cookies详讲

转载

zhen12321 于 2018-07-18 16:51:47 发布 402 收藏
分类专栏： java 保存


java 同时被 2 个专栏收录
89 篇文章 0 订阅
订阅专栏


保存
319 篇文章 3 订阅
订阅专栏

HTTP cookies，通常又称作"cookies"，已经存在了很长时间，但是仍旧没有被予以充分的理解。首要的问题是存在了诸多误区，认为cookies是后门程序或病毒，或压根不知道它是如何工作的。第二个问题是对于cookies缺少一个一致性的接口。尽管存在着这些问题，cookies仍旧在web开发中起着如此重要的作用，以至于如果cookie在没有可替代品出现的情况下消失，我们许多喜欢的Web应用将变得毫无用处。

cookies的起源

早期Web开发面临的最大问题之一是如何管理状态。简言之，服务器端没有办法知道两个请求是否来自于同一个浏览器。那时的办法是在请求的页面中插入一个token，并且在下一次请求中将这个token返回（至服务器）。这就需要在form中插入一个包含token的隐藏表单域，或着在URL的qurey字符串中传递该token。这两种办法都强调手工操作并且极易出错。

Lou Montulli,那时是网景通讯的一个雇员，被认为在1994年将“magic cookies”的概念应用到了web通讯中。他意图解决的是web中的购物车，现在所有购物网站都依赖购物车。他的最早的说明文档提供了一些cookies工作原理的基本信息该文档在RFC2109中被规范化（这是所有浏览器实现cookies的参考依据），并且最终逐步形成了REF2965.Montulli最终也被授予了关于cookies的美国专利。网景浏览器在它的第一个版本中就开始支持cookies，并且当前所有web浏览器都支持cookies。

cookie是什么？

坦白的说，一个cookie就是存储在用户主机浏览器中的一小段文本文件。Cookies是纯文本形式，它们不包含任何可执行代码。一个Web页面或服务器告之浏览器来将这些信息存储并且基于一系列规则在之后的每个请求中都将该信息返回至服务器。Web服务器之后可以利用这些信息来标识用户。多数需要登录的站点通常会在你的认证信息通过后来设置一个cookie，之后只要这个cookie存在并且合法，你就可以自由的浏览这个站点的所有部分。再次，cookie只是包含了数据，就其本身而言并不有害。

创建cookie

通过HTTP的Set-Cookie消息头，Web服务器可以指定存储一个cookie。Set-Cookie消息的格式如下面的字符串（中括号中的部分都是可选的）

```
1 Set-Cookie:value [ ;expires=date][ ;domain=domain][ ;path=path][ ;secure]
```

消息头的第一部分，value部分，通常是一个name=value格式的字符串。事实上，原始手册指示这是应该使用的格式，但是浏览器对cookie的所有值并不会按此格式校验。实际上，你可以指定一个不包含等号的字符串并且它同样会被存储。然而，通常性的使用方式是以name=value的格式（并且多数的接口只支持该格式）来指定cookie的值。

当一个cookie存在，并且可选条件允许的话，该cookie的值会在接下来的每个请求中被发送至服务器。cookie的值被存储在名为Cookie的HTTP消息头中，并且只包含了cookie的值，其它的选项全部被去除。例如：

```
1 Cookie : value
```

通过Set-Cookie指定的选项只是应用于浏览器端，一旦选项被设置后便不会被服务器重新取回。cookie的值与Set-Cookie中指定的值是完全一样的字符串；对于这些值不会有更近一步的解析或转码操作。如果在指定的请求中有多个cookies，那么它们会被分号和空格分开，例如：

```
1 Cookie:value1 ; value2 ; name1=value1
```

