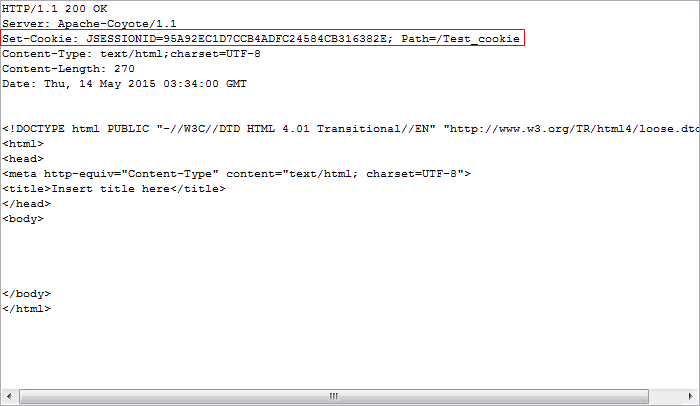
# Cookie

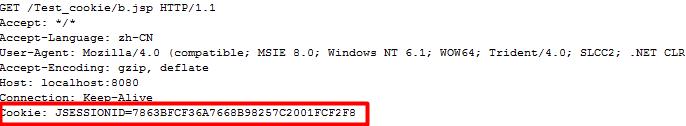
## Cookie简介

* HTTP是无状态协议，服务器不能记录浏览器的访问状态，也就是说服务器不能区分中两次请求是否由一个客户端发出。这样的设计严重阻碍的Web程序的设计。如：在我们进行网购时，买了一条裤子，又买了一个手机。由于http协议是无状态的，如果不通过其他手段，服务器是不能知道用户到底买了什么。而Cookie就是解决方案之一。
* Cookie实际上就是服务器保存在浏览器上的一段信息。浏览器有了Cookie之后，每次向服务器发送请求时都会同时将该信息发送给服务器，服务器收到请求后，就可以根据该信息处理请求。
* 例如：我们上文说的网上商城，当用户向购物车中添加一个商品时，服务器会将这个条信息封装成一个Cookie发送给浏览器，浏览器收到Cookie，会将它保存在内存中(注意这里的内存是本机内存，而不是服务器内存)，那之后每次向服务器发送请求，浏览器都会携带该Cookie，而服务器就可以通过读取Cookie来判断用户到底买了哪些商品。当用户进行结账操作时，服务器就可以根据Cookie的信息来做结算。
* Cookie的用途：
  + 网上商城的购物车
  + 保持用户登录状态
* Cookie的原理：
  + 总的来看Cookie像是服务器发给浏览器的一张“会员卡”，浏览器每次向服务器发送请求时都会带着这张“会员卡”，当服务器看到这张“会员卡”时就可以识别浏览器的身份。
  + 实际上这个所谓的“会员卡”就是服务器发送的一个响应头：



如图Set-Cookie这个响应头就是服务器在向服务器发“会员卡”，这个响应头的名字是Set-Cookie，后边JSESSIONID=95A92EC1D7CCB4ADFC24584CB316382E和 Path=/Test\_cookie，是两组键值对的结构就是服务器为这个“会员卡”设置的信息。浏览器收到该信息后就会将它保存到内存或硬盘中。

* + 当浏览器再次向服务器发送请求时就会携带这个Cookie信息：



这是浏览器发送的请求报文，中间画红框的就是Cookie信息，这里可以理解为浏览器这次带着“会员卡”再次访问服务器。

* + 于是服务器就可以根据Cookie信息来判断浏览器的状态。
* Cookie的缺点
  + Cookie最为请求或响应报文发送，无形中增加了网络流量。
  + Cookie是明文传送的安全性差。
  + 各个浏览器对Cookie有限制，使用上有局限

## 使用Cookie

* 使用Cookie主要是先创建Cookie对象，然后将Cookie信息发送给浏览器，浏览器收到Cookie后会自动保存，然后我们可以在下次浏览器发送请求时读取Cookie信息。

### 设置Cookie

* 创建一个CookieServlet
* 在Servlet的doPost()方法中编写如下代码：

|  |
| --- |
| //创建一个Cookie对象  Cookie cookie = **new** Cookie("username", "zhangsan");  //将Cookie对象放入response对象中  response.addCookie(cookie); |

