应正文的缓存大小。Servlet容器将使用一个缓存，其大小至少是请求的尺寸大小。这个方法必须在响应正文被写入之前调用，如果内容已经被写入或者响应对象已经被提交，这个方法将抛出IllegalStateException异常

**public void setCharacterEncoding(String charset)**

设置发送到客户端的响应的字符编码，例如：UTF-8。

**public void setContentLength(int len)**

对于HTTP Servlet，在响应中，设置内容正文的长度，这个方法设置HTTP Content-Length实体报头。

**public void setContentType(String type)**

设置要发送到客户端的响应的内容类型，此时响应应该还没有提交。给出的内容类型可以包括字符编码说明，例如：text/html;charset=UTF-8。如果这个方法在getWriter()方法被调用之前调用，那么响应的字符编码将仅从给出的内容类型中设置。这个方法如果在gerWriter()方法被调用之后或者在响应被提交之后调用，将不会设置响应的字符编码。在使用HTTP协议的情况下，这个方法设置Content-Type实体报头。

### ServletConfig接口

javax.servlet.ServletConfig接口表示Servlet实例的初始化数据。当Servlet容器初始化一个Servlet对象时，会为这个Servlet对象创建一个ServletConfig对象。用于Servlet容器调用init()方法是向其Servlet实例传入初始值。该对象保存着Servlet实例的初始化数据。

该接口定义了四个方法：

**public String getInitParameter(String name)**

返回名字为name的初始化参数的值，初始化参数在web.xml配置文件中进行配置。如果参数不存在，这个方法将返回null

**public Enumeration getInitParameterNames()**

返回Servlet所有初始化参数的名字的枚举集合。如果Servlet没有初始化参数，这个方法将返回一个空的枚举集合。

**public ServletContext getServletContext()**

返回Servlet上下文对象的引用，关于ServletContext的使用。

**public String getServletName()**

返回Servlet实例的名字。这个名字是在Web应用程序的部署描述符中指定的。如果一个没有注册的Servlet实例，这个方法返回的将是Servlet的类名。

### 一个简单的Servlet

编写HelloWorldServlet类

编写一个Servlet，实际上就是编写一个实现了javax.servlet.Servlet接口的类。

// 实现Servlet接口的Servlet类

**public** **class** HelloWorldServlet **implements** Servlet {

**private** ServletConfig config = **null**; //定义一个ServletConfig类型的变量

**public** **void** destroy() {} //重写destroy方法

**public** ServletConfig getServletConfig() { //重写getServletConfig方法

**return** config;

}

**public** String getServletInfo() { //重写getServletInfo方法

**return** **null**;

}

**public** **void** init(ServletConfig config) **throws** ServletException {

**this**.config = config;

}

**public** **void** service(ServletRequest req, ServletResponse res) **throws** ServletException, IOException {

PrintWriter out = res.getWriter();

out.print("Hello World");

out.close(); //关闭输出流

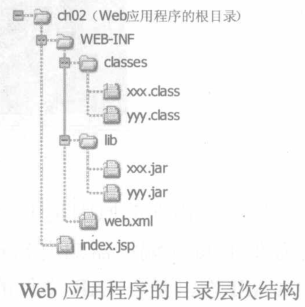
}

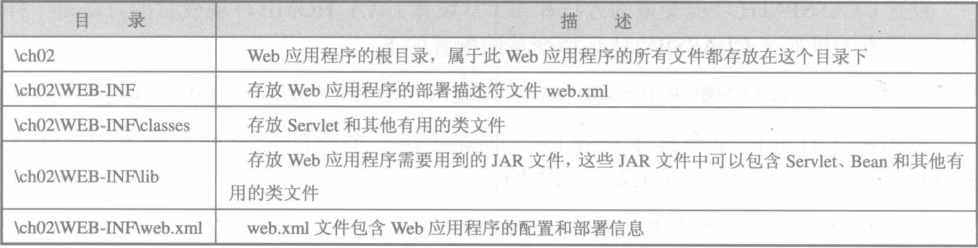
}

在Servlet中，主要的方法是service()，当客户端请求到来时，Servlet容器将调用Servlet实例的service()方法对请求进行处理。我们在service()方法中，首先通过ServletResponse类中的getWriter()方法调用得到一个PrintWriter类型的输出流对象out，然后调用out对象的println()方法向客户端发送字符串“Hello World”，最后关闭out对象。

### web应用程序目录结构

WEB-INF目录：





WEB-INF目录不属于web应用程序可以访问的上下文路径的一部分，对客户端来说，这个目录是不可见的，客户端无法直接访问该目录下数据。不过WEB-INF目录下的内容对于Servlet代码是可见的，在Servlet代码中可以通过调用ServletContext对象中的getResource()或者getResourceAsStream()方法来访问WEB-INF目录下的资源，也可以使用RequestDispatcher调用将WEB-INF目录下的内容呈现给客户端。

如果我们想在Servlet代码中访问保存在文件中的配置信息，而又不希望这些配置信息被客户端访问到，就可以把这个文件放在WEB-INF目录下。

### 不依赖协议的Servlet

#### GenericServlet类

我们通过实现Servlet接口来编写的Servlet类，这需要实现Servlet接口中定义的5个方法。为了简化Servlet的编写，在javax.servlet包中提供了一个抽象的类javax.servlet.GenericServlet。它给出了除service()方法外的其他4个方法的简单实现。GenericServlet类定义了一个通用的、不依赖于具体协议的Servlet，它实现了Servlet接口和ServletConfig接口。

public abstract class GenericServlet implements Servlet, ServletConfig, Serializable{}

如果要编写一个通用的Servlet，只需要从GenericServlet类继承，并实现其中的抽象方法service()

在GenericServlet类中，定义了两个重载的init()方法：

**public void init(ServletConfig config)**

**public void init()**

第一个init()方法是Servlet接口中init()方法的实现。在这个方法中，首先将ServletConfig对象保存在一个transient实例变量中，然后调用第二个不带参数的init()方法。

通常我们在编写继承自GenericServlet的Servlet类时，只需要重写第二个不带参数的init()方法就可以了。如果覆盖了第一init()方法，那么应该在子类的该方法中，包含一句super.init(config)代码的调用。

**在GenericServlet类中还定义了下列的方法:**

**public String getInitParameter(String name)**

返回名字为name的初始化参数的值，初始化参数在web.xml配置文件中进行配置。如果参数不存在，这个方法将返回null。

注意：这个方法只是为了方便而给出的，它实际上是通过调用ServletConfig对象的getInitParameter()方法来得到初始化参数的。

**public Enumeration getInitParameterNames()**

返回SERVLET所有初始化参数的名字的枚举集合。如果Servlet没有初始化参数，这个方法将返回一个空的枚举集合。

注意：这个方法只是为了方便而给出的，它实际上是通过调用ServletConfig对象的getInitParameterNames()方法来得到所有的初始化参数的名字

**public ServletContext getServletContext()**

返回Servlet上下文对象的引用，关于ServletContext的使用。

注意：这个方法只是为了方便而给出的，它实际上是通过调用ServletConfig对象的getServletContext()方法来得到的Servlet上下文对象的引用。

实例：

**public** **class** HelloWorldServlet **extends** GenericServlet {

**private** **static** **final** **long** ***serialVersionUID*** = -6092299089824464675L;

//实现GenericServlet抽象类中的service()方法

**public** **void** service(ServletRequest req, ServletResponse resp) **throws** ServletException, IOException {

//获取客户端提交的数据

String username = req.getParameter("username");

String pwd = req.getParameter("pwd");

//将结果返回给客户端

resp.setContentType("text/html;charset=utf-8"); //设置响应的字符编码及文档类型

PrintWriter out = resp.getWriter(); //获取输出流对象

out.println(username + ":" + pwd);

out.flush();

out.close();

}

}

### 基于HTTP协议的Servlet

#### HTTP协议

HTTP是一个属于应用层的面向对象的协议，由于其简捷、快速的方式，适用于分布式超媒体信息系统。它于1990年提出，经过几年的使用与发展，得到不断地完善和扩展。目前在WWW中使用的是HTTP/1.0的第六版，HTTP/1.1的规范化工作正在进行之中，而且HTTP-NG(Next Generation of HTTP)的建议已经提出。

HTTP协议的主要特点可概括如下：

1. 支持客户/服务器模式。
2. 简单快速：客户向服务器请求服务时，只需传送请求方法和路径。请求方法常用的有GET、HEAD、POST。每种方法规定了客户与服务器联系的类型不同。由于HTTP协议简单，使得HTTP服务器的程序规模小，因而通信速度很快。
3. 灵活：HTTP允许传输任意类型的数据对象。正在传输的类型由Content-Type加以标记。
4. 无连接：无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间。
5. 无状态：HTTP协议是无状态协议。无状态是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就较快。

##### HTTP协议详解之URL篇

http（超文本传输协议）是一个基于请求与响应模式的、无状态的、应用层的协议，常基于TCP的连接方式，HTTP1.1版本中给出一种持续连接的机制，绝大多数的Web开发，都是构建在HTTP协议之上的Web应用。

HTTP URL (URL是一种特殊类型的URI，包含了用于查找某个资源的足够的信息)的格式如下：

http://host[":"port][abs\_path]

http表示要通过HTTP协议来定位网络资源；host表示合法的Internet主机域名或者IP地址；port指定一个端口号，为空则使用缺省端口80；abs\_path指定请求资源的URI；如果URL中没有给出abs\_path，那么当它作为请求URI时，必须以“/”的形式给出，通常这个工作浏览器自动帮我们完成。

##### HTTP协议详解之请求篇

http请求由三部分组成，分别是：请求行、消息报头、请求正文

1. 请求行以一个方法符号开头，以空格分开，后面跟着请求的URI和协议的版本，格式如下：Method Request-URI HTTP-Version CRLF

其中 Method表示请求方法；Request-URI是一个统一资源标识符；HTTP-Version表示请求的HTTP协议版本；CRLF表示回车和换行（除了作为结尾的CRLF外，不允许出现单独的CR或LF字符）。

请求方法（所有方法全为大写）有多种，各个方法的解释如下：

GET：请求获取Request-URI所标识的资源

POST：在Request-URI所标识的资源后附加新的数据

HEAD：请求获取由Request-URI所标识的资源的响应消息报头

PUT：请求服务器存储一个资源，并用Request-URI作为其标识

DELETE：请求服务器删除Request-URI所标识的资源

TRACE：请求服务器回送收到的请求信息，主要用于测试或诊断

CONNECT：保留将来使用

OPTIONS：请求查询服务器的性能，或者查询与资源相关的选项和需求

应用举例：

GET方法：在浏览器的地址栏中输入网址的方式访问网页时，浏览器采用GET方法向服务器获取资源，

eg:GET /form.html HTTP/1.1 (CRLF)

POST方法要求被请求服务器接受附在请求后面的数据，常用于提交表单。

eg：POST /reg.jsp HTTP/ (CRLF)

Accept:image/gif,image/x-xbit,... (CRLF)

...

HOST:www.guet.edu.cn (CRLF)

Content-Length:22 (CRLF)

Connection:Keep-Alive (CRLF)

Cache-Control:no-cache (CRLF)

(CRLF) //该CRLF表示消息报头已经结束，在此之前为消息报头

user=jeffrey&pwd=1234 //此行以下为提交的数据

HEAD方法与GET方法几乎是一样的，对于HEAD请求的回应部分来说，它的HTTP头部中包含的信息与通过GET请求所得到的信息是相同的。利用这个方法，不必传输整个资源内容，就可以得到Request-URI所标识的资源的信息。该方法常用于测试超链接的有效性，是否可以访问，以及最近是否更新。

1. 请求报头后述
2. 请求正文(略)

##### HTTP协议详解之响应篇

在接收和解释请求消息后，服务器返回一个HTTP响应消息。

HTTP响应也是由三个部分组成，分别是：状态行、消息报头、响应正文

1. 状态行格式如下：

HTTP-Version Status-Code Reason-Phrase CRLF

其中，HTTP-Version表示服务器HTTP协议的版本；Status-Code表示服务器发回的响应状态代码；Reason-Phrase表示状态代码的文本描述。

状态代码有三位数字组成，第一个数字定义了响应的类别，且有五种可能取值：

1xx：指示信息--表示请求已接收，继续处理

2xx：成功--表示请求已被成功接收、理解、接受

3xx：重定向--要完成请求必须进行更进一步的操作

4xx：客户端错误--请求有语法错误或请求无法实现

5xx：服务器端错误--服务器未能实现合法的请求

常见状态代码、状态描述、说明：

200 OK //客户端请求成功

400 Bad Request //客户端请求有语法错误，不能被服务器所理解

401 Unauthorized //请求未经授权，这个状态代码必须和WWW-Authenticate报头域一起使用

403 Forbidden //服务器收到请求，但是拒绝提供服务

404 Not Found //请求资源不存在，eg：输入了错误的URL

500 Internal Server Error //服务器发生不可预期的错误

503 Server Unavailable //服务器当前不能处理客户端的请求，一段时间后可能恢复正常

eg：HTTP/1.1 200 OK （CRLF）

1. 响应报头后述
2. 响应正文就是服务器返回的资源的内容

##### HTTP协议详解之消息报头篇

HTTP消息由客户端到服务器的请求和服务器到客户端的响应组成。请求消息和响应消息都是由开始行（对于请求消息，开始行就是请求行，对于响应消息，开始行就是状态行），消息报头（可选），空行（只有CRLF的行），消息正文（可选）组成。

HTTP消息报头包括普通报头、请求报头、响应报头、实体报头。

每一个报头域都是由名字+“：”+空格+值 组成，消息报头域的名字是大小写无关的。

1. 普通报头

在普通报头中，有少数报头域用于所有的请求和响应消息，但并不用于被传输的实体，只用于传输的消息。

eg：

Cache-Control 用于指定缓存指令，缓存指令是单向的（响应中出现的缓存指令在请求中未必会出现），且是独立的（一个消息的缓存指令不会影响另一个消息处理的缓存机制），HTTP1.0使用的类似的报头域为Pragma。

请求时的缓存指令包括：no-cache（用于指示请求或响应消息不能缓存）、no-store、max-age、max-stale、min-fresh、only-if-cached;

响应时的缓存指令包括：public、private、no-cache、no-store、no-transform、must-revalidate、proxy-revalidate、max-age、s-maxage.

eg：

为了指示IE浏览器（客户端）不要缓存页面，服务器端的JSP程序可以编写如下：response.sehHeader("Cache-Control","no-cache");

//response.setHeader("Pragma","no-cache");作用相当于上述代码，通常两者//合用

这句代码将在发送的响应消息中设置普通报头域：Cache-Control:no-cache

Date普通报头域表示消息产生的日期和时间

Connection普通报头域允许发送指定连接的选项。例如指定连接是连续，或者指定“close”选项，通知服务器，在响应完成后，关闭连接

1. 请求报头

请求报头允许客户端向服务器端传递请求的附加信息以及客户端自身的信息。

常用的请求报头

Accept：请求报头域用于指定客户端接受哪些类型的信息。eg：Accept：image/gif，表明客户端希望接受GIF图象格式的资源；Accept：text/html，表明客户端希望接受html文本。

Accept-Charset：请求报头域用于指定客户端接受的字符集。eg：Accept-Charset:iso-8859-1,gb2312.如果在请求消息中没有设置这个域，缺省是任何字符集都可以接受。

Accept-Encoding：请求报头域类似于Accept，但是它是用于指定可接受的内容编码。eg：Accept-Encoding:gzip.deflate.如果请求消息中没有设置这个域服务器假定客户端对各种内容编码都可以接受。

Accept-Languag：请求报头域类似于Accept，但是它是用于指定一种自然语言。eg：Accept-Language:zh-cn.如果请求消息中没有设置这个报头域，服务器假定客户端对各种语言都可以接受。

Authorization：请求报头域主要用于证明客户端有权查看某个资源。当浏览器访问一个页面时，如果收到服务器的响应代码为401（未授权），可以发送一个包含Authorization请求报头域的请求，要求服务器对其进行验证。

Host：请求报头域主要用于指定被请求资源的Internet主机和端口号，它通常从HTTP URL中提取出来的，

eg：

我们在浏览器中输入：http://www.guet.edu.cn/index.html

浏览器发送的请求消息中，就会包含Host请求报头域，如下：

Host：www.guet.edu.cn

此处使用缺省端口号80，若指定了端口号，则变成：Host：www.guet.edu.cn:指定端口号

User-Agent

我们上网登陆论坛的时候，往往会看到一些欢迎信息，其中列出了你的操作系统的名称和版本，你所使用的浏览器的名称和版本，这往往让很多人感到很神奇，实际上，服务器应用程序就是从User-Agent这个请求报头域中获取到这些信息。User-Agent请求报头域允许客户端将它的操作系统、浏览器和其它属性告诉服务器。不过，这个报头域不是必需的，如果我们自己编写一个浏览器，不使用User-Agent请求报头域，那么服务器端就无法得知我们的信息了。

请求报头举例：

GET /form.html HTTP/1.1 (CRLF)

Accept:image/gif,image/x-xbitmap,image/jpeg,application/x-shockwave-flash,application/vnd.ms-excel,application/vnd.ms-powerpoint,application/msword,\*/\* (CRLF)

Accept-Language:zh-cn (CRLF)

Accept-Encoding:gzip,deflate (CRLF)

If-Modified-Since:Wed,05 Jan 2007 11:21:25 GMT (CRLF)

If-None-Match:W/"80b1a4c018f3c41:8317" (CRLF)

User-Agent:Mozilla/4.0(compatible;MSIE6.0;Windows NT 5.0) (CRLF)

Host:www.guet.edu.cn (CRLF)

Connection:Keep-Alive (CRLF)

(CRLF)

1. 响应报头

响应报头允许服务器传递不能放在状态行中的附加响应信息，以及关于服务器的信息和对Request-URI所标识的资源进行下一步访问的信息。

常用的响应报头：

Location：响应报头域用于重定向接受者到一个新的位置。Location响应报头域常用在更换域名的时候。

Server：响应报头域包含了服务器用来处理请求的软件信息。与User-Agent请求报头域是相对应的。下面是

Server响应报头域的一个例子：

Server：Apache-Coyote/1.1

WWW-Authenticate：响应报头域必须被包含在401（未授权的）响应消息中，客户端收到401响应消息时候，并发送Authorization报头域请求服务器对其进行验证时，服务端响应报头就包含该报头域。

eg：

WWW-Authenticate:Basic realm="Basic Auth Test!" //可以看出服务器对请求资源采用的是基本验证机制。

1. 实体报头

请求和响应消息都可以传送一个实体。一个实体由实体报头域和实体正文组成，但并不是说实体报头域和实体正文要在一起发送，可以只发送实体报头域。实体报头定义了关于实体正文（eg：有无实体正文）和请求所标识的资源的元信息。

常用的实体报头：

Content-Encoding：实体报头域被用作媒体类型的修饰符，它的值指示了已经被应用到实体正文的附加内容的编码，因而要获得Content-Type报头域中所引用的媒体类型，必须采用相应的解码机制。Content-Encoding这样用于记录文档的压缩方法，

eg：Content-Encoding：gzip

Content-Language：实体报头域描述了资源所用的自然语言。没有设置该域则认为实体内容将提供给所有的语言阅读

者。

eg：Content-Language:da

Content-Length：实体报头域用于指明实体正文的长度，以字节方式存储的十进制数字来表示。

Content-Type：实体报头域用语指明发送给接收者的实体正文的媒体类型。

eg：

Content-Type:text/html;charset=ISO-8859-1

Content-Type:text/html;charset=GB2312

Last-Modified：实体报头域用于指示资源的最后修改日期和时间。

Expires：实体报头域给出响应过期的日期和时间。为了让代理服务器或浏览器在一段时间以后更新缓存中(再次访问曾访问过的页面时，直接从缓存中加载，缩短响应时间和降低服务器负载)的页面，我们可以使用Expires实体报头域指定页面过期的时间。eg：Expires：Thu，15 Sep 2006 16:23:12 GMT

HTTP1.1的客户端和缓存必须将其他非法的日期格式（包括0）看作已经过期。eg：为了让浏览器不要缓存页面，我们也可以利用Expires实体报头域，设置为0，jsp中程序如下：response.setDateHeader("Expires","0");

#### HttpServlet类

在绝大多数的网络应用中，都是客户端（浏览器）通过HTTP协议去访问服务器端的资源，而我们所编写的Servlet也主要是应用于HTTP协议的请求和响应。为了快速开发应用于HTTP协议的Servlet类，Sun公司在java.servlet.http包中给我们提供了一个抽象的javax.servlet.http.HttpServlet类，它继承自GenericServlet抽象类，用于创建适合Web站点的HTTP Servlet。

public abstract class HttpServlet extends GenericServlet implements Serializable

在HttpServlet类中提供了两个重载的service()方法：

**public void service(ServletRequest req,ServletResponse res)**

**protected void service(HttpServletRequest req,HttpServletResponse resp)**

第一个service()方法是GenericServlet类中service()方法的实现。在这个方法中，首先将req和res对象转换为HttpServletRequest（继承自ServletRequest接口）和HttpServletResponse（继承自ServletResponse接口）类型，然后调用第二个service方法，对客户请求进行处理。

针对HTTP1.1中定义的7种请求方法GET、POST、HEAD、PUT、DELETE、TRACE、OPTIONS，HttpServlet分别提供了7种处理方法：

**public void doGet(HttpServletRequest req,HttpServletResponse resp)**

**public void doPost(HttpServletRequest req,HttpServletResponse resp)**

**public void doHead(HttpServletRequest req,HttpServletResponse resp)**

**public void doPut(HttpServletRequest req,HttpServletResponse resp)**

**public void doDelete(HttpServletRequest req,HttpServletResponse resp)**

**public void doTrace(HttpServletRequest req,HttpServletResponse resp)**

**public void doOptions(HttpServletRequest req,HttpServletResponse resp)**

这7个方法的参数类型及异常抛出类型与HttpServlet类中的第二个重载的service()方法是一致的。容器接收到一个针对HttpServlet对象的请求时，调用该对象中的方法顺序如下：

1. 调用公共的（public）service()方法。
2. 在公共的service()方法中，首先将参数类转换为HttpServletRequest和HttpServletResponse，然后调用保护的（protected）service()方法，将转换后的HttpServletRequest对象和HttpServletResponse对象作为参数传递进去。
3. 在保护的service()方法中，首先调用HttpServletRequest对象的getMethod()方法，获取HTTP请求方法的名字，然后根据请求方法的类型，调用相应的doXXX()方法。因此，我们在编写HttpServlet的派生类时，通常不需要去覆盖service()方法，而只需重写相应的doXXX()方法。HttpServlet类对TRACE和OPTIONS方法做了适当的实现，因此我们不需要去覆盖doTrace()和doOptions()方法。

#### HttpServletRequest接口

javax.servlet.http.HttpServletRequest接口继承自javax.servlet.ServletRequest。该接口定义了很多方法用于获取客户端提交的数据。在客户端提交数据到Web服务器的时候，Web服务器会将客户端提交的数据封装成HttpServletRequest对象作为参数提交给Servlet的service()方法中。

**HttpServletRequest接口中新增的常用方法：**

**public String getContextPath()**

返回请求URI中表示请求上下文的部分，上下文路径是请求URI的开始部分。上下文路径总是一斜杠（/）开头，但结束没有斜杠（/）。在默认（根）上下文中，这个方法返回空字符串””。例如：请求URI为“/samlie/test”，调用该方法返回路径“/sample”

**public Cookie[] getCookies()**

返回客户端在此次请求中发送的所有Cookie对象。

**public String getHeader(String name)**

返回名字为name的请求报头的值。如果请求中没有包含指定名字的报头，这个方法返回null。

**public Enumeration getHeaderNames()**

返回此次请求中包含的所有报头名字的枚举集合

**public Enumeration getHeaders(String name)**

返回名字为name的请求报头所有的值的枚举集合

**public String getMethod()**

返回此次请求中所有的HTTP方法的名字，例如：GET、POST、PUT

**public String getPathInfo()**

返回与客户端发送的请求URL相联系的额外的路径信息。额外的路径信息是跟在Servlet的路径之后、查询字符串之前的路径，并以斜杠（/）字符开始。例如：假定在web.xml文件中MyServlet类映射的URL是：/myservlet/\*，用户请求的URL是：http://localhost:8080/ch02/myservlet/test，当我们在HttpServletRequest对象上调用getPathInfo()时，该方法将返回/test。如果没有额外的路径信息，则返回null。

**public String getPathTranslated()**

将额外的路径信息转换为真实的路径。例如：D:\...\。

**public String getQueryString()**

返回请求URL中在路径后的查询字符串。如果在URL中没有查询字符串，该方法返回null。

例如：http://localhost:8080/ch02/logon.jsp?action=logon，方法返回action=logon

**public String getRequestURI()**

返回请求URL中从主机名到查询字符串之间的部分

**public StringBuffer getRequestURL()**

重新构造客户端用于发起请求的URL。返回的URL包括了协议、服务器的名字、端口号和服务器的路径，但是不包括查询字符串参数。

**public String getServletPath()**

返回请求URI中调用Servlet的部分。这部分的路径以斜杠（/）开始，包括了Servlet的名字或者路径，但是不包括额外的路径信息和查询字符串。

**public HttpSession getSession()**

返回和此次请求相关联的Session，如果没有给客户端分配Session，而create参数为true，则创建一个新的Session。如果参数为false，而此次请求没有一个有效的HttpSession，则返回nill

#### HttpServletResponse接口

HttpServletResponse接口继承自javax.servlet.ServletResponse。当Web服务器对客户端的请求作出相应的时候，Web服务会将服务端的响应数据封装进ServletResponse对象中作为参数提交给service()方法发送给客户端。

**HttpServletResponse接口中新增的常用方法：**

**public void addCookie(Cookie cookie)**

增加一个Cookie到响应中。这个方法可以被多次调用，用于设置多个Cookie

**public void addHeader(String name, String value)**

用给出的name和value，增加一个响应报头到响应中。

**public boolean containsHeader(String name)**

判断以name为名字的响应报头是否已经设置

**public String encodeRedirectURL(String url)**

使用Session ID对用于重定向的url进行编码，以便用于sendRedirect()方法中。如果该url不需要编码，则返回未改变得url

**public String encodeURL(String url)**

使用Session ID对指定的url进行编码。如果该url不需要编码，则返回为改变的url

**public void sendError(int sc)**

使用参数sc表示的状态代码发送一个错误响应到客户端，同时清除缓存。如果响应已经被提交，这个方法将抛出异常。

**public void sendError(int sc, String msg)**

使用指定的状态代码发送一个错误响应到客户端。服务器默认会创建一个包含了指定消息的服务器端错误页面作为响应，设置内容类型为“text/html”。如果Web应用程序已经声明了对应于指定状态代码的错误页面，则服务器会将这个页面发送给客户端，而不理会参数msg指定的错误消息。如果响应已经被提交，这个方法将抛出异常。

**public void sendRedirect(String location)**

发送一个临时的重定向响应到客户端，让客户端访问新的URL。如果指定的位置是相对URL，Servlet容器在发送响应到客户端之前，必须将相对URL转换为绝对URL。如果响应已经被提交，则抛出异常。

**public void setHeader(String name, String value)**

用给出的name和value，设置一个响应报头。如果这个报头已经被设置，新的值将覆盖先前的值。

**public void setStatus(int sc)**

为响应设置状态代码。

## 实例

实例一

编写一个带有表单的HTML页面，表单中提供了一个文本输入控件，让用户输入他的姓名，提交给Servlet进行处理。在Servlet中，我们从提交的信息中去除用户姓名，然后加上欢迎信息输出到客户端。欢迎信息作为Servlet的初始化参数在web.xml文件中配置

1. 编写welcome.html

<!DOCTYPE html>

<html>

<head>

<meta charset=*"UTF-8"*>

<title>欢迎您title>

</head>

<body>

<form action=*"welcome"* method=*"get"*>

请输入用户名：<input type=*"text"* name=*"username"* /><br />

<input type=*"submit"* value=*"提交"* />

</form>

</body>

</html>

1. 编写WelcomeServlet类

为了简化Servlet的编写，Sercletc API给我们提供了支持HTTP协议的javax.servlet.http.HttpServlet类，我们只需要从HttpServlet类派生一个子类，在子类中完成相应的功能就可以了

编写WelcomeServlet类，从HttpServlet类继承，重写doGet()方法。

**public** **class** FirstServlet **extends** HttpServlet {

**private** **static** **final** **long** ***serialVersionUID*** = 5608718856280140475L;

@Override

**protected** **void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {

**this**.doPost(req, resp);

}

@Override

**protected** **void** doPost(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {

req.setCharacterEncoding("utf-8"); //设置客户端提交的字符编码

String username = req.getParameter("username"); //获取客户端提交的数据

resp.setCharacterEncoding("utf-8"); //设置提交时的自编编码

resp.setContentType("text/html; charset=utf-8"); //设置输出的文件类型及字符编码

PrintWriter out = resp.getWriter(); //返回一个输出流

out.print("<html>");

out.print("<head><title>欢迎页面</title></head>");

out.print("<body>");

out.print("<h1>欢迎光临" + username + "</h1>");

out.print("</body>");

out.print("</html>");

out.flush();

out.close();

}

}

1. 编译WelcomeServlet.java
2. 部署WelcomeServlet类

第一种方式：使用注解的方式配置Servlet的映射。

@WebServlet("/welcome") //只能写在类的上面

**public** **class** FirstServlet **extends** HttpServlet {

**private** **static** **final** **long** ***serialVersionUID*** = 5608718856280140475L;

@Override

**protected** **void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {

**this**.doPost(req, resp);

}

@Override

**protected** **void** doPost(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {

//………………….

