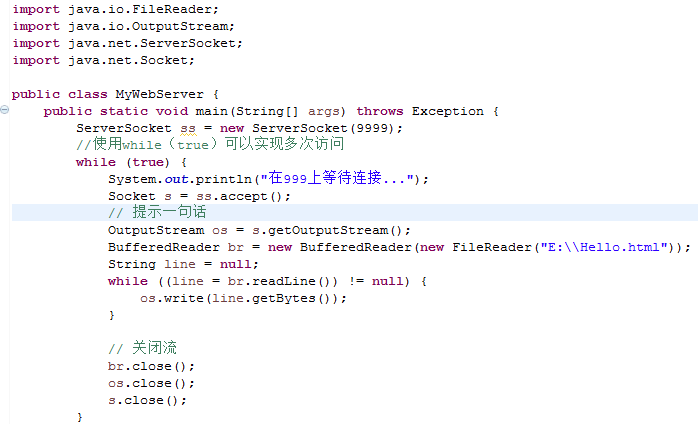
Tomcat 是Web应用服务器,是一个Servlet/JSP容器. Tomcat 作为Servlet容器,负责处理客户请求,把请求传送给Servlet,并将Servlet的响应传送回给客户.而Servlet是一种运行在支持Java语言的服务器上的组件. Servlet最常见的用途是扩展Java Web服务器功能,提供非常安全的,可移植的。

<http://www.cnblogs.com/zhaoyl/archive/2012/10/10/2718575.html>

 Web服务器的基本功能就是提供Web信息浏览服务。它只需支持HTTP协议、HTML文档格式及URL。应用程序服务器(简称应用服务器)，我们先看一下微软对它的定义："我们把应用程序服务器定义为“作为服务器执行共享业务应用程序的底层的系统软件”。 就像文件服务器为很多用户提供文件一样，应用程序服务器让多个用户可以同时使用应用程序（通常是客户创建的应用程序）"

模拟一个web服务器：



## URIs, URLs, and URNs

首先，URI，是uniform resource identifier，统一资源标识符，用来唯一的标识一个资源。而URL是uniform resource locator，统一资源定位器，它是一种具体的URI，即URL可以用来标识一个资源，而且还指明了如何locate这个资源。而URN，uniform resource name，统一资源命名，是通过名字来标识资源，比如mailto:java-net@java.sun.com。也就是说，URI是以一种抽象的，高层次概念定义统一资源标识，而URL和URN则是具体的资源标识的方式。URL和URN都是一种URI。

在Java的URI中，一个URI实例可以代表绝对的，也可以是相对的，只要它符合URI的语法规则。而URL类则不仅符合语义，还包含了定位该资源的信息，因此它不能是相对的，schema必须被指定。

ok，现在回答文章开头提出的问题，到底是imgUrl好呢，还是imgUri好？显然，如果说imgUri是肯定没问题的，因为即使它实际上是url，那它也是uri的一种。那么用imgUrl有没有问题呢？此时则要看它的可能取值，如果是绝对路径，能够定位的，那么用imgUrl是没问题的，而如果是相对路径，那还是不要用ImgUrl的好。总之，用imgUri是肯定没问题的，而用imgUrl则要视实际情况而定。

第二个，从HttpServletRequest的javadoc中可以看出，getRequestURI返回一个String，“the part of this request’s URL from the protocol name up to the query string in the first line of the HTTP request”，比如“POST /some/path.html?a=b HTTP/1.1”，则返回的值为”/some/path.html”。现在可以明白为什么是getRequestURI而不是getRequestURL了，因为此处返回的是相对的路径。而getRequestURL返回一个StringBuffer，“The returned URL contains a protocol, server name, port number, and server path, but it does not include query string parameters.”，完整的请求资源路径，不包括querystring。

总结一下：URL是一种具体的URI，它不仅唯一标识资源，而且还提供了定位该资源的信息。URI是一种语义上的抽象概念，可以是绝对的，也可以是相对的，而URL则必须提供足够的信息来定位，所以，是绝对的，而通常说的relative URL，则是针对另一个absolute URL，本质上还是绝对的。

注：这里的绝对(absolute)是指包含scheme，而相对(relative)则不包含scheme。

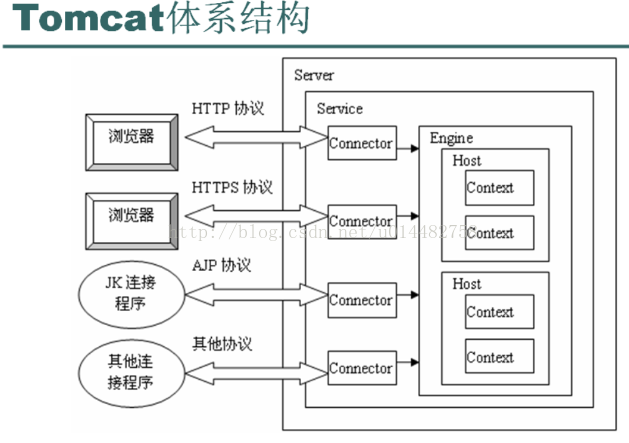


## 浏览器访问web原理

**输入域名时，首先去本机hosts文件中查找域名-IP的对应关系。如果没有查到，则去电信或者联通等dns服务器上查找对应关系。查找到之后，则去寻找IP地址对应的主机上的服务器，此处是tomcat。然后在tomcat服务器上进行解析，首先解析主机名www.wy.com(因为tomcat服务器可以管理多个主机)，然后解析web应用和资源名（html，jsp。。）解析完毕之后，tomcat服务器会获取相应的资源，然后将html界面返回给浏览器。**



## Tomcat体系结构



（1）Server   
Server表示整个的Catalina Servlet容器。Tomcat提供了Server接口的一个默认实现，这通常不需要用户自己去实现。在Server容器中，可以包含一个或多个Service组件。

（2）Service   
Service是存活在Server内部的中间组件，它将一个或多个连接器（Connector）组件绑定到一个单独的引擎（Engine）上。在Server中，可以包含一个或多个Service组件。多个不同的Connector可以共享同一个Engine。同一个Server中的多个Service之间没有相关性

Service也很少由用户定制，Tomcat提供了Service接口的默认实现，而这种实现既简单又能满足应用。

（3）Connector   
连接器（Connector）处理与客户端的通信，它负责接收客户请求，以及向客户返回响应结果。在Tomcat中，有多个连接器可以使用。

（4）Engine   
在Tomcat中，每个Service只能包含一个Servlet引擎（Engine）。引擎表示一个特定的Service的请求处理流水线。作为一个Service可以有多个连接器，引擎从连接器接收和处理所有的请求，将响应返回给适合的连接器，通过连接器传输给用户。用户允许通过实现Engine接口提供自定义的引擎，但通常不需要这么做。

（5）Host   
Host表示一个虚拟主机，一个引擎可以包含多个Host。注意Engine中必须有一个Host的名字和Engine的defaultHost属性匹配。用户通常不需要创建自定义的Host，因为Tomcat给出的Host接口的实现（类StandardHost）提供了重要的附加功能。

（6）Context   
一个Context表示了一个Web应用程序，运行在特定的虚拟主机中。什么是Web应用程序呢？在Sun公司发布的Java Servlet规范中，对Web应用程序做出了如下的定义：“一个Web应用程序是由一组Servlet、HTML页面、类，以及其他的资源组成的运行在Web服务器上的完整的应用程序。它可以在多个供应商提供的实现了Servlet规范的Web容器中运行”。一个Host可以包含多个Context（代表Web应用程序），每一个Context都有一个唯一的路径（Context Path）。用户通常不需要创建自定义的Context，因为Tomcat给出的Context接口的实现（类StandardContext）提供了重要的附加功能。

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[链接地址](javascript:goto();)\*\*\*\*   
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.2001-10-04.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

当直接通过IP地址访问网站时：通过<http://127.0.0.1>究竟访问的是哪个网站？

如何配置默认主机:

在tomcat/conf/server.xml 文件

<Engine name="Catalina" defaultHost="主机名">

## Tomcat Server处理一个HTTP请求的过程

1、用户点击网页内容，请求被发送到本机端口8080，被在那里监听的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Engine的回应。   
3、Engine获得请求localhost/test/index.jsp，匹配所有的虚拟主机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），名为localhost的Host获得请求/test/index.jsp，匹配它所拥有的所有的Context。Host匹配到路径为/test的Context（如果匹配不到就把该请求交给路径名为“ ”的Context去处理）。   
5、path=“/test”的Context获得请求/index.jsp，在它的mapping table中寻找出对应的Servlet。Context匹配到URL PATTERN为\*.jsp的Servlet,对应于JspServlet类。   
6、构造HttpServletRequest对象和HttpServletResponse对象，作为参数调用JspServlet的doGet（）或doPost（）.执行业务逻辑、数据存储等程序。   
7、Context把执行完之后的HttpServletResponse对象返回给Host。   
8、Host把HttpServletResponse对象返回给Engine。   
9、Engine把HttpServletResponse对象返回Connector。   
10、Connector把HttpServletResponse对象返回给客户Browser

