你管這破玩意兒叫Token?

菜鳥要飛 昨天

以下文章來源於碼海，作者碼海

碼海

一起進階，一起牛逼！

菜鳥要飛

自學教程、學習路線、面試寶典、源碼專題、開源項目，關注後回复"自學"有驚喜！和優秀程序員一起成長，讓最菜的菜鳥早日騰飛！

129篇原創內容

公眾號

上週我們在團隊內部首次採用了jwt（Json Web Token） token 這種no-session 的方式來作用戶的賬號驗證，發現網上很多文章對token 的介紹有誤，所以對cookie，session，token 作了一下對比（文中token指jwt token）相信大家看完肯定有收穫！

Cookie

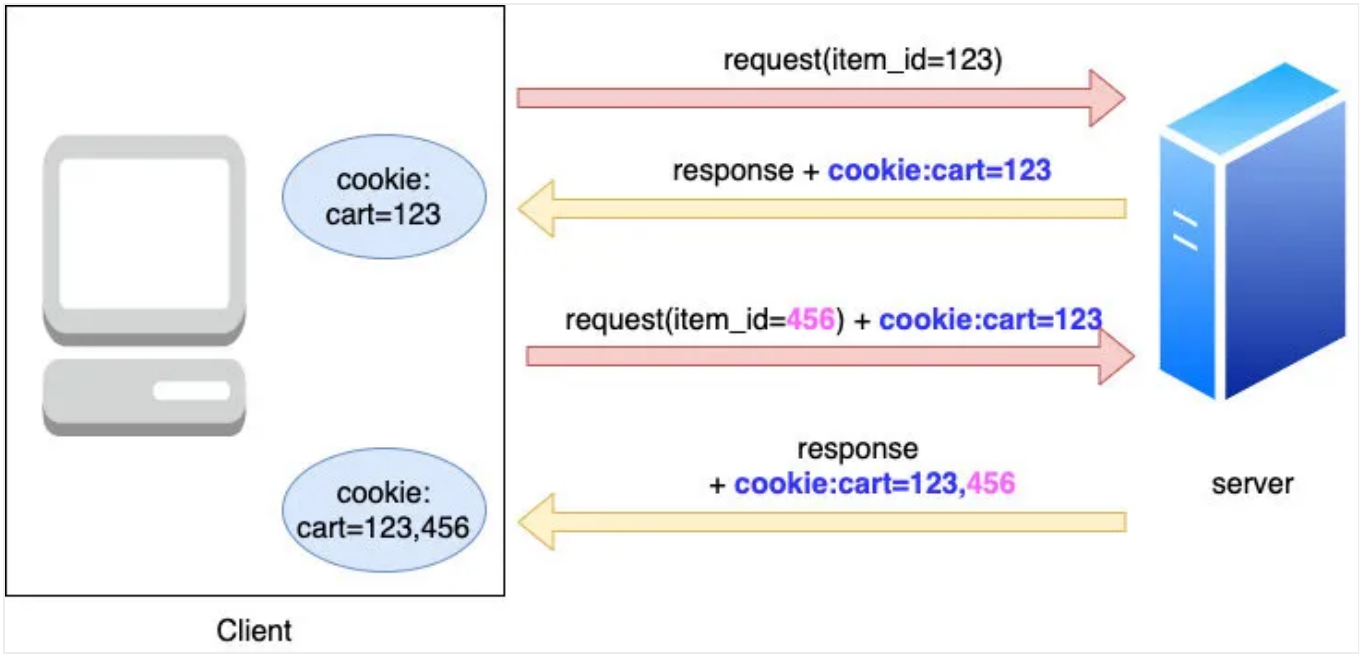
1991 年HTTP 0.9 誕生了，當時只是為了滿足大家瀏覽web 文檔的要求，所以只有GET 請求，瀏覽完了就走了，兩個連接之間是沒有任何联系的，這也是HTTP 為無狀態的原因，因為它誕生之初就沒有這個需求。

但隨著交互式Web 的興起（所謂交互式就是你不光可以瀏覽，還可以登錄，發評論，購物等用戶操作的行為），單純地瀏覽web 已經無法滿足人們的要求，比如隨著網上購物的興起，需要記錄用戶的購物車記錄，就需要有一個機制記錄每個連接的關係，這樣我們就知道加入購物車的商品到底屬於誰了，於是Cookie 就誕生了。

> Cookie，有时也用其复数形式 Cookies。类型为“小型文本文件”，是某些网站为了辨别用户身份，进行 Session 跟踪而储存在用户本地终端上的数据（通常经过加密），由用户客户端计算机暂时或永久保存的信息 。