服务器端框架通常会提供解析cookies的功能，并且通过编程方式获取cookies的值。

cookie编码（cookie encoding）

对于cookie的值进行编码一直都存在一些困惑。通常的观点是cookie的值必须被URL编码，但是这其实是一个谬误，尽管可以对cookie的值进行URL编码。原始的文档中指示仅有三种类型的字符必须进行编码：分号，逗号，和空格。规范中提到可以利用URL编码，但是并不是必须。RFC没有提及任何的编码。然而，几乎所有的实现方式都对cookie的值进行了一些列的URL编码。对于name=value的格式，name和value通常都单独进行编码并且不对等号“=”进行编码操作。

有效期选项（The expires option）

紧跟cookie值后面的每个选项都以分号和空格分割，并且每个选项都指定cookie何时应该被发送到服务器。第一个选项是expires，其指定了cookie何时不会再被发送到服务器端的，因此该cookie可能会被浏览器删掉。该选项所对应的值是一个格式为Wdy,DD-Mon--YYYY HH:MM:SS GMT的值，例如：

```
1 Set-Cookie:name=Nicholas;expires=Sat, 02 May 2009 23:38:25 GMT
```

在没有expires选项时，cookie的寿命仅限于单一的会话中。浏览器的关闭意味这一次会话的结束，所以会话cookie只存在于浏览器保持打开的状态之下。这就是为什么当你登录到一个web应用时经常看到一个checkbox，询问你是否选择存储你的登录信息：如果你选择是的话，那么一个expires选项会被附加到登录的cookie中。如果expires选项设置了一个过去的时间点，那么这个cookie会被立即删除。

domain选项（The domain option）

下一个选项是domain，指示cookie将要发送到哪个域或那些域中。默认情况下，domain会被设置为创建该cookie的页面所在的域名。例如本站中的cookie的domain属性的默认值为www.nczonline.com。domain选项被用来扩展cookie值所要发送域的数量。例如:

```
1 Set-Cookie:name=Nicholas;domain=nczonline.net
```

想象诸如Yahoo！这样的大型网站都会有许多以name.yahoo.com(例如：my.yahoo.com,finance.yahoo.com，等等)为格式的站点。单独的一个cookie可以简单的通过将其domain选项设置为yahoo.com而发送到所有这些站点中。浏览器会对domain的值与请求所要发送至的域名，做一个尾部比较（即从字符串的尾部开始比较），并且在匹配后发送一个Cookie消息头。

domain设置的值必须是发送Set-Cookie消息头的域名。例如，我无法向google.com发送一个cookie，因为这个产生安全问题。不合法的domain选项只要简单的忽略即可。

Path选项（The path option）

另一个控制何时发送Cookie消息头的方式是指定path选项。与domain选项相同的是，path指明了在发Cookie消息头之前必须在请求资源中存在一个URL路径。这个比较是通过将path属性值与请求的URL从头开始逐字符串比较完成的。如果字符匹配，则发送Cookie消息头，例如：

```
1 Set-Cookie:name=Nicholas;path=/blog
```

在这个例子中，path选项值会与/blog,/blogrool等等相匹配；任何以/blog开头的选项都是合法的。要注意的是只有在domain选项核实完毕之后才会对path属性进行比较。path属性的默认值是发送Set-Cookie消息头所对应的URL中的path部分。

secure选项（The secure option）

最后一个选项是secure。不像其它选项，该选项只是一个标记并且没有其它的值。一个secure cookie只有当请求是通过SSL和HTTPS创建时，才会发送到服务器端。这种cookie的内容意指具有很高的价值并且可能潜在的被破解以纯文本形式传输。例如

```
1 Set-Cookie:name=Nicholas;secure
```

现实中，机密且敏感的信息绝不应该在cookies中存储或传输，因为cookies的整个机制都是原本不安全的。默认情况下，在HTTPS链接上传输的cookies都会被自动添加上secure选项。

cookie的维护和生命周期（cookie maintenance and lifecycle）

任意数量的选项都可以在单一的cookie中指定，并且这些选项可以以任何顺序存在，例如

```
1 Set-Cookie:name=Nicholas; domain=nczonline.net; path=/blog
```