## Servlet特点

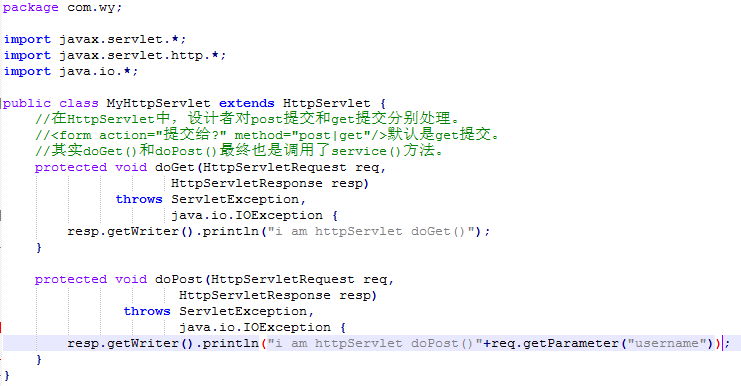
1. servlet 其实就是java程序(java类)
2. 该 java 程序(java 类)要遵循servlet开发规范
3. serlvet是运行在服务端，由服务器端（tomcat）调用和执行。
4. serlvet 功能强大,几乎可以完成网站的所有功能
5. 是学习jsp基础

## 开发servlet三种方法

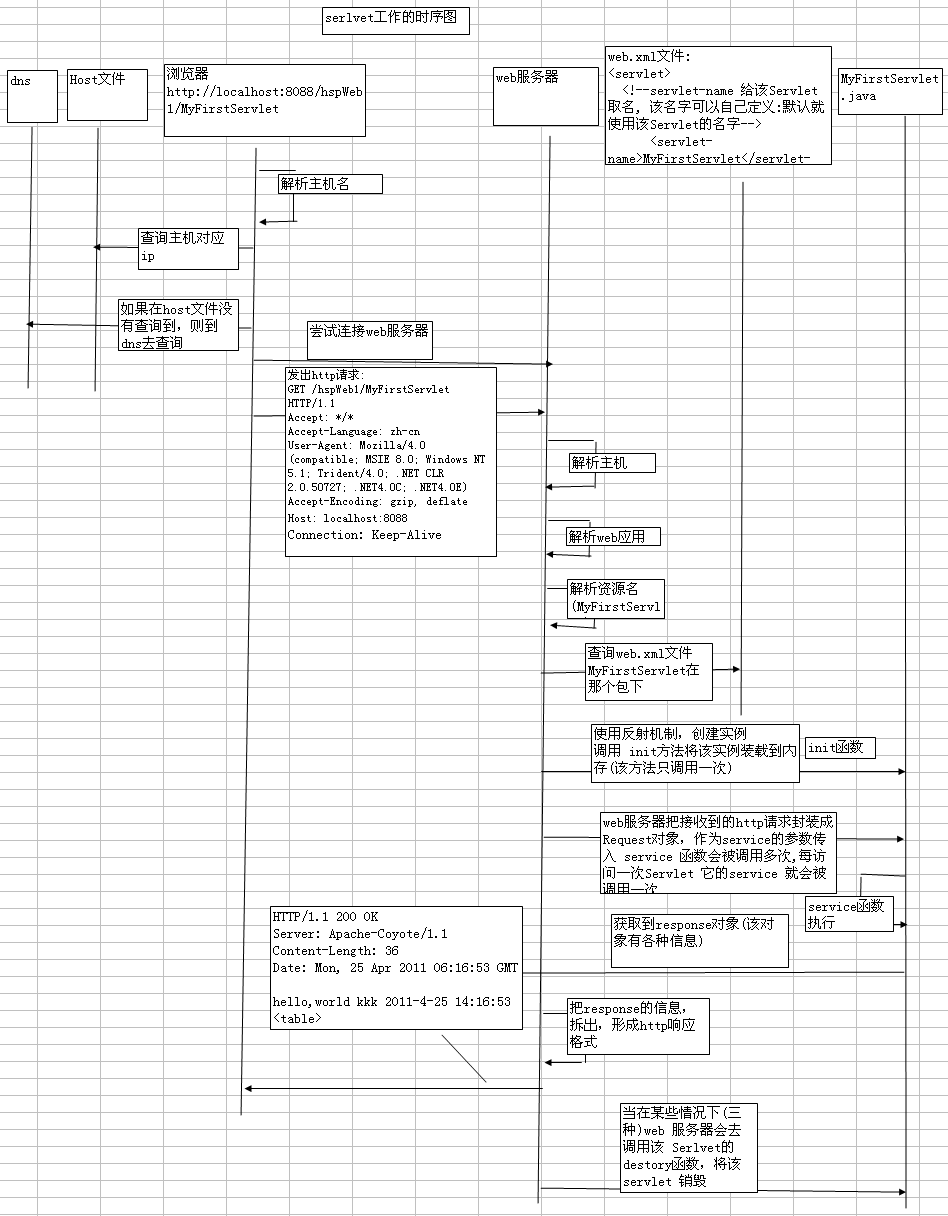
1. 实现 Servlet接口
2. 通过继承 GenericServlet
3. 通过继承 HttpServlet







## Servlet生命周期



面试题: 请简述servlet的生命周期(工作流程) 答:

标准版本：Servlet服务器是由WEB服务器调用，WEB服务器收到客户端的Servlet访问请求后，

**①**WEB服务器首先会检查是否已经装载并创建了该servlet实例对象。如果是直接进行第④步，否则执行第②步。

**②**装载并创建该Servlet的一个实例对象。

**③**调用Servlet实例对象的init()方法。（init方法在服务器装载servlet时执行，负责初始化实例对象）

**④**创建一个用于封装HTTP请求消息的HttpServletRequest对象和一个代表HTTP响应消息的HttpServletResponse对象，然后调用service()方法并将请求和响应作为参数传递进去。

1. WEB应用被停止或重启之前，Servlet引擎将卸载Servlet，在卸载之前调用Servlet的destroy()方法

以下情况会导致destroy（）方法调用：

1 tomcat重启 2 reload该webapp 3 重启电脑

## 创建Servlet对象的时机

1. Servlet容器启动时：读取web.xml配置文件中的信息，构造指定的Servlet对象，创建ServletConfig对象，同时将ServletConfig对象作为参数来调用Servlet对象的init方法。
2. 在Servlet容器启动后：客户首次向Servlet发出请求，Servlet容器会判断内存中是否存在指定的Servlet对象，如果没有则创建它，然后根据客户的请求创建HttpRequest、HttpResponse对象，从而调用Servlet 对象的service方法。
3. Servlet Servlet容器在启动时自动创建Servlet，这是由在web.xml文件中为Servlet设置的<load-on-startup>属性决定的。从中我们也能看到同一个类型的Servlet对象在Servlet容器中以单例的形式存在。

<servlet>

<servlet-name>Init</servlet-name>

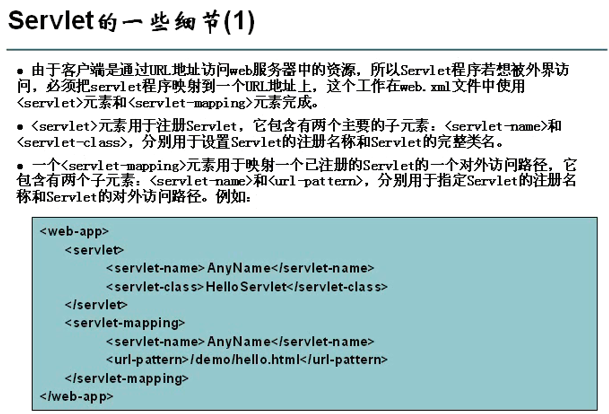
<servlet-class>org.xl.servlet.InitServlet</servlet-class>

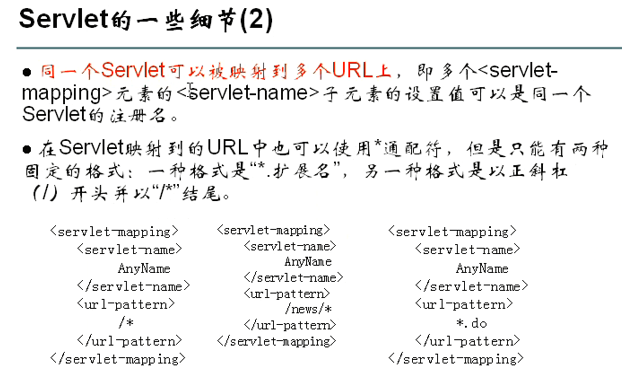
<load-on-startup>1</load-on-startup>

</servlet>

## get 提交 和 post的提交的区别

1. 从安全看 get<post GET请求的数据会附在URL之后（就是把数据放置在HTTP协议头中），以?分割URL和传输数据，参数之间以&相连 POST把提交的数据则放置在是HTTP包的包体中
2. 从提交内容看 get<post get 一般不要大于2k, post理论上无限制，但是在实际 开发中，建议不要大于64k
3. 从速度看 get>post.get要求服务器立即处理请求，而post请求可能形成一个队列请求。
4. Get可以保留uri中的参数，利于收藏