工作机制如下

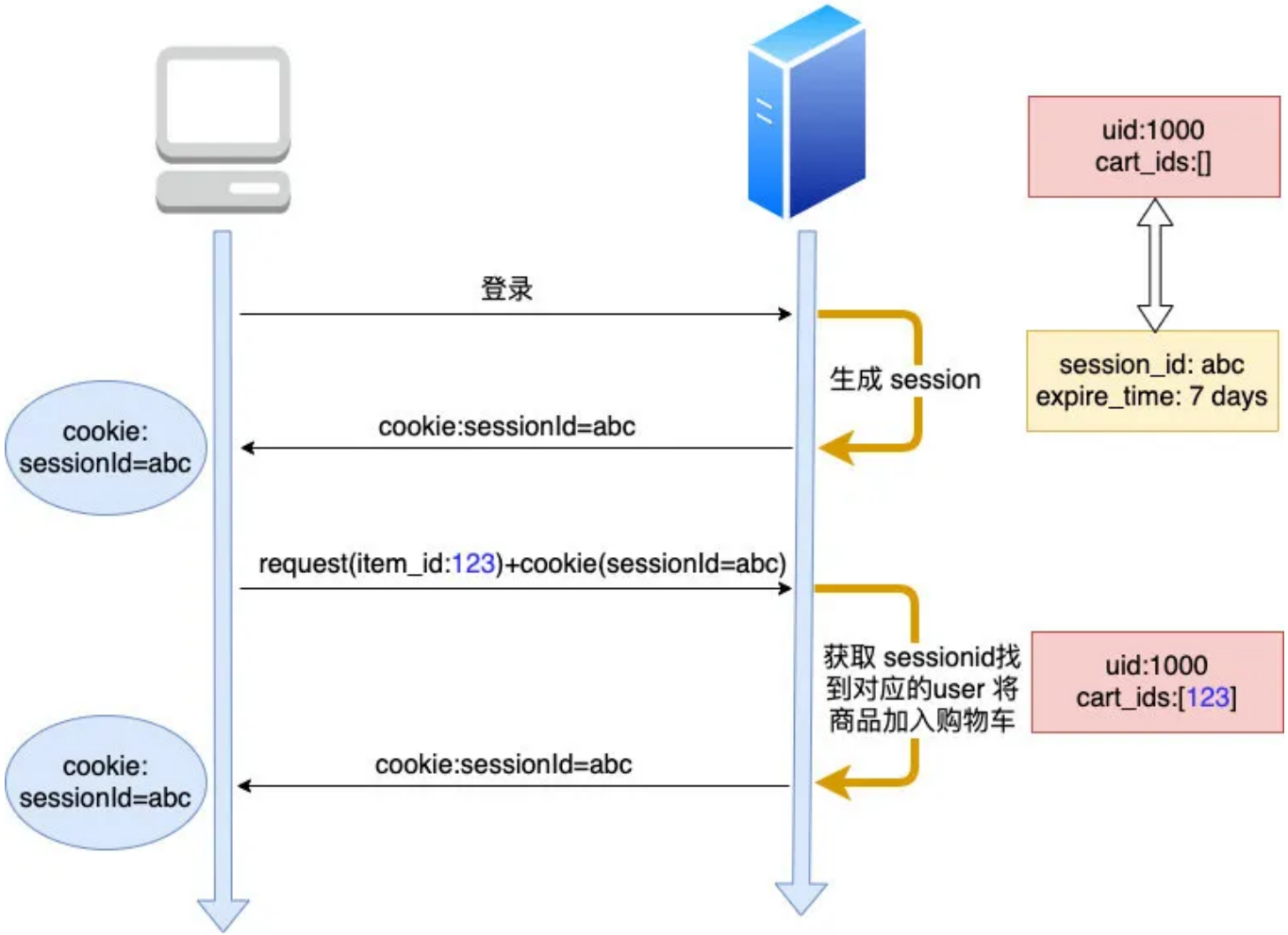
以加入购物车为例，每次浏览器请求后 server 都会将本次商品 id 存储在 Cookie 中返回给客户端，客户端会将 Cookie 保存在本地，下一次再将上次保存在本地的 Cookie 传给 server 就行了，这样每个 Cookie 都保存着用户的商品 id，购买记录也就不会丢失了



仔细观察上图相信你不难发现随着购物车内的商品越来越多，每次请求的 cookie 也越来越大，这对每个请求来说是一个很大的负担，我只是想将一个商品加入购买车，为何要将历史的商品记录也一起返回给 server ？购物车信息其实已经记录在 server 了，浏览器这样的操作岂不是多此一举？怎么改进呢

Session

仔细考虑下，由于用户的购物车信息都会保存在 Server 中，所以在 Cookie 里只要保存能识别用户身份的信息，知道是谁发起了加入购物车操作即可，这样每次请求后只要在 Cookie 里带上用户的身份信息，请求体里也只要带上本次加入购物车的商品 id，大大减少了 cookie 的体积大小，我们把这种能识别哪个请求由哪个用户发起的机制称为 Session（会话机制），生成的能识别用户身份信息的字符串称为 sessionId，它的工作机制如下



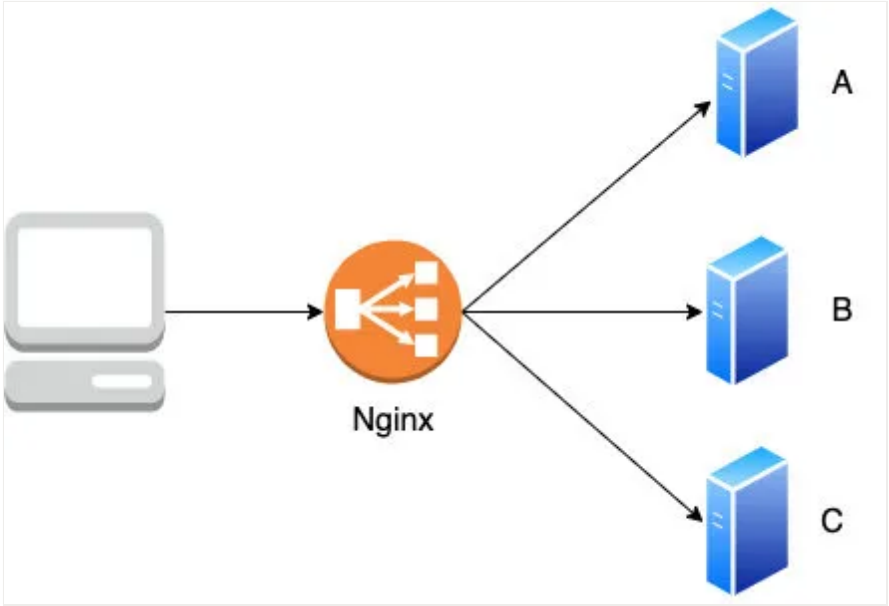
1. 首先用户登录，server 会为用户生成一个 session，为其分配唯一的 sessionId，这个 sessionId 是与某个用户绑定的，也就是说根据此 sessionid（假设为 abc） 可以查询到它到底是哪个用户，然后将此 sessionid 通过 cookie 传给浏览器
2. 之后浏览器的每次添加购物车请求中只要在 cookie 里带上 sessionId=abc 这一个键值对即可，server 根据 sessionId 找到它对应的用户后，把传过来的商品 id 保存到 server 中对应用户的购物车即可

