Kevin-air 关注此人

关注(2) 粉丝(6) 积分(9)

这个人很懒，啥也没写

.发送私信 .请教问题

博客分类

- 算法(0)
- 搜索(0)
- Logback(0)
- Java IO(0)
- Java 并发容器(3)
- 工作日志(0)
- go(0)
- Java(2)
- 转贴的文章(5)
- Java 集合(13)

阅读排行

1. 1. 深入理解 Session 与 Cookie
2. 2. Java集合<11> (Queue)
3. 3. Java集合<12> (Deque)
4. 4. Intellij IDEA 快捷键整理
5. 5. Java集合<10> (NavigableMap)
6. 6. IO - 同步，异步，阻塞，非阻塞
7. 7. Java集合<3> (Collection)
8. 8. 阻塞队列 BlockingQueue

最新评论

- @付乐： 查看»
- @Oscarfff：sessionID 与cookie 一起使用 查看»
- @Oscarfff：这篇文章真的很不错，不过有些地方还不是很明白 查看»

访客统计

- 今日访问：26
- 昨日访问：59
- 本周访问：107
- 本月访问：181
- 所有访问：12336

转 深入理解 Session 与 Cookie

发表于2年前(2014-01-14 11:32) 阅读（10576）| 评论（3） 53人收藏此文章, 我要收藏

赞9

摘要 Session 与 Cookie 不管是对 Java Web 的初学者还是熟练使用者来说都是一个令人头疼的问题。在初入职场时恐怕很多程序员在面试的时候都被问到过这个问题。其实这个问题回答起来既简单又复杂，简单是因为它们本身只是 HTTP 协议中的一个配置项，在 Servlet 规范中也只是对应到一个类而已；说它复杂原因在于当我们的系统大到需要用到很多 Cookie 的时候，我们不得不考虑 HTTP 协议对 Cookie 数量和大小的限制，那么如何才能解决这个瓶颈呢？ Session 也会有同样的问题，当我们的一个应用系统有几百台服务器的时候如何解决 Session 在多台服务器之间共享？...

session cookie

目录[-]

Session 与 Cookie 的作用都是为了保持访问用户与后端服务器的交互状态。它们有各自的优点，也有各自的缺陷，然而具有讽刺意味的是它们的优点和它们的使用场景又是矛盾的。例如，使用 Cookie 来传递信息时，随着 Cookie 个数的增多和访问量的增加，它占用的网络带宽也很大，试想假如 Cookie 占用 200 个字节，如果一天的 PV 有几亿，它要占用多少带宽？所以有大访问量的时候希望用 Session，但是 Session 的致命弱点是不容易在多台服务器之间共享，所以这也限制了 Session 的使用。

10.1 理解 Cookie

Cookie 的作用我想大家都知道，通俗地说就是当一个用户通过 HTTP 协议访问一个服务器的时候，这个服务器会将一些 Key/Value 键值对返回给客户端浏览器，并给这些数据加上一些限制条件，在条件符合时这个用户下次访问这个服务器的时候，数据又被完整地带回给服务器。

这个作用就像您去超市购物时，第一次给您办张购物卡，这个购物卡里存放了一些您的个人信息，下次您再来这个连锁超市时，超市会识别您的购物卡，下次直接购物就好了。

当初 W3C 在设计 Cookie 时实际上考虑的是为了记录用户在一段时间内访问 Web 应用的行为路径。由于 HTTP 协议是一种无状态协议，当用户的一次访问请求结束后，后端服务器就无法知道下一次来访问的还是不是上次访问的用户，在设计应用程序时，我们很容易想到两次访问是同一人访问与不同的两个人访问对程序设计和性能来说有很大的不同。例如，在一个很短的时间内，如果与用户相关的数据被频繁访问，可以针对这个数据做缓存，这样可以大大提高数据的访问性能。Cookie 的作用正是在此，由于是同一个客户端发出的请求，每次发出的请求都会带有第一次访问时服务端设置的信息，这样服务端就可以根据 Cookie 值来划分访问的用户了。

10.1.1 Cookie 属性项

当前 Cookie 有两个版本：Version 0 和 Version 1。通过它们有两种设置响应头的标识，分别是“Set-Cookie”和“Set-Cookie2”。这两个版本的属性项有些不同，表 10-1 和表 10-2 是两个版本的属性介绍。

表 10-1.Version 0 属性项介绍

属性项	属性项介绍
NAME=VALUE	键值对，可以设置要保存的 Key/Value，注意这里的 NAME 不能和其他属性项的名字一样
Expires	过期时间，在设置的某个时间点后该 Cookie 就会失效，如 expires=Wednesday, 09-Nov-99 23:12:40 GMT
Domain	生成该 Cookie 的域名，如 domain="xulingbo.net"
Path	该 Cookie 是在当前的哪个路径下生成的，如 path=/wp-admin/
Secure	如果设置了这个属性，那么只会在 SSH 连接时才会回传该 Cookie

表 10-2.Version 1 属性项介绍

属 性 项	属性项介绍
NAME=VALUE	与 Version 0 相同

Version	通过 Set-Cookie2 设置的响应头创建必须符合 RFC2965 规范，如果通过 Set-Cookie 响应头设置，默认值为 0，如果要设置为 1，则该 Cookie 要遵循 RFC 2109 规范
Comment	注释项，用户说明该 Cookie 有何用途
CommentURL	服务器为此 Cookie 提供的 URI 注释
Discard	是否在会话结束后丢弃该 Cookie 项，默认为 fasle
Domain	类似于 Version 0
Max-Age	最大失效时间，与 Version 0 不同的是这里设置的是在多少秒后失效
Path	类似于 Version 0
Port	该 Cookie 在什么端口下可以回传服务端，如果有多个端口，以逗号隔开，如 Port="80,81,8080"
Secure	类似于 Version 0

以上两个版本的 Cookie 中设置的 Header 头的标识符是不同的，我们常用的是 Set-Cookie：userName=“junshan”；Domain=“xulingbo.net”，这是 Version 0 的形式。针对 Set-Cookie2 是这样设置的：Set-Cookie2：userName=“junshan”；Domain=“xulingbo.net”；Max-Age=1000。但是在 Java Web 的 Servlet 规范中并不支持 Set-Cookie2 响应头，在实际应用中 Set-Cookie2 的一些属性项却可以设置在 Set-Cookie 中，如这样设置：Set-Cookie：userName=“junshan”；Version=“1”;Domain=“xulingbo.net”;Max-Age=1000。

