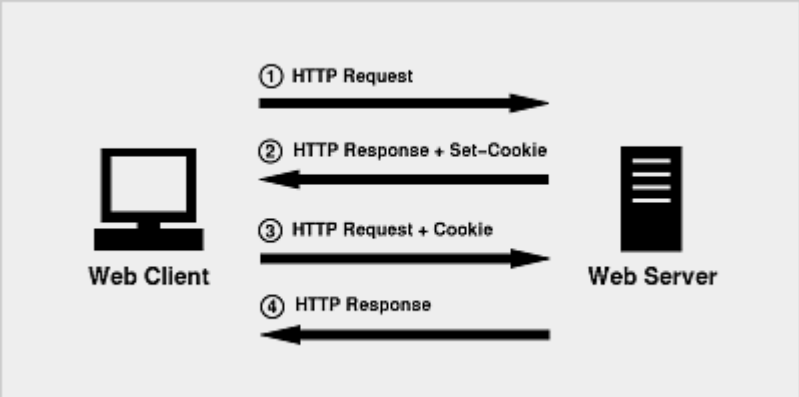
Cookie/Session的机制与安全

Cookie HTTP Node.js Session 内存数据库 Redis 表单 数据库

Cookie和Session是为了在无状态的HTTP协议之上维护会话状态，使得服务器可以知道当前是和哪个客户在打交道。本文来详细讨论Cookie和Session的实现机制，以及其中涉及的安全问题。

因为HTTP协议是无状态的，即每次用户请求到达服务器时，HTTP服务器并不知道这个用户是谁、是否登录过等。现在的服务器之所以知道我们是否已经登录，是因为服务器在登录时设置了浏览器的Cookie！Session则是借由Cookie而实现的更高层的服务器与浏览器之间的会话。

> Cookie是由网景公司的前雇员Lou Montulli在1993年发明的，现今Cookie已经广泛使用了。



Cookie 的实现机制

Cookie是由客户端保存的小型文本文件，其内容为一系列的键值对。 **Cookie是由HTTP服务器设置的，保存在浏览器中**， 在用户访问其他页面时，会在HTTP请求中附上该服务器之前设置的Cookie。 Cookie的实现标准定义在RFC2109: HTTP State Management Mechanism中。 那么Cookie是怎样工作的呢？下面给出整个Cookie的传递流程：

1. 浏览器向某个URL发起HTTP请求（可以是任何请求，比如GET一个页面、POST一个登录表单等）
2. 对应的服务器收到该HTTP请求，并计算应当返回给浏览器的HTTP响应。

   > HTTP响应包括请求头和请求体两部分，可以参见：读 HTTP 协议。

3. 在响应头加入`Set-Cookie`字段，它的值是要设置的Cookie。

在RFC2109 6.3 Implementation Limits中提到： UserAgent（浏览器就是一种用户代理）至少应支持300项Cookie， 每项至少应支持到4096字节，每个域名至少支持20项Cookie。

4. 浏览器收到来自服务器的HTTP响应。
5. 浏览器在响应头中发现`Set-Cookie`字段，就会将该字段的值保存在内存或者硬盘中。

   `Set-Cookie`字段的值可以是很多项Cookie，每一项都可以指定过期时间`Expires`。 默认的过期时间是用户关闭浏览器时。

6. 浏览器下次给该服务器发送HTTP请求时， 会将服务器设置的Cookie附加在HTTP请求的头字段`Cookie`中。

   浏览器可以存储多个域名下的Cookie，但只发送当前请求的域名曾经指定的Cookie， 这个域名也可以在`Set-Cookie`字段中指定）。

7. 服务器收到这个HTTP请求，发现请求头中有`Cookie`字段， 便知道之前就和这个用户打过交道了。
8. 过期的Cookie会被浏览器删除。

总之，服务器通过`Set-Cookie`响应头字段来指示浏览器保存Cookie， 浏览器通过`Cookie`请求头字段来告诉服务器之前的状态。 Cookie中包含若干个键值对，每个键值对可以设置过期时间。