## Servlet的映射路径：

　　　　　　　　url　　　　　　浏览器输入

精确匹配：　　/first　　　　　　http：//localhost:8080/day10\_web/first

　　　　　　/maodoer/first　　http://localhost:8080/day10\_web/maodoer/first

模糊匹配：　　/\*　　　　　　　　http://localhost:8080/day10\_web/任意字符

　　　　　　/itcast/\*　　　　　　http://localhost:8080/day10\_web/itcast/任意字符

　　　　　　\*.后缀名　　　　　　http://localhost:8080/day10\_web/任意路径.后缀名

　　　　　　\*.html(伪静态)　　　http://http://localhost:8080/day10\_web/ergrep.html

注意：

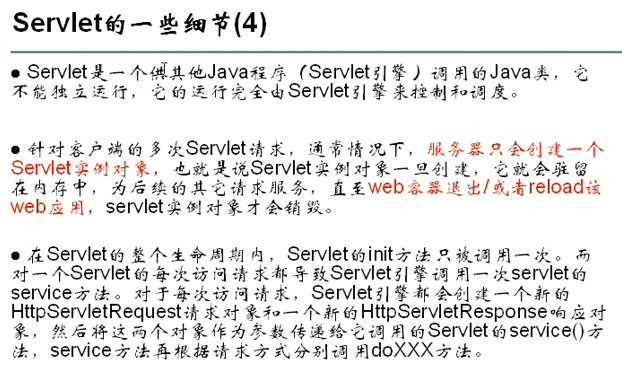
1）url-pattern一定以/开头或者以\*开头。

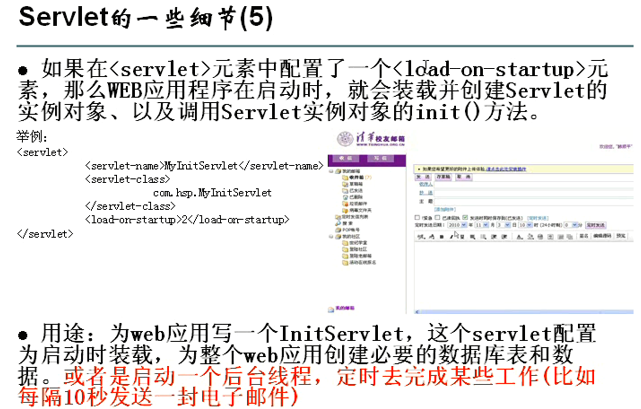
2）不能同时使用两种模糊匹配，例如/itcast/\*.do

3）当输入的url有多个servlet同时被匹配的情况下：

　　3.1精确匹配优先（长的最像优先被匹配）

　　3.2以后缀名结尾的模糊匹配级优先级最低



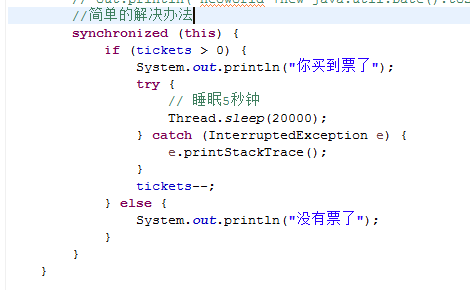


## Servlet单例问题

当Servlet被第1次访问后，就被加载到内存，以后该实例对各个请求服务.即在使用中是单例.

因为 Servlet是单例，因此会出现线程安全问题: 比如:

售票系统. 如果不加同步机制，则会出现问题:



这里我给大家一个原则:

1. 如果一个变量需要多个用户共享，则应当在访问该变量的时候，加同步机制

synchronized (对象){

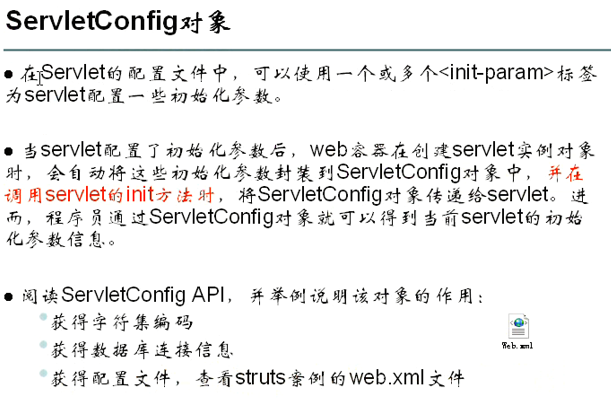
//同步代码

}

（2）如果一个变量不需要共享，则直接在 doGet() 或者 doPost()定义.这样不会存在线程安全问题

## ServletConfig

该对象主要用于读取 servlet的配置信息.

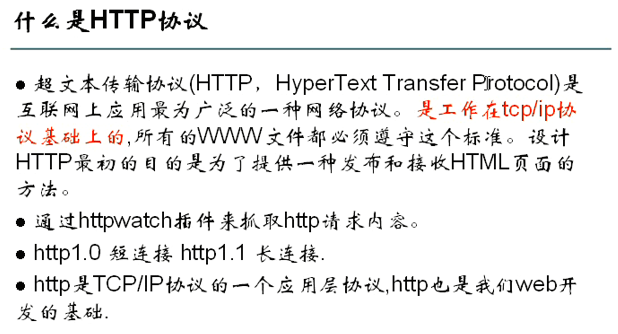




String encoding=this.getServletConfig().getInitParameter("encoding");

getServletConfig（）方法是在GenericServlet里面定义的。

## http协议



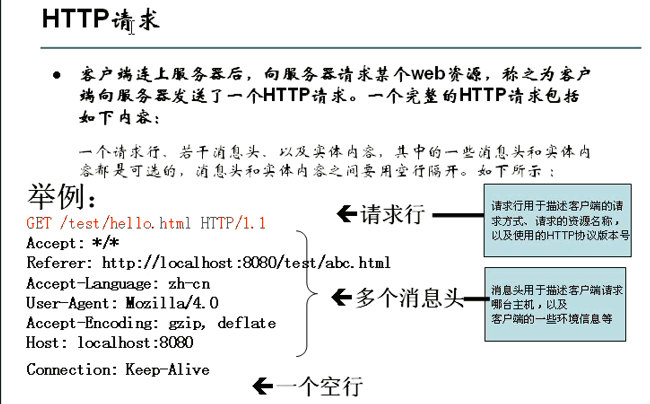
http1.0 称为短连接

http1.1 称为长连接.

所谓长和短指的是持续时间的。

长连接：是保存一段时间的连接，而不是数据发送完毕就立即断掉。

短连接：是发送完数据就断掉。





GET /test/hello.html HTTP/1.1 [**请求行**]

**Accept:， \*/\* [消息名] 消息名：内容**

**Referer: 。**[**http://localhost:8080/test/abc.html**](http://localhost:8080/test/abc.html)

**Accept-Language: zh-cn**

**User-Agent: Mozilla/4.0**

**Accept-Encoding: gzip, deflate**

**Host: http://www.sohu.com:80**

**Connection: Keep-Alive** [**消息头格式** （消息名: 内容 ）

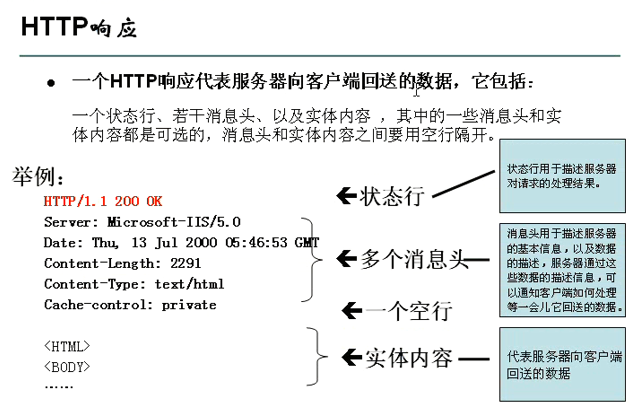
**特别说明**： 并不是每一次请求的消息头都一样.]

空行

发送的内容 [格式 : 内容名字=内容体]

请求方式

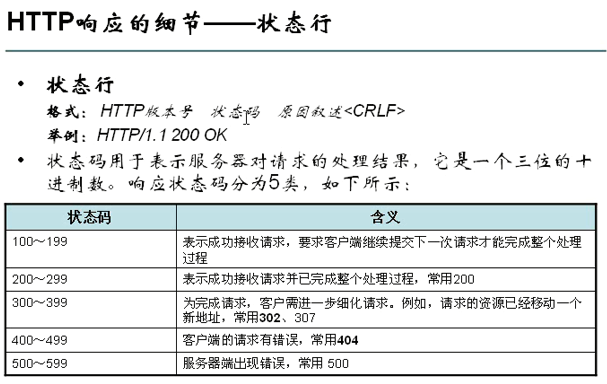
## http响应



response.setStatus(302);

response.setHeader("Location", "/servletPro/Servlet2");

// 上面两句话等价 response.sendRedirect("/servletPro/Servlet2");



http响应消息头详解

* + **Location**: <http://www.baidu.org/index.jsp> 【让浏览器重新定位到url】
  + **Server**:apache tomcat 【告诉浏览器我是tomcat】
  + **Content-Encoding**: gzip 【告诉浏览器我使用 gzip】
  + **Content-Length**: 80 【告诉浏览器会送的数据大小80节】
  + **Content-Language**: zh-cn 【支持中文】
  + **Content-Type**: text/html; charset=GB2312 [内容格式text/html; 编码gab2312]
  + **Last-Modified**: Tue, 11 Jul 2000 18:23:51 GMT 【告诉浏览器，该资源上次更新时间】
  + **Refresh**: 1;url=http://www.baidu.com 【过多久去，刷新到 http://www.baidu.com】
  + **Content-Disposition**: attachment; filename=aaa.zip 【告诉浏览器，有文件下载】
  + **Transfer-Encoding**: chunked 校验盒 [传输的编码]
  + **Set-Cookie**:SS=Q0=5Lb\_nQ; path=/search[后面详讲]
  + **Expires**: -1[告诉浏览器如何缓存页面**IE**]
  + **Cache-Control**: no-cache [告诉浏览器如何缓存页面**火狐**]
  + **Pragma**: no-cache [告诉浏览器如何缓存页面]
  + **Connection**: close/Keep-Alive **[保持连接 1.1是Keep-Alive]**
  + **Date**: Tue, 11 Jul 2000 18:23:51 GMT

**定时刷新Refresh使用（在ServletHttpServlet下）**



**文件下载 Content-Disposition**



**缓存讲解**

提出问题：浏

默认情况下，会缓存我们的页面，这样出现一个问题：如果我们的用户习惯把光标停留在地址栏，然后回车来取页面，就会默认调用cache中取数据。

有些网站要求及时性很高（如股票，基金），因此要求我们不缓存页面

代码：

//指定该页面不缓存 Ie

response.setDateHeader("Expires", -1);【针对IE浏览器设置不缓存】

//为了保证兼容性.