10.1.2 Cookie 如何工作

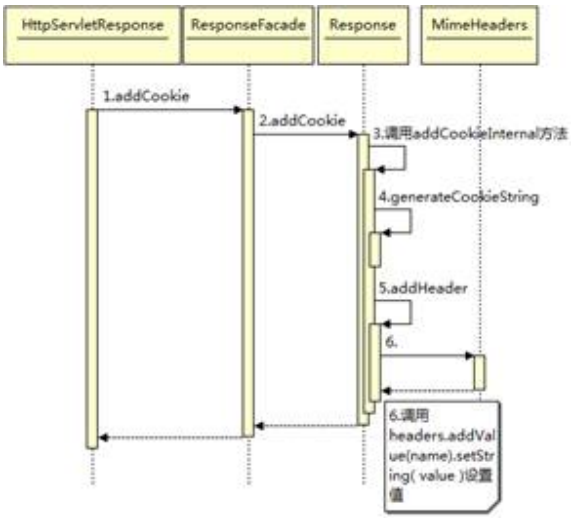
当我们用如下方式创建 Cookie 时：

```
String getCookie(Cookie[] cookies, String key) {
            if (cookies != null) {
                for (Cookie cookie : cookies) {
                    if (cookie.getName().equals(key)) {
                        return cookie.getValue();
                    }
                }
            }
            return null;
    }

    @Override
    public void doGet(HttpServletRequest request,
                                HttpServletResponse response)
                throws IOException, ServletException {
            Cookie[] cookies = request.getCookies();
            String userName = getCookie(cookies, "userName");
            String userAge = getCookie(cookies, "userAge");
            if (userName == null) {
                response.addCookie(new Cookie("userName", "junshan"));
            }
            if (userAge == null) {
                response.addCookie(new Cookie("userAge", "28"));
            }
            response.getHeaders("Set-Cookie");
}
```

Cookie 是如何加到 HTTP 的 Header 中的？当我们用 Servlet 3.0 规范来创建一个 Cookie 对象时，该 Cookie 既支持 Version 0 又支持 Version 1，如果您设置了 Version 1 中的配置项，即使您没有设置版本号，Tomcat 在最后构建 HTTP 响应头时也会自动将 Version 的版本设置为 1。下面看一下 Tomcat 是如何调用 addCookie 方法，图 10-1 是 Tomcat 创建 Set-Cookie 响应头的时序图。

图 10-1.Tomcat 创建 Set-Cookie 响应头的时序图



从图 10-1 中可以看出，真正构建 Cookie 是在 org.apache.catalina.connector. Response 类中完成的，调用 generateCookieString 方法将 Cookie 对象构造成一个字符串，构造的字符串的格式如 userName=“junshan”;Version=“1”；Domain=“xulingbo.net”；Max-Age=1000。然后将这个字符串命名为 Set-Cookie 添加到 MimeHeaders 中。

在这里有几点需要注意：

- 创建的 Cookie 的 NAME 不能和 Set-Cookie 或者 Set-Cookie2 的属性项值一样，如果一样会抛 IllegalArgumentException 异常。
- 创建 Cookie 的 NAME 和 VALUE 的值不能设置成非 ASSIC 字符，如果要使用中文，可以通过 URLEncoder 将其编码，否则将会抛 IllegalArgumentException 异常。
- 当 NAME 和 VALUE 的值出现一些 TOKEN 字符（如“\”、 “,”等）时，构建返回头会将该 Cookie 的 Version 自动设置为 1。
- 当该 Cookie 的属性项中出现 Version 为 1 的属性项时，构建 HTTP 响应头同样会将 Version 设置为 1。

不知道您有没有注意一个问题，就是当我们通过 response.addCookie 创建多个 Cookie 时，这些 Cookie 最终是在一个 Header 项中还是以独立的 Header 存在的，通俗地说也就是我们每次创建 Cookie 时是否都是创建一个以 NAME 为 Set-Cookie 的 MimeHeaders ？答案是肯定的。从上面的时序图中可以看出每次调用 addCookie 的时候，最终都会创建一个 Header，但是我们还不知道最终在请求返回时构造 HTTP 响应头是否将相同 Header 标识的 Set-Cookie 值进行合并。

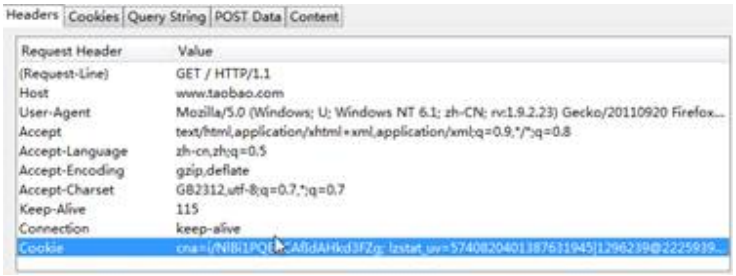
我们找到 Tomcat 最终构造 HTTP 响应头的代码，这段代码位于 org.apache.coyote.http11. Http11Processor 类的 prepareResponse 方法中，如下所示：

```
int size = headers.size();
for (int i = 0; i < size; i++) {
     outputBuffer.sendHeader(headers.getName(i), headers.getValue(i));
}
```

这段代码清楚地表示，在构建 HTTP 返回字节流时是将 Header 中所有的项顺序地写出，而没有进行任何修改。所以可以想象浏览器在接收 HTTP 协议返回的数据时是分别解析每一个 Header 项的。

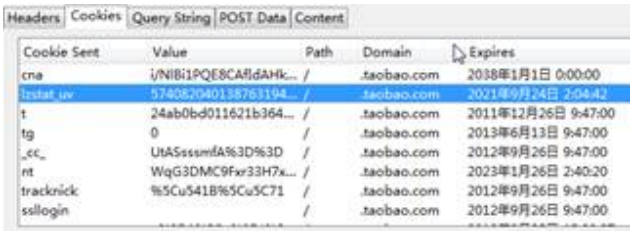
另外，目前很多工具都可以观察甚至可以修改浏览器中的 Cookie 数据。例如，在 Firefox 中可以通过 HttpFox 插件来查看返回的 Cookie 数据，如图 10-2 所示。