这个cooke有四个标识符：cookie的name，domain，path，secure标记。要想在将来改变这个cookie的值，需要发送另一个具有相同cookie name,domain,path的Set-Cookie消息头。例如：

```
1 Set-Cooke:name=Greg; domain=nczonline.net; path=/blog
```

这将以一个新的值来覆盖原来cookie的值。然而，仅仅只是改变这些选项的某一个也会创建一个完全不同的cookie，例如：

```
1 Set-Cookie:name=Nicholas; domain=nczonline.net; path=/
```

在返回这个消息头后，会存在两个同时拥有“name”的不同的cookie。如果你访问在www.nczonline.net/blog下的一个页面，以下的消息头将被包含进来：

```
1 Cookie：name=Greg;name=Nicholas
```

在这个消息头中存在了两个名为“name”的cookie，path值越详细则cookie越靠前。domain-path越详细则cookie字符串越靠前。假设我在ww.nczonline.net/blog下并且发送了另一个cookie，其设置如下：

```
1 Set-Cookie:name=Mike
```

那么返回的消息头现在则变为：

```
1 Cookie：name=Mike;name=Greg;name=Nicholas
```

由于包含“Mike”的cookie使用了域名（www.nczonline.net）作为其domain值并且以全路径（/blog）作为其path值，则它较其它两个cookie更加详细。

使用失效日期（using expiration dates）

当cookie创建时包含了失效日期，这个失效日期则关联了以name-domain-path-secure为标识的cookie。要改变一个cookie的失效日期，你必须指定同样的组合。当改变一个cookie的值时，你不必每次都设置失效日期，因为它不是cookie标识信息的组成部分。例如：

```
1 Set-Cookie:name=Mike;expires=Sat,03 May 2025 17:44:22 GMT
```

现在已经设置了cookie的失效日期，所以下次我想要改变cookie的值时，我只需要使用它的名字：

```
1 Set-Cookie:name=Matt
```

在cookie上的失效日期并没有改变，因为cookie的标识符是相同的。实际上，只有你手工的改变cookie的失效日期，否则其失效日期不会改变。这意味着在同一个会话中，一个会话cookie可以变成一个持久化cookie（一个可以在多个会话中存在的），反之则不可。为了要将一个持久化cookie变为一个会话cookie，你必须删除这个持久化cookie，这只要设置它的失效日期为过去某个时间之后再创建一个同名的会话cookie就可以实现。

需要记得的是失效日期是以浏览器运行的电脑上的系统时间为基准进行核实的。没有任何办法来来验证这个系统时间是否和服务器的时间同步，所以当服务器时间和浏览器所处系统时间存在差异时这样的设置会出现错误。

cookie自动删除（automatic cookie removal）

cookie会被浏览器自动删除，通常存在以下几种原因：

- 会话cooke(Session cookie)在会话结束时（浏览器关闭）会被删除
- 持久化cookie（Persistent cookie）在到达失效日期时会被删除
- 如果浏览器中的cookie限制到达，那么cookies会被删除以为新建cookies创建空间。详见我的另外一篇关于cookies restrictions的博客

对于任何这些自动删除来说，Cookie管理显得十分重要，因为这些删除都是无意识的。

Cookie限制条件（Cookie restrictions）

在cookies上存在了诸多限制条件，来阻止cookie滥用并保护浏览器和服务器免受一些负面影响。有两种cookies的限制条件：cookies的属性和cookies的总大小。原始的规范中限定每个域名下不超过20个cookies，早期的浏览器都遵循该规范，并且在IE7中有个更近一步的提升。在微软的一次更新中，他们在IE7中增加cookies的限制到50个，与此同时Opera限定cookies个数为30.Safari和Chrome对与每个域名下的cookies个数没有限制。