}

}

第二种方式：打开项目中WEB-INF目录下的web.xml文件，添加对本例中Servlet的配置

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns=*"http://xmlns.jcp.org/xml/ns/javaee"*

xsi:schemaLocation=*"http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_3\_1.xsd"*

id=*"WebApp\_ID"* version=*"3.1"*>

<!-- 定义的项目名称 -->

<display-name>javaee</display-name>

<!-- 定义的首页 -->

<welcome-file-list>

<welcome-file>index.html</welcome-file>

</welcome-file-list>

<!-- 定义的一个servlet -->

<servlet>

<!-- 自定义的servlet名 -->

<servlet-name>first</servlet-name>

<!-- 类的完整名称：包.类名 -->

<servlet-class>com.uc.javaweb.FirstServlet</servlet-class>

</servlet>

<servlet-mapping>

<!-- 必须与<servlet><servlet-name>子标签中定义的名字一样 -->

<servlet-name>first</servlet-name>

<!-- 自定义的映射路径，在名称的前面必须有一个"/"。当页面提交数据后Web容器会将页面中定义的提交文件路径和名称与此处的自定义路径名称作匹配当匹配成功后，则会根据定义的servlet名去找对应的完整类名，然后来处理提交的业务数据。

-->

<url-pattern>/welcome</url-pattern>

</servlet-mapping>

<!-- 定义多个servlet -->

..........

</web-app>

我们在<servlet>元素内部使用<init-param>子元素来为FirstServlet配置的初始化参数

<!-- 定义初始化值 -->

<init-param>

<param-name>key</param-name>

<param-value>value</param-value>

</init-param>

在Servlet中获取初始化的值：

String key = **this**.getInitParameter("key");

1. 访问WelcomeServlet

启动Tomcat服务器，打开浏览器，在地址栏中输入：http://localhost:8080/ch02/welcome.html

## 会话跟踪

Web应用是通过HTTP协议来进行通信的，而HTTP协议本身是基于请求/响应模式的，是无状态的协议。也就是说当客户端请求到来的时候，服务端做出相应后，服务端与客户端之间的连接就会关闭。这样下次提交请求的时候，服务端就不能区分这次提交的请求和之前提交的请求有无关联。

例如，我们要通过网上银行给朋友汇一笔钱，输入用户名和密码，服务器验证通过后，发回操作页面，然后连接关闭。在我们进行账户资金划账的时候，连接重新建立，但是服务器如何保证正在转账的账户就是我的账户呢？因为，此时可能还有很多人在进行转账的操作。

用户的请求发生在多个请求和响应之中，作为web服务器来说，必须能够采用一种机制来唯一地标识一个用户，同时记录该用户的状态，这是一个web应用程序典型的需求。

为了实现上述请求，需要以下两种机制：

1. 会话：服务器应当能都标识出来自单个客户的一系列请求，并把这些请求组成一个单独的工作“会话”。通过把特定请求与一个特定的工作会话相联系，在线银行应用程序就能都把一个用户与另一个用户区分开。
2. 状态：服务器应当能够记住前面请求的信息，以及对前一请求做出的处理信息。也就是说，服务器应当给每个会话联系状态。

Java Servlet API使用Session来跟踪会话和管理会话内的状态。利用Session，服务器可以把一个客户的所有请求联系在一起，并记住客户的操作状态。当客户第一次连接到服务器的时候，服务器为其建立一个Session，并分配给客户一个唯一的标识（Session ID），以后客户每次提交请求，都要将标识一起提交。服务器根据标识找出特定的Session，用这个Session记录客户的状态。

会话指的是客户端与服务端建立的一种连接机制，直到与服务器之间断开连接。在于Web中则是通过打开浏览器开始访问服务器资源，直到关闭浏览器不再访问服务器资源。这一个整个过程被称为一次会话。

### 会话跟踪的技术

常用的会话跟踪的方式：

1. Cookies
2. URL重写
3. 表单隐藏域
4. Session

#### Cookies

Cookies是最常用的跟踪用户会话的方式。Cookies是一种由服务器发送给客户的片段信息，存储在客户端浏览器的内存中或硬盘上，在客户随后对该服务器的请求中发回它。

Cookie有三个规范，原始的Netscape规范（版本0），RFC2109（HTTP状态管理机制，版本1）和RFC2965（HTTP状态管理机制，版本1，是对RFC2109的替代）。在Servlet2.4中的Cookie支持Netscape规范和RFC2109。为了确保最好的互操作性，默认情况下，Java Servlet API采用版本0（依照Netscape规范）创建Cookie。

Cookies以键-值对的方式记录会话跟踪的内容，服务器利用响应报头Set-Cookie来发送 Cookie信息

由于Cookie对用户是透明的，加上Cookie持久性高，可以长时间追踪用户（可以保存在用户机器的硬盘上），了解用户上网的习惯，这就造成了一些隐私权和安全性方面的问题，于是有些用户在使用浏览器时，会选择禁用Cookie，这样web服务器就无法利用Cookie来跟踪用户的会话了，所以也就有了第三种方法URL重写机制。

#### URL重写

当客户端禁用Cookie的时候，可以使用URL重写的机制来跟踪用户的会话。URL重写就是在URL中嵌入标识客户的Session ID，Servlet容器解释URL，取出Session ID，根据Session ID将请求与特定的Session关联。

Session ID被编码为URL字符串中的路径参数，在Servlet规范中，这个参数的名字必须是jsessionid，下面是一个包含了编码后的路径信息的URL的例子：

http://www.sumxin.org/bbs/index.jsp;jsessionid=1234?name=zhangsan&age=18

服务器将Session ID作为URL的一部分发送给客户端，客户端的请求URL中再传回来，这样，web服务器就可以跟踪用户的会话了。

要跟踪客户端的会话，就需要将所有发往客户端的URL进行编码，这可以通过调用HttpServletResponse接口中的encodeURL()方法和encodeRedirectURL()方法来实现，其中encodeRedirectURL()方法主要是在调用sendRedirect()方法之前使用

#### 表单隐藏域

该方法是在<input>中添加一个隐藏的字段

<input type=*"hidden"* name=*"sessionid"* value=*"12345"* />

那么在提交表单时，会把这个隐藏的名为sessionid的值为12345的域提交在GET或POST数据中，服务器收到我的请求后，会给一个客户端一个cookie中有

Cookie:JSESSIONID=E6047C8DB41EEA107256ECB443640C49

每次向服务器请求时，就带着我的JSESSIONID，服务器会识别我的JSESSIONID给我发送我需要的请求。

#### Session

Session是由Servlet容器产生的，是一种会话跟踪技术，能能解决用户在登录或者购物时多页面请求或存储相关用户信息。

比如这么个场景：你在打开淘宝，然后看到个想买的商品后，你点击立即购买，但是这时候会跳到登录那个新的界面，然后你登录之后，什么事都不用做，再次回到那个商品购买界面刷新下，你就发现这时候你再买这个商品就可以了。这就是session的作用。

多说一句：上面说到的cookie，其中就包含有session的ID，session是借助cookie来实现会话跟踪的，所以，如果cookie被删或者禁用的话，session这个会话跟踪技术也就无法使用了。

### Java Servlet API的会话跟踪

在Java Servlet API中，javax.servlet.http.HttpSession接口封装了Session的概念，Servlet容器提供了这个接口的实现。当一次会话开始的时候，Servlet容器就创建一个HttpSession对象，在HttpSession对象中存放客户的状态信息，Servlet容器为HttpSession对象分配一个唯一的Session ID，将其作为Cookie（或者作为URL的一部分，利用URL重写机制）发送给浏览器，浏览器在内存中保存这个Cookie。当客户再次发送HTTP请求时，浏览器将Cookie随请求一起发送，Servlet容器从请求对象中读取Session ID，然后根据Session ID找到对应的HttpSession对象，从而得到客户的状态信息。

一次会话Servlet容器就会创建一个唯一的Session，当关闭浏览器后重写访问服务器资源，Servlet容器又会生成一个新的Session。如果同时又多个访问服务器，则Servlet容器会给每个访问的会话生成多个唯一的Session。在Java Servlet API中，每个Session的作用域只在一次会话中有效。

#### HttpSession接口

HttpSession接口提供了下列方法：

**public Object getAttribute(String name)**

该方法用于获取保存在Session域中给定属性名（name）对应的属性值。

**public Enumeration getAttributeNames()**

该方法用于获取保存在Session域中的所有属性的名称，返回一个Enumeration对象。

**public void removeAttribute(String name)**

该方法通过给定的name参数来移除Session域中对应的属性值。

**public void setAttribute(String name, Object value)**

该方法用于给Session域中设置属性名和对应的值。

上面四个方法用于在HttpSession对象中读取、移除和设置属性，利用这些方法，可以在Session中维护客户的状态信息。

**public long getCreationTime()**

返回Session创建的时间，这个时间是从1970年1月1日00:00:00GMT以来的毫秒数

**public String getId()**

返回一个字符串，其中包含了分配给Session的唯一标识符。这个标识符是由Servlet容器分配的，与具体的实现有关。

**public long getLastAccessedTime()**

返回客户端最后一次发送给Session相关的请求的时间，这个时间是从1970年1月1日00:00:00GMT以来的毫秒数。这个方法可以用来确定客户端在两次请求之间的非活动时间。

**public int getMaxInactiveInterval()**

返回以毫秒为单位的最大时间间隔。这个时间值是Servlet容器在客户端的请求之间保持Session打开的最长时间间隔，超过这个时间间隔，Servlet容器将使Session失效。

**public void setMaxInactiveInterval(int interval)**

这个方法用于设置在Session失效之前，客户端的两个连接请求之间的最长时间间隔。如果设置一个负值，表示Session永远不会失效。web应用程序可以使用这个方法来设置Session的超时时间间隔，时间单位是秒。

**public ServletContext getServletContext()**

返回Session所属的ServletContext对象

**public void invalidate()**

这方法用于使会话失效。例如，用户在网上书店购买完图书后，可以选择退出系统，服务器端的web应用程序可以调用这个方法使Session失效，从而让用户不再与这个Session关联。

**public boolean isNew()**

如果客户端还不知道这个Session或者客户端没有选择加入Session，那么这个方法返回true。例如，服务器使用基于Cookie的Session，而客户端禁用了Cookie，那么对每一个请求，Session都是新的。

要得到一个Session对象，可以调用HttpServletRequest接口的getSession()方法

#### Session的生命周期

我们在进行Session编程的时候，在使用URL重写机制的时候要注意，为了保证会话跟踪的正确性，所有的链接和重定向语句中的URL都需要调用encodeURL()或encodeRedirectURL()方法进行编码。另外由于附加在URL中的Session ID是动态产生的，对每一个用户都是不同的，所以对于静态页面的相互跳转，URL重写机制就无能为力了，当然，我们也可以通过将静态页面转换为动态页面来解决这个问题。

在开发web应用程序的时候，如何去判断客户端是否禁用了Cookie，从而决定是否采用URL重写的机制去跟踪用户的会话。实际上，客户端是否禁用了Cookie，不需要我们去判断，Servlet容器会帮我们做这件事情。我们在开发web应用程序的时候，只需要对所有的链接和重定向语句中的URL都调用encodeURL()和encodeRedirectURL()方法进行编码就可以了。这两个方法在执行时，首先判断当前的Servlet事都已经执行了HttpSession对象的invalidate()方法，如果已经执行了，直接返回参数URL。接下来，判断客户端是否禁用了Cookie，如果没有禁用，则直接返回参数URL，如果禁用了Cookie，则在参数URL中附加Session ID，返回编码后的URL。

了解了encodeURL()或encodeRedirectURL()方法的工作原理，就可以结合基于Cookie和URL重写的机制来跟踪用户会话。如果一个web应用程序的功能实现以来于用户会话的跟踪，那么你可以将所有的页面实现为动态的，并在代码中使用URL重写机制。在运行时，Servlet容器会自动根据客户端的情况来选择会话跟踪的机制。

#### Cookie的应用

##### Cookie类

在Java Servlet API中给我们提供了javax.servlet.http.Cookie类，用于创建Cookie。这个类中主要有下列方法：

**public Cookie(String name, String value)**

用指定的名字和值构造一个Cookie

**public String getComment()**

**public void setComment(String purpose)**

上面两个方法用于获取和设置Set-Cookie响应报头的Comment属性

**public String getDomain()**

**public void setDomain(String pattern)**

上面两个方法用于获取和设置Set-Cookie响应报头的Domain属性

**public int getMaxAge()**

**public void setMaxAre(int expiry)**

上面两个方法用于获取和设置Set-Cookie响应报头的Max-Age属性

**public String getName()**

**public String getValue()**

上面两个方法用于获取Cookie的名字和值

**public void setValue(String newValue)**

在Cookie创建之后，为其设置新的值

**public String getPath()**

**public void setPath(String uri)**

上面两个方法用于获取和设置Set-Cookie响应报头的Path属性

**public boolean getSecure()**

如果浏览器在一个安全的协议上发送Cookie，这方法返回true

**public void setSecure(boolean flag)**

指示浏览器是否应该仅仅使用安全的协议来发送Cookie。例如HTTPS或者SSL。

**public int getVersion()**

**public void setVersion(int v)**

上面两个方法用于获取和设置Set-Cookie响应报头的Version属性，0表示Cookie遵从原始的Netscape规范，1表示Cookie遵从RFC2019规范。

如果要发送Cookie，可以调用HttpServletResponse接口的addCookie()方法。如果要从请求中取得Cookie，可以调用HttpServletRequest接口的getCookies()方法。

##### Session和Cookie

Session是一种服务器端技术，Session对象在服务器端创建，通常采用散列表来存储信息，例如：Tomcat的Session实现采用HashMap对象来存储属性名和属性值。

Cookie是有Netscape公司发明的、用于跟踪用户会话的一种方式。Cookie是由服务器发送给客户的片段信息，存储在客户端浏览器的内存中或硬盘中，在客户随后对该服务器的请求中发回它。

Session与Cookie的最大区别是，Session在服务器端保存信息，Cookie在客户端保存信息。为了跟踪用户的会话，服务器端在创建Session后，需要将Session ID交给客户端，在客户端下次请求时，将这个ID随请求一起发送回来。可以采用Cookie或URL重写的方式，将Session ID发送给客户端。

Session产生的会话Cookie将Session ID以Cookie的形式存放在浏览器内存中，当浏览器关闭后，存于浏览器内存中的Cookie消失，此次会话结束。打开新的浏览器会产生一个新的Session ID。所以不同进程的浏览器无法共享Session ID（同一进程的不同浏览器窗口可以共享Session ID）。

使用Cookie类生成的Cookie将Session ID存储在硬盘中，在Cookie失效之前，多个浏览器进程都可以共享Cookie中的信息（Session ID）。

#### HttpSessionBindingListener接口

如果一个对象实现了HttpSessionBindingListener接口，这个对象被绑定到Session中或者从Session中被删除时，Servlet容器会通知这对象，而这对象在接收到通知后，可以做一些初始化或清除状态的操作。

例如：在网络购物应用中，可以让购物车对象实现HttpSessionBindingListener接口，当顾客在选购商品时，web应用程序创建购物车对象，保存在Session中，并在购物车中放入预选的商品。当顾客没有结账就离开了站点，或者顾客在浏览器商品介绍的时候，Session超时，这个时候，Servlet容器就会通知购物车对象，它要从Session中被删除。购物车对象在得到通知后，可以把顾客选购的商品信息保存到数据库中。当顾客再次来到网站购物的时候，web应用程序将购物车对象保存（绑定）到Session中时，Servlet容器会通知购物车对象，此时，购物车对象可以从数据库中加载先前保存的商品信息。顾客会惊奇地发现，以前预购的商品信息任然存在。

**javax.servlet.http.HttpSessionBindingListener接口的方法**：

**public void valueBound(HttpSessionBindingEvent event)**

当对象正在被绑定到Session中，Servlet容器调用这个方法来通知该对象。

**public void valueUnbound(HttpSessionBindingEvent event)**

当从Session中删除对象时，Servlet容器调用这个方法来通知该对象

Servlet容器通过HttpSessionBindingEvent对象来通知实现了HttpSessionBindingListener接口的对象，而这个对象可以利用HttpSessionBindingEvent对象来访问与它相联系的HttpSession对象。

**javax.servlet.http.HttpSessionBindingEvent类提供了以下两种方法：**

**public HttpSessionBindingEvent(HttpSession session, String name)**

**public HttpSessionBindingEvent(HttpSession session, String name, Object value)**

上面两个构造方法构造一个事件对象，当一个对象被绑定到Session中或者从Session中被删除时，用这个事件对象来通知它。

**public String getName()**

返回绑定到Session中或者从Session中删除的属性的名字

**public Object getValue()**

返回被添加、删除或替换的属性的值。如果属性被添加或者被删除，这个方法返回属性的值。如果属性被替换，这方法返回属性先前的值。

**public HttpSession getSession()**

返回HttpSession对象

## Servlet上下文

运行在Java虚拟机中的每一个Web应用程序都有一个与之相关的Servlet上下文。Java Servlet API提供了一个ServletContext接口用来表示上下文。这个接口中定义了一组方法，Servlet可以使用这些方法与它的Servlet容器进行通信。例如：得到文件的MIME类型，转发请求，或者向日志文件中写入日志消息。

ServletContext对象是Web服务器中的一个已知路径的根。例如：Servlet上下文被定位于http://localhost:8080/ch06。以/ch06请求路径（称为上下文路径）开始的所有请求被发送到与此ServletContext关联的Web应用程序。ServletContext对象的作用域为Web应用程序的所共有，也就是在整个Web应用程序内都可以访问ServletContext对象内的数据。

### ServletContext接口

一个ServletContext对象表示一个Web应用程序的上下文。Servlet容器在Servlet初始化期间，向其传递ServletConfig对象，可以通过ServletConfig对象的getServletContext()方法来得到ServletContext对象。也可以通过GenericServlet类的getServletContext()方法得到ServletContext对象，不过GenericServlet类的getServletContext()也是调用ServletConfig对象的getServletContext()方法来得到这个对象的。

ServletContext接口定义了下面这些方法，Servlet容器提供了这个接口的实现：

**public Object getAttribute(String name)**

该方法通过给定的参数name（属性名）在ServletContext作用域中获取属性的值。返回一个Object类型的对象

**public Enumeration getAttributeNames()**

该方法用于获取ServletContext作用域中所有的属性的名称，返回一个Enumeration枚举。可以通过循环遍历Enumeration获取每个参数的名称。

**public void removeAttribute(String name)**

该方法通过给定的属性名称来移除ServletContext作用域中对应的属性值。

**public void setAttribute(String name, Object object)**

该方法用于向ServletContext作用域中设置属性和值。

上面4种方法用于读取、移除和设置共享属性，任何一个Servlet都可以设置某个属性，而同一个Web应用程序的另一个Servlet可以读取这个属性，不管这些Servlet是否为同一个客户进行服务。

**public ServletContext getContext(String uripath)**

该方法返回服务器上与指定的URL相对应的ServletContext对象。给出的uripath参数必须以斜杠（/）开始，被解释为相对于服务器文档根的路径。如果调用该方法访问一个受限制的ServletContext对象，那么该方法将返回null。

**public String getInitParameter(String name)**

该方法用于获取在web.xml配置文件中定义的初始化属性的值。其中name为配置文件中对应的属性名称。

**public Enumeration getInitParameterNames()**

该方法用于获取所有在web.xml配置文件中定义的属性的属性名称，返回一个Enumeration类型的对象。

上面2种方法可以获取为Servlet上下文定义的初始化参数，这些参数被整个Web应用程序所使用。可以在配置文件（web.xml）中使用<context-param>元素定义上下文的初始化参数。

**public int getMajorVersion()**

**public int getMinorVersion()**

上面两个方法用于返回Servlet容器支持的Java Servlet API的主版本和次版本号。例如。对于遵从Servlet2.4版本的容器，getMajorVersion()方法返回2，getMinorVersion()方法返回4。

**public String getMimeType(String file)**

该方法返回指定文件的MIME类型，如果类型是未知的，这个方法将返回null。MIME类型的检测是根据Servlet容器的配置，也可以在Web应用程序的配置文件中指定。

**public RequestDispatcher getRequestDispatcher(String path)**

该方法返回一个RequestDispatcher对象，作为给定路径上的资源的封装。可以使用RequestDispatcher对象将一个请求转发给其他资源进行处理，或者在响应中包含资源。要注意的是，传入的参数path必须以斜杠（/）开始，被解释为相对于当前上下文根的路径。

**public RequestDispatcher getNamedDispatcher(String name)**

该方法与getRequestDispatcher()方法类似。不同之处在于，该方法接受一个在配置文件中以<servlet-name>元素给出的Servlet（或JSP页面）的名字作为参数。

**public String getRealPath(String path)**

在一个Web应用程序中，资源用相对于上下文路径的路径来引用，这个方法可以返回资源在服务器文件系统上的真实路径（文件的绝对路径）。返回的真实路径的格式应该适合于运行这个Servlet容器的计算机和操作系统（包括正确的路径分隔符）。如果Servlet容器不能够将虚拟路径转换为真实的路径，这方法将返回null。

**public URL getResource(String path)**

该方法返回被映射到指定路径上的资源的URL。传入的参数path必须以斜杠（/）开始，被解释为相对于当前上下文根的路径。这个方法允许Servlet容器从任何来源的Servlet生成一个可用的资源。资源可以是在本地或远程文件系统上，在数据库中，或者在WAR文件中。如果没有资源映射到指定的路径上，该方法将返回null。

**public InputStream getResourceAsStream(String path)**

该方法与getResource()方法类似，不同之处在于，该方法返回资源的输入流对象。另外，使用getResourceAsStream()方法，元信息（如内容长度和内容类型）将丢失，而使用getResource()方法，元信息是可用的。

**public Set getResourcePaths(String path)**

该方法返回资源的路径列表，参数path必须是以斜杠（/）开始，指定用于匹配资源的部分路径。例如，一个Web应用程序包含了下列资源：

/welcome.html

/catalog/index.html

/catalog/products.html

/catalog/offers/books.html

/catalog/offers/music.html

/customer/login.jsp

/WEB-INF/web.xml

/WEB-INF/classes/com.acme.OrderServlet.class

如果调用getResourcePaths(“/”)，将返回[/welcome.html, /catalog/,/customer/,/WEB-INF/]。如果调用getResourcePaths(“/catalog/”)，将返回[/catalog/index.html, /catalog/products.html, /catalog/offers/]

**public String getServerInfo()**

该方法返回运行Servlet的容器的名称和版本

**public String getServletContextName()**

该方法返回在配置文件中使用<display-name>元素指定的对应当前ServletContext的Web应用程序的名称。

**public void log(String msg)**

**public void log(String message, Throwable throwable)**

ServletContext接口提供了上面两个记录日志的方法，第一个方法用于记录一般的日志，第二个方法用于记录指定异常的栈跟踪信息。

## Servlet跳转

### 请求转发

请求转发指的是在Web应用中，客户端请求可以统一为一个请求路径，然后通过Servlet（控制器）将客户端的不同请求转发给相应的Servlet或JSP页面，然后由他他们进行处理并产生对请求的相应。这一过程为请求转发。

要实现请求转发，就要用到**javax.servlet.RequestDispatcher**接口

#### RequestDispatcher接口

RequestDispatcher对象由Servlet容器创建，用于封装一个由路径所标识的服务器资源。利用RequestDispatcher对象，可以把请求转发给其他的Servlet或JSP页面。

RequestDispatcher接口定义了两种方法：

**public void forward(ServletRequest request, ServletResponse response)**

该方法用于将请求从一个Servlet传递给服务器上的另外的Servlet、JSP页面或者是HTML文件。在Servlet中，可以对请求做一个初步的处理，然后调用这个方法，将请求传递给其他的资源来输出响应。要注意的是，这个方法必须在响应被提交给客户端之前调用，否则的化，会抛出异常。forward()方法调用之后，原先在响应缓存中的没有提交的内容将被自动清除。

**public void include(ServletRequest request, ServletResponse response)**

该方法用于在响应中包含其他资源的内容。和forward()方法的区别在于：利用include()方法将请求转发给其他Servlet，被调用的Servlet对该请求做出的响应将并入原先的响应对象中，原先的Servlet还可以继续输出响应信息。而利用forward()方法将请求转发给其他Servlet，将由调用的Servlet负责对请求做出响应，而原Servlet的执行则终止。

#### 得到RequestDispatcher对象

有三种方法可以用来得到RequestDispatcher对象：

1. 利用ServletRequest接口中的getRequestDispatcher()方法：

**public** RequestDispatcher getRequestDispatcher(String path)

1. 利用ServletContext接口中的getNamedDispatcher()

**public** RequestDispatcher getNamedDispatcher(String name)

1. 利用ServletContext接口中的getRequestDispatcher()

**public** RequestDispatcher getRequestDispatcher(String path)

可以看到ServletRequest接口和ServletContext接口各自提供了一种同名的方法getRequestDispatcher()，且参数都是资源的路径名，不过ServletContext接口的getRequestDispatcher()方法的参数必须以斜杠（/）开始，被解释为相对于当前上下文根的路径。而ServletRequest的getRequestDispatcher()方法的参数不但可以上相对于上下文根的路径，而且可以是相对于当前Servlet的路径。

#### **注意事项**

1. 转发只能在同一个应用的组件之间进行，不可以转发给其他应用的地址。
2. 转发的地址可以用“相对地址”方式，也可以用“绝对地址”方式。但需要注意的是：用绝对地址方式时,应从应用名后(Context path)开始。例如：要转发到的地址为：http://192.168.5.100/tst/jsp/somewhere.jsp (其中tst为应用名)，对应的绝对地址为：“/jsp/somewhere.jsp”。这是很好理解的，因为转发只能转到本应用内的地址，所以绝对地址是没有必要包含应用名的。
3. 组件之间通过转发来分工协作，势必涉及数据的传递，可以通过request对象传递数据。request对象的setAttribute和getAttribute分别用于以“名称-对象对”的形式存取数据。
4. 在一个组件转发给另外一个组件之前，通过response输出的响应内容是没有意义的。在转发之前的通过out对象输出的内容最终不会输出到浏览器，这是由于RequestDispatcher在转发之前清空了输出缓冲区。但如果在转发之前输出的信息超出了缓冲区，或者调用了out对象的flush()方法，此响应内容已经输出到了客户端(称之为响应信息已提交)，此时如果实施转发操作会抛出运行时异常：Java.lang.IllegalStateException。

#### 请求转发的实例

编写一个PortalServlet，判断用户是否已经登录，如果没有登录，则调用RequestDispatcher接口的include()方法，将请求转发给LoginServlet，LoginServlet在响应中发送登录表单；如果已经登录，则调用RequestDispatcher接口的forward()方法，将请求转发给WelcomeServlet，向用户显示欢迎信息。

1. 编写PortalServlet类（控制器）

**public** **class** PortalServlet **extends** HttpServlet {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**protected** **void** doGet(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {

response.setContentType("text/html;charset=utf-8");

PrintWriter out = response.getWriter();

out.print("<html><head><title>");

out.print("登录页面");

out.print("</title></head><body>");

String name = request.getParameter("user");

String pwd = request.getParameter("password");

**if**("zhangsan".equals(name) && "123".equals(pwd)){

ServletContext context = **this**.getServletContext();

RequestDispatcher rd = context.getRequestDispatcher("/welcom");

rd.forward(request, response);

}**else**{

RequestDispatcher rd = request.getRequestDispatcher("login2");

rd.forward(request, response);

}

out.print("</body></html>");

out.flush();

out.close();

}

**protected** **void** doPost(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {

doGet(request, response);

}

}

1. 编写LoginServlet类

**public** **class** LoginServlet **extends** HttpServlet {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**protected** **void** doGet(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {

response.setContentType("text/html;charset=utf-8");

PrintWriter out = response.getWriter();

out.println("<form method=post action=protal>");

out.println("<table>");

out.println("<tr><td>请输入用户名</td><td><input type=text name=user></td></tr>");

out.println("<tr><td>请输入密码</td><td><input type=password name=password></td></tr>");

out.println("<tr><td><input type=reset value=重置></td><td><input type=submit value=登录></td></tr>");

out.println("</table>");

out.println("</form>");

out.flush();

out.close();

}

**protected** **void** doPost(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {

doGet(request, response);

}

}

1. 配置Servlet

<servlet>

<servlet-name>PortalServlet</servlet-name>

<servlet-class>com.uc.PortalServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>PortalServlet</servlet-name>

<url-pattern>/protal</url-pattern>

</servlet-mapping>

### 重定向

浏览器向某组件发出请求信息，Web容器在接收客户的请求后执行对应的Servlet组件。当执行到重定向的代码时，Web容器会将重定向的URL和302状态码封装进响应头中发送到浏览器。浏览器在接收到响应信息后，将URL解析出来再重新向Web容器发送新的URL请求。重定向的整个过程涉及两次“请求-响应”。

#### sendRedirect()

重定向可以通过response的sendRedirect(String url)方法来实现，注意String类型的参数url表示重定向到的地址，需要注意的是，如果表示重定向到本应用内的一个绝对地址时，要从应用名前开始，例如：tst应用中的某个组件要重定向到本应用内的/jsp/somewhere.jsp，则重定向的绝对地址应该是：“/tst/jsp/somewhere.jsp”这一点和转发中的绝对地址表示是不同的。

#### 注意事项

1. 使用response.sendRedirect时，前面不能有HTML输出

这并不是绝对的，不能有HTML输出其实是指不能有HTML被送到了浏览器。事实上现在的server都有cache机制，一般在8K（我是说JSP　SERVER），这就意味着，除非你关闭了cache，或者你使用了out.flush()强制刷新，那么在使用sendRedirect之前，有少量的HTML输出也是允许的。

1. response.sendRedirect之后，应该紧跟一句return;

我们已经知道response.sendRedirect是通过浏览器来做转向的，所以只有在页面处理完成后，才会有实际的动作。既然你已经要做转向了，那么后的输出还有什么意义呢？而且有可能会因为后面的输出导致转向失败。

### sendRedirect()和forward()方法的区别

HttpServletResponse接口的sendRedirect()方法和RequestDispatcher接口的forward()方法都可以利用另外的资源（Servlet、JSP页面、HTML文件）来为客户端进行服务，但是这两种方法有着本质上的区别

区别一：

sendRedirect()方法：该方法实际上是告诉浏览器另外资源所在的位置，让浏览器重新访问另外的资源。调用sendRedirect()方法，会在响应中设置Location响应报头。地址栏中的请求会发生变化。

forward()方法：该方法是浏览器发出请求后，服务器调用forward()方法将浏览器的请求转发给另外的资源。并在地址栏中显示的URL始终是原始请求的URL。