图 10-2.HttpFox 插件展示的 Header 数据



在 cookie 项中可以详细查看 Cookie 属性项，如图 10-3 所示。

图 10-3.HttpFox 插件展示的 Cookie 数据



前面主要介绍了服务端如何创建 Cookie，下面看一下如何从客户端获取 Cookie。

当我们请求某个 URL 路径时，浏览器会根据这个 URL 路径将符合条件的 Cookie 放在 Request 请求头中传回给服务端，服务端通过 request.getCookies() 来取得所有 Cookie。

10.1.3 使用 Cookie 的限制

Cookie 是 HTTP 协议头中的一个字段，虽然 HTTP 协议本身对这个字段并没有多少限制，但是 Cookie 最终还是存储在浏览器里，所以不同的浏览器对 Cookie 的存储都有一些限制，表 10-3 是一些通常的浏览器对 Cookie 的大小和数量的限制。

表 10-3.浏览器对 Cookie 的大小和数量的限制

浏览器版本	Cookie 数限制	Cookie 总大小限制
IE6	20 个 / 每个域名	4095 个字节
IE7	50 个 / 每个域名	4095 个字节

续表

浏览器版本	Cookie 数限制	Cookie 总大小限制
IE8	50 个 / 每个域名	4095 个字节
IE9	50 个 / 每个域名	4095 个字节
Chrome	50 个 / 每个域名	大于 80000
FireFox	50 个 / 每个域名	4097 个字

10.2 理解 Session

前面已经介绍了 Cookie 可以让服务端程序跟踪每个客户端的访问，但是每次客户端的访问都必须传回这些 Cookie，如果 Cookie 很多，这无形地增加了客户端与服务端的数据传输量，而 Session 的出现正是为了解决这个问题。

同一个客户端每次和服务端交互时，不需要每次都传回所有的 Cookie 值，而是只要传回一个 ID，这个 ID 是客户端第一次访问服务器的时候生成的，而且每个客户端是唯一的。这样每个客户端就有了一个唯一的 ID，客户端只要传回这个 ID 就行了，这个 ID 通常是 NANE 为 JSESIONID 的一个 Cookie。

10.2.1 Session 与 Cookie

下面详细讲一下 Session 如何基于 Cookie 来工作。实际上有三种方式能可以让 Session 正常工作：

- 基于 URL Path Parameter，默认支持。
- 基于 Cookie，如果没有修改 Context 容器的 cookies 标识，默认也是支持的。
- 基于 SSL，默认不支持，只有 connector.getAttribute("SSLEnabled") 为 TRUE 时才支持。

第一种情况下，当浏览器不支持 Cookie 功能时，浏览器会将用户的 SessionCookieName 重写到用户请求的 URL 参数中，它的传递格式如 /path/Servlet;name=value;name2=value2?Name3=value3，其中“Servlet；”后面的 K-V 就是要传递的 Path Parameters，服务器会从这个 Path Parameters 中拿到用户配置的 SessionCookieName。关于这个 SessionCookieName，如果在 web.xml 中配置 session-config 配置项，其 cookie-config 下的 name 属性就是这个 SessionCookieName 值。如果没有配置了 session-config 配置项，默认的 SessionCookieName 就是大家熟悉的“JSESSIONID”。需要说明的一点是，与 Session 关联的 Cookie 与其他 Cookie 没有什么不同。接着 Request 根据这个 SessionCookieName 到 Parameters 中拿到 Session ID 并设置到 request.setRequestedSessionId 中。

请注意，如果客户端也支持 Cookie，Tomcat 仍然会解析 Cookie 中的 Session ID，并会覆盖 URL 中的 Session ID。

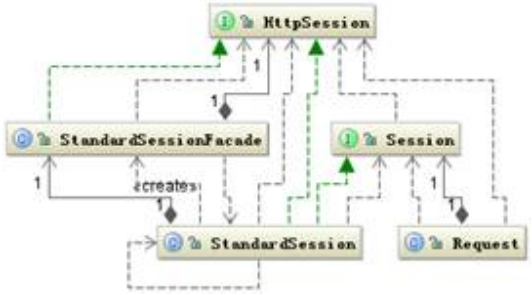
如果是第三种情况，将会根据 javax.servlet.request.ssl_session 属性值设置 Session ID。

10.2.2 Session 如何工作

有了 Session ID 服务端就可以创建 HttpSession 对象了，第一次触发通过 request.getSession() 方法。如果当前的 Session ID 还没有对应的 HttpSession 对象，那么就创建一个新的，并将这个对象加到 org.apache.catalina. Manager 的 sessions 容器中保存。Manager 类将管理所有 Session 的生命周期，Session 过期将被回收，服务器关闭，Session 将被序列化到磁盘等。只要这个 HttpSession 对象存在，用户就可以根据 Session ID 来获取这个对象，也就达到了状态的保持。

