深入了解 Token 认证的来龙去脉

2018-02-08 08:39:55 分类：Web开发 (/category/WebDev1024/)

来自：边城的博客 (https://my.oschina.net/jamesfancy/blog/1613994)，作者公众号"边城客栈"，Githubhttps://github.com/jamesfancy (https://github.com/jamesfancy)，微博@边城客栈-JFan

摘要: Token 是在服务端产生的。如果前端使用用户名/密码向服务端请求认证，服务端认证成功，那么在服务端会返回 Token 给前端。前端可以在每次请求的时候带上 Token 证明自己的合法地位

不久前，我在在前后端分离实践 (https://www.itcodemonkey.com/article/1917.html)中提到了基于 Token 的认证，现在我们稍稍深入一些。

通常情况下，我们在讨论某个技术的时候，都是从问题开始。那么第一个问题：

为什么要用 Token？

而要回答这个问题很简单——因为它能解决问题！

可以解决哪些问题呢？

1、Token 完全由应用管理，所以它可以避开同源策略

2、Token 可以避免 CSRF 攻击 (https://www.itcodemonkey.com/article/2125.html)

3、Token 可以是无状态的，可以在多个服务间共享

Token 是在服务端产生的。如果前端使用用户名/密码向服务端请求认证，服务端认证成功，那么在服务端会返回 Token 给前端。前端可以在每次请求的时候带上 Token 证明自己的合法地位。如果这个 Token 在服务端持久化（比如存入数据库），那它就是一个永久的身份令牌。

于是，又一个问题产生了：需要为 Token 设置有效期吗？

最新文章

1 苹果遭遇史上最严厉的泄...

2 开源 AI 技术潜在危机爆发...

3 OpenWall 正式开源 Linux ...

4 快播王欣出狱，多位大佬...

5 Google 已经能用 AI 来预...

 吗?

对于这个问题，我们不妨先看两个例子。一个例子是登录密码，一般要求定期改变密码，以防止泄漏，所以密码是有有效期的；另一个例子是安全证书。SSL 安全证书都有有效期，目的是为了解决吊销的问题，对于这个问题的详细情况，来看看知乎的回答 (https://www.zhihu.com/question/20803288)。所以无论是从安全的角度考虑，还是从吊销的角度考虑，Token 都需要设有效期。

那么有效期多长合适呢？

只能说，根据系统的安全需要，尽可能的短，但也不能短得离谱——想像一下手机的自动熄屏时间，如果设置为 10 秒钟无操作自动熄屏，再次点亮需要输入密码，会不会疯？如果你觉得不会，那就亲自试一试，设置成可以设置的最短时间，坚持一周就好（不排除有人适应这个时间，毕竟手机厂商也是有用户体验研究的）。