可以看到通过这种方式再也不需要在 cookie 里传所有的购物车的商品 id 了，大大减轻了请求的负担！

另外通过上文不难观察出 **cookie 是存储在 client 的，而 session 保存在 server**，sessionId 需要借助 cookie 的传递才有意义。

session 的痛点

看起来通过 cookie + session 的方式是解决了问题， 但是我们忽略了一个问题，上述情况能正常工作是因为我们假设 server 是单机工作的，但实际在生产上，为了保障高可用，一般服务器至少需要两台机器，通过负载均衡的方式来决定到底请求该打到哪台机器上。



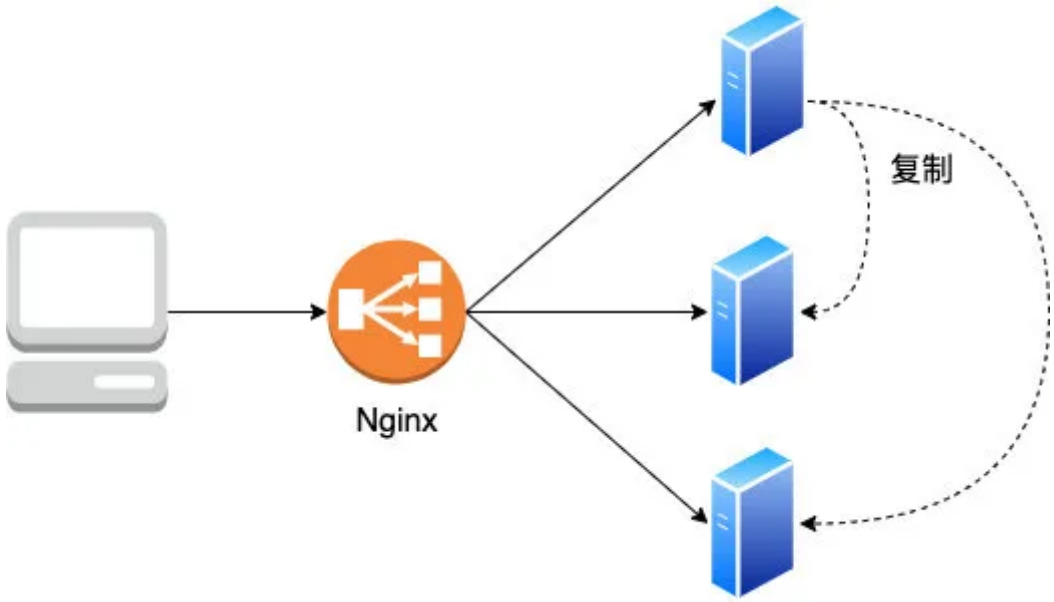
balance

如图示：客户端请求后，由负载均衡器（如 Nginx）来决定到底打到哪台机器

假设登录请求打到了 A 机器，A 机器生成了 session 并在 cookie 里添加 sessionId 返回给了浏览器，那么问题来了：下次添加购物车时如果请求打到了 B 或者 C，由于 session 是在 A 机器生成的，此时的 B,C 是找不到 session 的，那么就会发生无法添加购物车的错误，就得重新登录了，此时请问该怎么办。主要有以下三种方式

1、session 复制

A 生成 session 后复制到 B，C，这样每台机器都有一份 session，无论添加购物车的请求打到哪台机器，由于 session 都能找到，故不会有问题



balance (1)