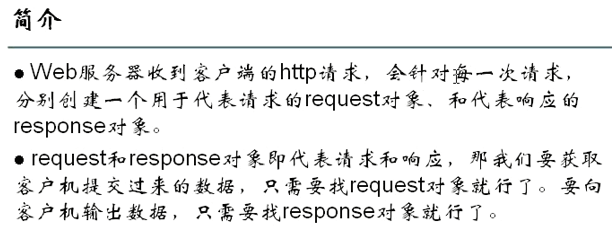
response.setHeader("Cache-Control", "no-cache");【针对火狐浏览器等】

response.setHeader("Pragma", "no-cache");//【其他浏览器】

有些网站要求网页缓存一定时间,比如缓存一天

response.setDateHeader("Expires", System.currentTimeMillis()+3600\*1000\*24);后面一个参数表示设置的缓存保持时间

## HttpServletResponse





HttpServletResponse的再说明

getWriter()

getOutputStream();

区别

1. getWriter() 用于向客户机回送**字符数据**
2. getOutputStream() 返回的对象，可以回送**字符数据**，也可以回送**字节数据**(二进制数据)

OutputStream os=response.getOutputStream();

os.write("hello,world".getBytes());

如何选择:

如果我们是回送**字符数据**，则使用 PrintWriter对象 ,效率高

如果我们是回送**字节数据**(binary date) ,则只能使用 OutputStream

☞ 这两个流不能同时使用.

比如：

OutputStream os=response.getOutputStream();

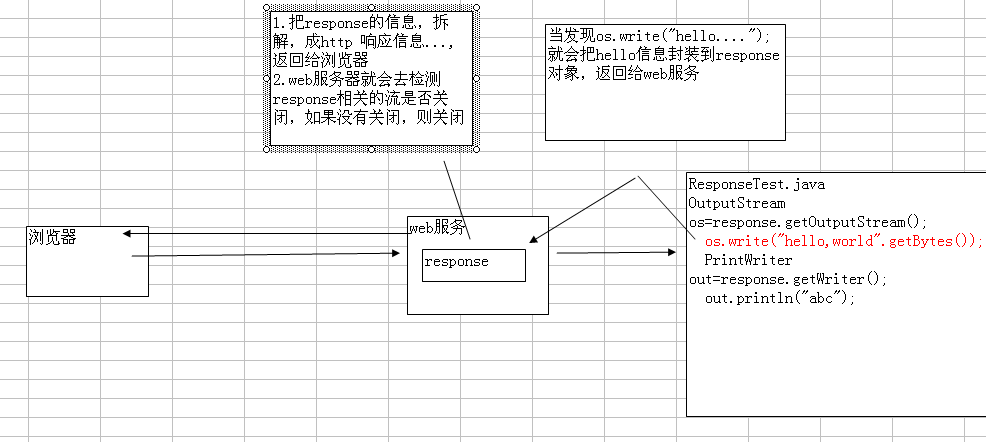
os.write("hello,world".getBytes());

PrintWriter out=response.getWriter();

out.println("abc");

就会报错:

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



**Web服务器会自动检查并关闭流**

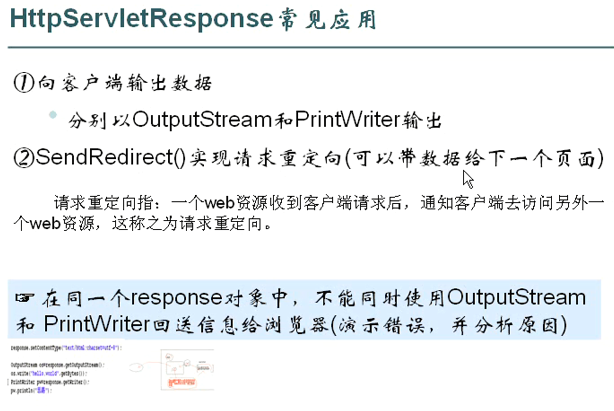
response细节

1. getOutStream和getWriter方法分别用于得到输出二进制数据、文本数据的ServletOutputStream、PrintWriter对象。

2. 在同一个response对象中getOutputStream和getWriter这两个方法互相排斥，调用其中一个方法后，就不能再调用另外一个方法。

3. Servlet程序向ServletOutputSteam或PrintWriter对象中写入的数据将被Servlet引擎从response里面获取，Servlet引擎将这些数据当作相应消息的正文，然后再与相应状态行和各相应头组合后输出到客户端。

4. Servlet的service方法结束后，Servlet引擎将检查getWriter或getOutputSteam方法返回的输出流对象是否已经调用close方法，如果没有，Servlet引擎将调用之关闭输出流对象。





**response.sendRedirect(“servlet的地址?参数名=参数值&参数名=参数值...”);**

**String 参数=request.getParameter(“参数名”);**

使用session()来传递字符参数和对象

A.传递字符串 //3. 使用session来传递数据给下个页面

放入session request.getSession ().setAttribute("loginUser",username);

取出session 在JSP中通过session取出 request.getSession.getAttribute("loginUser");

B．传递对象

## 中文乱码处理

1. 表单form
2. post

**在服务器端设置成浏览器端的编码方式。**

解决方法: **request.setCharacterEncoding("utf-8"); //gbk gb2312 big5**

1. get

写一个工具类:

**package** com.hsp.utils;

**public** **class** MyTools {

**public** **static** String getNewString(String str) {

String newString="";

**try** {

// newString=**new** String(str.getBytes("iso-8859-1"),"utf-8"); //中文传到浏览器是乱码

newString=**new** String(str.getBytes("iso-8859-1"),"gb2312");

} **catch** (Exception e) {

e.printStackTrace();

// 把iso-8859-1 转换成 utf-8

}

**return** newString;

}

}

1. 超链接 **本质是get提交**
2. sendRedirect() 发生乱码

response.sendRedirect(“servlet地址?username=顺平”);

session是不会主动产生乱码问题的。

**☞版本低导致的乱码**

**特别说明**，如果你的浏览器是 ie6 或以下版本，则我们的 ② 和 ③中情况会出现乱码(当中文是奇数的时候)

解决方法是 ：

String info=java.net.URLEncoder.encode("你好吗.jpg", "utf-8");

<a href=”<http://www.sohu.com?name=>”+ info >测试</a>

response.sendRedirect(“servlet地址?username=”+info);

**说明: 我们应当尽量使用post 方式提交;**

**返回浏览器显示乱码**

在服务端是中文，在response的时候，也要考虑浏览器显示是否正确,一般我们通过

response.setContentType(“text/html;charset=utf-8”); ok

**下载提示框中文乱码**

补充一个知识点: 当我们下载文件的时候，可能提示框是中文乱码

String temp=java.net.URLEncoder.*encode*("传奇.mp3","utf-8");

response.setHeader("Content-Disposition","attachment; filename="+temp);

**String temp=java.net.URLEncoder.encode(“传奇.mp3”,”utf-8”); //统一资源定位**

**Response.setHeader(“Content-Disposition”,”attachment;filename=”+temp);**

一、Servlet输出乱码

1. 用servlet.getOutStream字节流输出中文，假设要输出的是String str ="钓鱼岛是中国的，无耻才是日本的"。

1.1 若是本地服务器与本地客户端这种就不用说了，直接可以out.write(str.getBytes())可以输出没有问题。因为服务器中用str.getBytes()是采用默认本地的编码，比如GBK。而浏览器也解析时也用本地默认编码，两者是统一的，所以没有问题。

1.1 若服务器输出时用了, out.write(str.getBytes("utf-8"))。而本地默认编码是GBK时(比例在中国)，那么用浏览器打开时就会乱码。因为服务器发送过来的是utf-8的1010数据，而客户端浏览器用了gbk来解码，两者编码不统一，肯定是乱码。当然，你也可以自己将客户端浏览器的编码手工调用下(IE菜单是：查询View->编码encoding->utf-8)，但是这种操作很烂，最好由服务器输出响应头告诉，浏览器用哪种编码来解码。所以要在服务器的servlet中，增加response.setHeader("content-type","text/html;charset=utf-8")，当然也可直接用简单的response.setContentType("text/hmtl;charset=utf-8")。两种的操作是一样一样的。

2. 用servlet.getWirter字符流输出中文，假设要输出的是String str ="钓鱼岛是中国的，无耻才是日本的"。

2.1 若写成out.print(str)输出时，客户端浏览器显示的将全是多个?????的字符，代表在编码表中肯定就找不到相应的字符来显示。原因是：servlet.getWriter()得到的字符输出流，默认对字符的输出是采用ISO-8859-1，而ISO-8859-1肯定是不支持中文的。所以肯定要首先要做的第一件事：是要将服务器对象输出字符能支持中文的。其次服务器向客户端写回的响应头要告诉客户端是用了哪种编码表进行编码的。而实现这两个需求，只需要response.setContentType("text/hmtl;charset=utf-8")。就搞定了。特别注意：response.setContentType("text/html;charset=utf-8")要放在PrintOut out = response.getWriter()代码的前面，否则只是有告诉客户端用什么码表编码的功能，而服务器端还是用ISO-8859-1编码了。再特别提示下：在同一Servlet中的doGet或doPost方法中，不能既用response.getOutputStream又用response.getWriter，因为这两种response的响应输出字节流与字符流是冲突的，只能用其一。

二、Servlet文件下载，中文乱码情况。

关键是下载时响应头 content-disposition中attachment;filename=文件名。这个文件名filename不能是含有中文字符串的，要用URLEncoding编码进行编码，才能进行进行http的传输。如下代码示例：