区别二:

sendRedirect()方法和forward()方法还有一个区别是，sendRedirect()方法不但可以向位于同一主机上的不同Web应用程序之间进行重定向，而且可以将客户端重· 定向到其他服务器上的Web应用程序资源。而forward()方法则只能在同一web应用程序中进行转发处理。

## Servlet异常

在javax.servlet包中定义了两个异常类，ServletException和UnavailableException。

### ServletException类

ServletException类定义了一个通用的异常，可以被init()、service()、doXXX()方法抛出，这个类提供了4个构造方法和1个实例方法：

**public ServletException()**

该方法构造一个新的Servlet异常

**public ServletException(String message)**

该方法用指定的消息构造一个新的Servlet异常。这个消息可以被写入服务器的日志中或者显示给客户

**public ServletException(String message, Throwable rootCause)**

在Servlet执行时，如果有一个异常阻碍了Servlet的正常操作，那么这个异常就是根原因异常。如果需要在一个Servlet异常中包含根原因的异常，可以调用这个构造方法，同时包含一个描述消息。例如：可以在ServletException异常中嵌入一个java.sql.SQLException异常。

**public ServletException(Throwable rootCause)**

该方法同上，只是没有指定描述消息的参数。

**public Throwable getRootCause()**

该方法返回引起这个异常的异常，也就是返回根原因的异常。

### UnavailableException类

UnavailableException类是ServletException类的子类，该异常被Servlet抛出，用于向Servlet容器指示这个Servlet永久地或者暂时地不可用。

该类提供了2个构造方法和2个实例方法：

**public UnavailableException(String msg)**

该方法用一个给定的消息构造一个新的异常，指示Servlet永久不可用。

**public UnavailableException(String msg, int seconds)**

该方法用一个给定的消息构造一个新的异常，指示Servlet暂时不可用。其中的参数seconds指明在这个以秒为单位的时间内，Servlet不可用。如果Servlet不能估计出多长时间后它将恢复功能，可以传递一个负数或零给seconds参数。

**public int getUnavailableSeconds()**

该方法返回Servlet预期的暂时不可用的秒数。如果返回一个负数，表明Servlet永久不可用或者不能估计出Servlet多长时间不可用。

**public boolean isPermanent()**

该方法返回一个布尔值，用于指示Servlet是否是永久不可用。返回true，表明Servlet永久不可用，返回false，表明Servlet可用或者暂时不可用。

## Web应用程序的部署

Web应用程序的开发分为设计开发与配置部署两个阶段。通过部署，实现了组件与组件之间的松耦合，降低了Web应用程序的维护难度。

### 配置任意目录下的Web应用程序

一个Web容器可以运行多个Web应用程序，每个Web应用程序都是一个唯一的上下文根，上下文根如何部署是和具体的web容器相关的。对于Tomcat：%CATALINA\_HOME%\webapps目录下的每一个子目录都是一个独立的Web应用程序，这个子目录的名字就是这个Web应用程序的上下文根。

在部署和访问Web应用程序时，通过上下文路径来区分不同的Web应用程序。上下文路径以斜杠（/）开头，但结束没有斜杠（/）。在默认上下文中，这个路径将是空字符串。

### web.xml配置文件

前言：一般的web工程中都会用到web.xml，web.xml主要用来配置，可以方便的开发web工程。web.xml主要用来配置Filter、Listener、Servlet等。但是要说明的是web.xml并不是必须的，一个web工程可以没有web.xml文件。

* WEB工程加载web.xml过程 经过个人测试，WEB工程加载顺序与元素节点在文件中的配置顺序无关。即不会因为 filter 写在 listener 的前面而会先加载 filter。WEB容器的加载顺序是：ServletContext -> context-param -> listener -> filter -> servlet。并且这些元素可以配置在文件中的任意位置。加载过程顺序如下：

1. 启动一个WEB项目的时候，WEB容器会去读取它的配置文件web.xml，读取<listener>和<context-param>两个结点。
2. 紧急着，容器创建一个ServletContext（servlet上下文），这个web项目的所有部分都将共享这个上下文。
3. 容器将<context-param>转换为键值对，并交给servletContext。
4. 容器创建<listener>中的类实例，创建监听器。

* **web.xml文件元素详解**

1. **schema**

web.xml的模式文件是由Sun公司定义的，每个web.xml文件的根元素<web-app>中，都必须标明这个 web.xml使用的是哪个模式文件。其它的元素都放在<web-app></web-app>之中。

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns=*"http://xmlns.jcp.org/xml/ns/javaee"*

xsi:schemaLocation=*"http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_3\_1.xsd"*

id=*"WebApp\_ID"* version=*"3.1"*>

</web-app>

1. **<icon>Web应用图标**

指出IDE和GUI工具用来表示Web应用的大图标和小图标

<icon>

<small-icon>/images/app\_small.gif</small-icon>

<large-icon>/images/app\_large.gif</large-icon>

</icon>

1. **<display-name>Web应用名称**

提供GUI工具可能会用来标记这个特定的Web应用的一个名称

<display-name>TestJavaEE</display-name>

1. **<description>Web应用描述**

给出于此相关的说明性文本

1. **<distributable/>**

<distributable/>可以使用distributable元素来告诉servlet/JSP容器，Web容器中部署的应用程序适合在分布式环境下运行。

1. **<context-param>**

声明应用范围内的初始化参数。它用于向 ServletContext提供键值对，即应用程序上下文信息。我们的listener, filter等在初始化时会用到这些上下文中的信息。在servlet里面可以通过getServletContext().getInitParameter("context/param")得到

<context-param>

<description>参数描述</description>

<param-name>参数名</param-name>

<param-value>参数值</param-value>

</context-param>

1. **<filter>过滤器配置**

将一个名字与一个实现javaxs.servlet.Filter接口的类相关联

<filter>

<filter-name>过滤器名称（自定义）</filter-name>

<filter-class>完整的类限定名</filter-class>

<init-param>

<param-name>参数名称</param-name>

<param-value>参数值</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>过滤器名称（以<filter>中定义的保持一致）</filter-name>

<url-pattern>需要过滤的URL路径（自定义）</url-pattern>

</filter-mapping>

1. **<listener>监听器配置**

<listener>

<listener-class>完整的类限定名</listener-class>

</listener>

1. **<servlet>配置**

<servlet></servlet> 用来声明一个servlet的数据，主要有以下子元素：  <servlet-name></servlet-name> 指定servlet的名称  <servlet-class></servlet-class> 指定servlet的类名称  <jsp-file></jsp-file> 指定web站台中的某个JSP网页的完整路径  <init-param></init-param> 用来定义参数，可有多个init-param。在servlet类中通过getInitParamenter(String name)方法访问初始化参数  <load- on-startup></load-on-startup>指定当Web应用启动时，装载Servlet的次序。当值为正数或零时：Servlet容器先加载数值小的servlet，再依次加载其他数值大的servlet。当值为负或未定义：Servlet容器将在Web客户首次访问这个servlet时加载它。 <servlet-mapping></servlet-mapping> 用来定义servlet所对应的URL，包含两个子元素  <servlet-name></servlet-name> 指定servlet的名称  <url-pattern></url-pattern> 指定servlet所对应的URL

<servlet>

<servlet-name>PortalServlet</servlet-name>

<servlet-class>com.uc.PortalServlet</servlet-class>

<init-param>

<param-name>参数名</param-name>

<param-value>参数值</param-value>

</init-param>

<run-as>

<description>Security rols for anonymous access</description>

<role-name>tomcat</role-name>

</run-as>

<load-on-startup>指定当Web应用启动时，装载Servlet的次序</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>PortalServlet</servlet-name>

<url-pattern>/protal</url-pattern>

</servlet-mapping>

1. **<session-config>会话超时配置**

单位为分钟

<session-config>

<session-timeout>120</session-timeout>

</session-config>

1. **<mime-mapping>MIME类型配置**

<mime-mapping>

<extension>html</extension>

<mime-type>text/html</mime-type>

</mime-mapping>

1. **<welcome-file-list>欢迎文件页**

<welcome-file-list>

<welcome-file>index.html</welcome-file>

<welcome-file>index.jsp</welcome-file>

</welcome-file-list>

1. **<error-page>错误页面**

通过错误码来配置error-page

<!--配置了当系统发生404错误时，跳转到错误处理页面NotFound.jsp-->

<error-page>

<error-code>404</error-code>

<location>/NotFound.jsp</location>

</error-page>

通过异常的类型配置error-page

<!--配置了当系统发生java.lang.NullException（即空指针异常）时，跳转到错误处理页面error.jsp-->

<error-page>

<exception-type>java.lang.NullException</exception-type>

<location>/error.jsp</location>

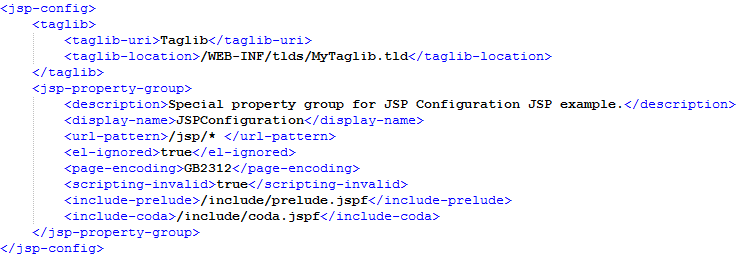
</error-page>

1. **TLD配置**
2. **<jsp-config>设置jsp**

<jsp-config> 包括 <taglib> 和 <jsp-property-group> 两个子元素。其中<taglib> 元素在 JSP 1.2 时就已经存在；而<jsp-property-group> 是JSP 2.0 新增的元素

<jsp-property-group> 元素主要有八个子元素，它们分别为：

1. <description>：设定的说明
2. <display-name>：设定名称
3. <url-pattern>：设定值所影响的范围，如： /CH2 或 /\*.jsp
4. <el-ignored>：若为 true，表示不支持 EL 语法
5. <scripting-invalid>：若为 true，表示不支持 <% scripting %>语法
6. <page-encoding>：设定 JSP 网页的编码
7. <include-prelude>：设置 JSP 网页的抬头，扩展名为 .jspf
8. <include-coda>：设置 JSP 网页的结尾，扩展名为 .jspf



对于Web 应用程式来说，Scriptlet 是个不乐意被见到的东西，因为它会使得HTML 与Java 程式码交相混杂，对于程式的维护来说相当的麻烦，必要的时候，可以在web.xml 中加上<script- invalid> 标签，设定所有的JSP 网页都不可以使用Scriptlet

* **Mapping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servlet A 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servlet A，也不会去理会其他的servlet了。
2. 最长路径匹配。例子：servlet A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servlet B。
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

以”/’开头和以”/\*”结尾的是用来做路径映射的。以前缀”\*.”开头的是用来做扩展映射的。所以，为什么定义”/\*.action”这样一个看起来很正常的匹配会错？因为这个匹配即属于路径映射，也属于扩展映射，导致容器无法判断。

## 数据库访问

### JDBC简介

JDBC是Java程序操作数据库的API，也是Java程序与数据库相交互的一门技术。JDBC是Java语言编写的类和接口组成，它对数据库的操作提供了基本方法，但对于数据库的细节操作由数据库厂商进行实现。使用JDBC操作数据库，需要数据库厂商提供数据库的驱动程序。

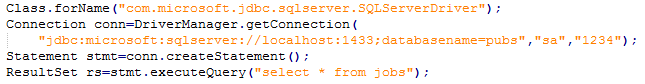
在JDBC3.0版本里，包括了两个包：java.sql和javax.sql

* java.sql这个包中的类和接口主要针对基本的数据库编程服务，如生成连接、执行语句，以及准备语句和运行批处理查询等。同时也有一些高级的处理，比如批处理更新、事务隔离和可滚动结果集等。
* javax.sql这个包主要为数据库方面的高级操作提供接口和类。如为连接管理、分布式事务和旧有的连接提供了更好的抽象，它引入了容器管理的连接池、分布式事务和行集等。

### JDBC API

JDBC API包含在JDK中，被分为两个包：java.sql和javax.sql。java.sql包定义了访问数据库的接口和类，其中一些接口由驱动程序供应商来实现。

简单访问数据库的代码：



#### 加载与注册数据库驱动

连接数据库前，需要将数据库厂商提供的数据库驱动类注册到JDBC的驱动管理器中。通常情况下，是通过将数据库驱动类加载到JVM来实现的。

加载JDBC驱动需要调用Class类的静态方法forName()，向其传递要加载的JDBC驱动的类名。Class.forName()

* + Driver接口

java.sql.Driver是所有JDBC驱动程序需要实现的接口。这个接口是提供给数据库厂商使用的。不同厂商实现Driver接口的类名是不相同的。

com.microsoft.jdbc.sqlserver.SQLServerDriver //SQL Server2000数据库的JDBC驱动类名

com.microsoft.sqlserver.jdbc.SQLServerDriver //SQL Server2005数据库的JDBC驱动类名

oracle.jdbc.driver.OracleDriver //Oralce数据库的JDBC驱动类名

com.mysql.jdbc.Driver //MySQL数据库的JDBC驱动类名

* + 加载JDBC驱动

Class类的静态方法forName()用于加载JDBC驱动，向该方法中传递JDBC驱动的类名即可。（在加载JDBC驱动之前需导入java.sql包）



* + 注册JDBC驱动

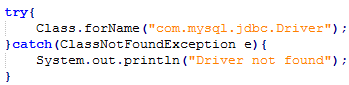
DriverManager类是驱动程序管理器类，用于管理驱动程序，主要作用于用户和驱动之间。该类中所有的方法都是静态方法。该类提供了registerDriver()方法用于注册驱动程序类。

语法格式：



通常我们不需要亲自调用registerDriver()方法来注册驱动程序类的实例，因为Driver接口的驱动程序类中都包含了静态代码块，这些静态代码块会调用DriverManager.registerDriver()方法来注册自身的一个实例。

实例：



#### 建立数据库连接

**getConnection()方法**

调用DriverManager类的getConnection()方法用于建立到数据库的连接，返回一个Connection对象。在DriverManager类中提供了3个重载的getConnection()方法。

* + public static Connection getConnection(String url) throws SQLException

该方法通过给出的数据库URL建立到数据库的连接。DriverManager类将从注册的JDBC驱动中选择一个合适的驱动，调用它的connec()方法建立到数据库的连接

* + public static Connection getConnection(String url, String user, String password) throws SQLException

该方法除了需要数据库的URL外，还需要用户名和密码来建立一个到数据库的连接

* + public static Connection getConnection(String url , Properties info) throws SQLException

该方法需要数据库的URL和java.util.Properties对象。Properties包含了用于特定数据库所需要的参数，以键-值对的方式指定连接参数。通常情况下至少需要指定user和password属性。

**JDBC URL**

URL用于标识一个被注册的驱动程序，驱动程序管理器通过这个URL选择正确的驱动程序，从而建立到数据库的连接

基本格式：JDBC协议+IP地址或域名+端口号+数据库名称

jdbc:sqlserver://127.0.0.1:1433;DatabaseName=pubs //SQL Server数据库的URL

jdbc:oracle:thin:@127.0.0.1:1521:ORCL //Oracle数据库的URL

jdbc:mysql://127.0.0.1:3306/databasename //MySQL数据库的URL

Connection对象时JDBC封装的数据库连接对象，只有创建此对象后，才可以对数据库进行相关操作

DriverManager.getConnection(url , user ,password);

Connection对象的获取需要用到DriverManager对象，DriverManager的getConnection()方法通过数据库连接URL、数据库用户名及密码创建Connection对象。



#### 访问数据库

在java.sql包中给我们提供了3个接口，用于实现对数据库的操作及访问。

##### Statement接口

通过调用Connection对象的createStatement()方法创建一个Statement对象。该对象用于执行静态的SQL语句，返回执行的结果。

* Statement createStatement() throws SQLException

该方法创建一个Statement对象，用于向数据库发送SQL语句。没有参数的SQL语句通常用Statement对象来执行。

在Statement接口中定义了下列方法用于执行SQL语句。

* ResultSet executeQuery(String sql) throws SQLException

该方法执行参数sql指定的SQL语句，返回一个ResultSet对象。ResultSet对象用于查看执行的结果。

* int executeUpdate(String sql) throws SQLException

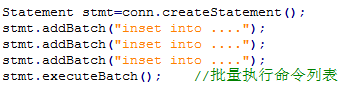
该方法执行参数sql指定的insert、update、delete语句。另外该方法也可以执行SQL DDL语句，例如：create table

* boolean execute(String sql) throws SQLException

该方法执行返回多个结果集的SQL语句。在某些情况下，一条SQL语句可以返回多个结果集或者更新行数。通常情况下，我们可以不用这种方法，除非要执行一个返回多个结果集的存储过程或者动态执行一个未知的SQL串。如果使用该方法，就必须使用getResultSet()或者getUpdateCount()方法来获取结果，并且调用getMoreResults()方法来访问下一个结果集。如果第一个结果是一个ResultSet对象，返回true；如果返回的是一个更新行数或者没有结果，则返回false。

* int[] executeBatch() throws SQLException

该方法允许我们向数据库提交一批命令，然后一起执行。如果所有的命令都成功执行，返回值是一个更新行数的数组。数组中的每一个int元素是按照加入命令的先后顺序存储的，表示了相应命令的更新行数。可以使用addBatch()方法将SQL命令加入到命令列表中。



##### ResultSet

调用Statement对象的executeQuery()方法创建了一个ResultSet对象。该对象以逻辑表格的形式封装了执行数据库操作的结果集，ResultSet接口由数据库厂商实现。ResultSet对象维护了一个指向当前结果集数据行的游标，初始的时候，游标在第一行之前，可以通过ResultSet对象的next()方法移动游标到下一行。

* boolean next() throws SQLException

该方法移动游标到下一行，如果新的数据行有效，返回true，否则返回false。通过判断该方法的返回值，可以循环读取结果集中的数据行，直到最后。

ResultSet接口中还定义了很多方法来获取当前行中列的数据，根据字段类型的不同，采用不同的方法来获取数据。

|  |  |  |  |
| --- | --- | --- | --- |
| getArray() | getAsciiStream() | getBigDecimal() | getBinaryStream() |
| getBlob() | getBoolean() | getByte() | getBytes() |
| getCharacterStream() | getClob() | getDate() | getDouble() |
| getFloat() | getInt() | getLong() | getObject() |
| getShort() | getString() | getTime() | getTimestamp() |

这些方法中，提供了两种形式的调用：一种是以列的索引作为参数（注意索引从1开始），另一种是以列的名字作为参数。



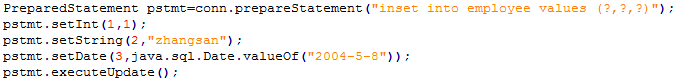
##### PreparedStatement

我们在程序中传递的SQL语句在执行之前都必须被预编译后才能被数据库引擎执行。如果重复执行参数不同的SQL语句，则效率比较低效。如果想用不同的参数来多次执行同一个SQL语句，则可以使用PreparedStatement对象。

PreparedStatement接口从Statement接口继承而来，它的对象表示一条预编译过的SQL语句。我们可以通过调用Connection对象的prepareStatement()方法来得到PreparedStatement对象

PreparedStatement对象所代表的SQL语句中的参数用问号（?）来表示，调用PreparedStatement对的setXXX()方法来设置这些参数。

setXXX()方法有两个参数值，第一个参数值是要设置的SQL语句中的参数索引（从1开始），第二个参数是要设置的SQL语句中的参数的值。



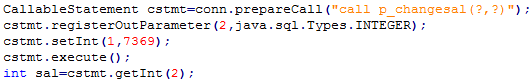
SQL数据类型与Java数据库类型的对比关系

|  |  |
| --- | --- |
| SQL数据类型 | Java数据类型 |
| INTEGER、INT | int |
| TINYINT、SMALLINT | short |
| BIGINT | long |
| DECIMAL、NUMERIC | java.math.BigDecimal |
| FLOAT | float |
| DOUBLE | double |
| CHAR、VARCHAR | String |
| BOOLEAN、BIT | boolean |
| DATE | java.sql.Date |
| TIME | java.sql.Time |
| TIMESTAMP | java.sql.Timestamp |
| BLOB | java.sql.Blob |
| CLOB | java.sql.Clob |
| ARRAY | java.sql.Array |

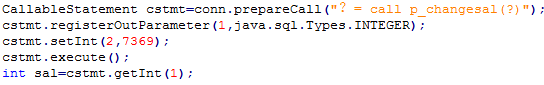
##### CallableStatement

CallableStatement对象用于执行SQL存储过程。CallableStatement接口从PreparedStatement接口继承而来，通过调用Connection对象的prepareCall()方法来得到CallableStatement对象。

在执行存储过程之前，凡是存储过程中类型为OUT的参数必须被注册，可以通过CallableStatement对象的registerOutParameter()方法来完成。对于类型为IN的参数，可以利用setXXX()方法来设置参数的值。



CallableStatement对象也可以用于执行函数



函数的参数总是IN类型，如果要接收函数的返回值，需要使用“?=”的语法形式，并调用registerOutParameter()方法注册函数的返回值。

##### ResultSetMetaData

ResultSetMetaData接口用于获取描述数据库表结构的元数据（在SQL中，用于描述数据库或者它的各个组成部分之一的数据称为元数据）

调用ResultSet对象的getMetaData()方法来得到ResultSetMetaData对象。

ResultSetMetaData接口定义常用的方法

* int getColumnCount() throws SQLException

该方法返回结果集中列的数量

* int getColumnDisplaySize(int column) throws SQLException

该方法返回列的最大字符宽度

* String getColumnName(int column) throws SQLException

该方法返回列的名字

* int getColumnType(int column) throws SQLExceptin

该方法返回列的SQL类型，该类型称作JDBC类型，在java.sql.Type类中定义。

* String getColumnTypeName(int column) throws SQLException

该方法返回列的数据库特定的类型名

* String getTableName(int column) throws SQLException

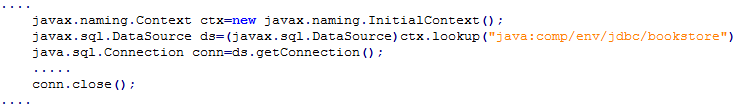
该方法返回列所属的表名

#### 事务处理

### JDBC数据源和连接池

在javax.sql包中，定义了DataSource接口，它提供了另一种方式建立数据库的连接。DataSource接口由驱动程序供应商来实现，利用DataSource来建立数据库的连接，不需要客户程序中加载JDBC驱动，也不需要使用java.sql.DriverManager类。在程序中通过向一个JNDI服务器查询来得到DataSource对象，然后调用DataSource对象的getConnection()方法来建立数据库的连接。DataSource对象可以看成是连接工厂，用于提供到此DataSource对象所表示的物理数据源的连接。

一个DataSource对象代表一个真正的数据源。根据DataSource的实现方法，数据源既可以是关系数据库、电子表格、还可以是一个表格形式的文件。当一个DataSource对象注册到名字服务中，应用程序就可以通过名字服务获得DataSource对象，并用它来产生一个与DataSource代表的数据源之间的连接。



javax.naming.Context接口表示一个命名上下文。这个接口中，定义了将对象和名字绑定，以及通过名字查询对象的方法。

javax.naming.InitialContext类是Context接口的实现类。

查询一个命名的对象，通过调用Context接口的lookup()方法，如下所示：

* + Object lookup(String name)throws NamingException

JNDI名称空间由一个初始的命名上下文（context）及其下的任意数目的子上下文组成。JDNI名称空间是分层次的，这与许多文件系统的目录/文件结构类似，初始上下文与文件系统的根类似，子上下文与子目录类似。JNDI层次的根是初始上下文，在这里通过变量ctx来表示。在初始上下文下由许多子上下文，其中之一就是jdbc，jdbc子上下文保留给JDBC数据源使用。逻辑数据源的名字可以在子上下文jdbc中，也可以在jdbc下的子上下文中。层次中的最后一级元素是注册的对象（这和文件相类似），在这个示例中是书库元的逻辑名，即bookstore。

java:comp/env是环境命名上下文（Environment Naming Context，ENC），引入它是为了了解JNDI命名冲突的问题。ENC将资源引用名和实际的JNDI名相分离，从而提高了J2EE应用的移植性。

javax.sql.DataSource接口可以有3中类型的实现。

* + 基本的实现------产生一个标准的连接对象，与调用DriverManager.getConnection()方法得到的连接对象一样，这是一个到数据库的物理连接。
  + 连接池实现------产生一个自动参与到连接池中的连接对象。这种实现需要和一个中间层连接池管理器一起工作。
  + 分布式事务实现------产生一个on股分布式事务的连接对象，这种连接对象几乎总是参与到连接池中。这种实现需要和一个中间层事务管理器和连接池管理器一起工作。

#### 数据库连接池

我们知道，建立数据库连接是相当耗时和耗费资源的，而且一个数据库服务器同时建立的连接数也是有限的，在大型的web应用中，可能同时会有成百上千访问数据库的请求，如果web应用程序为每一个客户请求分配一个数据库连接，将导致性能的急剧下降。为了你能够重复利用数据库连接，提高对请求的响应时间和服务器的性能，可以采用连接池技术。连接池技术预先建立多个数据库连接对象，然后将连接对象保存到连接池中，当客户请求到来是，从池中取出一个连接对象为客户服务，当请求完成后，客户程序调用close()方法，将连接对象放回池中。

在普通的数据库访问程序中，客户程序得到的连接对象是物理连接，调用连接对象的close()方法将关闭连接，而采用连接池技术，客户程序得到的连接对象是连接池中物理连接的一个句柄，调用连接对象的close()方法，物理连接并没有关闭，数据源的实现只是删除了客户程序中的连接对象和池中的连接对象之间的连接。

要使用数据源和连接池，必须要有数据源和连接池的实现，这在J2SE的类库中是没有提供的。Tomcat提供了数据源和连接池的实现（使用开源的DBCP连接池实现）JDBC数据库连接池的实现必须实现javax.sql.DataSource接口。

##### Tomcat实现数据库连接池

Tomcat服务器内置DBCP数据连接池，它是 apache 上的一个Java连接池项目。DBCP通过连接池预先同数据库建立一些连接放在内存中(即连接池中)，应用程序需要建立数据库连接时直接到从接池中申请一个连接使用，用完后由连接池回收该连接，从而达到连接复用，减少资源消耗的目的。

1. 配置连接池文件

配置连接池文件有两种方法，第一种直接配置Tomcat的conf/context.xml配置文件。具体配置如下：

在<Context>元素的内容中使用<Resource />元素来配置JDBC数据源

<Resource />元素的数据描述：

|  |  |
| --- | --- |
| **属性** | **描述** |
| name | 指定资源相对于java:comp/env上下文的JNDI名 |
| auth | 指定资源的管理者，它有两个可选的值：Application和Container。如果web应用程序在web.xml文件中使用<resource-ref>元素，那么这个属性是不需要的，如果使用<resource-env-ref>元素，那么这个属性是可选的。 |
| type | 指定资源所属的Java类的完整限定名 |
| maxActive | 指定在连接池中数据库连接的最大数目，指定这个值需要参照使用数据库所配置的最大连接数。取值为0，表示没有限制 |
| maxIdle | 指定在连接池中保留的空闲的数据库连接的最大数目。取值为-1，表示没有限制 |
| maxWait | 指定等待一个数据库连接成功为可用状态的最大时间，以毫秒为单位。如果设为-1.表示永久等待 |
| username | 指定连接数据库的用户名 |
| password | 指定连接数据库的用户密码 |
| driverClassName | 指定JDBC驱动程序类名 |
| url | 指定连接数据库的URL |



第二种是在项目的META-INF文件夹中新建一个context.xml文件，内容如上图

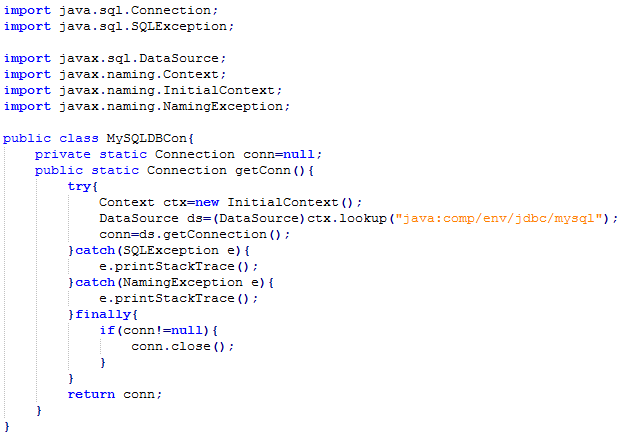
1. 添加数据库驱动程序 .jar

如果使用第一种方式配置context.xml配置文件，则需要将数据库驱动程序添加到Tomcat安装目录下的bil文件夹下。

如果使用第二种方式配置context.xml配置文件，则需要将数据库驱动程序添加到项目中的WEB-INF/lib文件夹下

1. 使用Tomcat服务器实现的DataSource接口连接内置DBCP池连接对象。

以下方法只实用于Tomcat服务器。

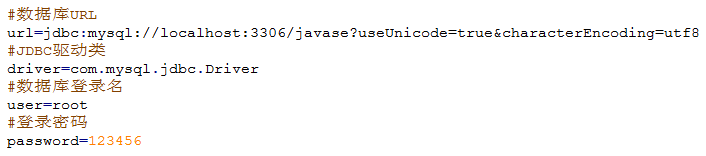


##### DBCP开源连接池

是Apache上的一个 Java连接池项目，也是 tomcat使用的连接池组件。单独使用dbcp需要导入common-dbcp.jar包。由于建立数据库连接是一个非常耗时耗资源的行为，所以通过连接池预先同数据库建立一些连接，放在内存中，应用程序需要建立数据库连接时直接到连接池中申请一个就行，用完后再放回去。dbcp没有自动的去回收空闲连接的功能。

1. 下载DBCP工具包：<http://commons.apache.org/proper/commons-dbcp/download_dbcp.cgi> 解压后将commons-dbcp2-2.1.1.jar 导入项目bil文件夹。
2. 实现DBCP数据库连接池的方式：
3. 使用简单版配置文件的方式（其中连接池参数需要代码来配置）

