大家都知道，HTTP 协议本身是无状态的，Session 的出现解决了这个问题，也被大多数 Web 端采用。 但它背后的实现原理你是否有兴趣了解呢，以及在它基础上的思维发散，和你聊聊。

**无状态的 HTTP**

大家都知道，我们目前使用的互联网应用层协议基本上都是基于 HTTP 和 HTTPS 的，它们的本身是无状态的， 只负责请求和响应。 我告诉服务器我需要什么，服务器返回给我相应的资源。 如果没有额外处理的话， 服务器是不知道你是谁，更无法根据你是谁给你展现和你相关的内容了。

HTTP 协议一开始被设计成这样还是有一些历史原因的，当时的互联网多用于学术交流，只用于文章信息的展现之类的事情，远没有现在这么丰富多彩。所以在当时的背景下 HTTP 协议被设计成这样其实也是很符合它的场景的。

但随着互联网应用越来越广泛，应用的形式也变得越来越多，我们的 Web 应用不只限于提供简单的信息展现了，还需要用户能够登录，可以在论坛发帖子，在购物网站买东西等等。 这就需要 HTTP 协议能够记录用户的状态。也就是我们现在熟悉的 Session 由来。

**Session 如何实现**

大多数 Web 框架都提供了操作 Session 的 API。 如果你有 Web 端开发的经验，那么你对这样的代码应该不会陌生：

session["name"] = "xxx";

我们给 session 对象设置一个值，比如用户登录成功后，给他设置一个用户名，这样我们这个用户的状态就保存下来了。 并且后续的请求中，这个状态都可以读取到。这一切就这么自动的发生了，并且习以为常。那么这一行代码背后发生了什么呢？ 就是下面几个步骤。

**生成 Session ID**

关于 Session 的工作原理， 其实还是值得我们了解一下的。 我们调用上面这行代码后，我们使用的 Web 框架首先会给当前请求创建一个 Session ID。 这个 Session ID 是通过一系列算法生成的一个唯一字符串。 这也是一个 Web 框架(PHP, Ruby on Rails, ExpressJS 等)提供的基础能力，每个框架生成 Sesssion ID 的具体实现算法可能有所差别，但整体流程都是一样的。

**建立服务端 Session 存储结构**

这个新生成的 Session ID 用于标识这次发起请求的用户，并存储到服务器的某个区域中（默认情况下会在内存中）。 这个 Session ID 同样也是本地存储的一个 key，比如我们上面代码中设置了 name 属性，就相当于在这个 key 中设置了对应的属性。 这样说起来可能有点抽象，举个例子 Session 在服务端的存储结构大概相当于这样：

{

"session-id-1": {

"name" : "xxx"

},

"session-id-2": {

"name" : "xxx"

}

//...

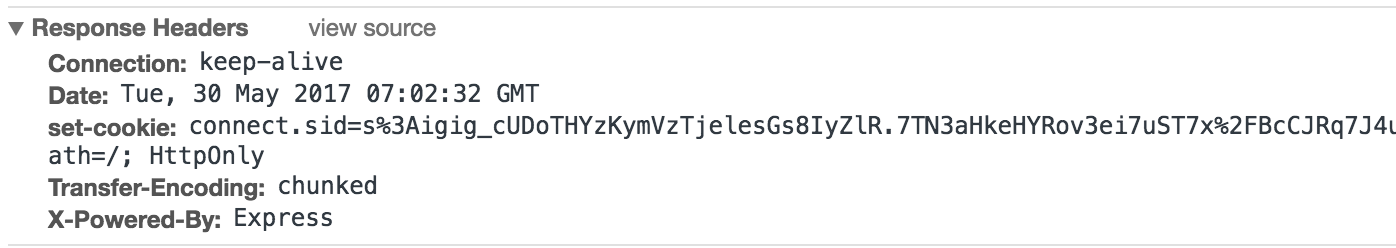
}

}

服务端会保存这样一个字典， 将每个用户的 Session ID 和对应的属性都记录下来。

**把 Session ID 返回给用户浏览器**

还继续我们刚才的流程，生成完 Session ID 并建立好本地存储结构后， 服务端会在返回给用户的 HTTP 响应消息中带上这个 Session ID:

[](http://7rf34y.com2.z0.glb.qiniucdn.com/c/4ea4d1e4e9aa0097d2e6882b90807738)

如上图， 通过 Response Header 的 set-cookie 带上，这样截图使用的是基于 Express 的 NodeJS 服务端框架， 这里面的 connect.sid=xxx 就是服务端给这个用户生成的 Session ID。