三、Servlet的response增加addCookie，cookie中value的中文码问题解决方法。

关于cookie的原理，见<http://blog.csdn.net/chenshufei2/article/details/8009992>。 若想将cookie中存放中文的值，必须用Base64编码后，发给客户浏览器端进入存储。而下次客户端浏览访问是带回来的cookie中的值，是经过Base64编码的，所以需要用Base64解码即可。 Base64编码主要是解决将特殊字符进行重新编码，编码成a-b、A-B、0-9、+与/，字符52，10个数字与一个+，一个/ 共64个字符。它的原理是将原来3个字节的内容编码成4个字节。主要是取字节的6位后，在前面补00组成一个新的字节。所以这样原来的3个字节共24，被编码成4个字节32位了。

具体代码示例如下：



四、获取请求参数乱码

GET方式的乱码：

如<a href=”/demo5/servlet/RD2?name=中国”>CN</a>，直接用request.getParameter得到的字符串strCN将会乱码，这也是因为GET方式是用http的url传过来的默认用iso-8859-1编码的，所以首先得到的strCn要再用iso-8859-1编码得到原文后，再进行用utf-8(看具体页面的charset是什么utf-8或gbk)进行解码即可。new String(strCn.getBytes(“ISO-8859-1”),“UTF-8”);

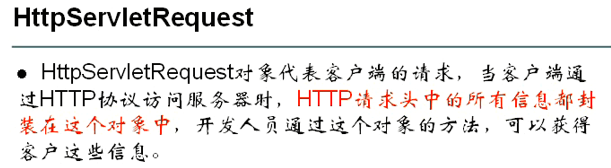


这种方式操作比较麻烦的是，有一个参数要用iso-8859-1编码一次再解码一次。

POST方式的乱码：只需要request.setCharacterEncoding("UTF-8"):即可。



## HttpServletRequest详解1

 HttpServletRequest对象的详解 该对象表示浏览器的请求(http请求), 当web 服务器得到该请求后，会把请求信息封装成一个HttpServletRequest 对象

* + getRequestURL方法返回客户端发出请求时的完整URL。
  + getRequestURI方法返回请求行中的资源名部分。
  + **getQueryString** 方法返回**请求行中的参数**部分(参数名+值)。

该函数可以获取请求部分的数据 比如 <http://localhost/web>名?username=abc&pwd=123

**request.getQueryString(); 就会得到 username=abc&pwd=123**

getRemoteAddr方法返回发出**请求的客户机的IP**地址。（可以对其进行封杀）

getRemoteHost方法返回发出请求的客户机的完整主机名 如果客户机没有在dns上注册(一般上网，要提供一个域名让别人去访问) 则返回ip地址

**pw.println("你的IP="+req.getRemoteAddr()+"<br>");**

**pw.println("你的机器名是"+req.getRemoteHost()+"<br>");**

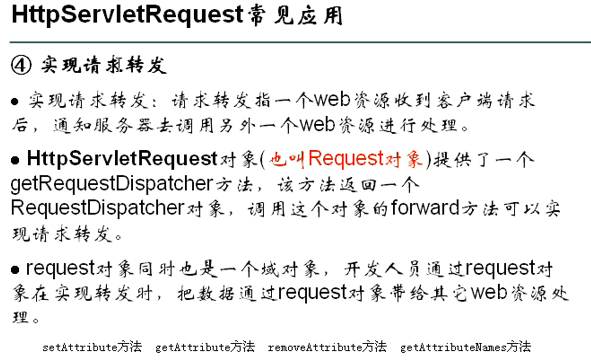
getRemotePort方法**返回客户机所使用的网络端口号**

**客户机的端口号是随机选择的，web服务器的端口号是一定的。**

getLocalPort方法返回web服务器所使用的网络端口号

getLocalAddr方法返回WEB服务器的IP地址。

getLocalName方法返回WEB服务器的主机名



请求转发requeset.getRequestDispatcher(资源地址).forward(request,response);

资源地址：不需要项目名。因为它只是在WEB服务器内部转发。

## sendRedirect() 和 forward 的区别

1. 叫法: sendRedirect() 重定向,转发 forward() 叫转向
2. 实际发生的位置不一样: sendRedirect 发生 浏览器; forward 发生 web服务器.
3. 用法不一样

request.getRequestDispatcher(“/资源URI”).forward(request,response)

response.sendRedirect(“/web应用/资源URI”);

1. 能够去URL 范围不一样

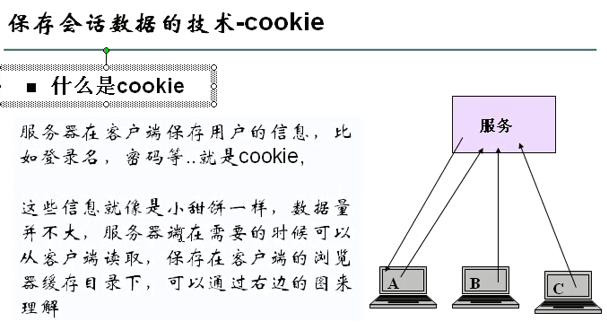
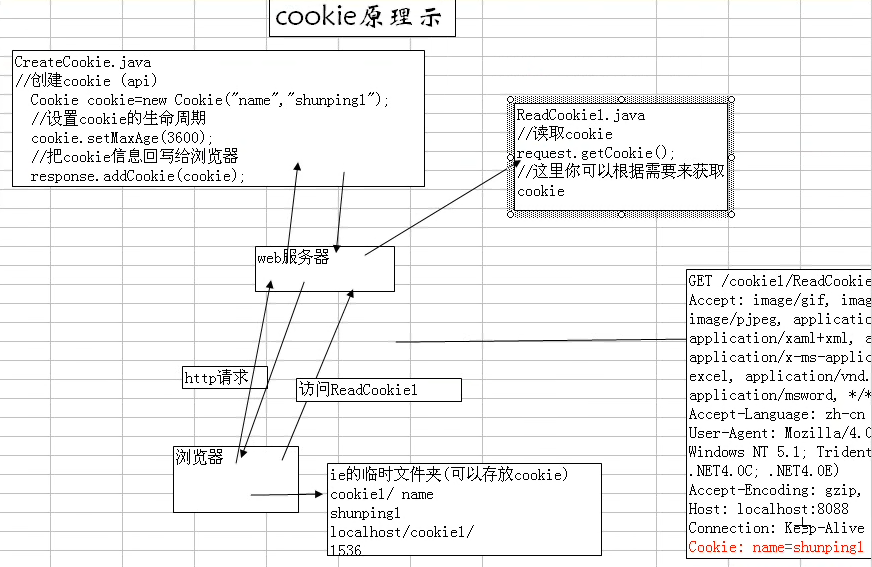
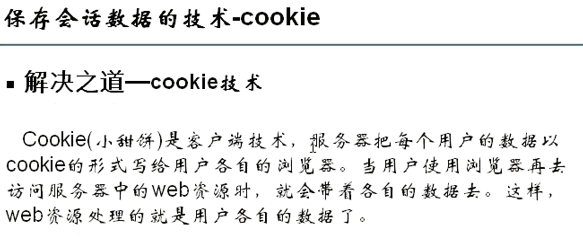
sendRedirect 可以去 外边URL; forward 只能去当前的WEB应用的资源

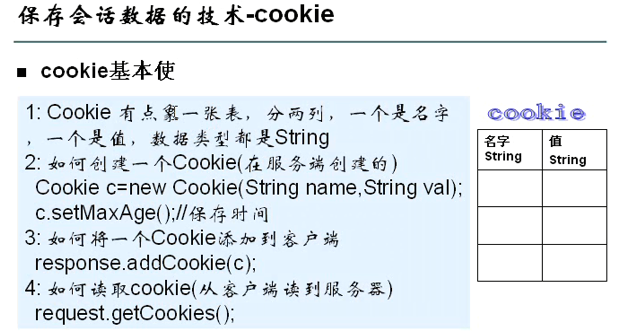


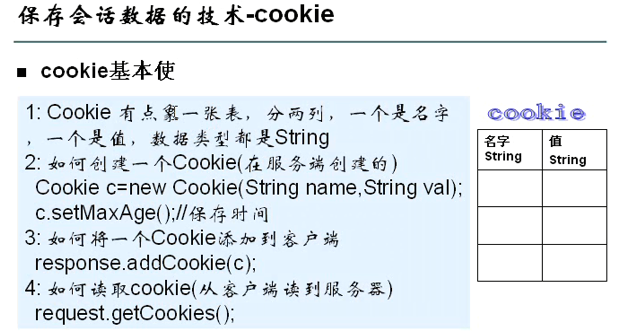
## Servlet单实例多线程

Servlet是单实例的，对于同一种业务请求只有一个实例，不同的业务请求可以通过分发来产生多个实例。为什么单实例可以处理多个请求？因为是单实例多线程的。

## Cookie







Cookie的小结

1. Cookie是在服务器端创建的。
2. Cookie是保存在浏览器这一端的。
3. Cookie的生命周期可以通过cookie.setMaxAge(2000); 设置。

如果不设置，该cookie的生命周期当浏览器关闭时就消亡。

1. Cookie是可以被多个浏览器共享的。
2. 怎么理解 我们可以把cookie想成一张表：

**如果cookie重名会有什么问题？ 重名会替换原来的值。**

**在浏览器端的临时文件夹里保存cookie信息，文件夹cookie1，是web应用的名字。**

1. 一个站点可以保存多个cookie 一般不超过20个.
2. **Cookie存放时是以明文方式存放的，因此安全性比较低。可以通过加密后再保存。**

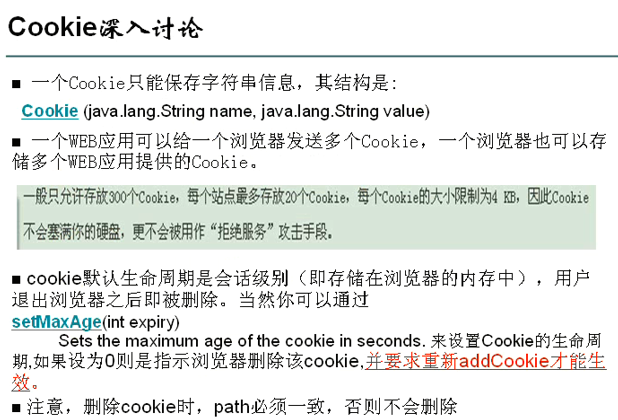
**-->补讲一个md5算法：密码都要加密存放，对用户输入的密码要进行MD5加密，再存放到数据库中去**。