发向服务器的所有cookies的最大数量（空间）仍旧维持原始规范中所指出的：4KB。所有超出该限制的cookies都会被截掉并且不会发送至服务器。

Subcookies

鉴于cookie的数量限制，开发者提出的subcookies的观点来增加cookies的存储量。Subcookies是一些存储在一个cookie的value中的一些name-value对，并且通常与以下格式类似：

```
1	name=a=b&c=d&e=f&g=h
```

这种方式允许在单个cookie中保存多个name-value对，而不会超过浏览器cookie的数量限制。通过这种方式创建cookies的负面影响是，需要自定义解析方式来提取这些值，相比较而言cookies的格式会更为简单。服务器端框架已开始支持subcookies的存储。我编写的YUI Cookie utility，支持在javascript中读/写subcookies

Javascript中的cookie（cookie In Javascript）

通过Javascript中的document.cookie属性，你可以创建，维护和删除cookies。当要创建cookies时该属性等同于Set-Cookie消息头，而在读取cookie时则等同于Cookie消息头。在创建一个cookie时，你需要使用和Set-Cookie期望格式相同的字符串：

```
1	document.cookie="name=Nicholas;domain=nczonline.net;path=/";
```

设置document.cookie属性的值并不会删除存储在页面中的所有cookies。它只简单的创建或修改字符串中指定的cookies。下次发送一个请求到服务器时，这些cookies（通过document.cookie设置的）会和其它通过Set-Cookie消息头设置的cookies一样发送至服务器。所有这些cookies并没有什么明确的不同之处。

要使用Javascript提取cookie的值时，只要从document.cookie中读取即可。返回的字符串与Cookie消息头中的字符串格式相同，所以多个cookies会被分号和字符串分割。例如：

```
1	name1=Greg; name2=Nicholas
```

鉴于此，你需要手工解析这个cookie字符串来提取真实的cookie数据。当前已有许多描述利用Javascript来解析cookie的资料，包括我的书，Professional Javascript，所以在这我就不再说明。通常利用已存在的Javascript库操作cookie会更简单，如YUI Cookie utility 来在Javascript中处理cookies而不要手工重新创建这些算法。

通过访问document.cookie返回的cookies遵循发向服务器的cookies一样的访问规则。要通过Javascript访问cookies，该页面和cookies必须在相同的域中，有相同的path，有相同的安全级别。

注意：一旦cookies通过Javascript设置后遍不能提取它的选项，所以你将不会知道domain，path，expiration日期或secure标记。

HTTP-Only cookies

微软的IE6 SP1在cookies中引入了一个新的选项：HTTP-only cookies.HTTP-Only背后的意思是告之浏览器该cookie绝不应该通过Javascript的document.cookie属性访问。设计该特征意在提供一个安全措施来帮助阻止通过Javascript发起的跨站脚本攻击(XSS)窃取cookie的行为（我会在另一篇博客中讨论安全问题，本篇如此已足够）。今天Firefox2.0.0.5+，Opera9.5+,Chrome都支持HTTP-Only cookies。3.2版本的Safari仍不支持。

要创建一个HTTP-Only cookie，只要向你的cookie中添加一个HTTP-Only标记即可：

```
1	Set-Cookie: name=Nicholas; HttpOnly
```

一旦设定这个标记，通过documen.coookie则不能再访问该cookie。IE同时更近一步并且不允许通过XMLHttpRequest的getAllResponseHeaders()或getResponseHeader()方法访问cookie，然而其它浏览器则允许此行为。Firefox在3.0.6中修复了该漏洞，然而仍旧有许多浏览器漏洞存在，complete browser support list列出了这些。

你不能通过JavaScript设置HTTP-only cookies，因为你不能再通过JavaScript读取这些cookies，这是情理之中的事情。

总结（conclusion）

为了高效的利用cookies，仍旧有许多要了解和弄明白的东西。对于一项创建于十多年前但仍旧如最初实现的那样被使用至今的技术来说，这是件多不可思议的事。本篇只是提供了一些每个人都应该知道的关于浏览器cookies的基本指导，但无论如何，也不是一个完整的参考。对于今天的web来说Cookies仍旧起着非常重要的作用，并且不恰当的管理cookies会导致各种安全性的问题，从最糟糕的用户体验到安全漏洞。我希望这篇手册能够激起一些关于cookies的不可思议的亮点。