Cookie 的安全隐患

Cookie提供了一种手段使得HTTP请求可以附加当前状态， 现今的网站也是靠Cookie来标识用户的登录状态的：

1. 用户提交用户名和密码的表单，这通常是一个POST HTTP请求。
2. 服务器验证用户名与密码，如果合法则返回200（OK）并设置`Set-Cookie`为`authed=true`。
3. 浏览器存储该Cookie。
4. 浏览器发送请求时，设置`Cookie`字段为`authed=true`。
5. 服务器收到第二次请求，从`Cookie`字段得知该用户已经登录。 按照已登录用户的权限来处理此次请求。

这里面的问题在哪里？

我们知道可以发送HTTP请求的不只是浏览器，很多HTTP客户端软件（包括curl、Node.js）都可以发送任意的HTTP请求，可以设置任何头字段。 假如我们直接设置`Cookie`字段为`authed=true`并发送该HTTP请求， 服务器岂不是被欺骗了？**这种攻击非常容易，Cookie是可以被篡改的！**

Cookie 防篡改机制

服务器可以为每个Cookie项生成签名，由于用户篡改Cookie后无法生成对应的签名， 服务器便可以得知用户对Cookie进行了篡改。一个简单的校验过程可能是这样的：

1. 在服务器中配置一个不为人知的字符串（我们叫它Secret），比如：`x$sfz32`。
2. 当服务器需要设置Cookie时（比如`authed=false`），不仅设置`authed`的值为`false`， 在值的后面进一步设置一个签名，最终设置的Cookie是`authed=false|6hTiBl7lVpd1P`。
3. 签名`6hTiBl7lVpd1P`是这样生成的：`Hash('x$sfz32'+'false')`。 要设置的值与Secret相加再取哈希。
4. 用户收到HTTP响应并发现头字段`Set-Cookie: authed=false|6hTiBl7lVpd1P`。
5. 用户在发送HTTP请求时，篡改了`authed`值，设置头字段`Cookie: authed=true|???`。 因为用户不知道Secret，无法生成签名，只能随便填一个。
6. 服务器收到HTTP请求，发现`Cookie: authed=true|???`。服务器开始进行校验：`Hash('true'+'x$sfz32')`，便会发现用户提供的签名不正确。

通过给Cookie添加签名，使得服务器得以知道Cookie被篡改。然而故事并未结束。

因为**Cookie是明文传输的**， 只要服务器设置过一次`authed=true|xxxx`我不就知道`true`的签名是`xxxx`了么， 以后就可以用这个签名来欺骗服务器了。因此Cookie中最好不要放敏感数据。 一般来讲Cookie中只会放一个Session Id，而Session存储在服务器端。

Session 的实现机制

Session 是存储在服务器端的，避免了在客户端Cookie中存储敏感数据。 Session 可以存储在HTTP服务器的内存中，也可以存在内存数据库（如redis）中， 对于重量级的应用甚至可以存储在数据库中。

我们以存储在redis中的Session为例，还是考察如何验证用户登录状态的问题。

1. 用户提交包含用户名和密码的表单，发送HTTP请求。
2. 服务器验证用户发来的用户名密码。
3. 如果正确则把当前用户名（通常是用户对象）存储到redis中，并生成它在redis中的ID。

   这个ID称为Session ID，通过Session ID可以从Redis中取出对应的用户对象， 敏感数据（比如`authed=true`）都存储在这个用户对象中。
4. 设置Cookie为`sessionId=xxxxxx|checksum`并发送HTTP响应， 仍然为每一项Cookie都设置签名。
5. 用户收到HTTP响应后，便看不到任何敏感数据了。在此后的请求中发送该Cookie给服务器。
6. 服务器收到此后的HTTP请求后，发现Cookie中有SessionID，进行放篡改验证。
7. 如果通过了验证，根据该ID从Redis中取出对应的用户对象， 查看该对象的状态并继续执行业务逻辑。

Web应用框架都会实现上述过程，在Web应用中可以直接获得当前用户。 相当于**在HTTP协议之上，通过Cookie实现了持久的会话。这个会话便称为Session。**

> 上一篇：jQuery中$()函数有几种用法
>
> 下一篇：减少页面回流与重绘（Reflow & Repaint）