**配置文件（jdbc.properties）**



**实现代码**



连接池的参数设置：

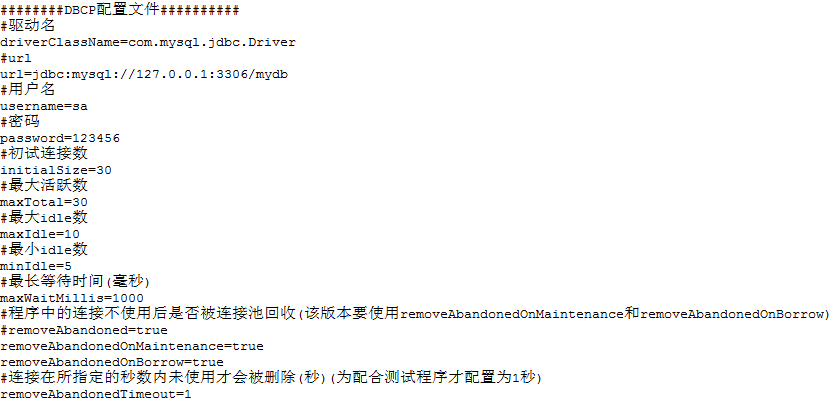
setInitialSize(int initialSize)

初始化连接，连接池启动时创建的初始化连接数量（默认值为0）

setMaxActive(int maxActive)

1. 使用标准的配置文件的方式

使用读取配置文件的方式，新建一个dbcp.properties配置文件，并且放在src根目录下。



1. 实现连接池：



##### C3P0开源连接池

C3P0是一个开源的JDBC连接池，它实现了数据源和JNDI绑定，支持JDBC3规范和JDBC2的标准扩展。c3p0是异步操作的，缓慢的JDBC操作通过帮助进程完成。扩展这些操作可以有效的提升性能。目前使用它的开源项目有hibernate，spring等。c3p0有自动回收空闲连接功能。

使用C3P0连接池具体实现步奏：

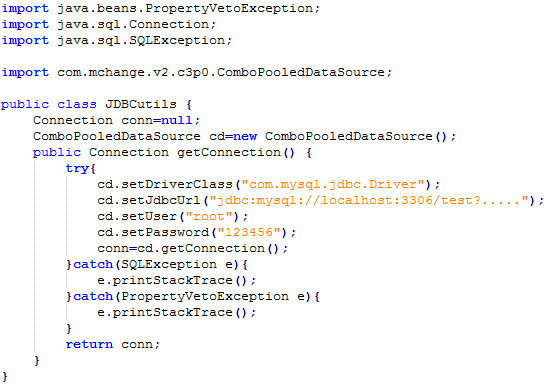
1. 下载C3P0工具包：https://sourceforge.net/projects/c3p0/files/latest/download 解压后将lib文件夹下的3个jar包导入到你的项目中。
2. 配置C3P0连接池

主要包括：初始化连接池时建立多少个连接、连接池最少多少个连接最多容纳多少连接、每个连接的生存时间、连接池能同时允许多少个操作进行，以及对具体数据库连接的配置：数据库的驱动、数据库的URL、数据库登录名、数据库密码、对这个数据库的连接池的细化配置（比如初始化时建立多少连接，最多最少连接数等等）。一个数据库的连接池配置用一个<named-config name="标识"> </name-config>节点来定义。在C3P0Utils中创建连接池时把 “标识” 作为连接池的构造函数的参数传入，则C3P0在配置文件中找到同名节点，按照这个节点的配置来创建相应配置的连接池。

c3p0的配置方式分为三种，分别是：

1. setXXX方法一个个地设置各个配置项
2. 类路径下提供一个c3p0.properties文件
3. 类路径下提供一个c3p0-config.xml文件

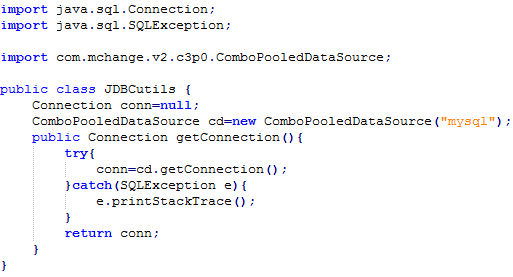
第一种：使用setXXX方法设置属性值：



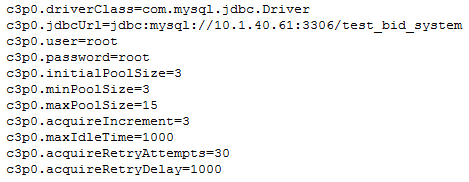
第二种：使用读取配置文件的方式,要求是配置文件必须命名为c3p0-config.xml,并且放在src根目录下，ComboPooledDataSource对象会自动读取名为c3p0-config.xml配置文件中的配置。配置文件如下:



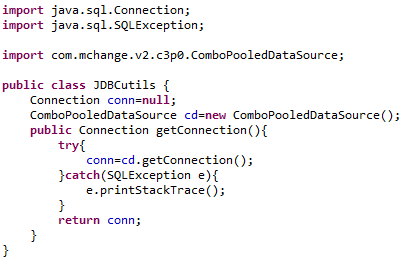
实现方法：



第三种：使用读取配置文件的方式,要求是配置文件必须命名为c3p0.properties,并且放在src目录下，ComboPooledDataSource对象会自动读取名为c3p0.properties配置文件中的配置配置文件如下:



实现方法：



##### Proxool

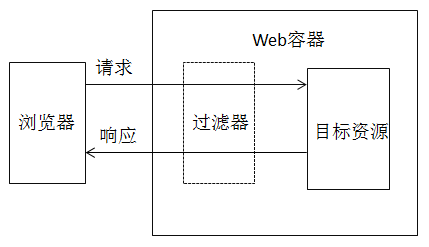
Proxool是一种Java数据库连接池技术，是sourceforge下的一个开源项目,这个项目提供一个健壮、易用的连接池，最为关键的是这个连接池提供监控的功能，方便易用，便于发现连接泄漏的情况

## Servlet的异常处理机制

## 开发线程安全的Servlet

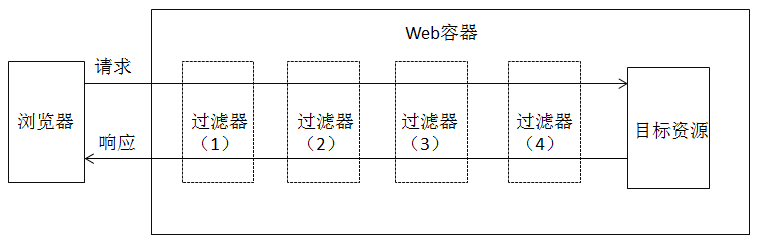
## 过滤器

过滤器（filter）是从Servlet2.3规范开始新增的功能，并在Servlet2.4规范中得到增强。过滤器，顾名思义，就是在源数据和目的数据之间起到过滤作用的中间组件。在Web应用程序中，过滤器是一个驻留在服务器端的Web组件，它可以截取客户端和资源之间的请求与响应信息，并对这些信息进行过滤。



当Web容器接收一个对资源的请求时，它将判断是否有过滤器与这个资源相关联。如果有，那么容器将把请求交给过滤器进行处理。在过滤器中，你可以改变请求的内容，或者从小设置请求的报头信息，然后再将请求发送给目标资源。当目标资源对请求作出响应时，容器同样会将响应先转发给过滤器，在过滤器中，你可以对响应的内容进行转换，然后再将响应发送到客户端。

在一个Web应用程序中，可以部署多个过滤器，这些过滤器组成了一个过滤器链。过滤器链中的每个过滤器负责特定的操作和任务，客户端的请求在这些过滤器之间传递，直到目标资源。



过滤器并不是必须要将请求传送到下一个过滤器（或目标资源），它也可以自行对请求进行处理，然后发送响应给客户单，或者将请求转发给另一个目标资源。

下面是过滤器在Web开发中的一些主要应用：

1. 对用户请求进行统一认证
2. 对用户的访问请求进行记录和审核
3. 对用户发送的数据进行过滤或替换
4. 转换图片格式
5. 对响应内容进行压缩，较少传输量
6. 对请求和响应进行加解密处理
7. 触发资源访问事件
8. 对XML的输出应用XSLT。

### Filter API

与过滤器开发相关的接口与类都包含在javax.servlet和javax.servlet.http包中，主要有下面的接口和类：

1. javax.servlet.Filter接口
2. javax.servlet.FilterConfig接口
3. javax.servlet.FilterChain接口
4. javax.servlet.ServletRequestWrapper类
5. javax.servlet.ServletResponseWrapper类
6. javax.servlet.http.HttpServletRequestWrapper类
7. javax.servlet.http.HttpServletResponseWrapper类

#### Filter接口

开发过滤器要实现javax.servlet.Filter接口，并提供了一个公开的不带参数的构造方法。在Filter接口中，定义了下面的三个方法：

**public void init(FilterConfig filterConfig)**

Web容器调用该方法来初始化过滤器。容器在调用该方法时，向过滤器传递FilterConfig对象，FilterConfig的用法和ServletConfig类似。利用FilterCofig对象可以得到ServletConfig对象，以及在部署描述符中配置的过滤器的初始化参数。

**public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)**

doFilter()方法类似于Servlet接口的Service()方法。当客户端请求目标资源的时候，容器就会调用与这个目标资源相关联的过滤器的doFilter()方法。在这个方法中，可以对请求和响应进行处理，实现过滤器的特定功能。在特定的操作完成后，可以调用chain.doFilter(request, response)将请求传给下一个过滤器，也可以直接向客户端返回响应信息，或者利用RequestDispatcher的forward()和include()方法，以及HttpServletResponse的sendRedirect()方法将请求重定向到其他资源。需要注意的是，这方法的请求和响应参数的类型是ServletRequest和ServletResponse，也就是说，过滤器的使用并不依赖于具体的协议。

**public void destroy()**

Web容器调用该方法指示过滤器的生命周期结束。在这个方法中，可以释放过滤器使用的资源。

与开发Servlet不同的是，Filter接口并没有相应的实现类可供继承，要开发过滤器，只能直接实现Filter接口。

#### FilterConfig接口

javax.servlet.FilterConfig接口用于在过滤器初始化时向其传递信息。FilterConfig接口由容器实现，容器将其实例作为参数传入过滤器对象的init()方法中。

在FilterConfig接口中，定义了4种方法：

**public String getFilterName()**

得到在配置文件（web.xml）中指定的过滤器的名字

**public String getInitParameter(String name)**

返回在配置文件中指定的名字为name的初始化参数的值。如果这个参数不存在，该方法将返回null。

**public Enumeration getInitParameterNames()**

返回过滤器的所有初始化参数的名字的枚举集合。如果过滤器没有初始化参数，这个方法将返回一个空的枚举集合。

**public ServletContext getServletContext()**

返回Servlet上下文对象的引用

#### FilterChain接口

javax.servletFilterChain接口由容器实现，容器将其实例作为参数传入过滤器对象的doFilter()方法中。过滤器对象使用FilterChain对象调用过滤器链中的下一个过滤器，如果该过滤器是链中最后一个过滤器，那么将调用目标资源。

FilterChain接口只有一个方法：

**public void doFilter(ServletRequest request, ServletResponse response)**

调用该方法将使过滤器链中的下一个过滤器被调用。如果调用该方法的过滤器是链中最后一个过滤器，那么目标资源被调用。

### 过滤器的部署

在实现一个过滤器后，需要配置文件（web.xml）中对过滤器进行配置，这是通过<filter>和<filter-maooing>元素来完成的

<filter>元素用于在Web应用程序中声明一个过滤器其中<description>、<display-name>、<icon>元素与配置Servlet中的元素的作用是相同的。<filter-name>元素用于为过滤器指定一个名字，该元素的内容不能为空。<filter-class>元素用于指定过滤器的完整的限定类名。<init-param>元素用于为过滤器指定初始化参数，它的子元素<param-name>指定参数的名字，<param-value>指定参数的值。在过滤器中，可以使用FilterConfig接口对象来访问初始化参数。

<filter>

<filter-name>guestbookFilter</filter-name>

<filter-class>com.uc.GuestbookFilter</filter-class>

<init-param>

<param-name>name</param-name>

<param-value>value</param-value>

</init-param>

</filter>

Servlet容器对配置文件中声明的每一个过滤器，只创建 一个实例。与Servlet类似，容器将在同一个过滤器实例上运行多个线程同时为多个请求服务，因此，开发过滤器时，也要注意线程安全的问题。如果在部署时，对同一个过滤器类声明了两次，那么容器将会创建两个相同的过滤器类的实例。

<filter-mapping>元素用于指定过滤器关联的URL样式或者Servlet。其中<filter-name>子元素的值必须是在<filter>元素中声明过的过滤器的名字。<url-pattern>元素和<servlet-name>元素可以选择一个；<url-pattern>元素指定过滤器关联的URL样式；<servlet-name>元素指定过滤器对应的Servlet。用户在访问<url-pattern>元素指定的URL（自定义URL）上的资源或<servlet-name>元素指定的Servlet时，该过滤器才会被容器调用。<filter-mapping>元素还可以包含0到4个<dispatcher>元素，<dispatcher>元素指定过滤器对应的请求方法，可以是REQUEST，INCLUDE，FORWARD和ERROR四种之一，默认为REQUEST。

1. REQUEST

当用户直接访问页面时，Web容器将会调用过滤器。如果目标资源是通过RequestDispatcher的include()或forward()方法访问时，那么该过滤器将不会被调用

1. INCLUDE

如果目标资源是通过RequestDispatcher的include()方法访问时，那么该过滤器将被调用。除此之外，过滤器不会被调用

1. FORWARD

如果目标资源是通过RequestDispatcher的forward()方法访问时，那么该过滤器将被调用。除此之外，过滤器不会被调用

1. ERROR

如果目标资源是通过声明式异常处理机制调用时，那么该过滤器将被调用。除此之外，过滤器不会被调用

<filter-mapping>

<filter-name>guestbookFilter</filter-name>

<url-pattern>/asd/\*</url-pattern>

<dispatcher>REQUEST</dispatcher>

<dispatcher>FORWARD</dispatcher>

</filter-mapping>

当访问的资源的URL是http://localhost:8080/chr/asd/...的资源则会先访问过滤器后再访问资源。

注意：在配置文件<url-pattern>元素中自定义的URL，如果配置为“/\*”，Web容器会将客户端对服务器所有资源（包括静态资源等）的访问都先提交给声明的过滤器。如果URL定义为/aaa/\*，则Web容器会将访问资源的URL中包含了/aaa/的URL访问先提交给该声明的过滤器，访问资源的URL不包含/aaa/的，则不会提交给过滤器。如果URL定义为/bbb/ccc.jsp，则Web容器只会将访问资源的URL中与/bbb/ccc.jsp完全匹配的先提交给过滤器。

### 过滤器的开发

1. 编写过滤器类SimpleFilter，它实现了Filter接口。

**public** **class** SimpleFilter **implements** Filter {

**public** **void** destroy() {

// **TODO** Auto-generated method stub

}

**public** **void** doFilter(ServletRequest request, ServletResponse response, FilterChain chain) **throws** IOException, ServletException {

response.setContentType("text/html;charset=utf-8");

PrintWriter out = response.getWriter();

out.println("before deFilter");

chain.doFilter(request, response);

out.println("after deFilter()");

out.close();

}

**public** **void** init(FilterConfig fConfig) **throws** ServletException {

// **TODO** Auto-generated method stub

}

}

1. 编写测试页面test.jsp

<!DOCTYPE html>

<html>

<head>

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

<meta http-equiv=*"Content-Type"* content=*"text/html; charset= UTF-8"*>

<title>测试页</title>

</head>

<body>

这是测试页面！

</body>

</html>

1. 部署过滤器

<filter>

<display-name>SimpleFilter</display-name>

<filter-class>com.uc.SimpleFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>SimpleFilter</filter-name>

<url-pattern>/test.jsp</url-pattern>

</filter-mapping>

当我们在访问test.jsp资源的时候，Web容器会先去访问过滤器，再去访问资源。在向客户端响应之前也会先访问过滤器，再将结果响应给客户端。

#### 验证用户有效的过滤器

对用户请求进行认证，这是Web应用程序中常见的功能。通常的做法是，当用户访问受保护的资源时，要求用户输入用户名和密码进行验证，验证通过后，将某个标志（fiag）保存到Session对象中；当用户再次访问受保护的资源时，则取出Session中的标记进行判断，如果用户已验证，则运行访问，如果没有则转到登录页面。

在对用户进行验证后的通常处理方式为：当用户验证成功后，向用户发送成功登录信息，并给出一个首页链接，让用户可以进入首页；当用户验证失败后，向用户发送错误信息，并给出一个返回到登录页面的链接，让用户可以重写登录。在实际操作中，有这样一种情况，当用户访问一个受保护的页面时，服务器端发送登录页面，用户的输入了正确的用户名和面后，希望能自动进入先前访问的页面，而不是进入首页。为了让用户在登录后直接进入先前的页面，我们需要在将登录页面发送给客户端之前保存用户先前访问页面的URL。下面的代码获取用户的请求URI和查询字符串，并保存到请求对象中，然后将请求转发给登录页面。

HttpServletRequest hsr = (HttpServletRequest) request;

String uri = hsr.getRequestURI(); //获取客户端的URI

String strQuery = hsr.getQueryString();

**if**(**null** != strQuery){

uri += strQuery;

}

hsr.setAttribute("uri", uri);

hsr.getRequestDispatcher("logon.jsp").forward(request, response);

在登录页面中，只需要包含一个隐含输入域，它的值为用户先前的请求URL

<input type=*"hidden"* name=*"origin\_uri"* value=*"*${requestScope.uri}*"* />

当用户提交登录表单时，我们就得到了用户先前的请求URL，在验证通过后，可以将客户端重定向到先前访问的页面。

#### 替换请求和响应的数据

在Servlet规范中，同定义了4个包装类：ServletRequestWrapper、ServletResponseWrapper、HttpServletRequestWrapper和HttpServletResponseWrapper，这四个包装类分别实现了请求或响应的接口。

**public class ServletRequestWrapper implements ServletRequest**

**public class ServletResponseWrapper implements ServletResponse**

**public class HttpServletRequestWrapper implements HttpServletRequest**

**public class HttpServletResponseWrapper implements HttpServletResponse**

从表面上看，这4个类就好像是真正的请求和响应类，不过实质上是：他们在构造方法中接收真正的请求或响应对象，然后利用该对象的方法来完成自己需要实现的方法。包装类是装饰设计模式的运用，装饰模式给我们提供了一种不使用继承而修改或增加现有对象功能的方法。

有了包装类，要改变请求和响应信息就变得非常简单了，我们只需要编写一个包装类的子类，然后覆盖想要修改的方法就可以了。

例如，我们想要为所有的请求添加一个查询字符串，可以编写一个类，从HttpServletRequestWrapper类继承，并重写getQueryString()方法。

**public** **class** MyRequestWrapper **extends** HttpServletRequestWrapper {

**public** MyRequestWrapper(HttpServletRequest request) {

**super**(request);

// **TODO** 自动生成的构造函数存根

}

**public** String getQueryString(){

String str = "abc = 123";

//利用super变量调用父类的同名方法，得到原有的查询字符串

String str2 = **super**.getQueryString();

**if**(**null** != str2){

str2 += "&" + str;

**return** str2;

}**else**{

**return** str;

}

}

}

然后在过滤器类的doFilter()方法中构造MyRequestWrapper类的对象，将其作为参数传递给chain.doFilter()方法

**public** **class** SimpleFilter **implements** Filter {

**public** **void** destroy() {

// **TODO** Auto-generated method stub

}

**public** **void** doFilter(ServletRequest request, ServletResponse response, FilterChain chain) **throws** IOException, ServletException {

HttpServletRequest httpReq = (HttpServletRequest) request;

chain.doFilter(**new** MyRequestWrapper(httpReq), response);

}

**public** **void** init(FilterConfig fConfig) **throws** ServletException {

// **TODO** Auto-generated method stub

}

}

在目标资源的代码中，如果调用request.getQueryString()就会得到附加了字符串“abc = 123”的查询字符串。

我们需要得到响应的内容需要注意响应的内容是通过字符或字节输出流对象向客户端输出的，而字符和字节流对象则是通过HttpServletResponse.getWriter()和HttpServletResponse.getOutputStream()方法得到的。正常情况下，响应的内容将被容器直接发送到客户端，要想得到响应的内容，就要替换默认的输出流对象，并且新的输出流对象应该是内存输出流对象，也就是当我们调用该输出流对象的write()方法时，数据被写到内存的缓冲区。我们可以使用java.io包中的ByteArrayOutputStream类，让数据写到字节数组中，同时重写HttpServletResponse类的getWriter()和getOutputStream()方法，返回构建在ByteArrayOutputStrean之上的PringWriter对象和ServletOutputStream对象。

## 监听器Listener

Listener是Servlet规范的另一个高级特性。Listener用于监听Java Web程序中的事件，例如创建、修改、删除Session、request、context等，并触发响应的事件。利用Listener能够用很少的代码实现很绚丽的效果。

Listener也是Servlet规范2.3中加入的，对应设计模式中的Listener模式，事件发生的时候会自动触发该事件对应的Listener。Listener主要用于对Session、request、context等进行监控。

### Listener的使用

使用Listener需要实现相应的Listener接口。应该触发Listener事件的时候，Tomcat会自动调用Listener的方法。编程者不需要关注Tomcat怎么回调，只要实现这些方法就可以了。

**public** **class** SessionListenerTest **implements** HttpSessionListener {

**public** **void** sessionCreated(HttpSessionEvent event) {

//创建Session时被调用

HttpSession session = event.getSession();

System.***out***.println("新建一个session:" + session);

}

**public** **void** sessionDestroyed(HttpSessionEvent event) {

//销毁Session前被调用

HttpSession session = event.getSession();

System.***out***.println("销毁一个session:" + session);

}

}

该代码实现了HttpSessionListener接口。创建Session会使服务器调用sessionCreated()方法，销毁Session（包括session超时自动销毁）时服务器会调用sessionDestroyed()方法，都以HttpSessionEvent对象作为参数。HttpSessionEvent代表了当前的时间，能从中获取到Session。Listener需要在web.xml中配置才生效

<listener>

<listener-class>com.uc.SessionListenerTest.java</listener-class>

</listener>

<listener>标签一般配置在<servlet>标签的前面。启动Web服务器后，每当创建一个Session或者销毁一个Session控制台都会打印出相应日志。

一个完整的Listener包括Listener类、<listener>配置。一个web.xml中可以配置多个Listener。同一个类型的Listener也可以配置多个，触发的时候服务器会顺次执行各个Listener的相应方法。

### Listener的分类

Servlet2.5规范中共有8种Listener，分别用于监听Session、context、request等的创建与销毁、属性变化等。另有一个Listener能够监听存放在Session中的对象。共有6中Event。

#### 监听对象的创建与销毁

HttpSessionListener、ServletContextListener、ServletRequestListener分别用于监控Session、context、request的创建与销毁。触发时机分别为：

1. HttpSessionListener：监听Session的创建与销毁。创建Session时执行sessionCreated(HttpSessionEvent event)方法。超时或者执行session.invalidate()时执行sessionDestroyed(HttpSessionEvent event)方法。该Listener可用于收集在线者信息。
2. ServletContextListener：监听context的创建与销毁。context代表当前的Web应用程序。服务器启动或者热部署war包时执行contextInitialized(ServletContextEvent event)方法。服务器关闭时或者只关闭该Web时会执行contextDestroyed(ServletContextEvent event)方法。该Listener可用于启动时获取web.xml里配置的初始化参数。
3. ServletRequestListener：监听request的创建与销毁。用户每次请求request都会执行requestInitialized(ServletRequestEvent event)方法。request处理完毕自动销毁前执行requestDestroyed(ServletRequestEvent event)方法。注意如果一个HTML页面内含有多个图片，则请求一次HTML页面可能会触发多次request事件。

**public** **class** ListenerTest **implements** ServletContextListener, HttpSessionListener, ServletRequestListener {

**public** **void** sessionCreated(HttpSessionEvent event) {

// 创建Session时被调用

}

**public** **void** sessionDestroyed(HttpSessionEvent event) {

// 销毁Session前被调用

}

**public** **void** requestInitialized(ServletRequestEvent event) {

// 创建request时被调用

}

**public** **void** requestDestroyed(ServletRequestEvent event) {

// 销毁request前被调用

}

**public** **void** contextInitialized(ServletContextEvent event) {

// 创建context时被调用

}

**public** **void** contextDestroyed(ServletContextEvent event) {

// 销毁context前被调用

}

}

#### 监听对象的属性变化

另一类Listener用于监听Session、context、request的属性变化，包括HttpSessionAttributeListener、ServletContextAttributeListener、ServletRequestAttributeListener。当向被监听对象中添加、更新、移除属性时，会分别指向xxxAdded()、xxxReplaced()、xxxRemoved()方法，xxx分别代表Session、context、request。

**public** **class** SessionAttributeListenerTest **implements** HttpSessionAttributeListener {

**public** **void** attributeAdded(HttpSessionBindingEvent arg0) {

// 添加属性时被调用

}

**public** **void** attributeRemoved(HttpSessionBindingEvent arg0) {

// 删除属性前被调用

}

**public** **void** attributeReplaced(HttpSessionBindingEvent arg0) {

// 修改属性时被调用

}

}

#### 监听Session内的对象

除了上面的6种Listener，还有两种Listener用于监控Session内的对象，分别是HttpSessionBindingListener与HttpSessionActivationListener。它们触发时机分别为：

1. HttpSessionBindingListener：当对象呗放到Session里是执行valueBound(HttpSessionBindingEvent event)方法。当对象被冲Session里移除时执行valueUnbound(HttpSessionBindingEvent event)方法。对象必须实现该Listener接口。
2. HttpSessionActivationListener：服务器关闭时，会将Session里的内容保存在硬盘上，这个过程叫做钝化。服务器重新启动时，会将Session内容从硬盘上重新加载。当Session里的对象被钝化时会执行sessionWillPassivate(HttpSessionEvent event)方法，当对象被重新加载时执行sessionDidActivate(HttpSessionEvent event)。对象必须实现该Listener接口。

这两种Listener常用于实现Session内的对象对Session的监听。

//同时实现多个接口

public class PersonInfo implements HttpSessionActivationListener, HttpSessionBindingListener,Serializable {

private static final long *serialVersionUID* = -1487919671274221919L;

private String name;

public void valueBound(HttpSessionBindingEvent event) {

//被放进Session前被调用

}

public void valueUnbound(HttpSessionBindingEvent event) {

//从session中移除后被调用

}

public void sessionDidActivate(HttpSessionEvent event) {

//从硬盘恢复后被调用

}

public void sessionWillPassivate(HttpSessionEvent event) {

// 即将被钝化到硬盘时被调用

}

}

与上面6种Listener不同，这两个Listener监听的是Session中的对象而非Session等，因此不需要在web.xml中声明。

# JSP技术

## JSP技术概述

### 什么是JSP

JSP（全称Java Server Pages）是由Sun公司倡导和许多公司参与共同创建的一种使软件开发者可以响应客户端请求，而动态生成HTML、XML或其他格式文档的Web网页的技术标准。

JSP技术是以Java语言作为脚本语言的，JSP网页为整个服务器端的Java库单元提供了一个接口来服务于HTTP的应用程序。JSP文件后缀名为（\*.jsp）。

JSP是一种动态网页开发技术。它使用JSP标签在HTML网页中插入Java代码。标签通常以<%开头、以%>结束。

JSP是一种Java Servlet，主要用于实现Java Web应用程序的用户界面部分。网页开发者们通过结合HTML代码、XHTML代码、XML元素以及嵌入JSP操作和命令来编写JSP。

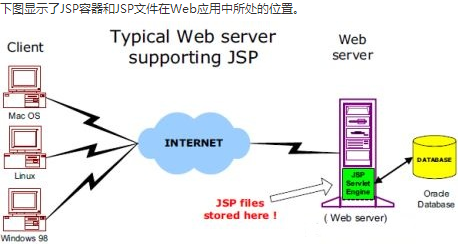
JSP通过网页表单获取用户输入数据、访问数据库及其他数据源，然后动态地创建网页。

### JSP技术原理

#### JSP结构

网络服务器需要一个JSP引擎，也就是一个容器来处理JSP页面。容器负责截获对JSP页面的请求。

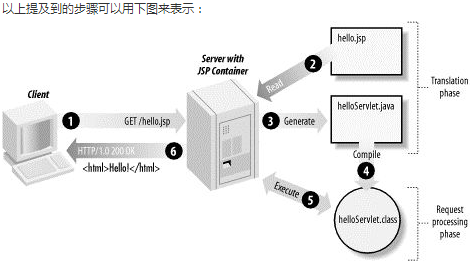
JSP容器与web服务器协同合作，为JSP的正常运行提供必要的运行环境和其他服务，并且能够正确识别专属于JSP页面的特殊元素。



#### JSP处理

以下步骤表明了Web服务器是如何使用JSP来创建网页的：

1. 就像其他普通的网页一样，您的浏览器发送一个HTTP请求给服务器
2. Web服务器识别出这是一个对JSP网页的请求，并且将该请求传递给JSP引擎。通过使用URL或者.jsp文件来完成。
3. JSP引擎从磁盘中载入JSP文件，然后将他们转化为servlet。这种转化只是简单地将所有模板文本改用println()语句，并且将所有的JSP元素转化成Java代码
4. JSP引擎将servlet编译成可执行类，并且将原始请求传递给servlet引擎。
5. Web服务器的某组件将会调用servlet引擎，然后载入并执行servlet类。在执行过程中，servlet产生HTML格式的输出并将其嵌于HTTPresponse中交给Web服务器。
6. Web服务器以静态HTML网页的形式将HTTPresponse返回到您的浏览器中。
7. 最终，Web浏览器处理HTTPresponse中动态产生的HTML网页，就好像在处理静态网页一样