写在后面的：

该文章是2009年的，到现在可能已经算一篇比较老的文章，但是文章中关于cookies的讲解十分详细，最近在学习cookies时，读到该文章，觉得值得大家一读，同时也作为自己的参考吧，所以将原文翻译为中文。文中较为详细的讲解了关于cookies的起源，所要解决的问题，以及cookies的本身的属性和每个属性的作用，以及相关浏览器对于cookies的实现情况和延伸情况，正如作者所表达的那样，cookies作为一项十几年前创建却一直沿用至今的技术，值得每个开发者思考并了解和弄明白关于cookies的一些基本信息和原理，译文中尽量保证还原原文本意，但水平有限，一些语句翻译的比较生涩，建议和原文对比阅读，效果可能会比较好一点。

关于作者： Nicholas C. Zakas，资深前端工程师，曾在yahoo工作近五年，并担任前端开发技术领导，他的著作有《javascript 高级程序设计》是一本高质量的前端技术著作。

英文原版

HTTP cookies, most often just called “cookies,” have been around for a while but are still not very well understood. The first problem is a lot of misconceptions, ranging from cookies as spyware or viruses to just plain ignorance over how they work. The second problem is a lack of consistent interfaces to work with cookies. Despite all of the issues surrounding them, cookies are such an important part of web development that, should they disappear without a replacement, many of our favorite web applications would be rendered useless.

Origin of cookies

One of the biggest issues in the early days of the web was how to manage state. In short, the server had no way of knowing if two requests came from the same browser. The easiest approach, at the time, was to insert some token into the page when it was requested and get that token passed back with the next request. This required either using a form with a hidden field containing the token or to pass the token as part of the URL’s query string. Both solutions were intensely manual operations and prone to errors.

Lou Montulli, an employee of Netscape Communications at the time, is credited with applying the concept of "magic cookies" to web communication in 1994. The problem he was attempting to solve was that of the web's first shopping cart, now a mainstay on all shopping sites. His original specification provides basic information about how cookies work, which was formalized in RFC 2109 (the reference for most browser implementations) and eventually evolved into RFC 2965. Montulli would also be granted a United States patent for cookies. Netscape Navigator supported cookies since its first version, and cookies are now supported by all web browsers.

What is a cookie?

Quite simply, a cookie is a small text file that is stored by a browser on the user's machine. Cookies are plain text; they contain no executable code. A web page or server instructs a browser to store this information and then send it back with each subsequent request based on a set of rules. Web servers can then use this information to identify individual users. Most sites requiring a login will typically set a cookie once your credentials have been verified, and you are then free to navigate to all parts of the site so long as that cookie is present and validated. Once again, the cookie just contains data and isn't harmful in and of itself.

Cooke creation

A web server specifies a cookie to be stored by sending an HTTP header called `Set-Cookie`. The format of the `Set-Cookie` header is a string as follows (parts in square brackets are optional):

```
Set-Cookie: <em>value</em>[; expires=<em>date</em>][; domain=<em>domain</em>][; path=<em>path</em>][; secur
```

The first part of the header, the value, is typically a string in the format `name=value`. Indeed, the original specification indicates that this is the format to use but browsers do no such validation on cookie values. You can, in fact, specify a string without an equals sign and it will be stored just the same. Still, the most common usage is to specify a cookie value as `name=value` (and most interfaces support this exclusively).