## Md5算法

 MD5，全名Message Digest Algorithm 5 ，中文名为消息摘要算法第五版，为计算机安全领域广泛使用的一种散列函数，用以提供消息的完整性保护。上面这段话话引用自百度百科，我的理解MD5是一种信息摘要算法，主要是通过特定的hash散列方法将文本信息转换成简短的信息摘要，压缩+加密+hash算法的结合体，是绝对不可逆的。

MD5以512位分组来处理输入的信息，且每一分组又被划分为16个32位子分组，经过了一系列的处理后，算法的输出由四个32位分组组成，将这四个32位分组级联后将生成一个128位散列值。

MD5算法是单向散列算法的一种。单向散列算法也称为HASH算法，是一种将任意长度的信息压缩至某一固定长度（称之为消息摘要）的函数(该压缩过程不可逆)。Hash函数可用于数字签名、信息完整性检查等用途。常见的散列算法还有SHA、RIPE-MD、HAVAL、N-Hash等。



**setMaxAge(负数),相当于该cookie的生命周期是会话级的。**

1. Cookie存放中文怎么处理

存放时：

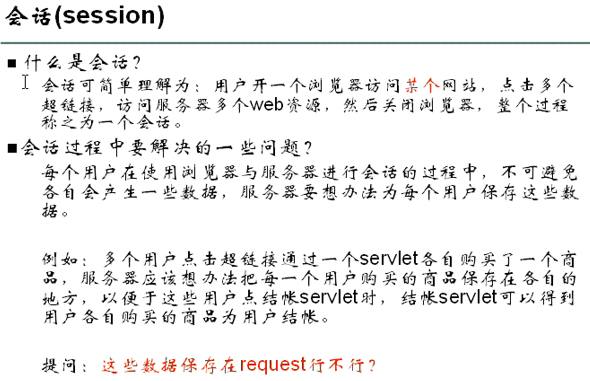
String val = java.net.URLEncoder.encode("顺平","UTF-8");

Cookie cookie = new Cookie("uname",val);

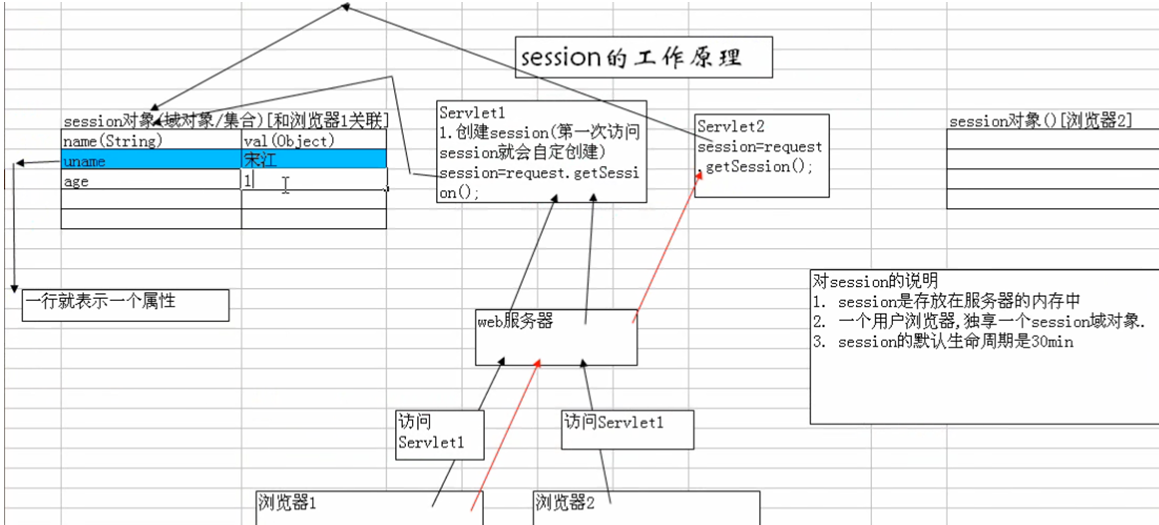
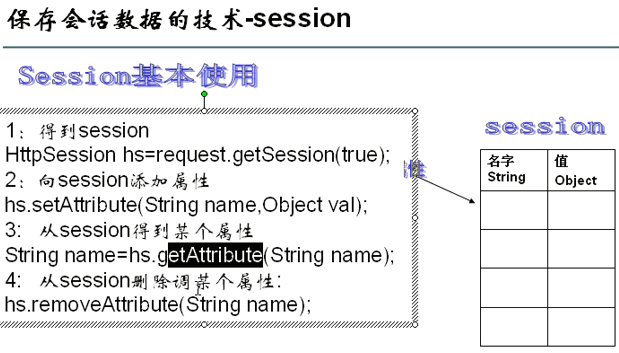
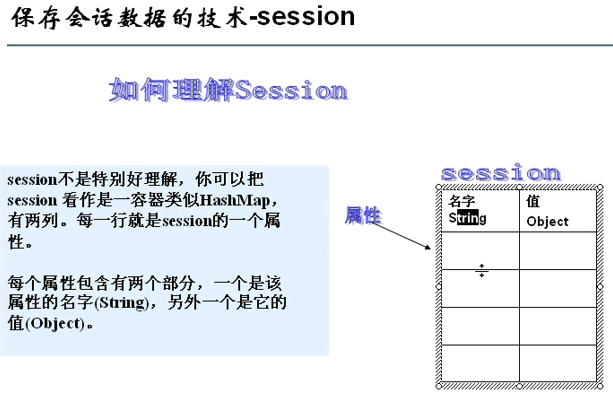
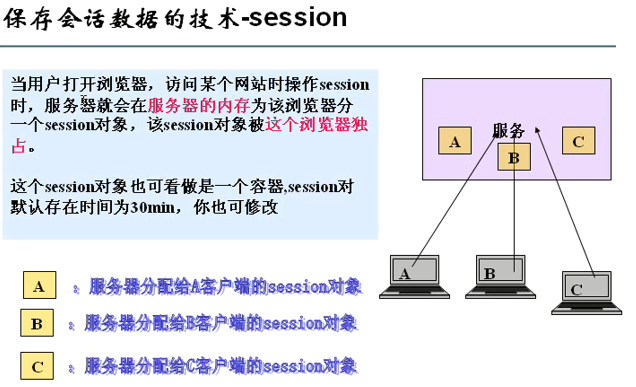
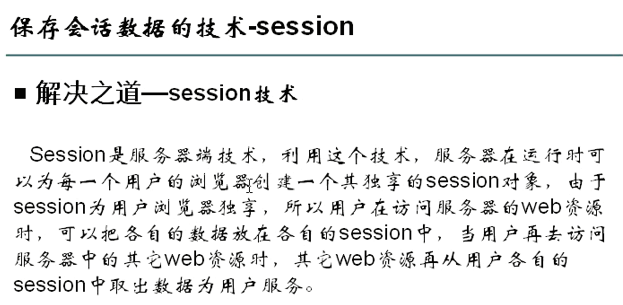
取出时：

String val = java.net.URLDecoder.*decode*(ckie.getValue(),"utf-8");

## Session



会话是针对一个网站而言。

 Session小结：

1. Session是存在服务器的内存中的。
2. 一个用户浏览器，独享一个session对象。
3. Session中的属性的默认生命周期是30min,可以通过修改web.xml来修改。

这样修改

一个地方是：tomcat/conf/web.xml

<session-config>

<session-timeout>30</session-timeout>

</session-config>

对所有的web应用生效。

第二个地方是：在单个web应用下的web.xml文件下添加或修改ession-config

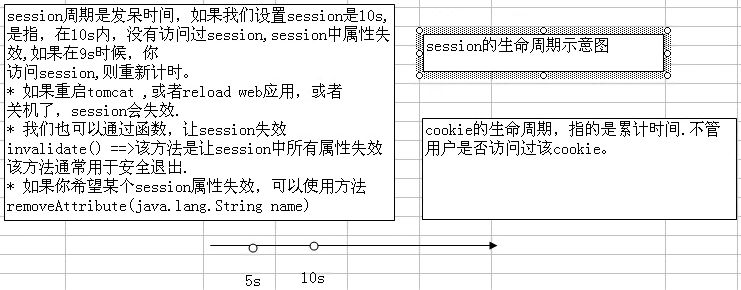
<session-config>

<session-timeout>10</session-timeout>

</session-config>

只对本web应用生效

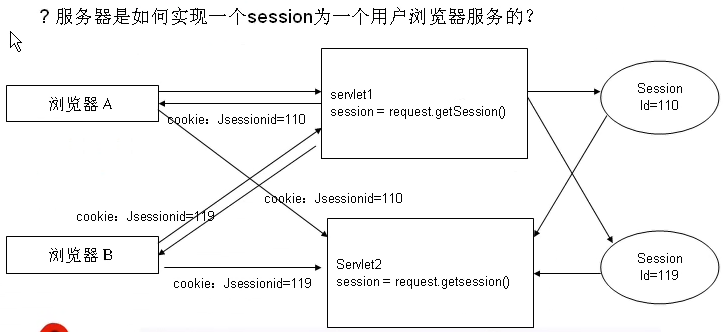
第三种方法是：session.setMaxInactiveInterval(600); //以秒为单位, 指的是发呆时间---连续不被使用的时间。如果在这个时间内被访问，则重新计时，但是若只要超过这个时间却没有被访问，session一个生命周期就结束。