* 在浏览器中访问该Servlet，会发现响应头中出现如下内容：

Set-Cookie: username=zhangsan

* 如此就成功的向浏览器设置了一个Cookie，当我们在刷新页面时会发现浏览器的请求头中出现如下代码：

Cookie: username=zhangsan

* 同样我们还可以同时设置多个Cookie：

|  |
| --- |
| //创建一个Cookie对象  Cookie cookie1 = **new** Cookie("username", "zhangsan");  Cookie cookie2 = **new** Cookie("password", "123456");  Cookie cookie3 = **new** Cookie("age", "20");  //将Cookie对象放入response对象中  response.addCookie(cookie1);  response.addCookie(cookie2);  response.addCookie(cookie3); |

* 浏览器会以一下形式发送Cookie：

Cookie: username=zhangsan; password=123456; age=20

* 总之，设置Cookie就是两个步骤：
  1. 创建Cookie对象
  2. 将Cookie对象加入到response中

### 读取Cookie

* 读取Cookie主要指从读取浏览器中携带的Cookie
* 只需要通过一个方法就可以得到浏览器中的Cookie，在Servlet或JSP中通过request对象调用getCookies()方法可以获得浏览器传送过来的所有Cookie对象，返回的是一个Cookie数组，通过遍历Cookie数组可以获得所有的Cookie信息。
* 如：

|  |
| --- |
| //通过request的getCookies()方法获的Cookie数组  Cookie[] cookies = request.getCookies();  //判断数组是否为空  **if**(cookies != **null**){  //如果不为空遍历数组  **for**(Cookie cookie : cookies){  //获取cookie的名字  String name = cookie.getName();  //获取cookie的键  String value = cookie.getValue();  System.***out***.println(name+"---"+value);  }  } |

### Cookie有效时间

* 经过上边的介绍我们已经知道Cookie是存储在浏览器中的，但是可想而知一般情况下浏览器不可能永远保存一个Cookie，一来是占用硬盘空间，再来一个Cookie可能只在某一时刻有用没必要长久保存。
* 所以我们还需要为Cookie设置一个有效时间。
* 通过Cookie对象的setMaxAge()可以设置Cookie的有效时间。
* 其中setMaxAge()接收一个int型的参数，来设置有效时间。参数主要有一下四种情况：
  + 设置为0，setMaxAge(0)
    - Cookie立即失效，下次浏览器发送请求将不会在携带该Cookie
  + 设置大于0，setMaxAge(60)
    - 表示有效的秒数60就代表60秒即1分钟，也就是Cookie在1分钟后失效。
  + 设置小于0，setMaxAge(-1)
    - 设置为负数表示当前会话有效。也就是关闭浏览器后Cookie失效
  + 不设置
    - 如果不设置失效时间，则默认当前会话有效。

### Cookie的路径

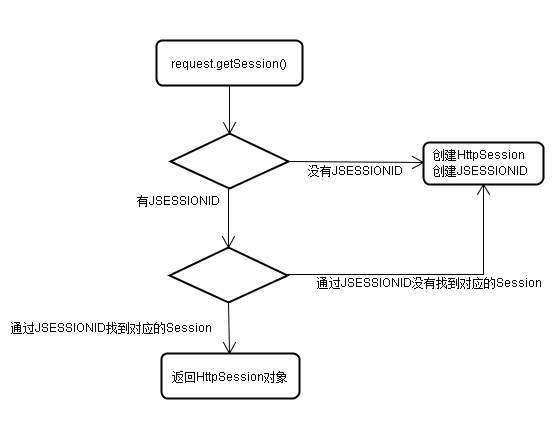
* Cookie的路径指告诉浏览器访问那些地址时该携带该Cookie，我们知道浏览器会保存很多不同网站的Cookie，比如百度的Cookie，新浪的Cookie，腾讯的Cookie等等。那我们不可能访问百度的时候携带新浪的Cookie，也不可能访问每个网站时都带上所有的Cookie这是不现实的，所以往往我们还需要为Cookie设置一个Path属性，来告诉浏览器何时携带该Cookie。
* 我们同过Cookie的setPath()来设置路径，这个路径是由浏览器来解析的所以/代表服务器的根目录。
* 如：
  + 设置为 /项目名/路径 🡪 cookie.setPath(“/项目名/路径”)
    - 这样设置只有访问“/项目名/路径”下的的资源才会携带Cookie
    - 如：/项目名/路径/1.jsp 、/项目名/路径/hello/2.jsp 等
  + 如果不设置，默认会在访问“/项目名”下的资源时携带
    - 如：“/项目名/index.jsp” 、 “/项目名/hello/index.jsp”