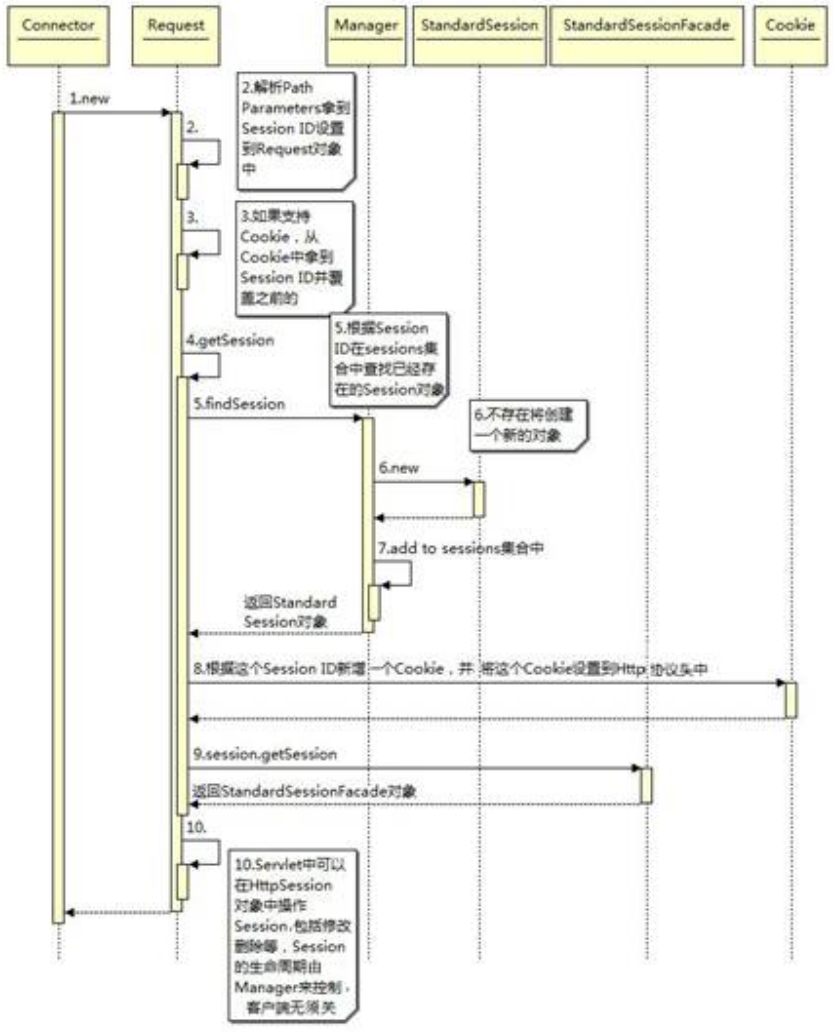
Session 相关类图如图 10-4 所示。

图 10-4.Session 相关类图



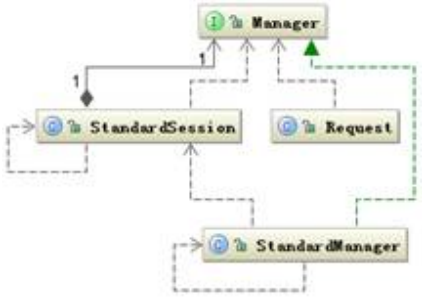
从图 10-4 中可以看出，从 request.getSession 中获取的 HttpSession 对象实际上是 StandardSession 对象的门面对象，这与前面的 Request 和 Servlet 是一样的原理。图 10-5 是 Session 工作的时序图。

图 10-5.Session 工作的时序图（查看大图）



从时序图中可以看出，从 Request 中获取的 Session 对象保存在 org.apache. catalina.Manager 类中，它的实现类是 org.apache.catalina.session.StandardManager，通过 requestedSessionId 从 StandardManager 的 sessions 集合中取出 StandardSession 对象。由于一个 requestedSessionId 对应一个访问的客户端，所以一个客户端，也就对应一个 StandardSession 对象，这个对象正是保存我们创建的 Session 值的。下面我们看一下 StandardManager 这个类是如何管理 StandardSession 的生命周期的。

图 10-6.StandardManager 与 StandardSession 的类关系图



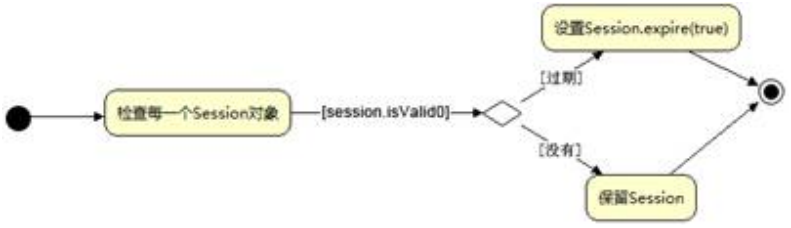
StandardManager 类负责 Servlet 容器中所有的 StandardSession 对象的生命周期管理。当 Servlet 容器重启或关闭时 StandardManager 负责持久化没有过期的 StandardSession 对象，它会将所有的 StandardSession 对象持久化到一个以“SESSIONS.ser”为文件名的文件中。到 Servlet 容器重启时，也就是 StandardManager 初始化时，会重新读取这个文件解析出所有 Session 对象，重新保存在 StandardManager 的 sessions 集合中。Session 恢复状态图如图 10-7 所示。

图 10-7.Session 恢复状态图

当 Servlet 容器关闭时 StandardManager 类会调用 unload 方法将 sessions 集合中的 StandardSession 对象写到“SESSIONS.ser”文件中，然后在启动时再按照上面的状态图重新恢复，注意要持久化保存 Servlet 容器中的 Session 对象，必须调用 Servlet 容器的 stop 和 start 命令，而不能直接结束（kill）Servlet 容器的进程，因为直接结束进程，Servlet 容器没有机会调用 unload 方法来持久化这些 Session 对象。

另外，StandardManager 中的 sessions 集合中的 StandardSession 对象并不是永远保存的，否则 Servlet 容器的内存将很容易被消耗尽，所以必须给每个 Session 对象定义一个有效时间，超过这个时间 Session 对象将被清除。在 Tomcat 中这个有效时间是 60（maxInactiveInterval 属性控制）秒，超过 60 秒该 Session 将会过期。检查每个 Session 是否失效是在 Tomcat 的一个后台线程中完成的（backgroundProcess() 方法中）。过期 Session 状态图如图 10-8 所示。

图 10-8.过期 Session 状态图



除了后台进程检查 Session 是否失效外，当调用 request.getSession() 时也会检查该 Session 是否过期。值得注意的是，request.getSession() 方法调用的 StandardSession 永远都会存在，即使与这个客户端关联的 Session 对象已经过期。如果过期，又会重新创建一个全新的 StandardSession 对象，但是以前设置的 Session 值将会丢失。如果您取到了 Session 对象但是通过 session.getAttribute 取不到前面设置的 Session 值，请不要奇怪，因为很可能它已经失效了，请检查一下 <Manager pathname="" maxInactiveInterval="60" /> 中 maxInactiveInterval 配置项的值，如果不想让 Session 过期可以设置为 -1。但是您要仔细评估一下，网站的访问量和设置的 Session 的大小，防止

将您的 Servlet 容器内存撑爆。如果不想自动创建 Session 对象，也可以通过 request.getSession(boolean create) 方法来判断该客户端关联的 Session 对象是否存在。

10.3 Cookie 安全问题

虽然 Cookie 和 Session 都可以跟踪客户端的访问记录，但是它们的工作方式显然是不同的，Cookie 通过把所有要保存的数据通过 HTTP 协议的头部从客户端传递到服务端，又从服务端再传回到客户端，所有的数据都存储在客户端的浏览器里，所以这些 Cookie 数据可以被访问到，就像我们前面通过 Firefox 的插件 HttpFox 可以看到所有的 Cookie 值。不仅可以查看 Cookie，甚至可以通过 Firecookie 插件添加、修改 Cookie，所以 Cookie 的安全性受到了很大的挑战。