When a cookie is present, and the optional rules allow, the cookie value is sent to the server with each subsequent request. The cookie value is stored in an HTTP header called `Cookie` and contains just the cookie value without any of the other options. Such as:

```
Cookie: value
```

The options specified with `Set-Cookie` are for the browser's use only and aren't retrievable once they have been set. The cookie value is the exact same string that was specified with `Set-Cookie`; there is no further interpretation or encoding of the value. If there are multiple cookies for the given request, then they are separated by a semicolon and space, such as:

```
Cookie: value1; value2; name1=value1
```

Server-side frameworks typically provide functionality to parse cookies and make their values available programmatically.

Cookie encoding

There is some confusion over encoding of a cookie value. The commonly held belief is that cookie values must be URL-encoded, but this is a fallacy even though it is the de facto implementation. The original specification indicates that only three types of characters must be encoded: semicolon, comma, and white space. The specification indicates that URL encoding may be used but stops short of requiring it. The RFC makes no mention of encoding whatsoever. Still, almost all implementations perform some sort of URL encoding on cookie values. In the case of `name=value` formats, the name and value are typically encoded separately while the equals sign is left as is.

The expires option

Each of the options after the cookie value are separated by a semicolon and space and each specifies rules about when the cookie should be sent back to the server. The first option is `expires`, which indicates when the cookie should no longer be sent to the server and therefore may be deleted by the browser. The value for this option is a date in the format `Wdy, DD-Mon-YYYY HH:MM:SS GMT` such as:

```
Set-Cookie: name=Nicholas; expires=Sat, 02 May 2009 23:38:25 GMT
```

Without the `expires` option, a cookie has a lifespan of a single session. A session is defined as finished when the browser is shut down, so session cookies exist only while the browser remains open. This is why you'll often see a checkbox when signing into a web application asking if you would like your login information to be saved: if you select yes, then an `expires` option is attached to the login cookie. If the `expires` option is set to a date that appears in the past, then the cookie is immediately deleted.

The domain option

The next option is `domain`, which indicates the domain(s) for which the cookie should be sent. By default, `domain` is set to the host name of the page setting the cookie, so the cookie value is sent whenever a request is made to the same host name.Â For example, the default domain for a cookie set on this site would be `www.nczonline.net`. The `domain` option is used to widen the number of domains for which the cookie value will be sent. Sample:

```
Set-Cookie: name=Nicholas; domain=nczonline.net
```

Consider the case of a large network such as Yahoo! that has many sites in the form of *name*.yahoo.com (e.g., my.yahoo.com, finance.yahoo.com, etc.). A single cookie value can be set for all of these sites by setting the`domain` option to simply `yahoo.com`. The browser performs a tail comparison of this value and the host name to which a request is sent (meaning it starts the comparison from the end of the string) and sends the corresponding `Cookie` header when there's a match.

The value set for the `domain` option must be part of the host name that is sending the `Set-Cookie` header. I couldn't, for example, set a cookie on google.com because that would introduce a security issue. Invalid`domain` options are simply ignored.

The path option

Another way to control when the `Cookie` header will be sent is to specify the `path` option. Similar to the domain option, `path` indicates a URL path that must exist in the requested resource before sending the `Cookie`header. This comparison is done by comparing the option value character-by-character against the start of the request URL. If the characters match, then the `Cookie` header is sent. Sample:

```
Set-Cookie: name=Nicholas; path=/blog
```

In this example, the `path` option would match `/blog`, `/blogroll`, etc.; anything that begins with `/blog` is valid. Note that this comparison is only done once the `domain` option has been verified. The default value for the`path` option is the path of the URL that sent the `Set-Cookie` header.

The secure option

The last option is `secure`. Unlike the other options, this is just a flag and has no additional value specified. A secure cookie will only be sent to the server when a request is made using SSL and the HTTPS protocol. The idea that the contents of the cookie are of high value and could be potentially damaging to transmit as clear text. Sample:

```
Set-Cookie: name=Nicholas; secure
```

In reality, confidential or sensitive information should never be stored or transmitted in cookies as the entire mechanism is inherently insecure. By default, cookies set over an HTTPS connection are automatically set to be secure.