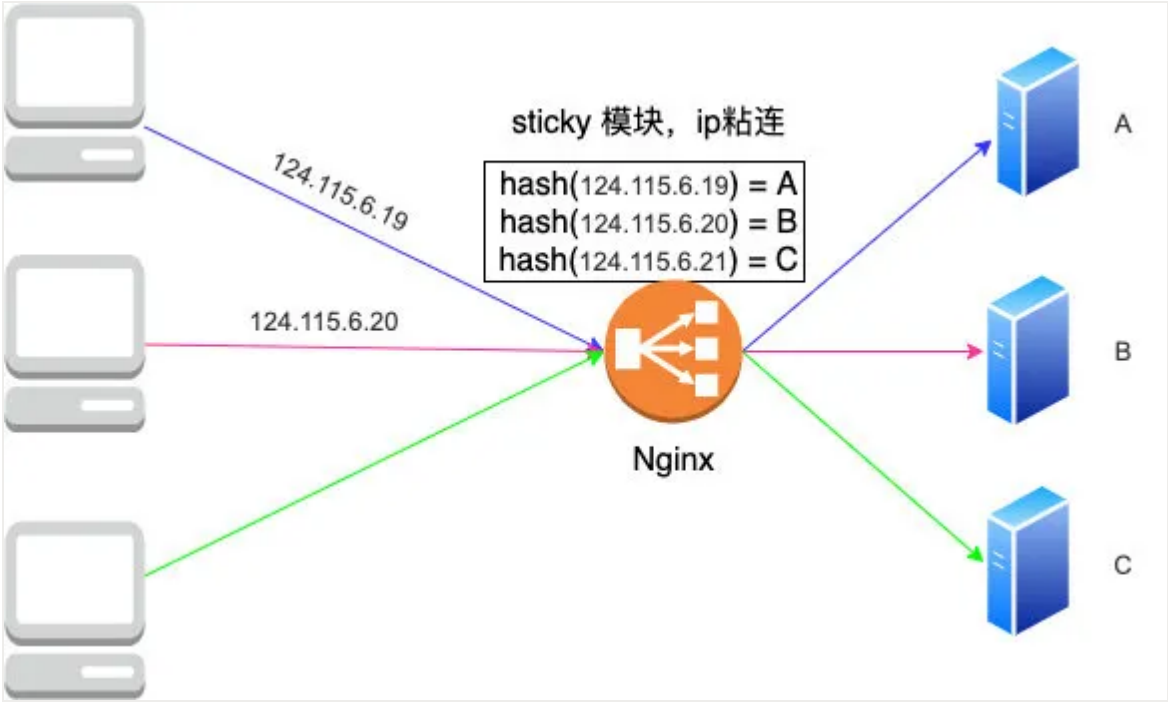
这种方式虽然可行，但缺点也很明显：

1. 同一样的一份 session 保存了多份，数据冗余
2. 如果节点少还好，但如果节点多的话，特别是像阿里，微信这种由于 DAU 上亿，可能需要部署成千上万台机器，这样节点增多复制造成的性能消耗也会很大。

> 2、session 粘连

这种方式是让每个客户端请求只打到固定的一台机器上，比如浏览器登录请求打到 A 机器后，后续所有的添加购物车请求也都打到 A 机器上，Nginx 的 sticky 模块可以支持这种方式，支持按 ip 或 cookie 粘连等等，如按 ip 粘连方式如下

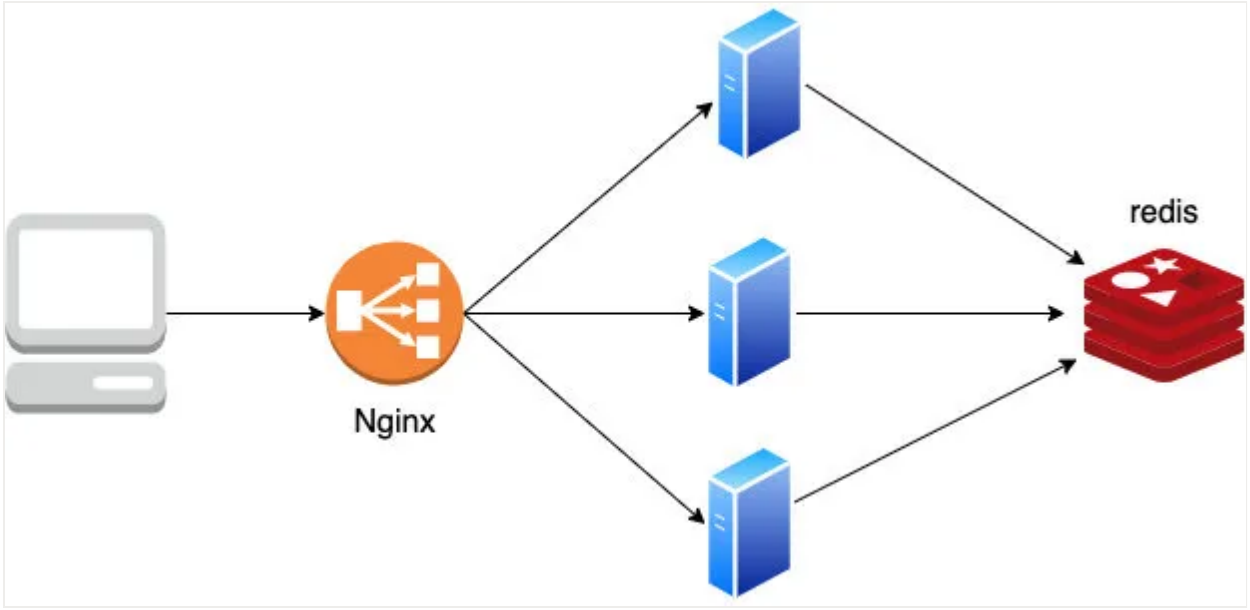
```
upstream tomcats {
    ip_hash;
    server 10.1.1.107:88;
    server 10.1.1.132:80;
}
```



这样的话每个 client 请求到达 Nginx 后，只要它的 ip 不变，根据 ip hash 算出来的值会打到固定的机器上，也就不存在 session 找不到的问题了，当然不难看出这种方式缺点也是很明显，对应的机器挂了怎么办？

> 3、session 共享

这种方式也是目前各大公司普遍采用的方案，将 session 保存在 redis，memcached 等中间件中，请求到来时，各个机器去这些中间件取一下 session 即可。