Session中可以存放多个属性。

Session可以存放对象。

**如果同一个用户浏览器向Session设置属性时名字相同会替换**。



http请求通过session和cookie保持会话。

Session生成时机：request对象调用getSession()方法时生成，服务器会为该session对象生成一个唯一的ID。

服务器端响应客户端请求时会在报头中设置set-cookie属性，该属性内容中有一个JSESSIONID即是session对象的标识，返回后由浏览器进行处理。

客户端再次发送请求时，浏览器会在报文头中自动追加cookie属性，该属性将jsessionid传到服务器端。在服务器端用request.getSession时会取得sessionid对应的对象而不会重新生成session。

如果客户端禁用了cookie，可以使用url地址重写的方法，把jsessionid放在url地址的后面，这样下次访问url时，就会通过这个jsessionid寻找session对象

1. session在何时被创建  
   一个常见的误解是以为session在有客户端访问时就被创建，然而事实是直到某server端程序调用HttpServletRequest.getSession(true)这样的语句时才被创建，注意如果JSP没有显示的使用 关闭session，则JSP文件在编译成Servlet时将会自动加上这样一条语句HttpSession session = HttpServletRequest.getSession(true);这也是JSP中隐含的session对象的来历。

由于session会消耗内存资源，因此，如果不打算使用session，应该在所有的JSP中关闭它。

1. ession在下列情况下被删除a.程序调用HttpSession.invalidate();或b.距离上一次收到客户端发送的session id时间间隔超过了session的超时设置;或c.服务器进程被停止（非持久session）
2. 3、如何做到在浏览器关闭时删除session  
   严格的讲，做不到这一点。可以做一点努力的办法是在所有的客户端页面里使用javascript代码window.oncolose来监视浏览器的关闭动作，然后向服务器发送一个请求来删除session。但是对于浏览器崩溃或者强行杀死进程这些非常规手段仍然无能为力。
3. 有个HttpSessionListener是怎么回事  
   你可以创建这样的listener去监控session的创建和销毁事件，使得在发生这样的事件时你可以做一些相应的工作。注意是session的创建和销毁动作触发listener，而不是相反。类似的与HttpSession有关的listener还有HttpSessionBindingListener，HttpSessionActivationListener和HttpSessionAttributeListener。
4. 存放在session中的对象必须是可序列化的吗  
   不是必需的。要求对象可序列化只是为了session能够在集群中被复制或者能够持久保存或者在必要时server能够暂时把session交换出内存。在Weblogic Server的session中放置一个不可序列化的对象在控制台上会收到一个警告。我所用过的某个iPlanet版本如果session中有不可序列化的对象，在session销毁时会有一个Exception，很奇怪。
5. 如何才能正确的应付客户端禁止cookie的可能性  
   对所有的URL使用URL重写，包括超链接，form的action，和重定向的URL，具体做法参见[6]  
   <http://e-docs.bea.com/wls/docs70/webapp/sessions.html#100770>
6. 开两个浏览器窗口访问应用程序会使用同一个session还是不同的session  
   参见第三小节对cookie的讨论，对session来说是只认id不认人，因此不同的浏览器，不同的窗口打开方式以及不同的cookie存储方式都会对这个问题的答案有影响。

## 跨应用程序的session共享

<http://blog.csdn.net/lingdushanke/article/details/5589056>

http://blog.csdn.net/vacblog/article/details/45044709

一个大项目被分割成若干小项目开发，为了能够互不干扰，要求每个小项目作为一个单独的web应用程序开发，可是到了最后突然发现某几个小项目之间需要共享一些信息，或者想使用session来实现SSO(single sign on)，在session中保存login的用户信息，最自然的要求是应用程序间能够访问彼此的session。  
  
然而按照Servlet 规范，session的作用范围应该仅仅限于当前应用程序下，不同的应用程序之间是不能够互相访问对方的session的。各个应用服务器从实际效果上都遵守了这一规范，但是实现的细节却可能各有不同，因此解决跨应用程序session共享的方法也各不相同。

首先来看一下Tomcat是如何实现web应用程序之间session的隔离的，从Tomcat设置的cookie路径来看，它对不同的应用程序设置的cookie路径是不同的，这样不同的应用程序所用的session id是不同的，因此即使在同一个浏览器窗口里访问不同的应用程序，发送给服务器的session id也可以是不同的。

**面试题：（关掉IE再打开IE上次购买的商品还在。-->涉及session的销毁时间）**

分析

我们的session生命周期是30min,该session不会随着浏览器关闭而销毁。而会到30分钟满后才会被服务器销毁。

销毁: 当发生以下三种情况时 session对象被销毁

                     (1)  用户距离上次发出请求的时间大于session的存活时间(session.setMaxInactiveInterval(interval))

                     (2)  调用session的 invalidate()方法

                     (3)  服务器关闭

这是由于session是基于cookie实现的,session在被创建后,会生成一个唯一的sessionid返回给浏览器 并保存在浏览器的cookie中,

接下来客户再次调用getsession(),服务器便会从客户端发送过来的cookie中查找name为sessionid的cookie是否存在, 若是存在 则通过该cookie的值来找到用户之前创建的session,若是不存在则创建一个新的session.然而保存sessionid的cookie默认是会话级别的,是保存在浏览器的内存中的,当浏览器关闭时这个cookie也就消失了。这个时候的cookie是会话级别的。所以再次打开一个新的浏览器 由于这个时候并不存在名为sessionid的cookie,所以服务器便会创建一个新的session,但是原来的session还是存在的!也就是说 这时候服务器中一共存在两个session

## 过滤器

实现方法：

1. 创建一个过滤器（继承Httpservlet 实现Filter接口）
2. 配置过滤器，默认不生效必须配置

**<filter>**

**<filter-name>MyFilter</filter-name>**

**<filter-class>org.gust.controller.filter.MyFilter</filter-class>**

**</filter>**

**<filter-mapping>**

**<filter-name>MyFilter</filter-name>**

**<url-pattern>/\*</url-pattern> // /\*表示所有网页都过滤**

**</filter-mapping>**

1. **在filter中的方法添加逻辑**

**public class MyFilter extends HttpServlet implements Filter {**

**@Override**

**protected void doGet(HttpServletRequest request, HttpServletResponse response)**

**throws ServletException, IOException {**

**processRequest(request, response);**

**}**

**@Override**

**protected void doPost(HttpServletRequest request, HttpServletResponse response)**

**throws ServletException, IOException {**

**processRequest(request, response);**

**}**

**private void processRequest(HttpServletRequest request, HttpServletResponse response) {**

**}**

**@Override**

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

**}**

**@Override**

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

**HttpServletRequest hsr = (HttpServletRequest) request;**

**//看看请求的资源是什么**

**String url = hsr.getRequestURI();**

**if (url.startsWith("/UserManager/CreateCode") || url.startsWith("/UserManager/Login") || url.startsWith("/UserManager/imgs")) {**

**//放行**

**chain.doFilter(request, response);**

**} else {**

**HttpSession session = hsr.getSession();**

**Users u = (Users) session.getAttribute("u");**

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

**chain.doFilter(request, response);**

**} else {**

**request.setAttribute("err", "请登录");**

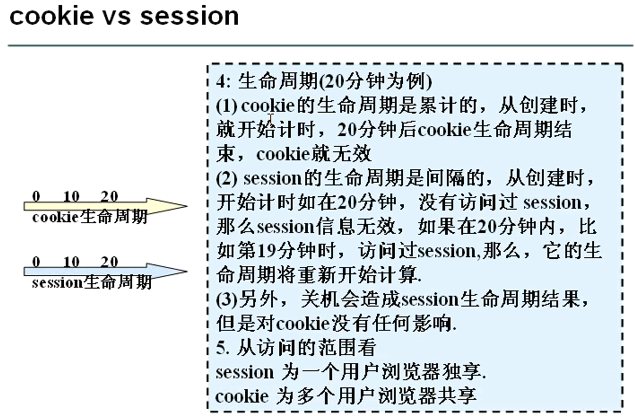
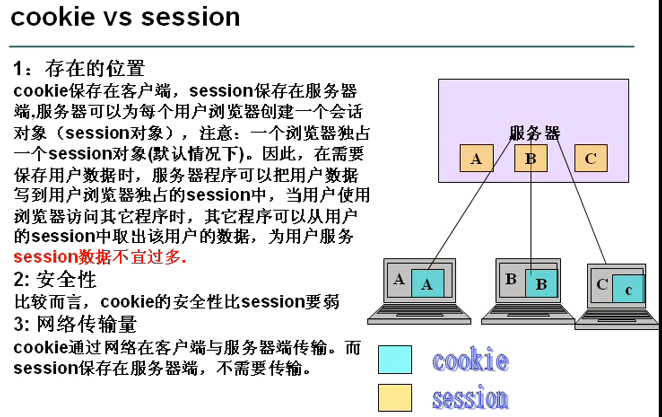
**request.getRequestDispatcher("/Login").forward(request, response);**