Cookie maintenance and lifecycle

Any number of options can be specified for a single cookie, and those options may appear in any order. For example:

```
Set-Cookie: name=Nicholas; domain=nczonline.net; path=/blog
```

This cookie has four identifying characteristics: the cookie `name`, the `domain`, the `path`, and the `secure` flag. In order to change the value of this cookie in the future, another `Set-Cookie` header must be sent using the same cookie name, domain, and path. For example:

```
Set-Cookie: name=Greg; domain=nczonline.net; path=/blog
```

This overwrites the original cookie's value with a new one. However, changing even one of these options creates a completely different cookie, such as:

```
Set-Cookie: name=Nicholas; domain=nczonline.net; path=/
```

After returning this header, there are now two cookies with a name of "name". If you were to access a page at `www.nczonline.net/blog`, the following header would be included in the request:

```
Cookie: name=Greg; name=Nicholas
```

There are two cookies in this header named "name", with the more specific `path` being returned first. The cookie string is always returned in order from most specific `domain-path-secure` tuple to least specific. Suppose I'm at `www.nczonline.net/blog` and set another cookie with default settings:

```
Set-Cookie: name=Mike
```

The returned header now becomes:

```
Cookie: name=Mike; name=Greg; name=Nicholas
```

Since the cookie with the value “Mike” uses the hostname (`www.nczonline.net`) for its domain and the full path (`/blog`) as its path, it is more specific than the two others.

Using expiration dates

When a cookie is created with an expiration date, that expiration date relates to the cookie identified by name-`domain`-`path`-`secure`. In order to change the expiration date of a cookie, you must specify the exact same tuple. When changing a cookie’s value, you need not set the expiration date each time because it’s not part of the identifying information. Example:

```
Set-Cookie: name=Mike; expires=Sat, 03 May 2025 17:44:22 GMT
```

The expiration date of the cookie has now been set, so the next time I want to change the value of the cookie, I can just use its name:

```
Set-Cookie: name=Matt
```

The expiration date on this cookie hasn’t changed, since the identifying characteristics of the cookie are the same. In fact, the expiration date won’t change until you manually change it again. That means a session cookie can become a persistent cookie (one that lasts multiple sessions) within the same session but the opposite isn’t true. In order to change a persistent cookie to a session cookie, you must delete the persistent cookie by setting its expiration date to a time in the past and then create a session cookie with the same name.

Keep in mind that the expiration date is checked against the system time on the computer that is running the browser. There is no way to verify that the system time is in sync with the server time and so errors may occur when there is a discrepancy between the system time and the server time.

Automatic cookie removal

There are a few reasons why a cookie might be automatically removed by the browser:

- Session cookies are removed when the session is over (browser is closed).
- Persistent cookies are removed when the expiration date and time have been reached.
- If the browser’s cookie limit is reached, then cookies will be removed to make room for the most recently created cookie. For more details, see my post on cookie restrictions.

Cookie management is important to avoid any of these automatic removal cases when they are unintended.

Cookie restrictions

There are a number of restrictions placed on cookies in order to prevent abuse and protect both the browser and the server from detrimental effects. There are two types of restrictions: number of cookies and total cookie size. The original specification placed a limit of 20 cookies per domain, which was followed by early browsers and continued up through Internet Explorer 7. During one of Microsoft's updates, they increased the cookie limit in IE 7 to 50 cookies. IE 8 has a maximum of 50 cookies per domain as well. Firefox also has a limit of 50 cookies while Opera has a limit of 30 cookies. Safari and Chrome have no limit on the number of cookies per domain.

The maximum size for all cookies sent to the server has remained the same since the original cookie specification: 4 KB. Anything over that limit is truncated and won't be sent to the server.