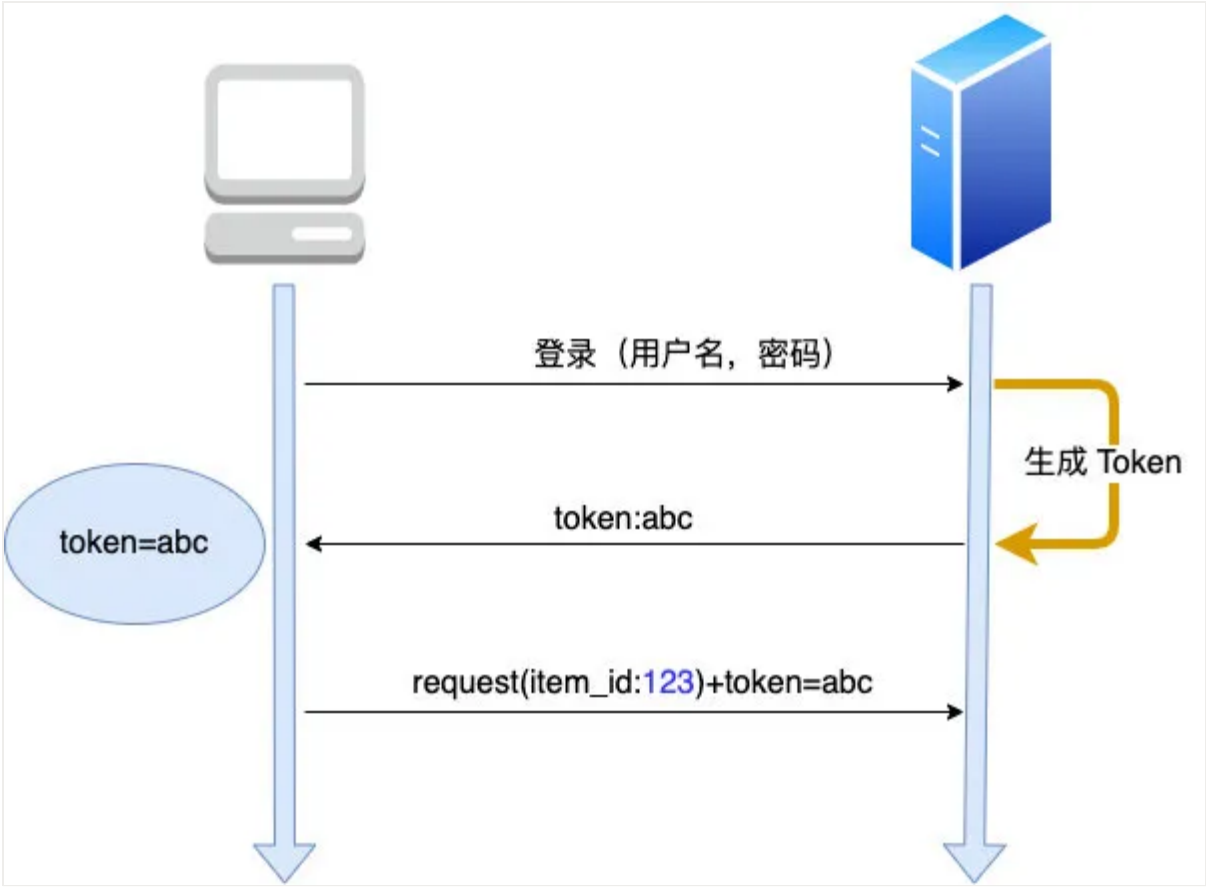


缺点其实也不难发现，就是每个请求都要去 redis 取一下 session，多了一次内部连接，消耗了一点性能，另外为了保证 redis 的高可用，必须做集群，当然了对于大公司来说，redis 集群基本都会部署，所以这方案可以说是大公司的首选了。

Token：no session!

通过上文分析我们知道通过在服务端共享 session 的方式可以完成用户的身份定位，但是不难发现也有一个小小的瑕疵：搞个校验机制我还得搭个 redis 集群？大厂确实 redis 用得比较普遍，但对于小厂来说可能它的业务量还未达到用 redis 的程度，所以有没有其他不用 server 存储 session 的用户身份校验机制呢，这就是我们今天要介绍的主角：token。

首先请求方输入自己的用户名，密码，然后 server 据此生成 token，客户端拿到 token 后会保存到本地，之后向 server 请求时在请求头带上此 token 即可。