一般情况下，JSP引擎会检查JSP文件对应的servlet是否已经存在，并检查JSP文件的修改日期是否早于servlet。如果JSP文件的修改日期早于对应的servlet，那么容器就可以确定JSP文件没有被修改过并且servlet有效。这使得整个流程与其他脚本语言相比要高效快捷一些。

总的来说，JSP网页就是用另一种方式来编写servlet而不用称为Java编程高手。除了解释阶段外，JSP网页几乎可以被当成一个普通的servlet来对待。

### JSP的生命周期

理解JSP底层功能的关键就是去理解它们所遵守的生命周期。JSP生命周期就是从创建到销毁的整个过程，类似于servlet生命周期，区别在于JSP生命周期还包括将JSP文件编译成servlet。

在JSP2.0规范中定义，JSP页面转换后的Servlet类必须实现javax.servlet.jsp.JspPage接口（与Servlet类似，Servlet类必须实现javax.servlet.Servlet接口），该接口继承自javax.servlet.Servlet接口。除了继承的方法外，JsPage接口还定义了下面两个方法：

* public void jspInit()

这个方法在JSP页面初始化时被调用，它类似于Servlet中的init()方法。页面编写者可以在JSP的声明元素中覆盖这个方法，以提供任何的初始化工作。

* public void jspDestroy()

在JSP页面将要被销毁时调用这个方法，它类似于Servlet中的destroy()方法。页面编写者可以在JSP的声明元素中覆盖这个方法，以提供任何的JSP清除工作。

因为绝大多数情况下，JSP页面都是使用HTTP协议，所以JSP页面转换后的Servlet类实际上必须实现javax.servlet.jsp.HttpJspPage接口，该接口继承自JsPage接口。除了继承的方法外，HttpJspPage接口还定义了一个方法：

* public void \_jspService(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response)throws javax.servlet.ServletException, java.io.IOException

这个方法对应于JSP页面的主体（body）部分，它类似于Servlet中的service()方法。这个方法由JSP容器自动定义，页面编写者不应当提供方法的实现。

## JSP语法详解

JSP文件主要由模板元素、指令元素、动作元素、脚本元素、声明、表达式、Scriptlets和JSP内建对象组成。

### JSP的构成

一个JSP文件由注释、模板元素、脚本元素、指令元素、动作元素5类元素构成。

#### 模板元素

模板元素是指JSP的静态HTML或者XML内容。它对JSP的显示是非常必要的，它遵循HTML或者XML语法规则。

<!DOCTYPE html>

<html>

<head>

<meta charset=*"UTF-8"*>

<title>Insert title here</title>

</head>

<body>

</body>

</html>

这些模板元素可以说是网页的框架，它影响页面的结构和美观程度。JSP编译时，它将把这些模板元素编译到Servlet里。当客户请求此JSP时，它会把这些模板元素一字不变地发送到客户端

比如：<html></head>在编译成servlet时会使用一下的代码替换：



#### 注释

JSP中的注释有多种情况，有JSP自带的注释规范，也有HTML/XML的注释规范

1. HTML/XML注释

<!-- 这是一段注释内容 -->

<!-- 当前时间为：<%=(**new** java.util.Date()).toLocaleString() %> -->

在客户端的HTML源代码中显示为：<!-- 当前时间为：2017-10-27 16:57:37 -->

1. 隐藏注释

隐藏注释写在JSP程序中，但不发给客户。

语法格式：

<%--此处为隐藏注释信息 --%>

用隐藏注释标记的字符会在JSP编译时被忽略。它不会显示在客户的浏览器中，也不会出现在源代码中。

1. Scriptlets中的注释

由于Scriptlets包含的是Java代码，所以Java中的注释规则在Scriptlets中也适用。常用的Java注释使用“//”表示单行注释，使用“/\*\* \*/”表示多行注释。

#### 指令元素(directive)

指令用于从JSP发送一个信息到容器上。它用来设置全局变量，声明类、要实现的方法和输出内容的类型等。它们并不向客户产生任何输出，所有的指令都在JSP整个文件范围内有效。指令元素为翻译阶段提供了全局信息。

语法格式：

<%@ include file=*"header.jsp"* %>

在起始符号<%@之后和结束符号%>之前，可以加空格，也可以不加，但是要注意的是，在起始符号中的<和%之间、%和@之间，以及技术符号中的%和>之间不能有任何的空格。指令元素有三种，分别是：页面(page)指令、include指令、taglib指令

1. **页面指令(page)**

页面指令用于定义JSP页面的相关属性，包括设置MIME类型，定义需要导入的包、错误页的指定等。一个JSP页面可以包含多个页面指令，在编译过程中，所有的页面指令都被抽出来同时应用到一个页面里。除了import外，其他的页面指定定义的属性/值只能出现一次。

语法格式：

<%@ page 属性=*"属性值"*;..... %>

JSP页面指令的属性：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 属性 | 描述 | | 默认值 | 例子 |
| **language** | | 指定当前页面使用的脚本语言，目前只能是“java” | java | language=”java” |
| **extends** | | 该属性用于指定JSP页面转换后的Servlet类从哪一个类继承，属性的值是完整的限定类名。通常不需要使用这个属性，JSP容器会提供转换后的Servlet类的父类。使用该属性时要格外小心，因为这可能会限制JSP容器为提升性能所作出的努力。 | 完整的限定类名 |  |
| **import** | | 导入要使用的Java类包。只是用“,”隔开包或者类列表。同样它总是被放在JSP文件的顶部 | 默认忽略（即不引入其他类或者包） | import=”java.io.\*,java.util.Hashtable” |
| **session** | | 该属性用于指定在JSP页面中是否可以使用Session对象，默认值是true。 | true/false | session=”true” |
| **buffer** | | 该属性用于设置out对象使用的缓冲区大小，默认为8KB，建议以8的倍数增加。如果设置为none，将不使用缓冲区。 | 不小于8KB，根据不同的服务器可设置 | buffer=”64kb” |
| **autoFlush** | | 指定当缓冲区满时，自动将缓冲区中的数据输出到客户端。如果设置成fale，当缓冲区满时会抛出异常。默认值为true | true/false | autoFlush=”true” |
| **info** | | 关于JSP页面的信息，定义一个字符串，可以使用servlet.getServletInfo()获得 | 默认忽略 | info=”一个测试页面” |
| **isErrorPage** | | 用于当前JSP页面设置成错误处理页面，错误处理页面用于处理另一个JSP页面的错误，也就是异常处理。只有设置为true时，才可以调用exception对象输出错误信息。 | false/true | isErrorPage=”fasle” |
| **errorPage** | | 定义此页面出现异常时调用另一个页面（错误处理页面） | 默认忽略 | errorPage=”error/error.jsp” |
| **isThreadSafe** | | 用来设置JSP文件是否能多线程使用。如果设置为true，那么一个JSP能够同时处理多个用户的请求；相反，如果设置为false。一个JSP只能一次处理一个请求。默认值为true | true | isThreadSafe=”true” |
| **contentType** | | 定义JSP字符编码和页面响应的MIME类型。该属性设置的是服务器响应给浏览器的编码规则。 | TYPE=text/html  CHARSET=iso8859-1 | contentType=”text/html;charset=gb2312” |
| **pageEncoding** | | JSP页面的字符编码。 | pageEncoding=  ”ISO-8859-1” | pageEncoding=”gb2312” |
| **isELlgnored** | | 制定EL（表达式语言）是否被忽略。如果为true，则容器忽略“${}”表达式的计算 | 默认值由web.xml描述文件的版本确定，Servlet2.3以前的版本将忽略 | isELlgnored=”true” |

这些页面指令的属性可以单独使用也可以同时使用。

<%@ page language=*"java"* import=*"java.util.\*"* session=*"true"*

buffer=*"8kb"* autoFlush=*"true"* info=*"a test directive jsp page"*

errorPage=*"error.jsp"* isErrorPage=*"false"* contentType=*"text/html; charset=UTF-8"*%>

<%@ page errorPage=*"error.jsp"* %>

<%@ page isELIgnored=*"false"* %>

1. **include指令**

JSP可以通过include指令来包含其他文件。被包含的文件可以是JSP文件、HTML文件或文本文件。这个包含的过程是静态的，称为静态包含。该包含融合的过程发生在将JSP代码转换成servlet代码之前进行。也就是转换后的servlet代码中是包含了被包含页面的内容（也就是使用了include指令的JSP页面在转换时，JSP容器会将被包含页面中的内容提取出来插入到包含的页面中再进行Servlet转换。最终依然是生成的一个Servlet Java文件）。

语法格式：

<%@ include file=*"被包含的文件路径"* %>

需要注意的是，一经编译，内容不可变，如果要改变被包含文件的内容，必须要重新编译JSP文件。

<%@ page language=*"java"* contentType=*"text/html; charset=utf-8"*

pageEncoding=*"utf-8"*%>

<%@ include file=*"head.jsp"* %>

<%@ include file=*"body.html"* %>

<%@ include file=*"footer.jsp"* %>

注意：在被包含的文件中最好不要使用<html>、</html>、<body>、</body>等标签，因为这会影响到原JSP文件中同样的标签，有时会导致错误。另外也不要定义相同的变量和方法，这样会导致不可预料的错误。

1. **taglib指令**

这个指令允许页面使用者自定义标签。首先用户要开发标签库，为标签库编写.tld配置文件，然后在JSP页面里使用自定义标签。这样，容器使用这个标签库确定在遇到定制标签时要怎么做。由于使用了标签，增加了代码的重用程度，比如可以把一些需要迭代显示的内容做成一个标签，在每次需要迭代显示时，就使用这个标签。使用标签页使页面易于维护。

语法格式：

<%@ taglib uri=*"uri"* prefix=*"prefixOfTag"* %>

**taglib指令有三个属性：**

1. uri：该属性唯一地标识和前缀（prefix）相关的标签库描述符，可是绝对或者相对的URI。这个URI被用于定位标签库描述的位置，
2. prefix:属性指定标签库的前缀，这些前缀不可以是：jsp、jspx、java、javax、sun、servlet、sunw。
3. tagdir:该属性指示前缀（prefix）将被用于标识安装在/WEB-INF/tags目录或其子目录下标签文件。一个隐含的标签库描述被使用。下面三种情况将发生转换错误：

属性的值不是以/WEB-INF/tags开始

属性的值没有指向一个已经存在的目录

该属性与uri属性一起使用。

<!DOCTYPE html>

<html>

<head>

<%@ page language=*"java"* contentType=*"text/html; charset=utf-8"*

pageEncoding=*"utf-8"*%>

<%@ taglib prefix=*"c"* uri=*"http://java.sun.com/jsp/jstl/core"* %>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=utf-8"*>

<title>???????</title>

</head>

<body>

<c:if test=*"*${sessionScope.test == 'hellking'}*"*>

${sessionScope.test}<br/>

</c:if>

</body>

</html>

运行此代码需要JSTL支持，需要将JSTL相关.jar包复制到\WEB-INF\lib目录下。

#### 脚本元素(scripting)

脚本元素包括三个部分：声明、脚本段和表达式。JSP2.0增加了EL表达式，作为脚本元素的另一个选择。声明脚本元素用于声明在其他脚本元素中可以使用的变量和方法，脚本段是一段Java代码，用于描述在堆请求的响应中要执行的动作，表达式脚本元素是Java语言中完整的表达式，在响应请求时被计算，计算的结果将被转换为字符串，插入到输出流中。

脚本元素主要包括：声明（declaration）、表达式（expression）、Scriptlets

1. **声明(declaration)**

就是在JSP程序中声明合法的变量和方法。位于“**<%!**”与“**%>**”之间的Java代码。在此标记中可以定义全局变量、方法、类。在整个文档中都可以使用。声明必须是完整的声明语句，遵照Java语言的语法。声明不会在当前的输出流中产生任何的输出。

<%!

**public** String getDate(){

**return** **new** java.util.Date().toLocaleString();

}

**private** **int** count = 10;

%>

虽然在<%!...%>中可以定义类或方法，但是尽量不要在JSP中定义类或方法。当JSP中需要类或者方法时，往往会通过JavaBean的形式调用。还有就是在声明变量的时候要注意，不要忘记变量名后面的分号，声明只在当前的JSP页面中有效。

1. **脚本段(scriptlets)**

脚本段是JSP中页面处理请求时执行的Java代码，位于“**<%**”和“**%>**”之间，在此中定义的变量为局部变量。它们是合法的Java代码。它可以产生输出，并将输出发送到客户的输出流里，也可以是一些流程控制语句。当然它中间还可以包含一些合法的Java注释。

<%

**int** i;

String color1 = "99ccff";

String color2 = "88cc33";

**for**(i = 1; i < count; i++){

String color = "";

**if**(i % 2 == 0){

color = color1;

}**else**{

color = color2;

}

out.print("<tr bgcolor=' + color + '><td>---</td></tr>");

}

%>

在JSP容器转换JSP页面为Servlet类时，页面中的代码段会按照出现的次序，一次被转换为\_jspService()方法中的代码，在脚本段中声明的变量，将成为\_jspService()方法中的局部变量。因此，脚本段中的变量是线程安全的。

1. **表达式(expression)**

是指位于“**<%=**”和“**%>**”之间的代码。主要功能是输出一个变量或一个具体的内容，所得的结果转换成字符串并与模板数据组合在一起。表达式在页面的位置，也就是该表达式计算结果所处的位置。（表达式语句结尾处没有分号）。

<%=getDate() %>

如果表达式的任何部分是一个对象，就可以使用toString()进行转换。表达式必须有一个返回值或本身就是一个对象。实际上表达式被转换成out.println()方法中的内容。

out.println()和<%=%>使用那个输出好：

<%

**int** rows = 10;

**int** cols = 10;

**for**(**int** x = 0; x < rows; x++){

%>

<tr/>

<%

**for**(**int** y = 0; y < cols; y++){

%>

<td><%=(x \* y) %></td>

<%

}

%>

</tr>

<%

}

%>

<%

**int** rows = 10;

**int** cols = 10;

out.print("<table border=\"1\" width=\"100%\">");

**for**(**int** x = 0; x < rows; x++){

out.print("<tr>");

**for**(**int** y = 0; y < cols; y++){

out.print("<td>" + (x \* y) + "</td>");

}

out.print("</tr>");

}

out.print("</table>");

%>

在JSP中使用<%=%>能将HTML标记与Java代码完全分离开，做到低耦合。

#### 动作元素(action)

与JSP指令元素不同的是，JSP动作元素在请求处理阶段起作用。JSP动作元素是用XML语法写成的。利用JSP动作可以动态地插入文件、重用JavaBean组件、把用户重定向到另外的页面、为Java插件生成HTML代码。动作元素可以有属性、可选的内容，与开始标签匹配的结束标签。动作元素也看可以是一个空标签，可以有属性。与XML和XHTML一样，JSP的标签页区分大小写。

JSP2.0规范定义了一些标准的动作，标准动作是一些标签，它们影响JSP运行时的行为和对客户端请求的响应，这些动作由JSP容器来实现。

从效果上来说，一个标准动作是写入到JSP页面中的一个标签。在页面被转换为Servlet期间，当JSP容器遇到这个标签，就用预定义的对应该标签的Java代码来代替它。

动作元素只有一种语法，它符合XML标准：

<jsp:action\_name attribute=*"value"* />

动作元素基本上都是预定义的函数，JSP规范定义了一系列的标准动作，它用JSP作为前缀，可用的标准动作元素如下：

|  |  |
| --- | --- |
| 语法 | 描述 |
| jsp:include | 在页面被请求的时候引入一个文件。 |
| jsp:useBean | 寻找或者实例化一个JavaBean。 |
| jsp:setProperty | 设置JavaBean的属性。 |
| jsp:getProperty | 输出某个JavaBean的属性。 |
| jsp:forward | 把请求转到一个新的页面。 |
| jsp:plugin | 根据浏览器类型为Java插件生成OBJECT或EMBED标记。 |
| jsp:element | 定义动态XML元素 |
| jsp:attribute | 设置动态定义的XML元素属性。 |
| jsp:body | 设置动态定义的XML元素内容。 |
| jsp:text | 在JSP页面和文档中使用写入文本的模板 |
| jsp:param | 以“名-值”对的形式为其他标签提供附加信息。它与jsp:include、jsp:forward、jsp:plugin一起使用 |
| jsp:fallback |  |
| jsp:params |  |
| jsp:invoke |  |
| jsp:doBody |  |
| jsp:output |  |
| jsp:root |  |
| jsp:declaration |  |
| jsp:scriptlet |  |
| jsp:expression |  |

常用的属性

所有的动作元素都有两个属性：id属性和scope属性

id属性是动作元素的唯一标识，可以在JSP页面中引用。动作元素创建的id值可以通过PageContext来调用。

scope属性用来识别动作元素的生命周期。id属性和scope属性有直接关系，scope属性定义了相关联id对象的寿命。scope属性有四个可能的值：(a)page、(b)request、(c)session、(d)application

1. **<jsp:include>动作元素**

该动作元素用于在当前页面中包含静态和动态的资源，一旦被包含的页面执行完毕，请求处理将再调用页面中继续进行。被包含的页面不能改变响应的代码或设置报头，这防止了对类似setCookie()这样的方法的调用，任何对这些方法的调用将被忽略。

语法格式：

不传递参数：

<jsp:include page=*"相对URL路径"* flush=*"true/false"* />

传递参数：

<jsp:include page=*"相对RUL路径"*>

<jsp:param value=*"value"* name=*"name"*/>

.....可以传递多个值

</jsp:include>

该动作元素与include指令不同的在于，include指令是在JSP文件被转换成Servlet的之前将被包含的文件内容提出出来插入到包含文件代码中，然后再统一进行编译。而<jsp:include />动作元素是在需要执行文件的时候才会引入处理后的其他文件（HTML文件按原文件引入，JSP文件按编译后的.class文件引入）。编译生成的servlet文件依然是相互独立的文件，代码不会融合在一个servlet中。

属性列表：

|  |  |
| --- | --- |
| 属性 | 描述 |
| page | 包含在页面中的文件的相对URL地址 |
| flush | 布尔属性，定义在包含资源前是否刷新缓冲区 |

实例

以下定义了两个文件date.jsp和main.jsp：

date.jsp

<%@ page language=*"java"* contentType=*"text/html; charset=utf-8"*

pageEncoding=*"utf-8"* %>

<p>

今天的日期是：<%=(**new** java.util.Date()).toLocaleString() %>

</p>

main.jsp

<!DOCTYPE html>

<html>

<head>

<%@ page language=*"java"* contentType=*"text/html; charset=utf-8"*

pageEncoding=*"utf-8"*%>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=utf-8"*>

<title></title>

</head>

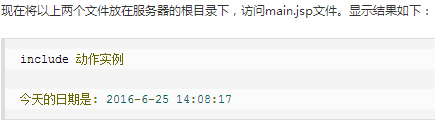
<body>

<h2>include动作实例</h2>

<jsp:include page=*"date.jsp"* flush=*"true"* />

</body>

</html>



1. **<jsp:useBean>动作元素**

该动作元素用于实例化JavaBean，或者定位一个已经存在的JavaBean实例，并把实例的引用赋值给一个变量。

语法格式：

<jsp:useBean id=*"name"* class=*"package.class"*/>

在类载入后，我们既可以通过 jsp:setProperty 和 jsp:getProperty 动作来修改和检索bean的属性。

属性列表：

|  |  |
| --- | --- |
| 属性 | 描述 |
| id | 用于标识JavaBean实例的名字，同时，该名字也是声明的脚本变量的名字，并被初始化为JavaBean实例的引用。要注意的是，指定的名字是区分大小写的，并遵循Java语言变量命名的规则 |
| class | 用于指定JavaBean对象的完整的限定类名 |
| scope | 指定Bean在那种上下文内可用，可以取下面的四个值之一：  page：默认值，表示该Bean只在当前页面内可用（保存在当前页面的PageContext内）。  request：表示该Bean在当前的客户请求内有效（保存在ServletRequest对象内）。  session：表示该Bean对当前HttpSession内的所有页面都有效。  application：表示该Bean对所有具有相同ServletContext的页面都有效。 |
| type | 指定定义的脚本变量的类型。这个类型可以是Bean类本身，它的父类，或者有Bean类实现的接口。该属性默认的值和class属性的值一样 |
| beanName | 指定Bean的名字。该名字被提供给java.beans.Beans类的instantiate()方法，来实例化一个JavaBean。 |

1. **<jsp:setProperty>动作元素**

该动作元素与<jsp:useBean>一起使用，用来设置已经实例化的Bean对象的属性和索引属性，<jsp:setProperty>标签使用Bean给定的setXXX()方法，在Bean中设置一个或多个属性值。在JSP中，经常使用<jsp:setProperty>动作元素将客户端提交的数据保存到JavaBean的属性中。

利用<jsp:setProperty>设置属性值有两种用法。

在<jsp:useBean>元素的外面（后面）使用：

<jsp:useBean id=*"name"* class=*"package.class"*/>

.....

<jsp:setProperty property=*"someProperty"* name=*"name"* .../>

此时，不管<jsp:useBean/>是找到了一个现有的Bean，还是新创建了一个Bean实例，<jsp:serProperty/>都会执行。

把<jsp:setProperty/>放入<jsp:useBean/>元素的内部：

<jsp:useBean id=*"myName"* class=*"package.class"*>

.....

<jsp:setProperty name=*"myName"* property=*"someProperty"*../>

</jsp:useBean>

此时，<jsp:setProperty/>只有在新建Bean实例时才会执行，如果是使用现有实例则不执行<jsp:setProperty/>

属性列表：

|  |  |
| --- | --- |
| 属性 | 描述 |
| name | Bean实例的名字，它必须是已经在<jsp:useBean>元素中通过id属性定义的名字。Bean的实例必须包含科协（具有setXXX()方法）的属性。 |
| property | 被设置的属性的名字。如果property属性的值是“\*”，标签就会在请求对象中查找所有的请求参数，看是否 有参数的名字和Bean属性的名字相同，如果找到匹配的参数和属性，就会按照正确的类型（自动进行类型转换）将参数的值设置为属性的值。如果一个参数的值为空，对应的属性的值不会被修改。 |
| param | 指定请求对象中参数的名字。在社会Bean的属性时，如果请求参数的名字和Bean属性的名字不相同，可以用param来指定参数的名字。如果没有使用param，那么就认为请求参数的名字和Bean属性的名字相同。在<jsp:setProperty>元素中，不能同时出现param和value属性 |
| value | 指定要赋值给Bean属性的值。可以用一个请求时属性表法师作为value属性的值。 |

1. **<jsp:getProperty>动作元素**

该属性用于提取指定Bean属性的值，转换成字符串，然后输出。如果属性值是一个对象，将调用该对象的toString()方法。

语法格式：

<jsp:useBean id=*"myName"* .../>

.....

<jsp:getProperty property=*"someProperty"* name=*"myName"*.../>

属性列表：

|  |  |
| --- | --- |
| 属性 | 描述 |
| name | Bean实例的名字，从这个实例中可以得到属性。（<jsp:useBean>标签中id属性的值） |
| property | 要得到的属性的名字。Bean的实例必须包含可读（具有getXXX方法）的属性 |

实例：

1. 编写一个User类名的JavaBean用于存放客户端提交的数据，其中对每个属性实现setXXX和getXXX方法
2. 编写一个reg.jsp页面用于将客户端提交的数据保存到User对象中。

<jsp:useBean id=*"user"* scope=*"session"* class=*"com.uc.pojo.User"*>

<jsp:setProperty name=*"user"* property=*"\*"* />

<jsp:setProperty name=*"user"* property=*"email"* param=*"mail"* />

</jsp:useBean>

1. 编写userinfo.jsp页面用于从User对象中获取用户提交的信息。

<jsp:useBean id=*"user"* scope=*"session"* class=*"com.uc.pojo.User"* />

你的姓名：<jsp:getProperty property=*"name"* name=*"user"*/>

你的性别：<%

**int** sex = user.getSex();

**if**(1 == sex){

out.print("男");

}**else** **if**(0 == sex){

out.print("女");

}

%>

你的学历：<jsp:getProperty property=*"education"* name=*"user"*/>

你的E-mail:<jsp:getProperty property=*"email"* name=*"user"*/>

1. **<jsp:forward>动作元素**

该动作元素用于在运行是将当前的请求转发给另外的JSP、Servlet或者静态资源文件。每当遇到此操作时，就停止执行当前的JSP，转而执行被转发的资源。请求被转向到的资源必须位于同JSP发送请求相同的上下文环境中。

语法格式：

不带参数的：

<jsp:forward page=*"相对URL地址"* />

带参数的：

<jsp:forward page=*"相对URL地址"* >

<jsp:param value=*"value"* name=*"name"*/>

...........多个参数

</jsp:forward>

属性列表：

|  |  |
| --- | --- |
| 属性 | 描述 |
| page | page属性包含的是一个相对URL。page的值可以直接给出，也可以在请求的时候动态计算，可以是一个JSP页面或者一个Servlet |

例如：

<%String target = "/login.jso" %>

<jsp:forward page=*"*<%=target %>*"* />

或者：

<jsp:forward page=*"/login.jsp"*>

<jsp:param value=*"hellking"* name=*"name"*/>

<jsp:param value=*"xxxxxx"* name=*"pwd"*/>

</jsp:forward>

跳转后可以使用JSP的内置对象request的getParameter()方法获得参数值。

参数值为：<%=request.getParameter("name") %>

参数值为：<%=request.getParameter("pwd") %>

1. **<jsp:plugin>、<jsp:params>和<jsp:fallback>动作元素**

该动作元素用于根据浏览器的类型，插入通过Java插件运行Java Applet所必需的<OBJECT>和<EMBED>元素。如果需要的插件不寻在，它会下载插件，然后执行Java组件。Java组件可以是一个applet或一个JavaBean。<jsp:plugin>标签根据客户端浏览器的类型被替换为<object>或<embed>标签。在<jsp:plugin>元素的内容中可以使用另外两个标签：<jsp:params>和<jsp:fallback>。

<jsp:params>是<jsp:plugin>动作的一部分，并且只能在<jsp:plugin>动作中使用。<jsp:params>动作包含一个或多个<jsp:param>动作，用于向Applet或JavaBean提供参数。

<jsp:fallback>是<jsp:plugin>动作的一部分，并且只能在<jsp:plugin>动作中使用。主要用于指定在Java插件不能启动时显示给用户的一段文件。如果插件能够启动，但是Applet或JavaBean没有发现或不能启动，那么浏览器会有一个出错信息提示。

实例：

<jsp:plugin code=*"MyApplet.class"* codebase=*"dirname"* type=*"applet"*

width=*"60"* height=*"80"*>

<jsp:params>

<jsp:param value=*"red"* name=*"fontcolor"*/>

<jsp:param value=*"black"* name=*"background"*/>

</jsp:params>

<jsp:fallback>

Unable to initialize Java Plugin

</jsp:fallback>

</jsp:plugin>

属性：

|  |  |
| --- | --- |
| 属性 | 描述 |
| type | 指定要执行的组件的类型，是JavaBean还是Applet。----(bean/applet) |
| code | 指定要执行的组件的完整的类名，以.class结尾。该名字要么相对于codebase，要么相对于当前页面 |
| codebase | 指定要执行的Java类所在的目录 |
| align | 指定组件对象的方法。  left---------------把组件放在网页左边，后面的文本会移至applet的右边  right--------------把组件放在网页右边，后面的文本会移至applet的左边  bottom-----------把组件的底部与当前行文本底部对齐  top---------------把组件的顶部与当前行顶部对齐  texttop-----------把组件的顶部与当前行文本顶部对齐  middle------------把组件的中部与当前行基线对齐  absmiddle--------把组件的中部与当前行中部对齐  baseline----------把组件的底部与当前行基线对齐  absbottom--------把组件的底部与当前行底部对齐 |
| archive | 指定以都好分割的Java归档文件列表。归档文件中可以包含组件要使用的类或需要的其他资源 |
| height和width | 指定组件的高度和宽度，以像素为单位 |
| hspace和vspace | 指定组件左右、上下留出的空间，以像素为单位 |
| jreversion | 指定组件运行时需要的JRE版本，默认值为1.2 |
| name | 指定组件的名字。 |
| nspluginurl | 指定对Netscape Navigator，可以下载JRE插件的URL。默认情况由实现定义 |
| iepluginurl | 指定对于Internet Explorer，可以下载JRE插件的URL。默认情况由实现定义 |

1. **<jsp:element>动作元素**

这个动作用于动态定义一个XML元素的标签。在<jsp:element>中，可以包含<jsp:attributee>和<jsp:body>。

语法格式：

<jsp:element name=*"name"*>

optional body

</jsp:element>

或者：

<jsp:element name=*"name"*>

jsp:attribute\*

jsp:body?