Subcookies

Due to the cookie number limit, developers have come up with the idea of subcookies to increase the amount of storage available to them. Subcookies are name-value pairs stored within a cookie value and typically have a format similar to the following:

```
name=a=b&c=d&e=f&g=h
```

This approach allows a single cookie to hold multiple name-value combinations without going over the browser's cookie limit. The downside to creating cookies in this format is that custom parsing is needed to extract the values rather than relying on the much simplier cookie format. Some server-side frameworks are beginning to support subcookie storage. The YUI Cookie utility, which I wrote, supports subcookies reading/writing from JavaScript.

Cookies in JavaScript

You can create, manipulate, and remove cookies in JavaScript by using the `document.cookie` property. This property acts as the `Set-Cookie` header when assigned to and as the `Cookie` header when read from. When creating a cookie, you must use a string that's in the same format that `Set-Cookie` expects:

```
document.cookie="name=Nicholas; domain=nczonline.net; path=/";
```

Setting the value of `document.cookie` *does not* delete all of the cookies stored on the page. It simply creates or modifies the cookie specified in the string. The next time a request is made to the server, these cookies are sent along with any others that were created via `Set-Cookie`. There is no perceivable difference between these cookies.

To retrieve cookie values in JavaScript, just read from the `document.cookie` property. The returned string is in the same format as the `Cookie` header value, so multiple cookies are separated by a semicolon and space. Example:

```
name1=Greg; name2=Nicholas
```

Because of this, you need to parse the cookie string manually to extract actual cookie data. There are numerous resources describing cookie parsing approaches for JavaScript, including my book, Professional JavaScript, so I won't go into it here. It's often easier to use an already-existing JavaScript library, such as the YUI Cookie utility to deal with cookies in JavaScript rather than recreating these algorithms by hand.

The cookies returned by accessing document.cookie follow the same access rules as cookies sent to the server. In order to access cookies via JavaScript, the page must be in the same domain and have the same path and have the same security level as specified by the cookie.

Note: It's not possible to retrieve the options for cookies once they've been set via JavaScript, so you'll have no idea what the `domain`, `path`, expiration date, or `secure` flag.

HTTP-Only cookies

Microsoft introduced a new option for cookies in Internet explorer 6 SP1: HTTP-only cookies. The idea behind HTTP-only cookies is to instruct a browser that a cookie should never be accessible via JavaScript through the `document.cookie` property. This feature was designed as a security measure to help prevent cross-site scripting (XSS) attacks perpetrated by stealing cookies via JavaScript (I'll discuss security issues with cookies in another post, this one is long enough as it is). Today, Firefox 2.0.0.5+, Opera 9.5+, and Chrome also support HTTP-only cookies. Safari as of 3.2 still does not.

To create an HTTP-only cookie, just add an `HttpOnly` flag to your cookie:

```
Set-Cookie: name=Nicholas; HttpOnly
```

Once this flag is set, there is no access via `document.cookie` to this cookie. Internet Explorer also goes a step further and doesn't allow access to this header using the `getAllResponseHeaders()` or`getResponseHeader()` methods on `XMLHttpRequest` while other browsers still permit it. Firefox fixed this vulnerability in 3.0.6 though there are still various browser vulnerabilities floating around (complete browser support list).

You cannot set HTTP-only cookies from JavaScript, which makes sense since you also can't read them from JavaScript.

Conclusion

There's a lot to know and understand about cookies in order to use them effectively. It's truly amazing how a technique created more than ten years ago is still in use in almost the exact same way as it was first implemented. This post is a guide to the basics that everyone should know about cookies in browsers but is not, in any way, a complete reference. Cookies are an important part of the web today and improperly managing them can lead to all kinds of issues from poor user experience to security holes. I hope that this writeup has shed some light on the magic of cookies.

Disclaimer: Any viewpoints and opinions expressed in this article are those of Nicholas C. Zakas and do not, in any way, reflect those of my employer, my colleagues, Wrox Publishing, O'Reilly Publishing, or anyone else. I speak only for myself, not for them.

原文出处