相信大家看了上图会发现存在两个问题

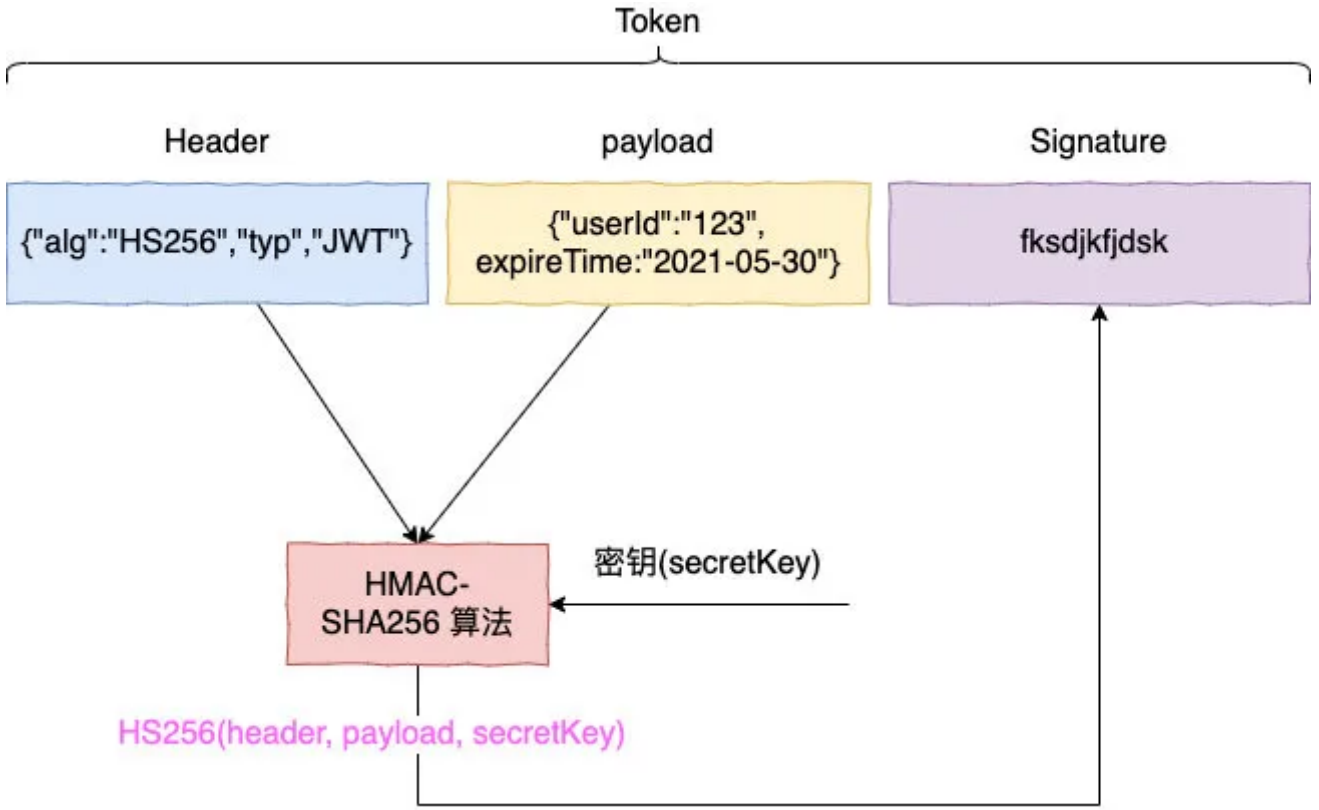
1、 token 只存储在浏览器中，服务端却没有存储，这样的话我随便搞个 token 传给 server 也行？

答：server 会有一套校验机制，校验这个 token 是否合法。

2、怎么不像 session 那样根据 sessionId 找到 userid 呢，这样的话怎么知道是哪个用户？

答：token 本身携带 uid 信息

第一个问题，如何校验 token 呢？我们可以借鉴 HTTPS 的签名机制来校验。先来看 jwt token 的组成部分



可以看到 token 主要由三部分组成

1. header：指定了签名算法
2. payload：可以指定用户 id，过期时间等非敏感数据

3. Signature: 签名，server 根据 header 知道它该用哪种签名算法，再用密钥根据此签名算法对 head + payload 生成签名，这样一个 token 就生成了。

当 server 收到浏览器传过来的 token 时，它会首先取出 token 中的 header + payload，根据密钥生成签名，然后再与 token 中的签名比对，如果成功则说明签名是合法的，即 token 是合法的。而且你会发现 payload 中存有我们的 userId，所以拿到 token 后直接在 payload 中就可获取 userid，避免了像 session 那样要从 redis 去取的开销

画外音：header，payload 实际上是以 base64 的形式存在的，文中为了描述方便，省去了这一步。

你会发现这种方式确实很妙，只要 server 保证密钥不泄露，那么生成的 token 就是安全的，因为如果伪造 token 的话在签名验证环节是无法通过的，就此即可判定 token 非法。

可以看到通过这种方式有效地避免了 token 必须保存在 server 的弊端，实现了分布式存储，不过需要注意的是，token 一旦由 server 生成，它就是有效的，直到过期，无法让 token 失效，除非在 server 为 token 设立一个黑名单，在校验 token 前先过一遍此黑名单，如果在黑名单里则此 token 失效，但一旦这样做的话，那就意味着黑名单就必须保存在 server，这又回到了 session 的模式，那直接用 session 不香吗。所以一般的做法是当客户端登出要让 token 失效时，直接在本地移除 token 即可，下次登录重新生成 token 就好。

另外需要注意的是 token 一般是放在 header 的 Authorization 自定义头里，不是放在 Cookie 里的，这主要是为了解决跨域不能共享 Cookie 的问题 （下文详述）