**}**

**}**

**}**

**}**

 Cookie vs Session

1. 存在的位置

Cookie存在客户端的临时文件夹

Session存在在服务器内存中,一个session域对象为一个用户浏览器服务。

1. 安全性

Cookie是以明文方式存放在客户端的，所以说安全性相对较弱.可以MD5加密再存放。

Session是存放服务器内存中的，安全性相对较强。

1. 网络传输量

Cookie会传递信息给服务器

Session属性值不会传递给客户端。

1. 生命周期

Cookie的生命周期是累积时间，即到点就失效。即我们给cookie设置setAge(30); 30秒后及失效。

Session 的声明周期间隔时间，即从最后一次访问后开始计时。即我们设置Session为20分钟，如果20分钟内没有访问，Session即失效。

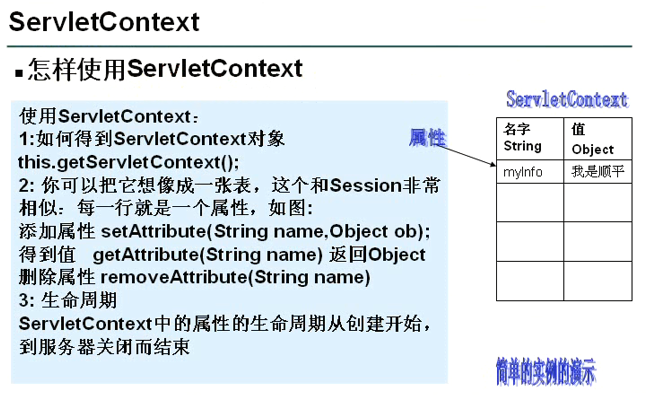
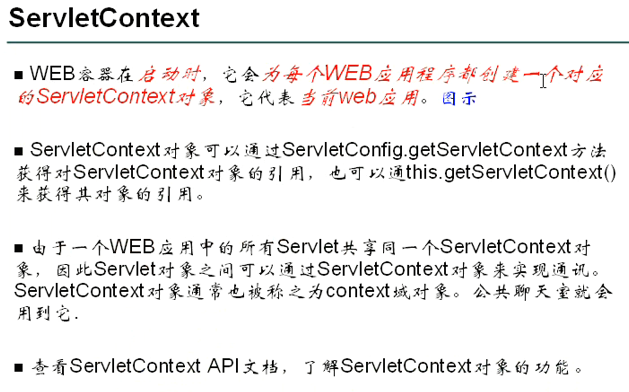
以下情况Session也会失效

1. 关闭tomcat
2. 重启web应用
3. 时间到
4. .调用session.invalidate();
5. 使用原则

因为Session会占用服务器内存，因此不要往session中存放过多过大的对象。

每个站点最多20个cookie 每个cookie大小限制在4K以内。

## ServletContext

1.  1. ServletContext 是在服务器创建
2. ServletContext被所有客户端共享
3. ServletContoext 当web应用启动时自动创建，
4. ServletContext 当web应用关闭 重启动 或服务器关闭时都造成ServletContext销毁

对ServletContext的用法小结：

//获取ServletContext的两种方法

this.getServletContext(); 或者 this.getServletConfig().ServletContext();

//添加属性

servletcontext.setAttribute(String , object);

//取出属性

servletcontext.getAttribute("属性名");

//删除

servletcontext.removeAttribute("属性名");

## 获取web应用的初始化参数

<servlet-name>Servlet2</servlet-name>

<servlet-class>com.hsp.Servlet2</servlet-class>

<!--如果这样配置， 相当于只能给该servlet使用 -->

<init-param>

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

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

</init-param>

</servlet>

</servlet>

<!-- 如果希望所有的Servlet都可以访问该配置-->

<context-param>

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

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

</context-param>

<servlet-mapping>

如何获取

String val=**this**.getServletContext().getInitParameter("name");

## 使用ServletContext实现跳转

目前跳转到页面有几种方法

1. response.sendRedirect("/web应用名/资源URL");

2. request.getRequestDispatcher("/资源URL").forward(request,response);

**区别** 1): getRequestDispatcher跳转发生在服务器而sendRedirect跳转发生在浏览器

2): 如果resquest.setAttribute("name","gust");希望下个页面可以使用其属性则用getRequestDispatcher

3): 如果session.setAttribute("uname","顺平");希望下个页面可以使用其属性则用两种方法都可以，建议使用getRequestDispatcher 因为效率高些

4): 如果我们要跳转到本应用外的URL则使用sendRedirect

3 **this**.getServletContext().getRequestDispatcher("/资源URL").forward(request,response);

跟第二种方法一样...

## 读取文件，和获取文件的路径

//读取文件 WebRoot下的资源文件

**import** java.io.InputStream;

**import** java.util.Properties;

InputStream is = **this**.getServletContext().getResourceAsStream("dbinfo.properties");

//创建properties

Properties pp = **new** Properties();

pp.load(is);

out.println("name="+pp.getProperty("username"));

//如果文件在src目录下要用类加载器去读

InputStream iis=Servlet4.**class**.getClassLoader().getResourceAsStream("dbinfo.properties");

InputStream is=Servlet类名.class.getClassLoader().getResourceAsStream("dbinfo.properties");

//获取文件全路径

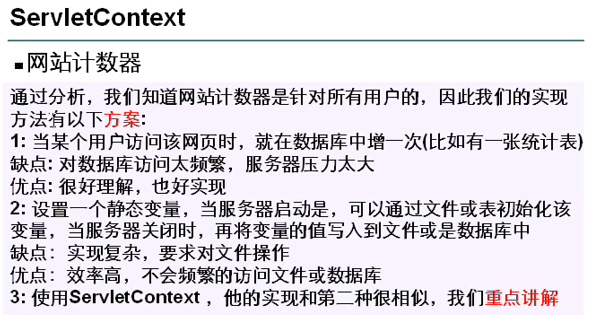
tomcat的主目录 \apache-tomcat-6.0.20\webapps\serContext\imgs\Sunset.jpg

String path=**this**.getServletContext().getRealPath("/imgs/a.jpg");

out.println("path="+path);

如果涉及到不同用户共享数据(量不大) 同时又不希望写入到数据库中， 可以考虑使用servletContext

防刷新 用response.sendRedirect("/web应用名/资源URL");跳转能防刷新





建立一个recoder.txt文件，用于保存访问量，这样可以保证稳定增长。

实现方法是

建立initservlet 用于初始化servletContext 和 在关闭tomcat时保存访问量

Init——把程序放入内存，启动之才能最后调用destory()

**配置文件中加入<load-on-startup>1</load-on-startup>**

<servlet-name>InitServlet</servlet-name>

<servlet-class>com.hsp.InitServlet</servlet-class>

**<load-on-startup>1</load-on-startup>**

</servlet>

WebRoot下创建**record.txt**

当用户登陆一次时，我们取出访问量，并加1

//向servletContext中取出属性并添加

String num= (String)this.getServletContext().getAttribute("nums");

this.getServletContext().setAttribute("nums",(Integer.parseInt(num)+1)+"");

在页面显示

String num = (String)this.getServletContext().getAttribute("nums");

Out.println("该页面被访问了"+num+"次");

如果我们的tomcat异常退出怎么办？

使用线程，定时把ServletContext中的值写入recoder.txt 比如10min

**package** com.hsp;

**import** java.io.\*;

**import** java.io.IOException;

**import** java.io.PrintWriter;

**import** javax.servlet.ServletException;

**import** javax.servlet.http.HttpServlet;

**import** javax.servlet.http.HttpServletRequest;

**import** javax.servlet.http.HttpServletResponse;

**public** **class** InitServlet **extends** HttpServlet {

@Override

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

**super**.destroy();

String path=**this**.getServletContext().getRealPath("/record.txt");

FileWriter fw=**null**;

BufferedWriter bw=**null**;

**try** { //把ServletContext值重新保存到文件

fw = **new** FileWriter(path);

bw = **new** BufferedWriter(fw);

// 从ServletContext中读取访问量

String nums = (String) **this**.getServletContext().getAttribute("nums");

bw.write(nums); //重新写回文件

} **catch** (Exception ex) {

// Logger.getLogger(InitServlet.class.getName()).log(Level.SEVERE, null, ex);

}**finally**{ //关闭流

**try** {

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

bw.close();

}

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

fw.close();

}

} **catch** (IOException ex) {

// Logger.getLogger(InitServlet.class.getName()).log(Level.SEVERE, null, ex);

}

}

}

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

FileReader fr=**null**;

BufferedReader br=**null**;

**try** { //从record.txt文件中读取浏览量 1. 得到该file的真实路径

String path=**this**.getServletContext().getRealPath("/record.txt");

//2.打开文件

fr = **new** FileReader(path);

//转为BufferedReader 再读一行

br = **new** BufferedReader(fr);

String nums = br.readLine();

//把nums放入ServletContext

**this**.getServletContext().setAttribute("nums",nums);

} **catch** (Exception ex) {

// Logger.getLogger(InitServlet.class.getName()).log(Level.SEVERE, null, ex);

}**finally**{ //一定要关闭流

**try** {

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

br.close();

}

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

fr.close();

}

} **catch** (IOException ex) {

// Logger.getLogger(InitServlet.class.getName()).log(Level.SEVERE, null, ex);

}

}

}

**protected** **void** processRequest(HttpServletRequest request, HttpServletResponse response)

**throws** ServletException, IOException {

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

PrintWriter out = response.getWriter();

**try** {

} **finally** {

out.close();

}

}

**protected** **void** doGet(HttpServletRequest request, HttpServletResponse response)

**throws** ServletException, IOException {

processRequest(request, response);

}

**protected** **void** doPost(HttpServletRequest request, HttpServletResponse response)

**throws** ServletException, IOException {

processRequest(request, response);

}

}

servletContext中的数据会长时间保存在服务器的内存中，所以不能向其中添加过大的数据