</jsp:element>

属性：

|  |  |
| --- | --- |
| 属性 | 描述 |
| name | 该属性用于指定动态产生的元素的名称 |

实例：

<jsp:element name=*"football"* />

执行后将产生一个空元素：

<football />

<jsp:element name=*"employee"* >

<jsp:attribute name=*"name"*>张三</jsp:attribute>

<jsp:body>张三是销售部的经理</jsp:body>

</jsp:element>

执行后将产生一个包含属性和内容的元素：

<employee name=*"张三"*>张三是销售部的经理</employee>

1. **<jsp:attribute>动作元素**

<jsp:attribute>动作主要由两个用途：一是用于在XML元素的内容中定义一个动作属性的值。二是在<jsp:element>动作中使用，指定输出元素的属性。

语法格式：

<jsp:attribute name=*"name"* trim=*"true/false"*>

optional body

</jsp:attribute>

属性：

|  |  |
| --- | --- |
| 属性 | 描述 |
| name | 用于指定元素属性的名称 |
| trim | 用于指定在<jsp:attribute>元素的内容前后出现的空白（包括空格、回车、换行、制表符）是否被JSP容器忽略。默认是true |

实例：

<jsp:forward>

<jsp:attribute name=*"page"*>reg.jsp</jsp:attribute>

</jsp:forward>

其作用于下列的一样：

<jsp:forward page=*"reg.jsp"* />

1. **<jsp:body>动作元素**

该动作元素用于定义元素的内容，与<jsp:element>动作元素一起使用。

语法格式：

<jsp:body>

optional body

</jsp:body>

1. **<jsp:text>动作元素**

该动作元素用于封装模板数据，它可以在模板数据允许出现的任何地方使用。<jsp:text>元素的作用和在JSP页面中直接书写模板数据一样。没有属性。

语法格式：

<jsp:text>template data</jsp:text>

在<jsp:text>动作中不能嵌套其他的动作和脚本元素，但是可以有EL表达式。

<jsp:text>Hell World</jsp:text>

执行后将产生输出：

Hello World

1. **<jsp:output>动作元素**

<jsp:output>只能在JSP文档和以XML语法编写的标签文件中使用，主要用于输出XML声明和文档类型声明。所谓JSP文档，是指使用XML语法编写的JSP页面。这意味着JSP文档是格式良好的、结构化的文档

1. **<jsp:param/>动作元素**

该动作元素用来以“名-值”对的形式为其他标签提供附加信息。它与<jsp:include/>、<jsp:forward/>和<jsp:plugin/>一起使用

语法格式：

<jsp:param value=*"value"* name=*"name"*/>

属性列表

|  |  |
| --- | --- |
| **属性** | **描述** |
| name | 为属性相关联的关键字 |
| value | 属性的值 |

1. 通过<jsp:include>动作元素搭配<jsp:param>动作元素，可以实现在包含文件的同时向被包含文件传递参数。

如何使用<jsp:param>动作元素传递一个参数

<body>

<p>接收到的参数</p>

<%

String name = request.getParameter("name");

String age = request.getParameter("age");

String sex = request.getParameter("sex");

%>

<%="name参数值为：" + name %>

<%="age参数值为：" + age %>

<%="sex参数值为：" + sex %>

</body>

ParamInclude2.jsp contentDemo2.jsp：

<body>

使用include动作元素包含一个包含JSP文件，并传递多个参数

<jsp:include page=*"contentDemo2.jsp"*>

<jsp:param value=*"Jame"* name=*"name"*/>

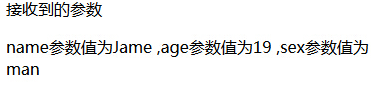
<jsp:param value=*"19"* name=*"name"*/>

<jsp:param value=*"man"* name=*"sex"*/>

</jsp:include>

</body>

运行结果：



1. 通过<jsp:forward>动作元素搭配<jsp:param>动作元素，可以实现在跳转页面的同时向跳转页面传递参数。

ParamForward2.jsp ForwardedDemo2.jsp

<body>

<h2>跳转到的页面</h2>

<p>接收到的参数</p>

<%

String name = request.getParameter("name");

String age = request.getParameter("age");

String sex = request.getParameter("sex");

%>

<%="name参数值为：" + name %>

<%="age参数值为：" + age %>

<%="sex参数值为：" + sex %>

</body>

<body>

使用forward动作元素跳转到另一个JSP文件，并传递参数

<jsp:forward page=*"ForwardedDemo2.jsp"*>

<jsp:param value=*"Jame"* name=*"name"*/>

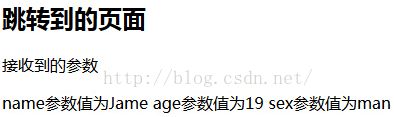
<jsp:param value=*"19"* name=*"age"*/>

<jsp:param value=*"man"* name=*"sex"*/>

</jsp:forward>

</body>

运行结果：



被包含的JSP文件通过使用request.getParameter()方法，来接受这些参数值，并赋值给字符串变量，最后通过JSP表达式输出变量的值。

### JSP内置对象

JSP内置对象是JSP容器为每个页面提供的Java对象，这些内部对象不需要由开发者实例化，他们由容器实现和管理，在所有的JSP页面总都使用内部对象。所有的隐式对象只对Scriptlet或者表达式有用，在JSP声明中不可用。

|  |  |
| --- | --- |
| 对象 | 描述 |
| request | HttpServletRequest类的实例 |
| response | HttpServletResponse类的实例 |
| out | PrintWriter类的实例，用于把结果输出至网页上 |
| session | HttpSession类的实例 |
| application | ServletContext类的实例，与应用上下文有关 |
| config | ServletConfig类的实例 |
| pageContext | PageContext类的实例，提供对JSP页面所有对象以及命名空间的访问 |
| page | 类似于Java类中的this关键字 |
| Exception | Exception类的对象，代表发生错误的JSP页面中对应的异常对象 |

#### request对象

request对象是javax.servlet.http.HttpServletRequest 类的实例。每当客户端提交一个请求到JSP页面时，JSP引擎就会创建一个新的request对象用来封装客户端发送来的请求数据。其中该对象中保存着客户端发送的请求数据或者参数。

来自客户端的请求经过Servlet容器封装进request对象中，request对象将作为jspService()方法的一个参数由容器传递给被请求的JSP页面，从而实现在被请求的JSP页面中包含封装数据或参数后的request对象。

在被请求的JSP页面中想要提取request对象中封装的数据或参数，只需要调用request对象的方法即可。

格式：request.方法

request对象的主要方法：

isUserInRole(String role)：判断认证后的用户名是否属于逻辑的role中的成员

getAttribute(String name)：返回由name指定的属性值，如果指定的属性值不存在，则会返回一个null值。

getAttributeNames()：返回request对象的所有属性的名字集合，其结果是一个枚举的实例。

getCookies()：返回客户端的所有Cookie对象，结果是一个Cookie数组

getCharacterEncoding()：返回请求中的字符编码方式

getContentLength()：返回请求的Body的长度，如果不确定长度，返回-1

getHeader(String name)：获得HTTP协议定义的文件头信息。

getHeaders(String name)：返回指定名字的request Header的所有值，其结果是一个枚举的实例。

getHeaderNames()：返回所有request Header的名字，其结果是一个枚举的实例。

getInputStream()：返回请求的输入流，用于获得请求中的数据。

getMethod()：获得客户端向服务端传送数据的方法，如GET、POST、HEADER、TRACE等。

getParameter(String name)：获得客户端传送给服务器端的参数值，该参数是由name指定的，通常是表单中的参数。

getParameterNames()：获得客户端传送给服务器端的所有参数的名字，其结果是一个枚举的实例。

getParameterValues(String name)：获得指定参数的所有值，参数由name指定。

getProtocol()：获取客户端向服务器端传送数据所依据的协议名称。

gerQueryString()：获得查询字符串，该字符串是由客户端以GET方法向服务器端传送的。

getRequestURI()：获取发出请求字符串的客户端地址。

getRemoteAddr()：获取客户端的IP地址

getRemoteHost()：获取客户端的名字

getSession([Boolean create])：返回和请求相关的session。create参数是可选的。当有参数create且这个参数的值为true时，如果客户端还没有创建session，那么将创建一个新的session。

getServerName()：获取服务器的名字

getServletPath()：获取客户端所请求的脚本文件的文件路径。

getServerPort()：获取服务器端的端口号

removeAttribute(String name)：删除请求中的一个属性。

setAttribute(String name, Object objt)：设置名字为name的request参数的值，该值是由java.lang.Object类型的oblt指定的。

使用request对象取得页面传递参数

<body>

<%

String param = request.getParameter("param");

%>

<a href=*"URL.jsp?param=Hello"*>点击这个链接</a>

<%="你提交的参数为：" + param %>

</body>

#### response对象

response对象是javax.servlet.http.HttpServletResponse类的实例。当服务器创建request对象时会同时创建用于响应这个客户端的response对象。该对象封装了JSP产生的响应，然后被发送给客户端以响应客户的请求。该对象由容器生成，作为jspService()方法的参数被传入JSP。因为输出流是缓冲的，所以可以设置HTTP状态码和response头。

response对象的主要方法：

addCookie(Cookie cook)：添加一个Cookie对象，用来保存客户端的用户信息。

addHeader(String name,String value)：添加HTTP文件头信息。该Header将传到客户端，如果已经存在同名的Header，将会覆盖已有的Header。

containsHeader(String name)：判断指定名字的HTTP文件头是否已经存在，然后返回真假布尔值。

encodeURL()：使用sessionId封装URL。如果没有必要封装URL，返回原值。

flushBuffer()：强制把当前缓冲区的内容发送到客户端。

getBufferSize()：返回缓冲区的大小。

getOutputStream()：返回到客户端的输出流对象。

sendError(int)：向客户端发送错误的信息。例如，404是指网页不寻在或者请求的页面无效。

sendRedirect(String location)：把响应发送到另一个位置进行处理。

setContentType(String contentType)：设置响应的MIME类型。

setHeader(String name ,String value)：设置指定名字的HTTP文件头的值，如果该值已经存在，则新值会覆盖原有的旧值。

#### out对象

out对象是 javax.servlet.jsp.JspWriter 类的实例，用来在response对象中写入内容。它表示为客户打开的输出流。PrintWriter使用它向客户端发送输出流。主要用来向客户端输出数据，代表输出流的对象。

out对象的主要方法：

out.print(boolean)、out.println(boolean)：输出Boolean类型的数据。

out.print(char)、out.println(char)：输出Char类型的数据。

out.print(char[])、out.println(char[])：输出Char[]类型的数据。

out.print(double)、out.println(double)：输出Double类型的数据。

out.print(float)、out.println(float)：输出Float类型的数据。

out.print(int)、out.println(int)：输出Int类型的数据。

out.print(long)、out.println(long)：输出Long类型的数据。

out.print(Object)、out.println(Object)：输出Object类型的数据。

out.print(String)、out.println(String)：输出String类型的数据。

out.newLine()：输出一个换行字符。

out.flush()：输出缓冲区里的数据。

out.close()：关闭输出流。

out.clearBuffer()：清除缓冲区里的数据，并把数据输出到客户端。

out.clear()：清除缓冲区里的数据，但不会把数据输出到客户端。

out.getBufferSize()：获得缓冲区的大小

out.getRemaining()：忽地缓冲区中没有被占用的空间的大小。

out.isAutoFlush()：返回布尔值。如果AutoFlush为真，则返回true；反之为false.

#### session对象

session对象是 javax.servlet.http.HttpSession 类的实例。该对象用来保存每个用户信息，以便跟踪每个用户的操作状态。其中，session信息保存在容器里，session的ID保存在客户机上的Cookie中。在许多服务器上，如果浏览器支持Cookie的话，就直接使用Cookie。但是如果不支持或废除了Cookie的话，就自动转化为URL-rewriting（重写URL，这个URL包含客户端的信息），session自动为每个流程提供了方便的存储信息的方法。

在一般情况下，用户首次登录系统时容器会给此用户分配一个唯一的标识的session id。这个ID用于区分其他的用户。当用户退出系统时，这个session就会自动消失。

session对象的主要方法：

getAttribute(String name)：获取与指定名字name相联系的属性。

getAttributeNames()：返回session对象中存储的每一个属性对象，其结果为一个枚举类的实例。

getCreationTime()：返回session被创建的时间，最小单位为千分之一秒。

getId()：此方法返回唯一的标识。每个session的ID是不同的。

getLastAccessedTime()：返回和当前session对象相关的客户端最后发送请求的时间，最小单位为千分之一秒。

getMaxInactiveInterval()：返回总时间（秒），负值表示session永远不会超时。它的值为该session对象的生存时间。

invalidate()：销毁这个session对象，使得和它绑定的对象都失效。

isNew()：如果客户端不接受使用session，那么每个请求中都会产生一个session对象。

removeAttribute(String name)：删除与指定name相联系的属性。

setAttribute(String name , java.lang.Object value)：设置指定名字name的属性值为value，并将之存储到session对象中。

#### application对象

application对象直接包装了servlet的ServletContext类的对象，是javax.servlet.ServletContext 类的实例。这个对象在JSP页面的整个生命周期中都代表着这个JSP页面。这个对象在JSP页面初始化时被创建，随着jspDestroy()方法的调用而被移除。

application对象的主要方法：

getAttribute(String name)：返回由name指定的名字的application对象的属性的值。

getAttributeNames()：返回所有的application对象的属性的名字，其结果是一个枚举的实例。

getInitParameter(String name)：返回由name指定的名字的application对象的某个熟悉的初始值。

getServletInfo()：返回Servlet编译器的当前版本的信息。

setAttribute(String name , Object object)：设置由name指定名字的application对象的属性的值object。

#### config对象

config对象是 javax.servlet.ServletConfig 类的实例，直接包装了servlet的ServletConfig类的对象。它表示Servlet的配置。当一个Servlet初始化时，容器把某些信息通过此对象传递给这个Servlet。

config对象的常用方法：

getServletContext()：返回执行者的Servlet上下文。

getServletName()：返回Servlet的名字。

getInitParameter(String name)：返回名字为name的初始参数的值。

getInitParameterNames()：返回这个JSP的所有的初始参数的名字。

#### pageContext 对象

pageContext对象是javax.servlet.jsp.PageContext 类的实例，用来代表整个JSP页面。它为JSP页面包装页面的上下文，管理对属于JSP中特殊可见部分中已命名对象的访问。它的创建和初始化都是由容器完成的。JSP页面里可以直接使用pageContext对象的句柄。pageContext对象的getXXX()、setXXX()和findXXX()方法可以用来根据不同的对象范围实现对这些对象的管理。

pageContext对象的主要功能是管理对属于JSP中特殊可见部分中已经命名对象的访问。

pageContext对象中常用的方法：

forward(String relativeUrlPath)：把页面重定向到另一个页面或者Servlet组件上。

getAttribute(String name[,int scope])：scope参数是可选的。该方法用来检索一个特定的已经命名的对象的范围，并且还可以通过调用getAttributeNamesInScope()方法，检索对某个特定范围的每个属性String字符串名称的枚举。

getException()：返回当前的exception对象。

gerRequest()：返回当前的request对象。

getResponse()：返回当前的reponse对象。

getServletConfig()：返回当前页面的ServletConfig对象。

getServletContext()：返回ServletContext对象。这个对象对所有的页面都是共享的。

getSession()：返回当前页面的session对象。

findAttribute()：用来按照页面、请求、会话及应用程序范围的顺序实现对某个已经命名属性的搜索。

setAttribute()：用来设置默认页面范围或特定对象范围之中的已命名对象。

removeAttribute()：用来删除默认页面范围或特定对象返回之中的已命名对象。

#### page 对象

page对象时java.lang.Object类的一个实例。它值的是JSP实现类的实例，也就是说，它是JSP本身。通过这个可以对它进行访问。

JSP实现类对象的一个句柄，只有在JSP页面的范围之内才是合法的。

#### exception 对象

exception 对象是java.lang.Throwable类的一个实例。它指的是运行时的异常，也就是被调用的错误页面的结果，只有在错误页面中才可以使用。

## 在JSP中使用JavaBean

### JavaBean的基本概念

JavaBean是用Java编写的类，它实现了将HTML代码与Java代码相分离，将Java代码单独封装成为一个处理某种业务逻辑的类，然后在JSP页面中调用此类，就可以降低HTML代码与Java代码之间的耦合度，简化JSP页面，提高Java程序代码的重用性和灵活性。实现了面向对象的开发模式。

* 使用JavaBean有以下几个优点：
* 可将HTML和Java代码分离，便于维护。
* 利用JavaBean的优点，将程序中经常用到的代码写成一个JavaBean组件，当在JSP使用时，直接调用JavaBean组件来执行所要的功能，大大节省了开发的时间和成本。
* 开发一个标准的JavaBean组件必须具有以下几个特性：
* 它必须是一个公共的(public)类
* 它必须有一个默认的构造方法，也就是不带参数的构造方法。在实例JavaBean对象时，需要调用默认的构造方法。该方法可以显式定义，如果不定义，解析器会自动创建一个默认的构造方法，从而可以通过new关键字直接对其进行实例化。
* 类中所有的属性都必须封装，即使用private声明。
* 它必须提供公共的setXXX()方法和getXXX()方法来让外部程序设置和获取JavaBean的属性。
* 它必须实现java.io.Serializable或者java.io.Externalizable接口，以支持序列化。

实例：

**public** **class** User {

**private** String name;

**private** String pwd;

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getPwd() {

**return** pwd;

}

**public** **void** setPwd(String pwd) {

**this**.pwd = pwd;

}

}

### JavaBean的属性

属性是JavaBean组件内部状态的抽象表示，外部程序使用属性来设置和获取JavaBean组件的状态。为了让外部程序能够知道JavaBean提供了那些属性，JavaBean的编写必须遵循标准的命名方法。

**属性命名规则**

例如，一个String类型的name属性，它对应的方法如下：

**public** **class** User {

**private** String name;

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

}

也就是为每一个属性添加get和set方法，其中属性名字的第一个字母大写，然后在名字前面相应地加上“get”和“set”。这样的属性是可读写的属性。如果一个属性只有get方法，那么这个属性是只读属性；如果一个属性只有set方法，那么这个属性是只写属性。

get/set命名方式有一个例外，就是对于boolean类型的属性，应该使用is/set命名方式（也可以使用get/set命名方式）。

**属性的类型**

JavaBean有4种类型的属性：

* 简单属性（simple property）
* 索引属性（indexed property）
* 绑定属性（bound property）
* 约束属性（constrained property）

在JSP中，支持JavaBean的简单属性和索引属性，绑定属性和约束属性主要用于图形界面编程中。

* 简单属性

简单属性就是接收单个值的属性。只要采用get/set命名约定即可。

* 索引属性

索引属性就是获取和设置数组时使用的属性。要运用索引属性，需要提供两对get/set方法，一对用于数组，另一对用于数组中的元素

语法格式：

**public** **class** User {

**private** String[] interest;

**public** String[] getInterest() {

**return** interest;

}

**public** **void** setInterest(String[] interest) {

**this**.interest = interest;

}

**public** String getInterest(**int** i){

**return** **this**.interest[i];

}

**public** **void** setInterest(**int** i, String str) {

**this**.interest[i] = str;

}

}

### 在JSP中使用JavaBean

在JSP中可以像使用普通类一样访问JavaBean，在脚本元素中实例化类的对象，调用对象的方法。为了充分利用JavaBean的特性，JSP提供了3个动作元素<jsp:useBean>、<jsp:setProperty>、<jsp:getProperty>来访问JavaBean。

* **<jsp:useBean>**

<jsp:useBean>动作元素用来实例化JavaBean，或者定位一个已经存在的JavaBean实例，并把实例的引用赋给一个变量

<jsp:useBean id=*"user"* scope=*"session"* class=*"com.uc.javaweb.pojo.User"* />

<jsp:useBean>元素各属性的含义：

* id

用于标识JavaBean实例的名字，同时，该名字也是声明的脚本变量的名字，并被初始化为JavaBean实例的引用。

* scope

指定JavaBean的有效使用范围，一共有page、request、session、application 4种属性范围。

* class

指明JavaBean对象所对应的包.类名称。

* type

指定引用该对象的变量的类型，它必须是Bean类的名字、超类名字、该类所实现的接口名字之一。请记住变量的名字是由id属性指定的。

* beanName

指定Bean的名字。如果提供了type属性和beanName属性，允许省略class属性。

<body>

<jsp:useBean id=*"user"* scope=*"session"* class=*"com.uc.javaweb.pojo.User"* />

<%

user.setName("yyy");

user.setPwd("mmm");

%>

<%="姓名：" + user.getName()%>

<%="密码：" + user.getPwd()%>

</body>

* **<jsp:setProperty>**

<jsp:setProperty>动作元素和<jsp:useBean>一起使用，用来设置JavaBean的简单属性和索引属性。<jsp:setProperty>动作使用Bean中的setXXX()方法，在Bean中设置一个或多个属性值。在JSP中，经常使用<jsp:setProperty>动作元素将客户端提交的数据保存在JavaBean的属性中。

语法格式：

<jsp:setProperty name=*"BeanName"* property=*"name"* value=*"value"*/>

<jsp:setProperty>元素各属性的含义：

* name：Bean实例的名字，它必须是已经在<jsp:useBean>元素中通过id属性定义的名字。Bean的实例必须包含可写（具有

setrXXX()方法）的属性。

* property：被设置的属性名字。如果property属性的值是“\*”，标签就会在请求对象中查找所有的请求参数，看是否有参数的

名字和Bean属性的名字相同，如果找到匹配的参数和属性就会按照正确的类型（自动进行类型转换）将参数的值设置为属性的值。如果一个参数的值为空（” ”），对于的属性的值不会被修改。

* param：指定请求对象中参数的名字。在设置Bean的属性时，如果请求参数的名字和Bean属性的名字相同，可以用param来

指定参数的名字。如果没有使用param，那么就认为请求参数的名字和Bean属性的名字相同。在<jsp:setProperty>元素中，不能同时出现param和value属性

* value：指定要赋给Bean属性的值。可以用一个请求时属性表达式作为value属性的值。
* **<jsp:getProperty>**

<jsp:getProperty>动作用来访问一个Bean的属性，并把属性的值转化成一个String然后发送到输出流中。如果属性是一个对象，将调用该对象的toString()方法。

语法格式：

<jsp:getProperty name=*"BeanName"* property=*"name"*/>

* name：Bean实例的名字，从这个实例中可以得到属性。（<jsp:useBean>标签中id属性的值）
* property：要得到的属性的名字。Bean的实例必须包含可读（具有getXXX()方法）的属性。

## EL表达式

表达式语言（Expression Language，简称EL），它最初定义在JSTL1.0规范中，在JSP2.0之后，EL已经正式成为JSP规范的一部分，并增加了新的特性。在JSP页面中使用表达式语言，可以简化对变量和对象的访问。

JSP表达式语言（EL）使得访问存储在JavaBean中的数据变得非常简单。JSP EL既可以用来创建算术表达式也可以用来创建逻辑表达式。在JSP EL表达式内可以使用整型数，浮点数，字符串，常量true、false，还有null。

### 语法

EL的语法简单，使用方便。所有的EL表达式都是以“${”开始，以“}”结束，例如${expr}。EL可以直接在JSP页面的模板文件中使用，也可以作为元素属性的值，还可以在自定义或者标准动作元素的内容中使用，但不能在脚本元素中使用。

#### “[ ]”和“.”操作符

EL使用“[]”和“.”操作符来访问数据，${expr-a.identifier-b}等价于${expr-a[“identifyier-b”]}。

例如：访问JavaBean对象user的属性name，可以写成如下两种形式：

${user.name}

${user["name"]}

#### 算术操作符

在EL中，有5个算术操作符

|  |  |  |  |
| --- | --- | --- | --- |
| 算术操作符 | 说明 | 实例 | 结果 |
| + | 加 | ${23 + 5} | 28 |
| - | 减 | ${23 – 5} | 18 |
| \* | 乘 | ${23 \* 5} | 115 |
| /（或div） | 除 | ${23 / 5}或${23 div 5} | 4.6 |
| %(或mod) | 取模(求余) | ${23 % 5}或${23 mod 5} | 3 |

需要注意的是，对于除法运算&{A / B}，如果A和B为null，返回0；如果A和B的类型是BigDecimal或BigInteger，将强制转换为BigDecimal，然后返回A.divide(B,BigDecimal.ROUND\_HALF\_UP)。对于其他情况，则将A和B强制转换为Double，然后进行相除。这就是为什么我们在表格中看到${23 / 5}的结果是4.6，而不是4的原因。

#### 关系操作符

在EL中，有6个关系操作符

|  |  |  |  |
| --- | --- | --- | --- |
| 关系操作符 | 说明 | 示例 | 结果 |
| ==或eq | 等于 | ${23 == 5}或${23 eq 5} | false |
| !=或ne | 不等于 | ${23 != 5}或${23 ne 5} | true |
| <或lt | 小于 | ${23 < 5}或${23 lt 5} | false |
| >或gt | 大于 | ${23 > 5}或${23 gt 5} | true |
| <=或le | 小于等于 | ${23 <= 5}或${23 le 5} | false |
| >=或ge | 大于等于 | ${23 >= 5}或${23 ge 5} | true |

#### 逻辑操作符

在EL中，有3个逻辑操作符

|  |  |  |  |
| --- | --- | --- | --- |
| 逻辑操作符 | 说明 | 示例 | 结果 |
| &&或and | 逻辑与 | 如果A为true，B为false，则A && B（或A and B） | false |
| ||或or | 逻辑或 | 如果A为true，B为false，则A || B（或A or B） | true |
| !或not | 逻辑非 | 如果A为true，则!A（或not A） | false |

#### Empty操作符

Empty操作符是一个前缀操作符，用于检测一个值是否为null或者为empty。

例如，变量A不存在，则${empty A}返回的结果为true

#### 条件操作符

EL中的条件操作符是“?”，例如：${A ? B : C}，如果A为true，计算B并返回其结果，如果A为fasle，计算C并返回其结果。

#### 圆括号

圆括号用于改变执行的优先级。例如：${23 \* (5 - 2)}

#### 操作符的优先级

[] . **>** () **>** -(unary) not ! empty **>** \* / div % mod **>** + - (binary) **>** < > <= >= lt gt le ge **>** == != eq ne **>** && and **>** || or **>** ? :

### 隐含对象

在EL中定义了11个隐含对象

#### pageContext

javax.servlet.PageContext对象。利用pageContext，可以访问ServletContext、Request、Response和Session等对象。

例如：

${pageContext.servletContext.serverInfo}

${pageContext.request.requestURI}

${pageContext.response.characterEncoding}

${pageContext.session.creationTime}

#### pageScope

类型是java.util.Map，将页面范围内的属性名和它的值进行映射。主要用于获取页面范围内的属性的值。

例如：

${pageScope.user}

如果user是一个JavaBean对象，还可以直接取出其属性值

例如：

${pageScope.user.name}

#### requestScope

类型是java.util.Map，将请求范围内的属性名和它的值进行映射。主要用于获取请求范围内的属性的值。

例如：

${requestScope.user.age}

#### sessionScope

类型是java.util.Map，将请求范围内的属性名和它的值进行映射。主要用于获取请求范围内的属性的值。

例如：

${sessionScope.user.education}

#### applicationScope

类型是java.util.Map，将应用程序范围内的属性名和它的值进行映射。主要用于获取应用程序范围内的属性的值。

例如：

${applicationScope.user.email}

#### param

类型是java.util.Map。将请求中的参数的名字和单个的字符串值进行映射。主要用于获取请求的参数，等同于调用ServletRequest.getParameter(String name)。

例如：一个请求URL：http://localhost:8080/ch16/user.jsp?name=zhangsan，要得到请求参数name的值。可以按照如下方式调用

${param.name}

#### paramValues

类型是java.util.Map，将请求中的参数的名字和一个包含了该参数所有值的String类型的数组进行映射。主要用于获取请求中的参数的值，等同于调用ServletRequest.getParameterValues(String name)

#### header

类型是java.util.Map，将请求报头的名字和单个的字符串值进行映射。主要用于获取请求报头的值，等同于调用ServletRequest.getHeader(String name)。

例如：

${header["User-Agent"]}

需要注意的是，对于包含连接字符（-）或其他一些特殊字符的字符串只能用“[ ]”操作符，而不能用“.”操作符，这也是“[ ]”和“.”操作符的区别。

#### headerValues

类型是java.util.Map，将请求报头的名字和一个包含该报头所有值的String类型的数组进行映射。主要用于获取请求报头的值，等同于调用ServletRequest.getHeaders(String name)。

#### cookie

类型是java.util.Map，将Cookie的名字和一个Cookie对象进行映射。主要有用于获取Cookie对象。如同调用HttpServletRequest.getCookies()后，从返回的Cookie数组中找到匹配名字的第一个Cookie对象。

例如：要得到一个名为userinfo的Cookie对象。

${cookie.userinfo }

如果要得到Cookie中的值，可以按照如下方式调用：

${cookie.userinfo.value}

initParam

类型是java.util.Map，将上下文的初始化参数的名字和它们的值进行映射。主要用于获取Web应用程序初始化参数的值，等同于调用ServletContext.getInitParameter(String name)。

例如，在web.xml文件中，使用<context-param>元素配置了一个driver参数，要得到它的值，可以按照如下方式调用：

${initParam.driver}

在JSP中的等价调用是：application.getInitParameter(“driver”)

### 命名变量

在EL中，对于命名变量值的查找是通过PageContext.findAttribute(String)方法来完成的。

例如，

${user}

这个表达式将按照page、request、session、application范围的顺序查找命名的属性user，如果属性没有找到，将返回null。我们也可以利用pageScope、requestScope、sessionScope和applicationScope指定范围，

例如：

${sessionScope.user}

### 保留字

|  |  |  |  |
| --- | --- | --- | --- |
| and | eq | gt | true |
| instanceof | or | ne | le |
| false | empty | not | lt |
| ge | null | div | mod |

### 函数

在EL中，允许定义和使用函数。语法格式如下：

ns:func(a1, a2, ..., an)

其中前缀ns必须匹配包含了函数的标签库的前缀，func是函数的名字，a1,a2,…,an是函数的参数。函数的定义和使用机制类似于标签库。

例如，我们编写一个将字符串从一种字符集转换为GBK编码的函数，首先编写一个Java类型，将这个函数声明为公开的静态的。

**public** **class** MyFunc {

**public** **static** String toGBK(String str, String charset){

String st="";

**try** {

st = **new** String(str.getBytes(charset), "GBK");

} **catch** (UnsupportedEncodingException e) {

// **TODO** 自动生成的 catch 块

e.printStackTrace();

}

**return** st;

}

}

然后在标签库描述符文件中对函数进行声明（myfuncs.tld）

<taglib xmlns=*"http://java.sun.com/xml/ns/j2ee"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://java.sun.com/xml/ns/j2ee web-jsptaglibrary\_2\_0.xsd"*

version=*"2.0"*>

<tlib-version>1.0</tlib-version>

<uri>/myfuncs</uri>

<function>

<name>toGBK</name>

<function-class>com.uc.MyFunc</function-class>\

<function-signature>

java.lang.String tiGBK(java.lang.String, java.lang.String)

</function-signature>

</function>

</taglib>

接下来可以编写一个测试页面（test.jsp）

<!DOCTYPE html>

<html>

<head>

<%@ page language=*"java"* contentType=*"text/html; charset=ISO-8859-1"*

pageEncoding=*"ISO-8859-1"*%>

<%@ taglib uri=*"/myfuncs"* prefix=*"myfn"* %>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=ISO-8859-1"*>

<title>???????</title>

</head>

<body>

欢迎你，${myfn:toGBK(param.username, "ISO-8859-1")}!