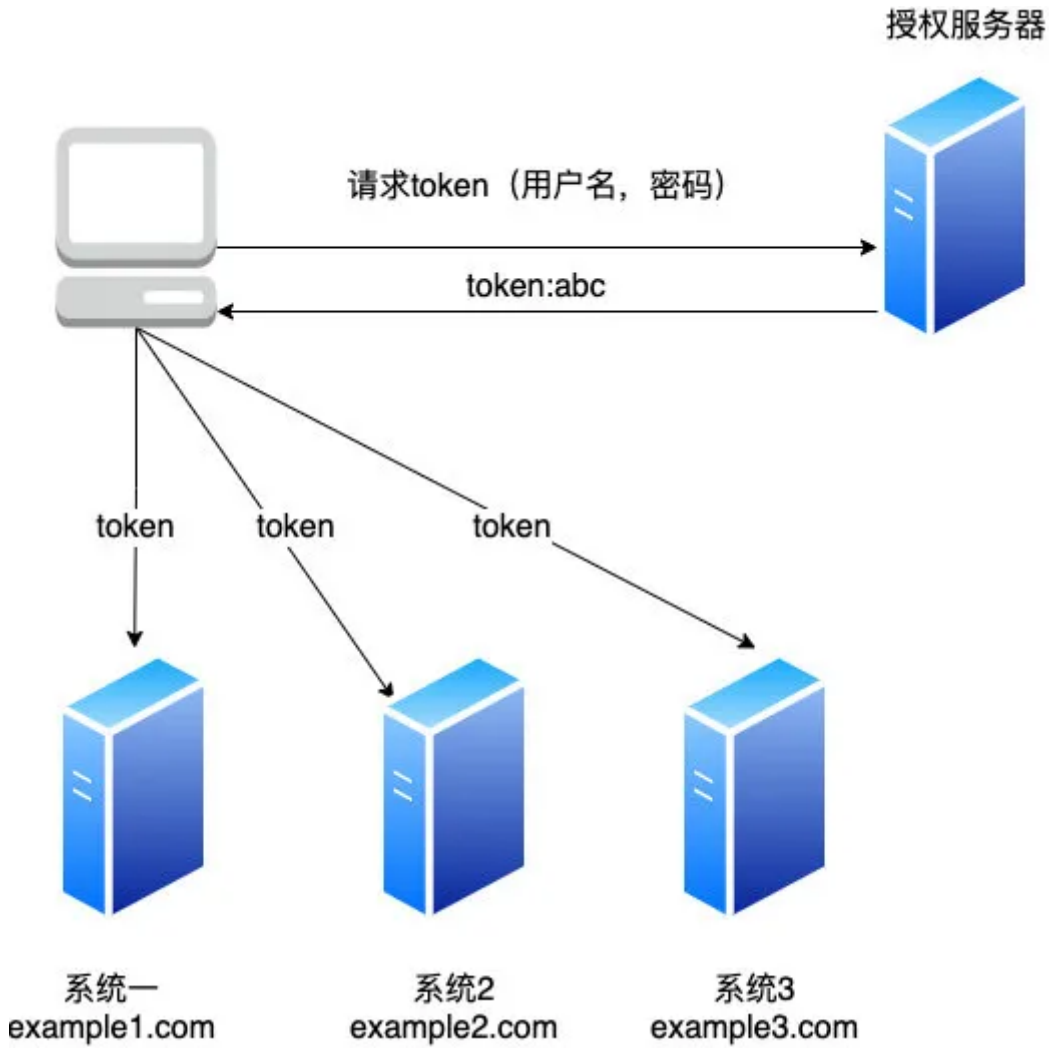
Cookie 与 Token 的简单总结

> Cookie 有哪些局限性?

1、 Cookie 跨站是不能共享的，这样的话如果你要实现多应用（多系统）的单点登录（SSO），使用 Cookie 来做需要的话就很困难了（要用比较复杂的 trick 来实现，有兴趣的话可以看文末参考链接）

画外音：所谓单点登录，是指在多个应用系统中，用户只需要登录一次就可以访问所有相互信任的应用系统。

但如果用 token 来实现 SSO 会非常简单，如下



只要在 header 中的 authorize 字段（或其他自定义）加上 token 即可完成所有跨域站点的认证。

2、 在移动端原生请求是没有 cookie 之说的，而 sessionid 依赖于 cookie，sessionid 就不能用 cookie 来传了，如果用 token 的话，由于它是随着 header 的 authoriize 传过来的，也就不存在此问题，换句话说token 天生支持移动平台，可扩展性好

综上所述，token 具有存储实现简单，扩展性好这些特点。

> token 有哪些缺点

那有人就问了，既然 token 这么好，那为什么各个大公司几乎都采用共享 session 的方式呢，可能很多人是第一次听到 token，token 不香吗？token 有以下两点劣势：

1、 token 太长了

token 是 header, payload 编码后的样式，所以一般要比 sessionId 长很多，很有可能超出 cookie 的大小限制（cookie 一般有大小限制的，如 4kb），如果你在 token 中存储的信息越长，那么 token 本身也会越长，这样的话由于你每次请求都会带上 token，对请求来是个不小的负担

2、 不太安全

网上很多文章说 token 更安全，其实不然，细心的你可能发现了，我们说 token 是存在浏览器的，再细问，存在浏览器的哪里？既然它太长放在 cookie 里可能导致 cookie 超限，那就只好放在 local storage 里，这样会造成安全隐患，因为 local storage 这类的本地存储是可以被 JS 直接读取的，另外由上文也提到，token 一旦生成无法让其失效，必须等到其过期才行，这样的话如果服务端检测到了一个安全威胁，也无法使相关的 token 失效。