相比较而言 Session 的安全性要高很多，因为 Session 是将数据保存在服务端，只是通过 Cookie 传递一个 SessionID 而已，所以 Session 更适合存储用户隐私和重要的数据。

10.4 分布式 Session 框架

从前面的分析可知，Session 和 Cookie 各自有优点和缺点。在大型互联网系统中，单独使用 Cookie 和 Session 都是不可行的，原因很简单。因为如果使用 Cookie，可以很好地解决应用的分布式部署问题，大型互联网应用系统一个应用有上百台机器，而且有很多不同的应用系统协同工作，由于 Cookie 是将值存储在客户端的浏览器里，用户每次访问都会将最新的值带回给处理该请求的服务器，所以也就解决了同一个用户的请求可能不在同一台服务器处理而导致的 Cookie 不一致的问题。

10.4.1 存在哪些问题

这种“谁家的孩子谁抱走”的处理方式的确是大型互联网的一个比较简单但是的确可以解决问题的处理方式，但是这种处理方式也会带来了很多其他问题，如：

- 客户端 Cookie 存储限制。随着应用系统的增多 Cookie 数量也快速增加，但浏览器对于用户 Cookie 的存储是有限制的。例如，IE7 之前的 IE 浏览器，Cookie 个数的限制是 20 个，后续的版本，包括 Firefox 等，Cookie 个数的限制都是 50 个。总大小不超过 4KB，超过限制就会出现丢弃 Cookie 的现象发生，这会严重影响应用系统的正常使用。
- Cookie 管理的混乱。在大型互联网应用系统中，如果每个应用系统都自己管理每个应用使用的 Cookie，将会导致混乱，由于通常应用系统都在同一个域名下，Cookie 又有上面一条提到的限制，所以没有统一管理很容易出现 Cookie 超出限制的情况。
- 安全令人担忧。虽然可以通过设置 HttpOnly 属性防止一些私密 Cookie 被客户端访问，但是仍然不能保证 Cookie 无法被篡改。为了保证 Cookie 的私密性通常会对 Cookie 进行加密，但是维护这个加密 Key 也是一件麻烦的事情，无法保证定期来更新加密 Key 也是带来安全性问题的一个重要因素。

当以上问题不能再容忍下去的时候，就不得不想其他办法处理了。

10.4.2 可以解决哪些问题

既然 Cookie 有以上这些问题，Session 也有它的好处，为何不结合使用 Session 和 Cookie 呢？下面是分布式 Session 框架可以解决的问题：

- Session 配置的统一管理。
- Cookie 使用的监控和统一规范管理。

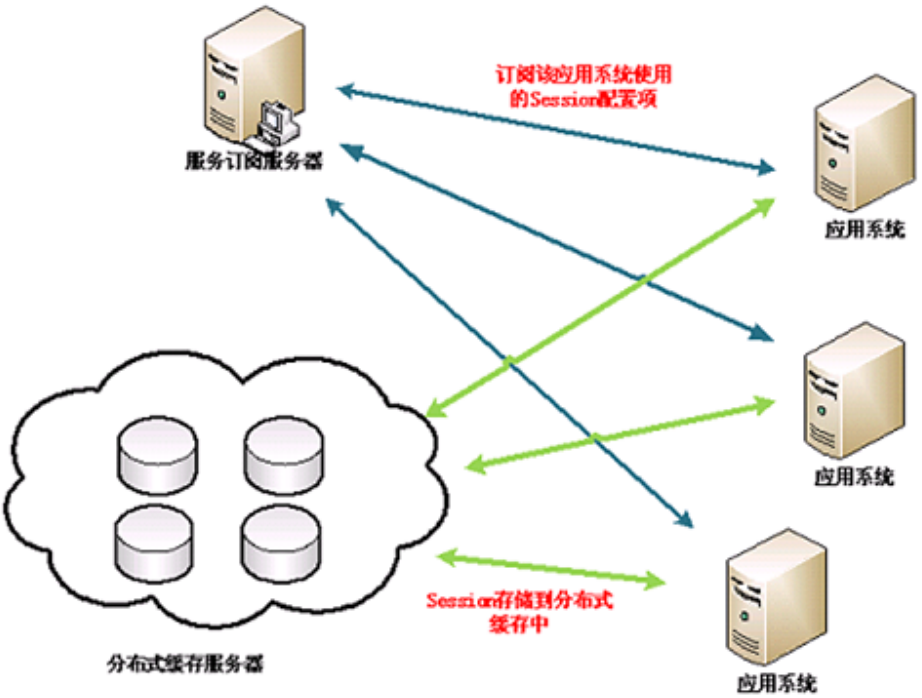
- Session 存储的多元化。
- Session 配置的动态修改。
- Session 加密 key 的定期修改。
- 充分的容灾机制，保持框架的使用稳定性。
- Session 各种存储的监控和报警支持。
- Session 框架的可扩展性，兼容更多的 session 机制如 wapSession。
- 跨域名 Session 与 Cookie 如何共享，现在同一个网站可能存在多个域名，如何将 Session 和 Cookie 在不同的域名之间共享是一个具有挑战性的问题。

10.4.3 总体实现思路

分布式 Session 框架的架构图如图 10-9 所示。

为了达成上面所说的几点目标，我们需要一个服务订阅服务器，在应用启动时可以从这个订阅服务器订阅这个应用需要的可写 Session 项和可写 Cookie 项，这些配置的 Session 和 Cookie 可以限制这个应用能够使用哪些 Session 和 Cookie，甚至可以控制 Session 和 Cookie 可读或者可写。这样可以精确地控制哪些应用可以操作哪些 Session 和 Cookie，可以有效控制 Session 的安全性和 Cookie 的数量。

图 10-9.Session 框架的架构图



如 Session 的配置项可以为如下形式：

```
<session>
          <key>sessionID</key>
        <ookiekey>sessionID</ookiekey  >
          <lifeCycle>9000</lifeCycle>
        <base64>true</base64>
</session  >
```

Cookie 的配置可以为如下形式：

```
<cookie>
          <key>cookie</key>
          <lifeCycle></lifeCycle>
          <type>1</type>
          <path>/wp</path>
        <domain>xulingbo.net</  domain>
          <decrypt>false</decrypt>
        <httpOnly>false</  httpOnly  >
</cookie>
```