</body>

</html>

## JSP标准标签库(JSTL)

JSTL规范由Sun公司制定，Apache的Jakarta小组负责实现。EL最初定义在JSTL1.0规范中，在JSP2.0之后，EL已经正式成为JSP规范的一部分。在JSTL1.1规范中，已经没有了EL的部分，不过，在JSTL中仍然可以使用EL。

JSTL的目标是为了简化JSP页面的设计。JSTL的使用为页面设计人员和程序开发人员的分工协作提供了便利。JSTL虽然叫做标准标签库，但实际上是由5个不同功能的标签库组成。在JSTL1.1规范中，为这5个标签库分别指定了不同的URI，并对标签库的前缀作出了约定。

|  |  |  |
| --- | --- | --- |
| 功能范围 | URI | 前缀 |
| core | http://java.sun.com/jsp/jstl/core | c |
| I18N | http://java.sun.com/jsp/jstl/fmt | fmt |
| SQL | http://java.sun.com/jsp/jstl/sql | sql |
| XML | http://java.sun.com/jsp/jstl/xml | x |
| Funcions | http://java.sun.com/jsp/jstl/functions | fn |

### 配置JSTL

要想在项目中使用JSTL标签库，需下载JSTL相应的jar包导入到项目中去：jstl.jar和standard.jar。jstl.jar中包含的是JSTL规范中定义的接口和相关的类；standard.jar中包含的是Jakarta小组对JSTL的实现和JSTL中5个标签库的TLD文件。

### Core标签库

Core标签库主要包括了一般用途的标签、条件标签、迭代标签和URL相关的标签。在JSP页面中使用Core标签库，要使用taglib指令，指定引用的标签库。

<%@ taglib uri=*"http://java.sun.com/jsp/jstl/core"* prefix=*"c"* %>

#### 一般用途的标签

一般用途的标签有<c:out>、<c:set>、<c:remove>和<c:catch>。

##### <c:out>

<c:out>用于计算一个表达式并将结果输出到当前的JspWriter对象。<c:out>标签的功能类似于JSP的表达式<%=expression%>，或者EL表达式${el-expression}。

1. 语法

语法1：没有标签体

<c:out value=*"value"* [escapeXml=*"{true|false}"*][default=*"defaultValue"*] />

语法2：有标签体

<c:out value=*"value"* [escapeXml=*"{true|false}"*]>

default value

</c:out>

在TLD中，该标签的<body-content>元素设置为JSP

1. 属性

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 类型 | 是否接受动态的值 | 描述 |
| value | Object | true | 被计算的表达式 |
| escapeXml | boolean | true | 确定在结果字符串中的字符“<”、“>”、“’”、“””和“&”是否应该被转换为对应的字符串引用或预定义实体引用。默认值是true |
| default | Object | true | 如果value为null，那么将使用和这个默认值 |

除了一下两种情况，JSTL中的标签的属性值总是被指定为动态的。

* + XML标签的select属性，这个属性在JSTL是保留的，指定一个字符串字面量，表示一个XPath表达式。
  + 被JSLT标签导出的范围变量的名字和范围属性。

所谓动态的值是指属性的值可以是Java表达式、EL表达式，或者通过<jsp:attribute>设置的值。

1. Null和错误处理

如果value为null，将输出default的值。如果没有指定default值，将输出空字符串。

1. 说明

如果属性value计算的结果不是java.io.Reader对象，那么这个结果将被强制转换为String对象，然后输出到当前的JspWriter对象。如果计算的结果是javaio.Reader对象，那么数据首先从Reader对象中读取，然后写入到当前的JspWriter对象中。

如果属性escapeXml是true，那么字符串“<”、“>”、“’”、“””和“&”将按照表中进行转换。

|  |  |
| --- | --- |
| 字符串 | 字符串实体代码 |
| < | &lt; |
| > | &gt; |
| ‘ | &#039; |
| “ | &#034; |
| & | &amp; |

默认值可以通过属性default指定，也可以直接在标签中指定

1. 实例

<c:out value=*"*${customer.address.city}*"* default=*"unknown"* />

<c:out value=*"*${sessionScope.book.desc}*"* escapeXml=*"false"*>

no description

</c:out>

##### <c:set>

<c:set>用于设置范围变量（即范围属性）的值或者JavaBean对象的属性。

1. 语法

语法1：使用value属性设置范围变量的值

<c:set value=*"value"* var=*"varName"* [scope=*"{page|request|session|application}"*] />

语法2：使用标签体的内容设置范围变量的值

<c:set var=*"varName"* [scope=*"{page|request|session|application}"*] >

body content

</c:set>

语法3：使用value属性设置target对象的属性

<c:set value=*"value"* target=*"target"* property=*"propertyName"* />

语法4：使用标签体的内容设置target对象的属性

<c:set target=*"target"* property=*"propertyName"*>

body content

</c:set>

在TLD中，该标签的<body-content>元素设置JSP

1. 属性

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 类型 | 是否接受动态的值 | 描述 |
| value | Object | true | 被计算的表达式 |
| var | String | false | 被导出的保存了value属性计算结果的范围变量的名称。这个范围变量的类型是属性，value指定的表达式计算结果的类型 |
| scope | String | false | var的JSP范围。默认值是page |
| target | Object | true | 要设置属性的对象。必须是JavaBean对象（相应的属性有setter方法）或者java.util.Map对象 |
| property | String | true | 要设置的target对象的属性的名称 |

1. Null和错误处理

语法3和语法4在下列情况下会抛一个异常

* target为null
* target不是java.util.Map对象或者支持setter方法的JavaBean对象

如果value为null

语法1：由var和scope定义的范围变量将被移除

如果指定了scope，那么将按照PageContext.removeAttribute(varName, scope)方法的行为移除范围变量。否则，按照PageContext.removeAttribute(varName)方法的行为移除范围变量。

语法3：如果target是一个Map对象，则从Map中移除property所标识的键—值对。如果target是一个JavaBean对象，则将property指定的属性设置为null。

1. 说明

语法1和语法2设置由var和scope所标识的范围变量的值。

语法3和语法4；如果target是java.util.Map对象，则设置与property所标识的键对应的元素的值。如果这个元素不存在，则把它加入到Map对象中。否则，设置JavaBean对象的属性的值。

1. 示例

<c:set var=*"user"* value=*"zhangsan"* scope=*"session"* />

<c:set var=*"name"*>

zhangsan

</c:set>

<c:set target=*"*${user}*"* property=*"age"* value=*"18"* />

<c:set target=*"*${preferences}*"* property=*"color"* >

${param.color}

</c:set>

##### <c:remove>

<c:remove>用于移除范围变量

1. 语法

<c:remove var=*"varName"* [scope=*"{page|request|session|application}"*]/>

1. 属性

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 类型 | 是否接受动态的值 | 描述 |
| var | String | false | 要移除的范围变量的名称 |
| scope | String | false | var的JSP范围。默认值是page |

1. 说明

如果没有指定scope属性，那么将按照PageContext.removeAttribute(varName)方法的行为移除范围变量。如果指定了scope属性，那么将按照PageContext.removeAttribute(varName, scope)方法的行为移除范围变量。

1. 实例

<c:remove var=*"user"* scope=*"session"* />

##### <c:catch>

<c:catch>用于捕获在其中嵌套的操作所抛出的异常对象（java.lang.Throwable对象），并将异常信息保存到变量中。

1. 语法

<c:catch [var=*"varName"*]>

nested actions

</c:catch>

在TLD中，该标签的<body-content>元素设置为JSP

1. 属性

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 类型 | 是否接受动态的值 | 描述 |
| var | String | false | 指定被导出的范围变量的名字，该范围变量保存了从嵌套的操作中抛出的异常。这个范围变量的类型是抛出的异常的类型 |

1. 描述

<c:catch>允许页面作者以一种统一的方式来处理任何操作抛出的异常。我们将可能抛出异常的代码放置在开始标签<c:catch>和结束标签</c:catch>之间，如果其中的代码抛出异常，异常将被捕获，并被保存到var所标识的范围变量中，该变量总是由page范围。如果没有异常发生，而var所标识的范围变量存在，那么它将被移除。

如果没有指定var属性，异常只是简单地被捕获，异常信息并不会被保存。

1. 示例

<c:catch var=*"exception"*>

<%

**int** i = 5;

**int** j = 0;

**int** k = i / j;

%>

</c:catch>

<c:out value=*"*${exception}*"* /><p>

<c:out value=*"*${exception.message}*"* /><p>

<!-- ${exception.message}相当于调用exception.getMessage() -->

#### 条件标签

条件标签包括<c:if>、<c:choose>、<c:when>和<c:otherwise>

##### <c:if>

<c:if>用于实现Java语言中if语句的功能。

1. 语法

语法1：没有标签体

<c:if test=*"testCondition"* var=*"varName"* [scope=*"{page|request|session|application}"*] />

语法2：有标签体

<c:if test=*"testCondition"* [var=*"varName"*][scope=*"{page|request|session|application}"*]>

body content

</c:if>

在TLD中，该标签的<body-content>元素设置为JSP。

1. 属性

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 类型 | 是否接受动态的值 | 描述 |
| test | boolean | true | 测试的条件，用于判断标签体是否应该被执行 |
| var | String | false | 被导出的保存了测试条件结果值的范围变量的名字。这范围变量的类型是Boolean |
| scope | String | false | var的JSP范围。默认值是page |

1. 约束

如果指定了scope属性，那么必须指定var属性。

1. 说明

如果属性test计算为true，那么标签体将被JSP容器执行，执行的结果将被输出到当前的JspWriter对象。对于语法1，var属性是必须要提供的，在标签执行后，可以用保存了条件结果的范围变量做进一步的判断。

示例

<c:if test=*"*${user.visitCount == 1}*"*>

this is your first visit. welcome to the site!

</c:if>

<c:if test=*"*${param.name == 'admin'}*"* var=*"result"* />

<c:out value=*"*${result}*"* />

##### <c:choose>

<c:choose>、<c:when>和<c:otherwise>一起实现互斥条件的执行，类似于Java语言的if/else if/else语句。

1. 语法

<c:choose>

body content(<when> and <otherwise> subtags)

</c:choose>

在TLD中，该标签的<body-content>元素设置为JSP。

1. 约束

<c:choose>是作为<c:when>和<c:otherwise>的父标签使用的，除了空白字符外，<c:choose>的标签体只能包含这两个标签。

1. 说明

在运行时，判断<c:when>标签的测试条件是否为true，第一个测试条件为true的<c:when>标签的标签体将被JSP容器执行。如果没有满足条件的<c:when>标签，那么<c:otherwise>的标签体将被执行。

##### <c:when>

<c:when>作为<c:choose>的子标签，表示一个可选的条件

1. 语法

<c:when test=*"testCondition"*>

body content

</c:when>

在TLD中，该标签的<body-content>元素设置为JSP

1. 属性

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 类型 | 是否接受动态的值 | 描述 |
| test | boolean | true | 测试的条件，用于判断标签体是否应该被执行。 |

1. 约束

<c:when>标签必须有一个直接的父标签<c:choose>，而且必须在同一个父标签下的<c:otherwise>标签之前出现。

1. 说明

在运行时，判断<c:when>标签的测试条件是否为true，第一个测试条件为true的<c:when>标签的标签体将被JSP容器执行。

##### <c:otherwise>

<c:otherwise>作为<c:choose>的子标签，表示最后的选择

1. 语法

<c:otherwise>

conditional block

</c:otherwise>

在TLD中，该标签的<body-content>元素设置为JSP

1. 约束

<c:otherwise>标签必须有一个直接的父标签<c:choose>，而且必须是<c:choose>标签中最后一个嵌套的标签。

1. 说明

在运行时，如果没有满足条件的<c:when>标签，那么<c:otherwise>的标签体将被JSP容器执行。

1. 示例

<c:choose>

<c:when test=*"*${param.name == 'zhangsan'}*"*>

${param.name} is manager!

</c:when>

<c:when test=*"*${param.name == 'lisi'}*"*>

${param.name} is salesman!

</c:when>

<c:otherwise>

${param.name} is employee!

</c:otherwise>

</c:choose>

#### 迭代标签

迭代标签有<c:forEach>和<c:forTokens>

##### <c:forEach>

<c:forEach>用于对包含了多个对象的集合进行迭代，重复执行它的标签体，或者重复迭代固定的次数。

1. 语法

语法1：对包含了多个对象的集合进行迭代

<c:forEach [var=*"varName"*] items=*"collection"*[varStatus=*"varStatusName"*][begin=*"begin"*][end=*"end"*][step=*"step"*]>

body content

</c:forEach>

语法2：迭代固定的次数

<c:forEach [var=*"varName"*][varStatus=*"varStatusName"*]begin=*"begin"* end=*"end"*[step=*"step"*]>

body content

</c:forEach>

在TLD中，该标签的<body-content>元素设置为JSP

1. 属性

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 类型 | 是否接受动态的值 | 描述 |
| var | String | false | 被导出的保存了当前迭代条目的范围变量的名字。这个范围变量的范围是NESTED，它的类型依赖于集合中的对象 |
| items | 数组、字符串和各种集合类型 | true | 要迭代的集合对象 |
| varStatus | String | false | 被导出的保存了迭代状态的范围变量的名字。被导出对象的类型是javax.servlet.jsp.jstl.core.LoopTagStatus，这个范围变量的范围是NESTED |
| begin | int | true | 如果指定了items，那么就从指定索引处的项目开始迭代，集合中第一个条目的索引是0。果没有指定items，那么将从指定的索引值开始迭代 |
| end | int | true | 如果指定了items，那么迭代将终止于指定索引处的项目（包含该项目）。如果没有指定items，那么将在索引达到指定的值时结束迭代 |
| step | int | true | 迭代的步长。默认的步长是1 |

1. 约束

如果指定了begin属性，则begin必须大于等于0。如果指定了end属性，而end小于begin，那么循环将不会执行。如果指定了step属性，则step必须大于等于1。

1. Null和错误处理

如果items是null，那么它被当做一个空的集合，这时，迭代不会执行

1. 说明

如果begin属性大于或等于items的大小，则迭代不会执行。items支持的类型如下：

* 数组，包括基本数据类型的数组和对象数组。对于基本数据类型的数组，迭代的当前条目将自动被转换为对应的封装类型。例如，int被转换为Integer，float被转换为Float。
* java.util.Collection接口的实现
* java.util.Iterator接口的实现
* java.util.Enumeration接口的实现
* java.util.Map接口的实现

通过var属性所导出的范围变量的类型是Map.Entry，这个变量将具有key和value两个属性，分别用于得到存储在Map中的键和值。

* String

一列以逗号（,）分隔的字符串值。在迭代时，将以逗号字符作为分隔符，连续地进行处理。

对于语法1，除了基本类型的数组和Map外，通过var属性所导出的范围变量的类型就是集合中对象的类型。对于语法2，导出的范围变量的类型是Integer。

示例

<c:forEach var=*"entry"* items=*"*${myHashMap}*"*>

下一个元素的键是${entry.key}

下一个元素的值是${entry.value}<p>

</c:forEach>

这个例子将迭代输出Map中存储的键—值对。

<table>

<c:forEach var=*"user"* items=*"*<%=arrList %>*"* varStatus=*"status"*>

<tr>

<td>${status.count}</td>

<td>${status.index}</td>

<td>${status.first}</td>

<td>${status.last}</td>

<td>${user.name}</td>

</tr>

</c:forEach>

</table>

<c:forEach var=*"i"* begin=*"100"* end=*"110"*>

${i}

</c:forEach>

这个例子将输出100~110（包含110）之间的整数数字。如果添加属性step=”2”，那么输出的结果将是：100 102 104 106 108 110。

##### <c:forTokens>

<c:forTokens>用于迭代字符串中由分隔符分隔的各成员

1. 语法

## 自定义标签库

# 应用篇

## 中文乱码与国际化

## 开发安全的Web应用程序

## 分页显示

在开发过程中，经常做的一件事，也是最基本的事，就是从数据库中查询数据，然后在客户端显示出来。当数据少时，可以在一个页面内显示完成。然而，如果查询记录是几百条、上千条呢？直接一个页面显示完全的话，表格得多长啊。。。。。。这时，我们可以用分页技术。

下面谈谈分页效果的实现，思路有三种：

### JS前端实现分页

纯JS实现分页。一次性查询记录并加载到html的table中。然后通过选择性地显示某些行来达到分页显示的目的。这是一种伪分页，障眼法而已。只能用于数据少的情况下。一旦数据多了，十几万条数据加载到html中会变得很慢。而且不实时，一次加载完后数据就写死在页面了，若数据库中有变化，浏览器端显示的仍是上次加载过来的数据。

首先：用table来显示查询出来的记录们，全部显示。

<table width=*"500"* id=*"idData"*>

<%

String user\_id, user\_name, user\_sex, user\_phone, user\_age;

**while** (sqlRst.next()) {

user\_id = sqlRst.getString(1);

user\_name = sqlRst.getString(2);

user\_sex = sqlRst.getString(3);

user\_phone = sqlRst.getString(4);

user\_age = sqlRst.getString(5);

%>

<tr>

<td><%=user\_id%></td>

<td><%=user\_name%></td>

<td><%=user\_sex%></td>

<td><%=user\_phone%></td>

<td><%=user\_age%></td>

</tr>

<%

}

%>

</table>

<br/>

<table width=*"60%"* align=*"right"*>

<tr><td><div id=*"changePages"* name=*"changePages"*></div></td></tr>

</table>

然后，在JS中修改table中某些行显示，某些行隐藏。

<script type=*"text/javascript"*>

**function** goPage(pno,psize){

**var** itable = document.getElementById("idData");//获取table

**var** num = itable.rows.length;//得到记录总数

**var** totalPage = 0;

**var** pageSize = psize;//一页显示pageSize条记录

//计算总页数

**if**(num/pageSize > parseInt(num/pageSize)){

totalPage=parseInt(num/pageSize)+1;

}**else**{

totalPage=parseInt(num/pageSize);

}

//当前页数

**var** currentPage = pno;

//获取当前页第一条、最后一条记录的行号

**var** startRow = (currentPage - 1) \* pageSize+1;

**var** endRow = currentPage \* pageSize;

endRow = (endRow > num)? num : endRow;

//修改table中当前页对应的行的属性为显示，非本页的记录为隐藏

**for**(**var** i=1;i<(num+1);i++){

**var** irow = itable.rows[i-1];

**if**(i>=startRow && i<=endRow){

irow.style.display = "block";

}**else**{

irow.style.display = "none";

}

}

//分页页码列表

**var** tempStr = "共"+num+"条记录 分"+totalPage+"页 当前第"+currentPage+"页";

**if**(currentPage>1){

tempStr += "<a href=\"#\" onClick=\"goPage("+(1)+","+psize+")\">首页</a>";

tempStr += "<a href=\"#\" onClick=\"goPage("+(currentPage-1)+","+psize+")\"><上一页</a>"

}**else**{

tempStr += "首页";

tempStr += "<上一页";

}

**if**(currentPage<totalPage){

tempStr += "<a href=\"#\" onClick=\"goPage("+(currentPage+1)+","+psize+")\">下一页></a>";

tempStr += "<a href=\"#\" onClick=\"goPage("+(totalPage)+","+psize+")\">尾页</a>";

}**else**{

tempStr += "下一页>";

tempStr += "尾页";

}

document.getElementById("changePages").innerHTML = tempStr;

}

</script>

### 利用结果集分批显示。

就是说，我们可以执行一个数据库查询操作，得到结果集rs。然后，通过指针的移动来显示当前页面的记录。这样，就可以以rs.absolute(当前页面号\*每页记录数)定位到当前页的第一条记录，然后通过while循环显示n条记录（n为每页显示记录数）。在跳页时，只需修改currentPage，即可在重定位到下一页时把当前页面号改掉，重新定位记录指针，通过while遍历显示n条记录。与JS选择性显示不同，这里是选择性遍历。与JS分页不同的是，这里分页每次跳页修改的是遍历的指针，每次跳页都要进行一次全面查询。同样地，不适合大数据量查询。这里比JS分页优化的地方在于——实时性。每次跳页都会查询一次数据库，保证数据的实时性。

### 利用SQL分页查询功能

## 验证码

## 上传和下载

### 上传

在Web应用系统开发中，文件上传和下载功能是非常常用的功能，今天来讲一下JavaWeb中的文件上传和下载功能的实现。

对于文件上传，浏览器在上传的过程中是将文件以流的形式提交到服务器端的，如果直接使用Servlet获取上传文件的输入流然后再解析里面的请求参数是比较麻烦，所以一般选择采用apache的开源工具common-fileupload这个文件上传组件。这个common-fileupload上传组件的jar包可以去apache官网上面下载，也可以在struts的lib文件夹下面找到，struts上传的功能就是基于这个实现的。common-fileupload是依赖于common-io这个包的，所以还需要下载这个包。

1. 文件上传页面好消息提示页面

文件上传页面（upload.jsp）

<form action=*"UploadServlet "* method=*"post"* enctype=*"multipart/form-data"*>

上传文件1：<input type=*"file"* name=*"file1"* />

上传文件2：<input type=*"file"* name=*"file2"* />

<input type=*"submit"* value=*"上传"* />

</form>

消息提示页面（message.jsp）

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=UTF-8"*>

<title>消息提示</title>

</head>

<body>

${message}

</body>

</html>

1. 编写处理上传文件的Servlet（UploadServlet）

@WebServlet("/UploadServlet")

**public** **class** UploadServlet **extends** HttpServlet {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**protected** **void** doGet(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {

//得到上传文件的保存目录，将上传的文件存放于WEB-INF目录下，不允许外界直接访问，保证上传文件的安全

String path = **this**.getServletContext().getRealPath("/WEB-INF/upload");

//判断文件夹是否存在，如果不存在则先创建文件夹

File file = **new** File(path);

**if**(!file.exists()) {

file.mkdir();

}

//消息提示

String message = "";

**try** {

//使用Apache文件上传组件处理文件上传步骤

//1.创建一个DiskFileItemFactory工厂

DiskFileItemFactory factory = **new** DiskFileItemFactory();

//设置工厂的缓冲区的大小，当上传的文件大小超过缓冲区大小时，就会生成一个临时文件存放到指定的临时目录中。

factory.setSizeThreshold(1024 \* 100); //设置缓冲区大小为100KB，如果不指定，那么缓冲区的大小默认10KB

//设置上传时生成的临时文件的保存目录(JDK自带的临时文件存放目录，也可以自建)

factory.setRepository(**new** File(System.*getProperty*("java.io.tmpdir")));

//2.设置一个文件上传解析器

ServletFileUpload upload = **new** ServletFileUpload(factory);

//监听文件上传进度

upload.setProgressListener(**new** ProgressListener() {

**public** **void** update(**long** pBytesRead, **long** pContentLength, **int** arg2) {

System.***out***.println("文件大小为：" + pContentLength + ",当前已处理：" + pBytesRead);

/\*\*

\* 文件大小为：14608,当前已处理：4096

\*文件大小为：14608,当前已处理：7367

\*文件大小为：14608,当前已处理：11419

\*文件大小为：14608,当前已处理：14608

\*/

}

});

//解决上传文件名的中文乱码

upload.setHeaderEncoding("UTF-8");

//3.判断提交上来的数据是否是上传表单的数据

**if**(!ServletFileUpload.*isMultipartContent*(request)) {

//按照传统方式获取数据

**return**;

}

//设置上传单个文件的大小的最大值。

upload.setFileSizeMax(1024 \* 1024);

//设置上传文件总量的最大值，最大值=同时上传的多个文件的最大值的和。

upload.setSizeMax(1024 \* 1024 \* 10);

//4.使用ServletFileUpload解析器解析上传数据，解析结果返回的是一个List<FileItem>集合，每一个FileItem对应一个Form表单的输入项

List<FileItem> list = upload.parseRequest(request);

**for** (FileItem fileItem : list) {

//如果fileItem中封装的是普通输入项的数据

**if**(fileItem.isFormField()) {

String name = fileItem.getFieldName();

//解决普通输入项的数据的中文论码

String value = fileItem.getString("UTF-8");

//value = new String(value.getBytes("iso8859-1"),"UTF-8");

System.***out***.println(name + "=" + value);

}**else** {//如果fileitem中封装的是上传文件

//得到上传的文件名称，

String filename = fileItem.getName();

System.***out***.println(filename);

**if**(filename == **null** || filename.trim().equals("")) {

**continue**;

}

//注意：不同的浏览器提交的文件名是不一样的，有些浏览器提交上来的文件名是带有路径的，如：c:\a\b\1.txt，而有些只是单纯的文件名，如：1.txt

//处理获取到的上传文件的文件名的路径部分，只保留文件名部分

filename = filename.substring(filename.lastIndexOf("\\") + 1);

//得到上传文件的扩展名

String fileExtName = filename.substring(filename.lastIndexOf(".") + 1);

//如果需要限制上传的文件类型，那么可以通过文件的扩展名来判断上传的文件类型是否合法

System.***out***.println("上传的文件的扩展名是："+fileExtName);

//获取fileItem中的上传文件的输入流

InputStream in = fileItem.getInputStream();

//得到文件保存的名称

String saveFilename = makeFileName(filename);

//得到文件的保存目录

String realSavePath = makePath(saveFilename, path);

//创建一个文件输出流

FileOutputStream out = **new** FileOutputStream(realSavePath + "\\" + saveFilename);

//创建一个缓冲区

**byte** buffer[] = **new** **byte**[1024];

//判断输入流中的数据是否已经读完的标识

**int** len = 0;

//循环将输入流读入到缓冲区当中，(len=in.read(buffer))>0就表示in里面还有数据

**while**((len=in.read(buffer))>0){

//使用FileOutputStream输出流将缓冲区的数据写入到指定的目录(savePath + "\\" + filename)当中

out.write(buffer, 0, len);

}

//关闭输入流

in.close();

//关闭输出流

out.close();

//删除处理文件上传时生成的临时文件

// fileItem.delete();

message = "文件上传成功！";

}

}

} **catch** (FileUploadException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**private** String makeFileName(String filename){

//为防止文件覆盖的现象发生，要为上传文件产生一个唯一的文件名

**return** UUID.*randomUUID*().toString() + "\_" + filename;

}

/\*\*

\* 为防止一个目录下面出现太多文件，要使用hash算法打散存储

\*/

**private** String makePath(String filename,String savePath){

//得到文件名的hashCode的值，得到的就是filename这个字符串对象在内存中的地址

**int** hashcode = filename.hashCode();

**int** dir1 = hashcode&0xf; //0--15

**int** dir2 = (hashcode&0xf0)>>4; //0-15

//构造新的保存目录

String dir = savePath + "\\" + dir1 + "\\" + dir2; //upload\2\3 upload\3\5

//File既可以代表文件也可以代表目录

File file = **new** File(dir);

//如果目录不存在

**if**(!file.exists()){

//创建目录

file.mkdirs();

}

**return** dir;

}

**protected** **void** doPost(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {

// **TODO** Auto-generated method stub

doGet(request, response);

}

}

### 下载

1. 列出提供下载的文件资源

将Web应用中的文件资源提供给用户进行下载，首先要在界面中列出文件目录下的所有文件，当用户点击文件下载超链接时就进行下载。

@WebServlet("/ListFileServlet")

**public** **class** ListFileServlet **extends** HttpServlet {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**protected** **void** doGet(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {

//获取上传文件的目录

String uploadPath = **this**.getServletContext().getRealPath("WEB-INF/upload");

//存储要下载的文件名

Map<String, String> fileNameMap = **new** HashMap<>();

//递归遍历filepath目录下的所有文件和目录，将文件的文件名存储到map集合中

listfile(**new** File(uploadPath), fileNameMap); //File既可以代表文件也可以代表目录

//将Map集合发送到listfile.jsp页面进行显示

request.setAttribute("fileNameMap", fileNameMap);

request.getRequestDispatcher("/listfile.jsp").forward(request, response);

}

**public** **void** listfile(File file, Map<String, String> map) {

//如果file代表的不是文件，而是一个目录

**if**(!file.isFile()) {

//列出该目录下的所有文件和目录

File[] files = file.listFiles();

//遍历files[]数组

**for** (File f : files) {

//递归调用

listfile(f,map);

}

}**else** {

/\*\*

\* 处理文件名，上传后的文件是以uuid\_文件名的形式去重新命名的，去除文件名的uuid\_部分

\* file.getName().indexOf("\_")检索字符串中第一次出现"\_"字符的位置，如果文件名类似于：9349249849-88343-8344\_阿\_凡\_达.avi

\* 那么file.getName().substring(file.getName().indexOf("\_")+1)处理之后就可以得到阿\_凡\_达.avi部分

\*/

String realName = file.getName().substring(file.getName().indexOf("\_")+1);

//file.getName()得到的是文件的原始名称，这个名称是唯一的，因此可以作为key，realName是处理过后的名称，有可能会重复

map.put(file.getName(), realName);

}

}

**protected** **void** doPost(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {

// **TODO** Auto-generated method stub

doGet(request, response);

}

}

这里简单说一下ListFileServlet中listfile方法，listfile方法是用来列出目录下的所有文件的，listfile方法内部用到了递归，在实际开发当中，我们肯定会在数据库创建一张表，里面会存储上传的文件名以及文件的具体存放目录，我们通过查询表就可以知道文件的具体存放目录，是不需要用到递归操作的，这个例子是因为没有使用数据库存储上传的文件名和文件的具体存放位置，而上传文件的存放位置又使用了散列算法打散存放，所以需要用到递归，在递归时，将获取到的文件名存放到从外面传递到listfile方法里面的Map集合当中，这样就可以保证所有的文件都存放在同一个Map集合当中。

1. 展示下载文件的listfile.jsp页面

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

<%@ taglib prefix=*"c"* uri=*"http://java.sun.com/jsp/jstl/core"* %>

<!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>Insert title here</title>