# HttpSession

* 使用Cookie有一个非常大的局限，就是如果Cookie很多，则无形的增加了客户端与服务端的数据传输量。而且由于浏览器对Cookie数量的限制，注定我们不能再Cookie中保存过多的信息，于是Session出现。
* Session的作用就是在服务器端保存一些用户的数据，然后传递给用户一个名字为JSESSIONID的Cookie，这个JESSIONID对应这个服务器中的一个Session对象，通过它就可以获取到保存用户信息的Session。

## Session的工作原理

* Session的创建
  + Session的创建时机是在request.getSession()方法第一次被调用时。
  + Session被创建后，同时还会有一个名为JSESSIONID的Cookie被创建。
  + 这个Cookie的默认时效就是当前会话。
* Session的使用
  + Session被创建后，对应的Cookie被保存到浏览器中，之后浏览器每次访问项目时都会携带该Cookie。
  + 当我们再次调用时会根据该JSESSIONID获取已经存在的Cookie，而不是在创建一个新的Cookie。
  + 如果Cookie中有JSESSIONID，但是JSESSIONID没有对应的Session存在，则会重新创建一个HttpSession对象，并重新设置JSESSIONID。



## Session时效

* 基本原则
  + Session对象在服务器端不能长期保存，它是有时间限制的，超过一定时间没有被访问过的Session对象就应该释放掉，以节约内存。所以Session的有效时间并不是从创建对象开始计时，到指定时间后释放——而是从最后一次被访问开始计时，统计其“空闲”的时间。
* 默认设置
  + 在全局web.xml中能够找到如下配置：

|  |
| --- |
| <!-- ==================== Default Session Configuration ================= -->  <!-- You can set the default session timeout (in minutes) for all newly -->  <!-- created sessions by modifying the value below. -->  <session-config>  <session-timeout>30</session-timeout>  </session-config> |

说明Session对象默认的最长有效时间为30分钟。

* 手工设置
  + session.setMaxInactiveInterval(int seconds)
  + session.getMaxInactiveInterval()
* 强制失效
  + session.invalidate()
* 可以使Session对象释放的情况
  + Session对象空闲时间达到了目标设置的最大值，自动释放
  + Session对象被强制失效
  + Web应用卸载
  + 服务器进程停止

## URL重写

* 在整个会话控制技术体系中，保持JSESSIONID的值主要通过Cookie实现。但Cookie在浏览器端可能会被禁用，所以我们还需要一些备用的技术手段，例如：URL重写。
* URL重写其实就是将JSESSIONID的值以固定格式附着在URL地址后面，以实现保持JSESSIONID，进而保持会话状态。这个固定格式是：URL;jsessionid=xxxxxxxxx
* 例如：

|  |
| --- |
| targetServlet;jsessionid=F9C893D3E77E3E8329FF6BD9B7A09957 |

* 实现方式：
  + response.encodeURL(String)
  + response.encodeRedirectURL(String)
  + 例如：

|  |
| --- |
| //1.获取Session对象  HttpSession session = request.getSession();    //2.创建目标URL地址字符串  String url = "targetServlet";    //3.在目标URL地址字符串后面附加JSESSIONID的值  url = response.encodeURL(url);    //4.重定向到目标资源  response.sendRedirect(url); |

## Session的活化和钝化

* Session机制很好的解决了Cookie的不足，但是当访问应用的用户很多时，服务器上就会创建非常多的Session对象，如果不对这些Session对象进行处理，那么在Session失效之前，这些Session一直都会在服务器的内存中存在。那么就，就出现了Session活化和钝化的机制。
* Session钝化：Session在一段时间内没有被使用时，会将当前存在的Session对象序列化到磁盘上，而不再占用内存空间。
* Session活化：Session被钝化后，服务器再次调用Session对象时，将Session对象由磁盘中加载到内存中使用。
* 如果希望Session域中的对象也能够随Session钝化过程一起序列化到磁盘上，则对象的实现类也必须实现java.io.Serializable接口。不仅如此，如果对象中还包含其他对象的引用，则被关联的对象也必须支持序列化，否则会抛出异常：java.io.NotSerializableException