统一通过订阅服务器推送配置可以有效地集中管理资源，所以可以省去每个应用都来配置 Cookie，简化 Cookie 的管理。如果应用要使用一个新增 Cookie，可以通过一个统一的平台来申请，申请通过才将这个配置项增加到订阅服务器。如果是一个所有应用都要使用的全局 Cookie，那么只需将这个 Cookie 通过订阅服务器统一推送过去就行了，省去了要在每个应用中手动增加 Cookie 的配置。

关于这个订阅服务器现在有很多开源的配置服务器，如 Zookeeper 集群管理服务器，可以统一管理所有服务器的配置文件。

由于应用是一个集群，所以不可能将创建的 Session 都保存在每台应用服务器的内存中，因为如果每台服务器有几十万的访问用户，服务器的内存肯定不够用，即使内存够用，这些 Session 也无法同步到这个应用的所有服务器中。所以要共享这些 Session 必须将它们存储在一个分布式缓存中，可以随时写入和读取，而且性能要很好才能满足要求。当前能满足这个要求的系统有很多，如 MemCache 或者淘宝的开源分布式缓存系统 Tair 都是很好的选择。

解决了配置和存储问题，下面看一下如何存取 Session 和 Cookie。

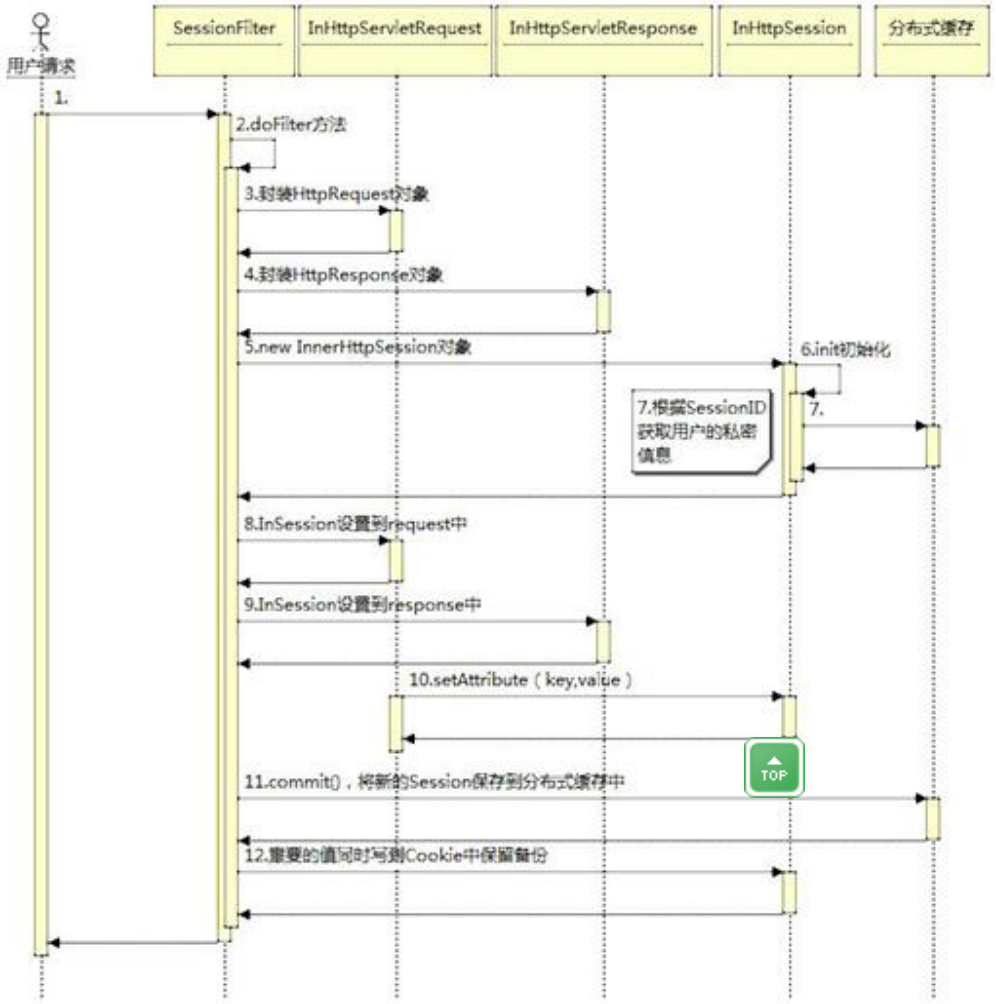
既然是一个分布式 Session 的处理框架，必然会重新实现 HttpSession 的操作接口，使得应用操作 Session 的对象都是我们实现的 InnerHttpSession 对象，这个操作必须在进入应用之前完成，所以可以配置一个 filter 拦截用户的请求。

先看一下如何封装 HttpSession 对象和拦截请求，图 10-10 是时序图。

我们可以在应用的 web.xml 中配置一个 SessionFilter，用于在请求到达 MVC 框架之前封装 HttpServletRequest 和 HttpServletResponse 对象，并创建我们自己的 InnerHttpSession 对象，把它设置到 request 和 response 对象中。这样应用系统通过 request.getHttpSession() 返回的就是我们创建的 InnerHttpSession 对象了，我们可以拦截 response 的 addCookies 设置的 Cookie。