这样，用户的浏览器得到这个 Cookie 后，再下一次请求同一个网站的时候就会在请求中带上这个 Cookie。

如果你对 Cookie 的细节不熟悉的话，不用多想，你可以理解成这样 — 你用的所有浏览器都会有这个逻辑，收到 set-cookie 响应头后，就会把里面的内容保存下来，下一次再访问同样的站点时候，就会把之前保存的 cookie 再重新发送回去。当然，关于 Cookie 的细节还有很多知识，不过理解我的这个简单解释就足够了，感兴趣的话可以再深入研究。

**客户端发送 Session ID**

就会像我们刚说的，浏览器下一次再请求这个网站时，会把之前保存的 Session ID 再重新发给服务端。 这时候服务端就会用这个 Session ID 和它之前建立的存储结构中进行匹配，如果这个 Session ID 是之前合法创建的，那么就可以从服务端存储中得到用户之前的状态了，比如登录用户名之类的。 比如:

{

//...

"session-id-1": {

"name" : "xxx"

}

//...

}

假设上面是我们服务端建立的本地 Session 数据存储，如果 Session ID 正确匹配，就能找到对应这个用户的 name 属性了。

**整体步骤**

上面给大家介绍的就是 Session 整体的工作过程。 大概分为这几个步骤：

1. 浏览器第一次请求网站， 服务端生成 Session ID。
2. 把生成的 Session ID 保存到服务端存储中。
3. 把生成的 Session ID 返回给浏览器，通过 set-cookie。
4. 浏览器收到 Session ID， 在下一次发送请求时就会带上这个 Session ID。
5. 服务端收到浏览器发来的 Session ID，从 Session 存储中找到用户状态数据，会话建立。
6. 此后的请求都会交换这个 Session ID，进行有状态的会话。

**扩展知识**

看完这一套流程后，是不是对我们开始的那一行代码背后发生的事情了解的更通透了呢。还有几个值得讨论的地方也和大家聊聊。

**1. Session ID**

首先就是 Session ID，如果你理解了前面的介绍后，就会得到一个知识，在整个会话过程中，最重要的就是 Session ID。一个相对成熟的 Web 应用，往往会同时处理成百上千，甚至更大量的用户同时在线。 这就对 Session ID 的创建有一个非常重要的要求，那就是在保证生成性能的同时，不能重复！

可以想象，如果你的 Web 框架在生成 Session ID 的时候重复了，会发生什么事情 — 用户会误登录进别人的账号， 这个后果还是非常严重的。 好在现在成熟的 Web 框架都考虑到了这个问题，你知识框架的使用者，所以你不必过于担心。但了解背后的这个原理以及思维方式还是有助于你写出更安全的程序的。

**2. Session 数据存储**

另外一个要聊聊的就是 Session 数据的存储。 通常情况下，如果你不明确的设置， 大多数 Web 框架会把 Session 数据存放到内存中。如果你的 Web 应用用户量不大的话，这也不成问题。 但如果你的用户数比较大的话，就可能发生一个事情 — 内存不够用了。

这很正常，内存容量是非常宝贵的，假设每个用户的 Session 数据是 100K， 1万个用户就会大概占用 1G 的存储空间，如果你的 Session 数据清理机制也恰巧比较慢的话，内存非常容易被占满。

这就需要你在设计比较大并发量的站点时，要考虑 Session 的存储方式，比如把它们保存到硬盘文件系统中，或者数据库中。 所以你在开发一个 Web 应用的时候，如果你的用户量很大，你需要有这个意识。

另外 Session 放到内存中还有一个弊端，如果你的 Web 服务器发生重启，那么所有的 Session 状态都会被情况，会在一定程度上影响用户体验。

**3. 传输安全**

最后再聊聊传输安全，有一种叫做 Session ID 劫持的，假如 Session ID 是基于 HTTP 协议传输的，因为是明文传输，那么它就可能被中间的路由器劫持。 攻击者得到 Session ID 后，把它带到自己的请求中，就能够进入你的账户。

所以一些 Web 框架还提供了 Session 的一些安全保护，比如间隔时间内动态刷新 Session ID，加上 Token 等。但这些也无法完全保证不被中间人看到。 其实从这个角度也间接体现了为什么 HTTPS 这么重要。

**总结**

这次跟大家聊了一下 HTTP Session 的原理和整个工作过程。 透过对它的了解，不仅是对细节的掌握，更重要的是这些知识能够帮助我们理清对技术整体的思维方式。 包括我们最后说的 Session ID 生成机制。为什么把 Session 放到内存中会有问题，这样才会理解框架为什么要提供硬盘和数据库之类的其他 Session 存储方式。无论你使用什么框架，什么语言，这些原理性的东西都是不变的。了解的多了，你也就越来越不用焦虑学哪种技术有前途这个问题了。

如果你觉得这篇文章有帮助，还可以关注微信公众号 swift-cafe，会有更多我的原创内容分享给你~

本站文章均为原创内容，如需转载请注明出处，谢谢。