所以 token 更适合一次性的命令认证，设置一个比较短的有效期

误解: Cookie 相比 token 更不安全，比如 CSRF 攻击

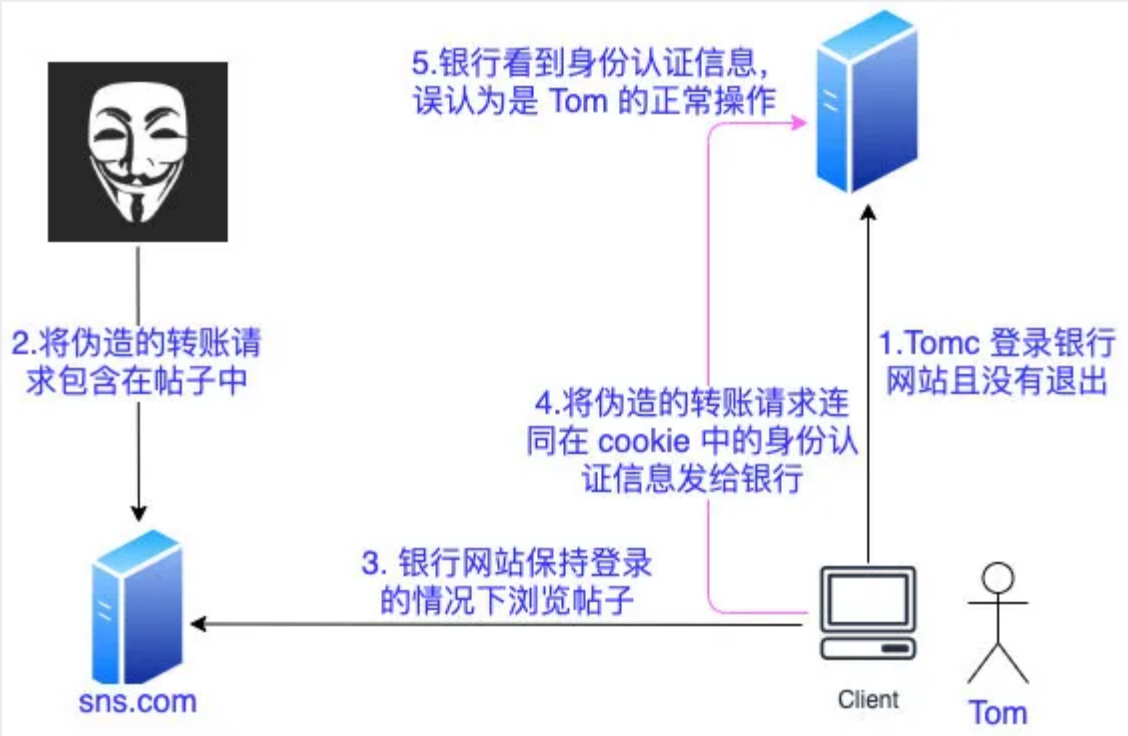
首先我们需要解释下 CSRF 攻击是怎么回事

攻击者通过一些技术手段欺骗用户的浏览器去访问一个自己曾经认证过的网站并运行一些操作（如发邮件，发消息，甚至财产操作如转账和购买商品）。由于浏览器曾经认证过（cookie 里带来 sessionId 等身份认证的信息），所以被访问的网站会认为是真正的用户操作而去运行。

比如用户登录了某银行网站（假设为 **http://www.examplebank.com/**，并且转账地址为 **http://www.examplebank.com/withdraw?amount=1000&transferTo=PayeeName**），登录后 cookie 里会包含登录用户的 sessionid，攻击者可以在另一个网站上放置如下代码

```
<img src="http://www.examplebank.com/withdraw?account=Alice&amount=1000&for=Badman">
```

那么如果正常的用户误点了上面这张图片，由于相同域名的请求会自动带上 cookie，而 cookie 里带有正常登录用户的 sessionid，类似上面这样的转账操作在 server 就会成功，会造成极大的安全风险



CSRF 攻击的根本原因在于对于同样域名的每个请求来说，它的 cookie 都会被自动带上，这个是浏览器的机制决定的，所以很多人据此认定 cookie 不安全。

使用 token 确实避免了CSRF 的问题，但正如上文所述，由于 token 保存在 local storage，它会被 JS 读取，**从存储角度来看**也不安全（实际上防护 CSRF 攻击的正确方式是用 CSRF token）

所以不管是 cookie 还是 token，从存储角度来看其实都不安全，都有暴露的风险，我们所说的安全更多的是强调传输中的安全，可以用 HTTPS 协议来传输， 这样的话请求头都能被加密，也就保证了传输中的安全。

其实我们把 cookie 和 token 比较本身就不合理，一个是存储方式，一个是验证方式，正确的比较应该是 session vs token。

总结

session 和 token 本质上是没有区别的，都是对用户身份的认证机制，只是他们实现的校验机制不一样而已（一个保存在 server，通过在 redis 等中间件获取来校验，一个保存在 client，通过签名校验的方式来校验），多数场景上使用 session 会更合理，但如果在单点登录，一次性命令认证上使用 token 会更合适，最好在不同的业务场景中合理选型，才能达到事半功倍的效果。

巨人的肩膀

- Cookie Session跨站无法共享问题(单点登录解决方案)：https://blog.csdn.net/wtopps/article/details/75040224
- http://cryto.net/~joepie91/blog/2016/06/13/stop-using-jwt-for-sessions/

这是我部署的机器人，请勿调戏！

推荐阅读

嘘！刚刚发现了一个山寨版某库....

可怕！公司部署了一个东西，悄悄盯着你…

发小被绿，我竭尽所学黑科技，动用云控捉奸寻找证据....

终于把废旧电脑变成了服务器！差点被女票拿去换洗脸盆，真香！

喜欢此内容的人还喜欢

技术总监：这25种烂代码，发现一个罚1000块！

漫話編程

Facebook開源C++11 組件庫，真香!

C語言與C++編程

一文看懂Nginx 架構

程序員的那些事