</head>

<body>

<!-- 遍历Map集合 -->

<c:forEach items=*"*${fileNameMap}*"* var=*"me"*>

<c:url value=*"/servlet/DownLoadServlet"* var=*"downurl"*>

<c:param name=*"filename"* value=*"*${me.key}*"*></c:param>

</c:url>

${me.value}<a href=*"*${downurl}*"*>下载</a>

<br/>

</c:forEach>

</body>

</html>

访问ListFileServlet，就可以在listfile.jsp页面中显示提供给用户下载的文件资源

1. 实现文件下载

编写一个用于处理文件下载的Servlet，DownLoadServlet。

@WebServlet("/DownLoadServlet")

**public** **class** DownLoadServlet **extends** HttpServlet {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**protected** **void** doGet(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {

//得到要下载的文件名

String fileName = request.getParameter("filename"); //23239283-92489-阿凡达.avi

fileName = **new** String(fileName.getBytes("iso8859-1"), "utf-8");

//上传的文件都是保存在/WEB-INF/upload目录下的子目录中

String fileSaveRootPath = **this**.getServletContext().getRealPath("/WEB-INF/upload");

//通过文件名找到文件的所在目录

String path = findFileSavePathByFileName(fileName, fileSaveRootPath);

//得到要下载的文件

File file = **new** File(path + "\\" + fileName);

//如果文件不存在

**if**(!file.exists()) {

request.setAttribute("message", "您要下载的资源已删除！！");

request.getRequestDispatcher("/message.jsp").forward(request, response);

**return**;

}

//处理文件名

String realname = fileName.substring(fileName.indexOf("\_") + 1);

//设置响应头，控制浏览器下载该文件

response.setHeader("content-disposition", "attachment;filename=" + URLEncoder.*encode*(realname, "UTF-8"));

//读取要下载的文件，保存到文件输入流

FileInputStream in = **new** FileInputStream(path + "\\" + fileName);

//创建输出流

OutputStream out = response.getOutputStream();

//创建缓冲区

**byte**[] buffer = **new** **byte**[1024];

**int** len = 0;

//循环将输出流中的内容读取到缓冲区中

**while**((len = in.read(buffer)) > 0) {

//输出缓冲区的内容到浏览器，实现文件下载

out.write(buffer, 0, len);

}

out.flush();

//关闭流

in.close();

out.close();

}

**public** String findFileSavePathByFileName(String filename,String saveRootPath){

**int** hashcode = filename.hashCode();

**int** dir1 = hashcode&0xf; //0--15

**int** dir2 = (hashcode&0xf0)>>4; //0-15

String dir = saveRootPath + "\\" + dir1 + "\\" + dir2; //upload\2\3 upload\3\5

File file = **new** File(dir);

**if**(!file.exists()){

//创建目录

file.mkdirs();

}

**return** dir;

}

**protected** **void** doPost(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {

// **TODO** Auto-generated method stub

doGet(request, response);

}

}

## 防止表单重复提交

## 图片的缩放

## 图片添加水印和文字

# server.xml文件

# web.xml文件

# JSON

JSON(JavaScript Object Notation, JS 对象标记) 是一种轻量级的数据交换格式。它基于 ECMAScript (w3c制定的js规范)的一个子集，采用完全独立于编程语言的文本格式来存储和表示数据。简洁和清晰的层次结构使得 JSON 成为理想的数据交换语言。 易于人阅读和编写，同时也易于机器解析和生成，并有效地提升网络传输效率。

## 什么是 JSON

1. JSON 指的是 JavaScript 对象表示法（JavaScript Object Notation）
2. JSON 是轻量级的文本数据交换格式
3. JSON 独立于语言：JSON 使用 Javascript语法来描述数据对象，但是 JSON 仍然独立于语言和平台。JSON 解析器和 JSON 库支持许多不同的编程语言。 目前非常多的动态（PHP，JSP，.NET）编程语言都支持JSON。
4. JSON 具有自我描述性，更易理解

## JSON与XML之间的比较

1. 可读性

JSON和XML的可读性可谓不相上下，一边是简易的语法，一边是规范的标签形式，很难分出胜负。

1. 可扩展性

XML天生有很好的扩展性，JSON当然也有，没有什么是XML可以扩展而JSON却不能扩展的。不过JSON在Javascript主场作战，可以存储Javascript复合对象，有着xml不可比拟的优势。

1. 编码难度

XML有丰富的编码工具，比如Dom4j、JDom等，JSON也有提供的工具。无工具的情况下，相信熟练的开发人员一样能很快的写出想要的xml文档和JSON字符串，不过，xml文档要多很多结构上的字符。

1. 解码难度

XML的解析方式有两种：

一是通过文档模型解析，也就是通过父标签索引出一组标记。例如：xmlData.getElementsByTagName("tagName")，但是这样是要在预先知道文档结构的情况下使用，无法进行通用的封装。

另外一种方法是遍历节点（document 以及 childNodes）。这个可以通过递归来实现，不过解析出来的数据仍旧是形式各异，往往也不能满足预先的要求。

凡是这样可扩展的结构数据解析起来一定都很困难。

JSON也同样如此。如果预先知道JSON结构的情况下，使用JSON进行数据传递简直是太美妙了，可以写出很实用美观可读性强的代码。如果你是纯粹的前台开发人员，一定会非常喜欢JSON。但是如果你是一个应用开发人员，就不是那么喜欢了，毕竟xml才是真正的结构化标记语言，用于进行数据传递。

而如果不知道JSON的结构而去解析JSON的话，那简直是噩梦。费时费力不说，代码也会变得冗余拖沓，得到的结果也不尽人意。但是这样也不影响众多前台开发人员选择JSON。因为json.js中的toJSONString()就可以看到JSON的字符串结构。当然不是使用这个字符串，这样仍旧是噩梦。常用JSON的人看到这个字符串之后，就对JSON的结构很明了了，就更容易的操作JSON。

以上是在Javascript中仅对于数据传递的xml与JSON的解析。在Javascript地盘内，JSON毕竟是主场作战，其优势当然要远远优越于xml。如果JSON中存储Javascript复合对象，而且不知道其结构的话，我相信很多程序员也一样是哭着解析JSON的。

1. 除了上述之外，JSON和XML还有另外一个很大的区别在于有效数据率。JSON作为数据包格式传输的时候具有更高的效率，这是因为JSON不像XML那样需要有严格的闭合标签，这就让有效数据量与总数据包比大大提升，从而减少同等数据流量的情况下，网络的传输压力 。

## JSON语法

### JSON语法规则

在 JS 语言中，一切都是对象。因此，任何支持的类型都可以通过 JSON 来表示，例如字符串、数字、对象、数组等。但是对象和数组是比较特殊且常用的两种类型：

1. 数据在名称/值对中
2. 数据由逗号分隔
3. 花括号保存对象
4. 方括号保存数组

### JSON 名称/值对

JSON 键值对是用来保存 JS 对象的一种方式，和 JS 对象的写法也大同小异，键/值对组合中的键名写在前面并用双引号 "" 包裹，使用冒号 : 分隔，然后紧接着值：

{"firstName": "John"}

这很容易理解，等价于这条 JavaScript 语句：

{firstName = "John"}

### JSON 值

JSON的值可以是：数值（整数或浮点数）、字符串（在双引号中）、逻辑值（true或false）、数组（在方括号中）、对象（在花括号中）、null。

数值（整数或浮点数）：

{"number":123}

{"number":12.3}

字符串（在双引号中）：

{"name":"John"}

逻辑值（true或false）：

{ "flag":true }

数组（在方括号中）：

{

"sites": [

{ "name":"菜鸟教程" , "url":"www.runoob.com" },

{ "name":"google" , "url":"www.google.com" },

{ "name":"微博" , "url":"www.weibo.com" }

]

}

对象（在圆括号中）：

{"name":"菜鸟教程" , "url":"www.runoob.com"}

JSON null（JSON 可以设置 null 值）：

{ "runoob":null }

## JSON 使用 JavaScript 语法

因为 JSON 使用 JavaScript 语法，所以无需额外的软件就能处理 JavaScript 中的 JSON。

通过 JavaScript，您可以创建一个对象数组，并像这样进行赋值：

var sites = [

{ "name":"runoob" , "url":"www.runoob.com" },

{ "name":"google" , "url":"www.google.com" },

{ "name":"微博" , "url":"www.weibo.com" }

];

可以像这样访问 JavaScript 对象数组中的第一项（索引从 0 开始）：

sites[0].name;

返回的内容是：

runoob

可以像这样修改数据：

<p>第一个网站名称: <span id=*"name1"*></span></p>

<p>第一个网站修改后的名称: <span id=*"name2"*></span></p>

<script>

**var** sites = [

{ "name":"runoob" , "url":"www.runoob.com" },

{ "name":"google" , "url":"www.google.com" },

{ "name":"微博" , "url":"www.weibo.com" }

];

document.getElementById("name1").innerHTML=sites[0].name;

// 修改网站名称

sites[0].name="菜鸟教程";

document.getElementById("name2").innerHTML=sites[0].name;

</script>

执行结果：

第一个网站名称: runoob

第一个网站修改后的名称: 菜鸟教程

## JSON文件

JSON 文件的文件类型是 ".json"

JSON 文本的 MIME 类型是 "application/json"

## JSON对象

JSON 对象使用在大括号({})中书写。对象可以包含多个 key/value（键/值）对。key 必须是字符串，value 可以是合法的 JSON 数据类型（字符串, 数字, 对象, 数组, 布尔值或 null）。key 和 value 中使用冒号(:)分割。每个 key/value 对使用逗号(,)分割。

{ "name":"runoob", "alexa":10000, "site":null }

### 访问对象值

访问对象的值可以使用点号（.）来访问：

<p id=*"demo"*></p>

<script>

**var** myObj, x;

myObj = { "name":"runoob", "alexa":10000, "site":**null** };

x = myObj.name;

document.getElementById("demo").innerHTML = x;

</script>

执行结果：

runoob

还可以使用方括号（[]）来访问对象的值：

<p id=*"demo"*></p>

<script>

**var** myObj, x;

myObj = myObj = { "name":"runoob", "alexa":10000, "site":**null** };

x = myObj["name"];

document.getElementById("demo").innerHTML = x;

</script>

执行结果：

runoob

### 循环对象

使用for-in来循环对象的属性：

<p id=*"demo"*></p>

<script>

**var** myObj = { "name":"runoob", "alexa":10000, "site":**null** };

**for** (x **in** myObj) {

document.getElementById("demo").innerHTML += x + "<br>";

}

</script>

执行结果：

name  
alexa  
site

在 for-in 循环对象的属性时，使用中括号（[]）来访问属性的值：

<p id=*"demo"*></p>

<script>

**var** myObj = { "name":"runoob", "alexa":10000, "site":**null** };

**for** (x **in** myObj) {

document.getElementById("demo").innerHTML += myObj[x] + "<br>";

}

</script>

执行结果：

runoob

10000

null

### 嵌套 JSON 对象

在JSON对象中可以包含另外的JSON对象：

myObj = {

"name":"runoob",

"alexa":10000,

"sites": {

"site1":"www.runoob.com",

"site2":"m.runoob.com",

"site3":"c.runoob.com"

}

}

可以使用点号(.)或者中括号([])来访问嵌套的 JSON 对象。

<p id=*"demo"*></p>

<script>

myObj = {

"name":"runoob",

"alexa":10000,

"sites": {

"site1":"www.runoob.com",

"site2":"m.runoob.com",

"site3":"c.runoob.com"

}

}

document.getElementById("demo").innerHTML += myObj.sites.site1 + "<br>";

// 或者

document.getElementById("demo").innerHTML += myObj.sites["site1"];

</script>

执行结果：

www.runoob.com

www.runoob.com

### 修改对象的值

可以使用点号(.)来修改 JSON 对象的值：

<p id=*"demo"*></p>

<script>

**var** myObj, i, x = "";

myObj = {

"name":"runoob",

"alexa":10000,

"sites": {

"site1":"www.runoob.com",

"site2":"m.runoob.com",

"site3":"c.runoob.com"

}

}

myObj.sites.site1 = "www.google.com";

**for** (i **in** myObj.sites) {

x += myObj.sites[i] + "<br>";

}

document.getElementById("demo").innerHTML = x;

</script>

执行结果：

www.google.com

m.runoob.com

c.runoob.com

还可以使用中括号([])来修改 JSON 对象的值：

<p id=*"demo"*></p>

<script>

**var** myObj, i, x = "";

myObj = {

"name":"runoob",

"alexa":10000,

"sites": {

"site1":"www.runoob.com",

"site2":"m.runoob.com",

"site3":"c.runoob.com"

}

}

myObj.sites["site1"] = "www.google.com";

**for** (i **in** myObj.sites) {

x += myObj.sites[i] + "<br>";

}

document.getElementById("demo").innerHTML = x;

</script>

### 删除对象属性

可以使用 delete 关键字来删除 JSON 对象的属性：

<script>

**var** myObj, i, x = "";

myObj = {

"name":"runoob",

"alexa":10000,

"sites": {

"site1":"www.runoob.com",

"site2":"m.runoob.com",

"site3":"c.runoob.com"

}

}

**delete** myObj.sites.site1;

**for** (i **in** myObj.sites) {

x += myObj.sites[i] + "<br>";

}

document.getElementById("demo").innerHTML = x;

</script>

执行结果：

m.runoob.com

c.runoob.com

还可以使用中括号([])来删除 JSON 对象的属性：

<p id=*"demo"*></p>

<script>

**var** myObj, i, x = "";

myObj = {

"name":"runoob",

"alexa":10000,

"sites": {

"site1":"www.runoob.com",

"site2":"m.runoob.com",

"site3":"c.runoob.com"

}

}

**delete** myObj.sites["site1"];

**for** (i **in** myObj.sites) {

x += myObj.sites[i] + "<br>";

}

document.getElementById("demo").innerHTML = x;

</script>

执行结果：

m.runoob.com

c.runoob.com

## JSON 数组

JSON 数组在中括号中书写。JSON 中数组值必须是合法的 JSON 数据类型（字符串, 数字, 对象, 数组, 布尔值或 null）。JavaScript 中，数组值可以是以上的 JSON 数据类型，也可以是 JavaScript 的表达式，包括函数，日期，及 undefined。

[ "Google", "Runoob", "Taobao" ]

### JSON 对象中的数组

对象属性的值可以是一个数组：

{

"name":"网站",

"num":3,

"sites":[ "Google", "Runoob", "Taobao" ]

}

可以使用索引值来访问数组：

<p id=*"demo"*></p>

<script>

**var** myObj, x;

myObj = {

"name":"网站",

"num":3,

"sites":[ "Google", "Runoob", "Taobao" ]

}

x = myObj.sites[0];

document.getElementById("demo").innerHTML = x;

</script>

执行结果：

Google

### 循环数组

可以使用 for-in 来访问数组：

<p id=*"demo"*></p>

<script>

**var** myObj, i, x = "";

myObj = {

"name":"网站",

"num":3,

"sites":[ "Google", "Runoob", "Taobao" ]

};

**for** (i **in** myObj.sites) {

x += myObj.sites[i] + "<br>";

}

document.getElementById("demo").innerHTML = x;

</script>

执行结果：

Google

Runoob

Taobao

也可以使用 for 循环：

<p id=*"demo"*></p>

<script>

**var** myObj, i, x = "";

myObj = {

"name":"网站",

"num":3,

"sites":[ "Google", "Runoob", "Taobao" ]

};

**for** (i = 0; i < myObj.sites.length; i++) {

x += myObj.sites[i] + "<br>";

}

document.getElementById("demo").innerHTML = x;

</script>

执行结果：

Google

Runoob

Taobao

### 嵌套 JSON 对象中的数组

JSON 对象中数组可以包含另外一个数组，或者另外一个 JSON 对象：

myObj = {

"name":"网站",

"num":3,

"sites": [

{ "name":"Google", "info":[ "Android", "Google 搜索", "Google 翻译" ] },

{ "name":"Runoob", "info":[ "菜鸟教程", "菜鸟工具", "菜鸟微信" ] },

{ "name":"Taobao", "info":[ "淘宝", "网购" ] }

]

}

可以使用 for-in 来循环访问每个数组：

<p id=*"demo"*></p>

<script>

**var** myObj, i, j, x = "";

myObj = {

"name":"网站",

"num":3,

"sites": [

{ "name":"Google", "info":[ "Android", "Google 搜索", "Google 翻译" ] },

{ "name":"Runoob", "info":[ "菜鸟教程", "菜鸟工具", "菜鸟微信" ] },

{ "name":"Taobao", "info":[ "淘宝", "网购" ] }

]

}

**for** (i **in** myObj.sites) {

x += "<h1>" + myObj.sites[i].name + "</h1>";

**for** (j **in** myObj.sites[i].info) {

x += myObj.sites[i].info[j] + "<br>";

}

}

document.getElementById("demo").innerHTML = x;

</script>

执行结果：

**Google**

Android

Google 搜索

Google 翻译

**Runoob**

菜鸟教程

菜鸟工具

菜鸟微信

**Taobao**

淘宝

网购

### 修改数组值

可以使用索引值来修改数组值：

<p id=*"demo"*></p>

<script>

**var** myObj, i, x = "";

myObj = {

"name":"网站",

"num":3,

"sites":[ "Google", "Runoob", "Taobao" ]

};

myObj.sites[1] = "Github";

**for** (i **in** myObj.sites) {

x += myObj.sites[i] + "<br>";

}

document.getElementById("demo").innerHTML = x;

</script>

执行结果：

Google

Github

Taobao

### 删除数组元素

可以使用 delete 关键字来删除数组元素：

<p id=*"demo"*></p>

<script>

**var** myObj, i, x = "";

myObj = {

"name":"网站",

"num":3,

"sites":[ "Google", "Runoob", "Taobao" ]

};

**delete** myObj.sites[1];

**for** (i **in** myObj.sites) {

x += myObj.sites[i] + "<br>";

}

document.getElementById("demo").innerHTML = x;

</script>

执行结果：

Google

Taobao

## JSON.parse()

JSON 通常用于与服务端交换数据。在接收服务器数据时一般是字符串。我们可以使用 JSON.parse() 方法将数据转换为 JavaScript 对象。

### 语法

JSON.parse(text[, reviver])

参数说明：

1. text:必需， 一个有效的 JSON 字符串。
2. reviver: 可选，一个转换结果的函数， 将为对象的每个成员调用此函数。

### JSON 解析实例

例如我们从服务器接收了以下数据：

{ "name":"runoob", "alexa":10000, "site":"www.runoob.com" }

我们使用 JSON.parse() 方法处理以上数据，将其转换为 JavaScript 对象：

**var** obj = JSON.parse('{ "name":"runoob", "alexa":10000, "site":"www.runoob.com" }');

解析完成后，我们就可以在网页上使用 JSON 数据了：

<p id=*"demo"*></p>

<script>

**var** obj = JSON.parse('{ "name":"runoob", "alexa":10000, "site":"www.runoob.com" }');

document.getElementById("demo").innerHTML = obj.name + "：" + obj.site;

</script>

执行结果：

runoob：www.runoob.com

### 从服务端接收 JSON 数据

我们可以使用 AJAX 从服务器请求 JSON 数据，并解析为 JavaScript 对象。

<p id=*"demo"*></p>

<script>

**var** xmlhttp = **new** XMLHttpRequest();

xmlhttp.onreadystatechange = **function**() {

**if** (**this**.readyState == 4 && **this**.status == 200) {

myObj = JSON.parse(**this**.responseText);

document.getElementById("demo").innerHTML = myObj.name;

}

};

xmlhttp.open("GET", "/try/ajax/json\_demo.txt", **true**);

xmlhttp.send();

</script>

执行结果：

网站

### 从服务端接收数组的 JSON 数据

如果从服务端接收的是数组的 JSON 数据，则 JSON.parse 会将其转换为 JavaScript 数组：

<p id=*"demo"*></p>

<script>

**var** xmlhttp = **new** XMLHttpRequest();

xmlhttp.onreadystatechange = **function**() {

**if** (**this**.readyState == 4 && **this**.status == 200) {

myArr = JSON.parse(**this**.responseText);

document.getElementById("demo").innerHTML = myArr[1];

}

};

xmlhttp.open("GET", "/try/ajax/json\_demo\_array.txt", **true**);

xmlhttp.send();

</script>

执行结果：

Runoob

### 异常

#### 解析数据

JSON 不能存储 Date 对象。如果你需要存储 Date 对象，需要将其转换为字符串。之后再将字符串转换为 Date 对象。

<p id=*"demo"*></p>

<script>

**var** text = '{ "name":"Runoob", "initDate":"2013-12-14", "site":"www.runoob.com"}';

**var** obj = JSON.parse(text);

obj.initDate = **new** Date(obj.initDate);

document.getElementById("demo").innerHTML = obj.name + "创建日期: " + obj.initDate;

</script>

执行结果：

Runoob创建日期：Sat Dec 14 2013 08:00:00 GMT+0800 (中国标准时间)

我们可以启用 JSON.parse 的第二个参数 reviver，一个转换结果的函数，对象的每个成员调用此函数。

<p id=*"demo"*></p>

<script>

**var** text = '{ "name":"Runoob", "initDate":"2013-12-14", "site":"www.runoob.com"}';

**var** obj = JSON.parse(text, **function** (key, value) {

**if** (key == "initDate") {

**return** **new** Date(value);

} **else** {

**return** value;

}});

document.getElementById("demo").innerHTML = obj.name + "创建日期：" + obj.initDate;

</script>

执行结果：

Runoob创建日期：Sat Dec 14 2013 08:00:00 GMT+0800 (中国标准时间)

#### 解析函数

JSON 不允许包含函数，但你可以将函数作为字符串存储，之后再将字符串转换为函数。

<p id=*"demo"*></p>

<script>

**var** text = '{ "name":"Runoob", "alexa":"function () {return 10000;}", "site":"www.runoob.com"}';

**var** obj = JSON.parse(text);

obj.alexa = eval("(" + obj.alexa + ")");

document.getElementById("demo").innerHTML = obj.name + " Alexa 排名：" + obj.alexa();

</script>

执行结果：

Runoob Alexa 排名：10000

## JSON.stringify()

JSON 通常用于与服务端交换数据。在向服务器发送数据时一般是字符串。我们可以使用 JSON.stringify() 方法将 JavaScript 对象转换为字符串。

### 语法

JSON.stringify(value[, replacer[, space]])

参数说明：

1. value:

必需， 一个有效的 JSON 字符串。

1. replacer:

可选。用于转换结果的函数或数组。

如果 replacer 为函数，则 JSON.stringify 将调用该函数，并传入每个成员的键和值。使用返回值而不是原始值。如果此函数返回 undefined，则排除成员。根对象的键是一个空字符串：""。

如果 replacer 是一个数组，则仅转换该数组中具有键值的成员。成员的转换顺序与键在数组中的顺序一样。当 value 参数也为数组时，将忽略 replacer 数组。

1. space:

可选，文本添加缩进、空格和换行符，如果 space 是一个数字，则返回值文本在每个级别缩进指定数目的空格，如果 space 大于 10，则文本缩进 10 个空格。space 有可以使用非数字，如：\t。

### JavaScript 对象转换

例如我们向服务器发送以下数据：

**var** obj = { "name":"runoob", "alexa":10000, "site":"www.runoob.com"};

使用 JSON.stringify() 方法处理以上数据，将其转换为字符串：

**var** myJSON = JSON.stringify(obj);

myJSON 为字符串。我们可以将 myJSON 发送到服务器：

<p id=*"demo"*></p>

<script>

**var** obj = { "name":"runoob", "alexa":10000, "site":"www.runoob.com"};

**var** myJSON = JSON.stringify(obj);

document.getElementById("demo").innerHTML = myJSON;

</script>

执行结果：

{"name":"runoob","alexa":10000,"site":"www.runoob.com"}

### JavaScript 数组转换

我们也可以将 JavaScript 数组转换为 JSON 字符串：

**var** arr = [ "Google", "Runoob", "Taobao", "Facebook" ];

**var** myJSON = JSON.stringify(arr);

myJSON 为字符串。我们可以将 myJSON 发送到服务器：

<p id=*"demo"*></p>

<script>

**var** arr = [ "Google", "Runoob", "Taobao", "Facebook" ];

**var** myJSON = JSON.stringify(arr);

document.getElementById("demo").innerHTML = myJSON;

</script>

执行结果：

["Google","Runoob","Taobao","Facebook"]

### 异常

解析数据

JSON 不能存储 Date 对象。JSON.stringify() 会将所有日期转换为字符串。之后你可以再将字符串转换为 Date 对象。

<p id=*"demo"*></p>

<script>

**var** obj = { "name":"Runoob", "initDate":**new** Date(), "site":"www.runoob.com"};

**var** myJSON = JSON.stringify(obj);

document.getElementById("demo").innerHTML = myJSON;

</script>

执行结果：

{"name":"Runoob","initDate":"2017-11-07T10:37:33.972Z","site":"www.runoob.com"}

### 解析函数

JSON 不允许包含函数，JSON.stringify() 会删除 JavaScript 对象的函数，包括 key 和 value。

<p id=*"demo"*></p>

<script>

**var** obj = { "name":"Runoob", "alexa":**function** () {**return** 10000;}, "site":"www.runoob.com"};

**var** myJSON = JSON.stringify(obj);

document.getElementById("demo").innerHTML = myJSON;

</script>

执行结果：

{"name":"Runoob","site":"www.runoob.com"}

我们可以在执行 JSON.stringify() 函数前将函数转换为字符串来避免以上问题的发生：

<p id=*"demo"*></p>

<script>

**var** obj = { "name":"Runoob", "alexa":**function** () {**return** 10000;}, "site":"www.runoob.com"};

obj.alexa = obj.alexa.toString();

**var** myJSON = JSON.stringify(obj);

document.getElementById("demo").innerHTML = myJSON;

</script>

执行结果：

{"name":"Runoob","alexa":"function () {return 10000;}","site":"www.runoob.com"}

不建议在 JSON 中使用函数。

## JSON使用

### Servlet中使用JSON向客户端传递数据

#### 手动编写JSON格式

在Servlet中想将数据通过JSON格式传递给客户端，可以将数据手动编写成JSON对应的格式再通过输出流的方式输出给客户端。

@WebServlet("/showClassGradeByGradeId")

**public** **class** ShowClassGradeByGradeId **extends** HttpServlet {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

/\*\*

\* **@see** HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)

\*/

**protected** **void** doGet(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {

String id = request.getParameter("id");

**if**(**null** != id){

//根据客户端提交的id值查询出结果集合

IClassGradeService icgs = **new** ClassGradeServiceImpl();

List<ClassGrade> lists = icgs.getClassGradeByGradeId(Integer.*valueOf*(id));

//将结果集合中的数据循环遍历，生成JSON格式的字符串

String mass = "{\"classGrade\":[";

response.setContentType("application/json; charset=utf-8"); //设置JSON的MIME类型

**if**(!lists.isEmpty()){

**for**(**int** i = 0; i < lists.size(); i++){

**if**((i + 1) == lists.size()){

mass += "{\"name\":\"" + lists.get(i).getName() + "\",\"id\":\"" + lists.get(i).getId() + "\"}";

}**else**{

mass += "{\"name\":\"" + lists.get(i).getName() + "\",\"id\":\"" + lists.get(i).getId() + "\"},";

}

}

mass += "]}";

}**else**{

mass = "{\"type\" : \"no\"}";

System.***out***.println("没有对应的数据");

}

//获取输出流，将JSON格式的字符串输出给浏览器

PrintWriter out = response.getWriter();

out.write(mass);

}

}

/\*\*

\* **@see** HttpServlet#doPost(HttpServletRequest request, HttpServletResponse response)

\*/

**protected** **void** doPost(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {

// **TODO** Auto-generated method stub

doGet(request, response);

}

}

客户端直接通过JS代码或者jquery代码获取

jquery使用Ajax获取服务端提交的JSON

**function** change(){

$.ajax({

type: "POST",

url: "showClassGradeByGradeId",

data: "id=" + $(".grade").val(),

success: **function**(msg){

**var** str="<option value=\"no\">---------请选择班级---------</option>";

**for**(**var** i = 0; i < msg.classGrade.length; i++){

str += "<option value=\"" + msg.classGrade[i].id + "\">" + msg.classGrade[i].name + "</option>";

}

$(".classes").html(str);

}

});

}

#### **使用第三方jar包实现对象与JSON之间的互转**。

1. **Json-lib.jar包**

Json-lib 是一个 Java 类库（官网：http://json-lib.sourceforge.net/）可以实现如下功能：

1. 转换 javabeans, maps, collections, java arrays 和 XML 成为 json 格式数据
2. 转换 json 格式数据成为 javabeans 对象
3. **Json-lib 需要的 jar 包：**
4. commons-beanutils-1.8.3.jar
5. commons-collections-3.2.2.jar
6. commons-lang-2.6.jar
7. commons-logging-1.2.jar
8. ezmorph-1.0.6.jar
9. json-lib-2.4-jdk15.jar
10. **Json-lib 的使用**
11. 将 Array 解析成 Json 串。使用 JSONArray.formObject(Object obj)可以解析 Array 类型：

**public** **class** JsonTest {

**public** **static** **void** main(String[] args) {

//将字符串数组解析成为JSON对象

String[] str = { "Jack", "Tom", "90", "true" };

JSONArray json = JSONArray.*fromObject*(str);

System.***err***.println(json);

//将对象数组解析成为JSON对象

Object[] o = { "北京", "上海", 89, **true**, 90.87 };

json = JSONArray.*fromObject*(o);

System.***err***.println(json);

//将String类型的集合解析成为JSON对象

List<String> list = **new** ArrayList<String>();

list.add("Jack");

list.add("Rose");

json = JSONArray.*fromObject*(list);

System.***err***.println(json);

//将Object类型的集合解析成为JSON对象

Set<Object> set = **new** HashSet<Object>();

set.add("Hello");

set.add(**true**);

set.add(99);

json = JSONArray.*fromObject*(set);

System.***err***.println(json);

}

}

运行结果：

["Jack","Tom","90","true"]

["北京","上海",89,true,90.87]

["Jack","Rose"]

[99,"Hello",true]

### 把 JSON 文本转换为 JavaScript 对象

# Ajax技术