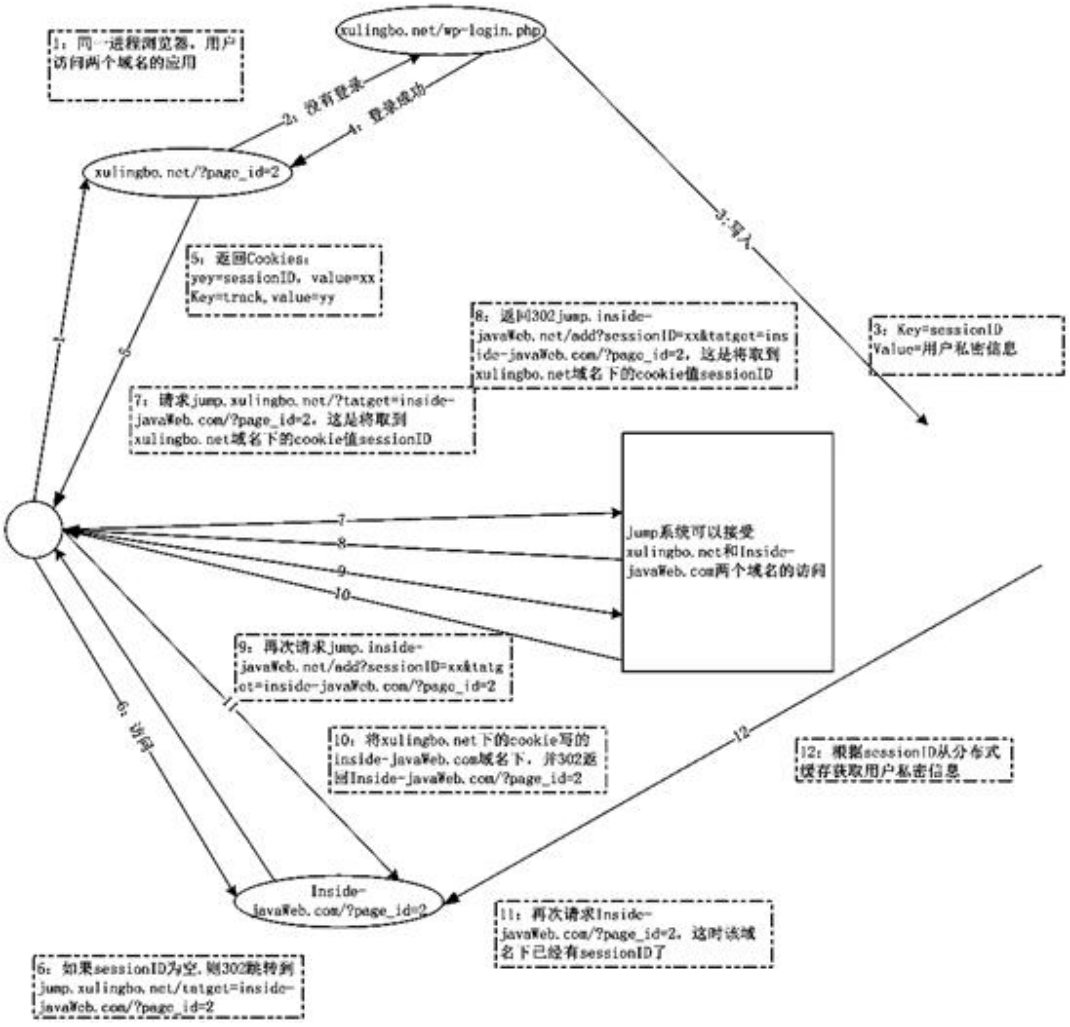
在时序图中，应用创建的所有 Session 对象都会保存在 InnerHttpSession 对象中，当用户的这次访问请求完成时，Session 框架将会把这个 InnerHttpSession 的所有内容再更新到分布式缓存中，以便于这个用户通过其他服务器再次访问这个应用系统。另外，为了保证一些应用对 Session 稳定性的特殊要求可以将一些非常关键的 Session 再存储到 Cookie 中，如当分布式缓存存在问题时，可以将部分 Session 存储到 Cookie 中，这样即使分布式缓存出现问题也不会影响关键业务的正常运行。

图 10-10.HttpSession 拦截请求时序图（查看大图）



还有一个非常重要的问题就是如何处理跨域名来共享 Cookie 的问题。我们知道 Cookie 是有域名限制的，也就是一个域名下的 Cookie 不能被另一个域名访问，所以如果在一个域名下已经登录成功，如何访问到另外一个域名的应用且保证登录状态仍然有效，这个问题大型网站应该经常会遇到。如何解决这个问题呢？下面介绍一种处理方式，如图 10-11 所示。

图 10-11.跨域名同步 session（查看大图）



从图中可以看出，要实现 Session 同步，需要另外一个跳转应用，这个应用可以被一个或者多个域名访问，它的主要功能是从一个域名下取得 sessionID，然后将这个 sessionID 同步到另外一个域名下。这个 sessionID 其实就是一个 Cookie，相当于我们经常遇到的 JSESSIONID，所以要实现两个域名下的 Session 同步，必须要将同一个 sessionID 作为 Cookie 写到两个域名下。

总共 12 步，一个域名不用登录就取到了另外一个域名下的 Session，当然这中间有些步骤还可以简化，也可以做一些额外的工作，如可以写一些需要的 Cookie，而不仅仅只传一个 sessionID。

除此之外，该框架还能处理 Cookie 被盗取的问题。如您的密码没有丢失，但是您的账号却有可能被别人登录的情况，这种情况很可能就是因为您登录成功后，您的 Cookie 被别人盗取了，盗取您的 Cookie 的人将您的 Cookie 加入到他的浏览器，然后他就可以通过您的 Cookie 正常访问您的个人信息了，这是一个非常严重的问题。在这个框架中我们可以设置一个 Session 签名，当用户登录成功后我们根据用户的私密信息生成的一个签名，以表示当前这个唯一的合法登录状态，然后将这个签名作为一个 Cookie 在当前这个用户的浏览器进程中和服务器传递，用户每次访问服务器都会检查这个签名和从服务端分布式缓存中取得的 Session 重新生成的签名是否一致，如果不一致，显然这个用户的登录状态不合法，服务端将清除这个 sessionID 在分布式缓存中的 Session 信息，让用户重新登录。

10.5 Cookie 压缩

Cookie 是在 HTTP 的头部，所以通常的 gzip 和 deflate 针对 HTTP Body 的压缩不能压缩 Cookie，如果 Cookie 量非常大，可以考虑将 Cookie 也做压缩，压缩方式是将 Cookie 的多个 k/v 对看成普通的文本，做文本压缩。压缩算法同样可以使用 gzip 和 deflate 算法，但是需要注意的一点是，根据 Cookie 的规范，Cookie 中不能包含控制字符，仅仅只能包含 ASCII 码为（34 ~ 126）的可见字符。所以要将压缩后的结果再进行转码，可以进行 Base32 或者 Base64 编码。

可以配置一个 Filter 在页面输出时对 Cookie 进行全部或者部分压缩，如下代码所示：

```
private void compressCookie(Cookie c, HttpServletResponse res) {
            try {
                  ByteArrayOutputStream bos = null;
                  bos = new ByteArrayOutputStream();
                  DeflaterOutputStream dos = new DeflaterOutputStream(bos);
                  dos.write(c.getValue().getBytes());
                  dos.close();
                  System.out.println("
                  before compress length:" + c.getValue(). getBytes().length);
                  String compress = new sun.misc.BASE64Encoder().encode(bos. toByteArray());
                  res.addCookie(new Cookie("compress", compress));
                  System.out.println("after compress length:" + compress.getBytes(). length);
            } catch (IOException e) {
                  e.printStackTrace();
            }
}
```

上面的代码是用 DeflaterOutputStream 对 Cookie 进行压缩的，Deflater 压缩后再进行 BASE64 编码，相应地用 InflaterInputStream 进行解压。

```
private void unCompressCookie(Cookie c) {
            try {
                  ByteArrayOutputStream out = new ByteArrayOutputStream();
                  byte[] compress =
            new sun.misc.BASE64Decoder().decodeBuffer(new String(c.getValue().getBytes()));
                  ByteArrayInputStream bis = new ByteArrayInputStream(compress);
                  InflaterInputStream inflater = new InflaterInputStream(bis);
                  byte[] b = new byte[1024];
                  int count;
                  while ((count = inflater.read(b)) >= 0) {
                        out.write(b, 0, count);
                  }
                  inflater.close();
                  System.out.println(out.toByteArray());;
            } catch (Exception e) {
                  e.printStackTrace();
            }
}
```

2KB 大小的 Cookie 压缩前与压缩后字节数相差 20% 左右，如果您的网站的 Cookie 在 2KB~3KB 左右，一天有 1 亿的 PV，那么一天就能够产生 4TB 的带宽流量了，从节省带宽成本来说压缩还是很有必要的。

10.6 表单重复提交问题

网站中在很多地方都有表单重复提交问题，一种情况是用户在网速慢的情况下可能会重复提交表单，还有就是恶意用户通过程序来发送恶意请求，在这些情况下都要设计一个防止表单重复提交的机制。

要能够防止表单重复提交，就要标识用户的每一次访问请求，使得每一次访问对服务端来说都是唯一确定的。为了标识用户的每次访问请求，可以在用户请求一个表单域时增加一个隐藏表单项，这个表单项的值每次都是唯一的 token，如：

```
<form id=”form” method=”post”>
<input type=hidden name=“crsf_token” value=“xxxx”/>
</form>
```

当用户在请求时生成这个唯一的 token 时，同时将这个 token 保存在用户的 Session 中，等用户提交请求时检查这个 token 和当前的 Session 中保存的 token 是否一致。如果一致，说明没有重复提交，否则用户提交上来的 token 已经不是当前的这个请求的合法 token。其工作过程如图 10-12 所示。

图 10-12.工作过程

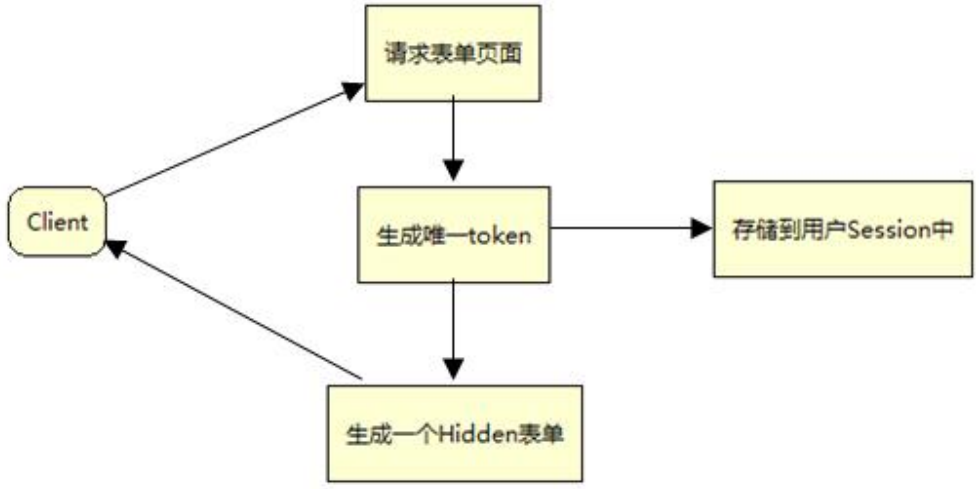
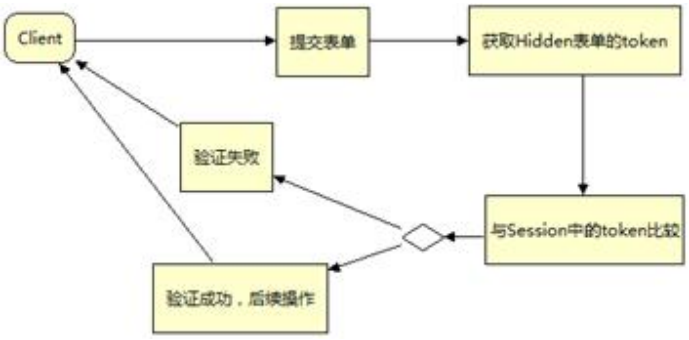


图 10-12 是用户发起对表单页面的请求过程，生成唯一的 token 需要一个算法，最简单的就是可以根据一个种子作为 key 生成一个随机数，并保存在 Session 中，等下次用户提交表单时做验证。验证表单的过程如图 10-13 所示。

图 10-13.验证表单的过程



当用户提交表单时会将请求时生成的 token 带回来，这样就可以和 Session 中保存的 token 做对比，从而确认这次表单验证是否合法。

10.7 总结

Cookie 和 Session 都是为了保持用户访问的连续状态，之所以要保持这种状态，一方面是为了方便业务实现，另一方面就是简化服务端程序设计，提高访问性能，但是这也带来了另外一些挑战，如安全问题、应用的分布式部署带来的 Session 的同步问题及跨域名 Session 的同步等一系列问题。本章分析了 Cookie 和 Session 的工作原理，并介绍了一致分布式 Session 的解决方案。

原文地址：http://www.ibm.com/developerworks/cn/java/books/javaweb_xlb/10/index.html

- « 上一篇
- 下一篇 »

评论3

- 
  1楼：Oscarfff 发表于 2015-05-09 17:16 回复此评论
  这篇文章真的很不错，不过有些地方还不是很明白
- 
  2楼：Oscarfff 发表于 2015-05-09 17:18 回复此评论
  sessionID 与cookie 一起使用
- 
  3楼：付乐 发表于 2015-07-22 17:46 回复此评论

插入： 表情 开源软件 发表评论

关闭插入表情

关闭相关文章阅读

- 2014/08/07 cookie和session
- 2015/08/26 session和cookie
- 2015/12/21 cookie,session
- 2015/07/06 cookie 和session 的区别详解 ...
- 2015/08/10 Http Session和Cookie...