然后新问题产生了，如果用户在正常操作的过程中，Token 过期失效了，要求用户重新登录……用户体验岂不是很糟糕？

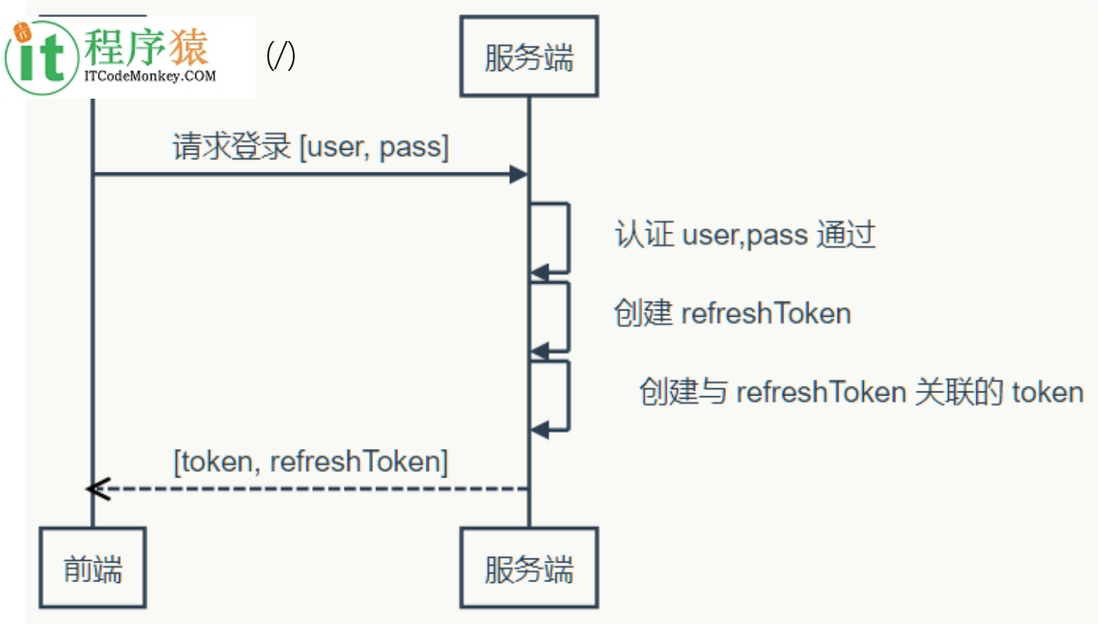
为了解决在操作过程不能让用户感到 Token 失效这个问题，有一种方案是在服务器端保存 Token 状态，用户每次操作都会自动刷新（推迟）Token 的过期时间——Session 就是采用这种策略来保持用户登录状态的。然而仍然存在这样一个问题，在前后端分离、单页 App 这些情况下，每秒种可能发起很多次请求，每次都去刷新过期时间会产生非常大的代价。如果 Token 的过期时间被持久化到数据库或文件，代价就更大了。所以通常为了提升效率，减少消耗，会把 Token 的过期时保存在缓存或者内存中。

还有另一种方案，使用 Refresh Token，它可以避免频繁的读写操作。这种方案中，服务端不需要刷新 Token 的过期时间，一旦 Token 过期，就反馈给前端，前端使用 Refresh Token 申请一个全新 Token 继续使用。这种方案中，服务端只需要在客户端请求更新 Token 的时候对 Refresh Token 的有效性进行一次检查，大大减少了更新有效期的操作，也就避免了频繁读写。当然 Refresh Token 也是有有效期的，但是这个有效期就可以长一点了，比如，以天为单位的时间。

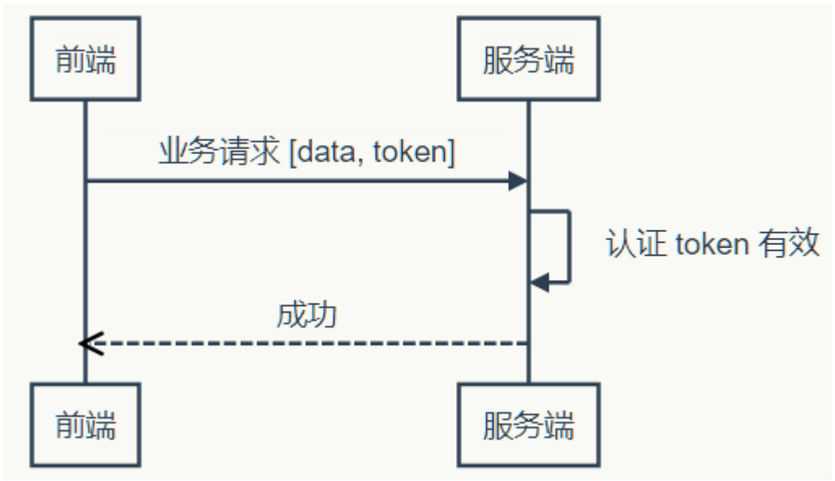
时序图表示

使用 Token 和 Refresh Token 的时序图如下：

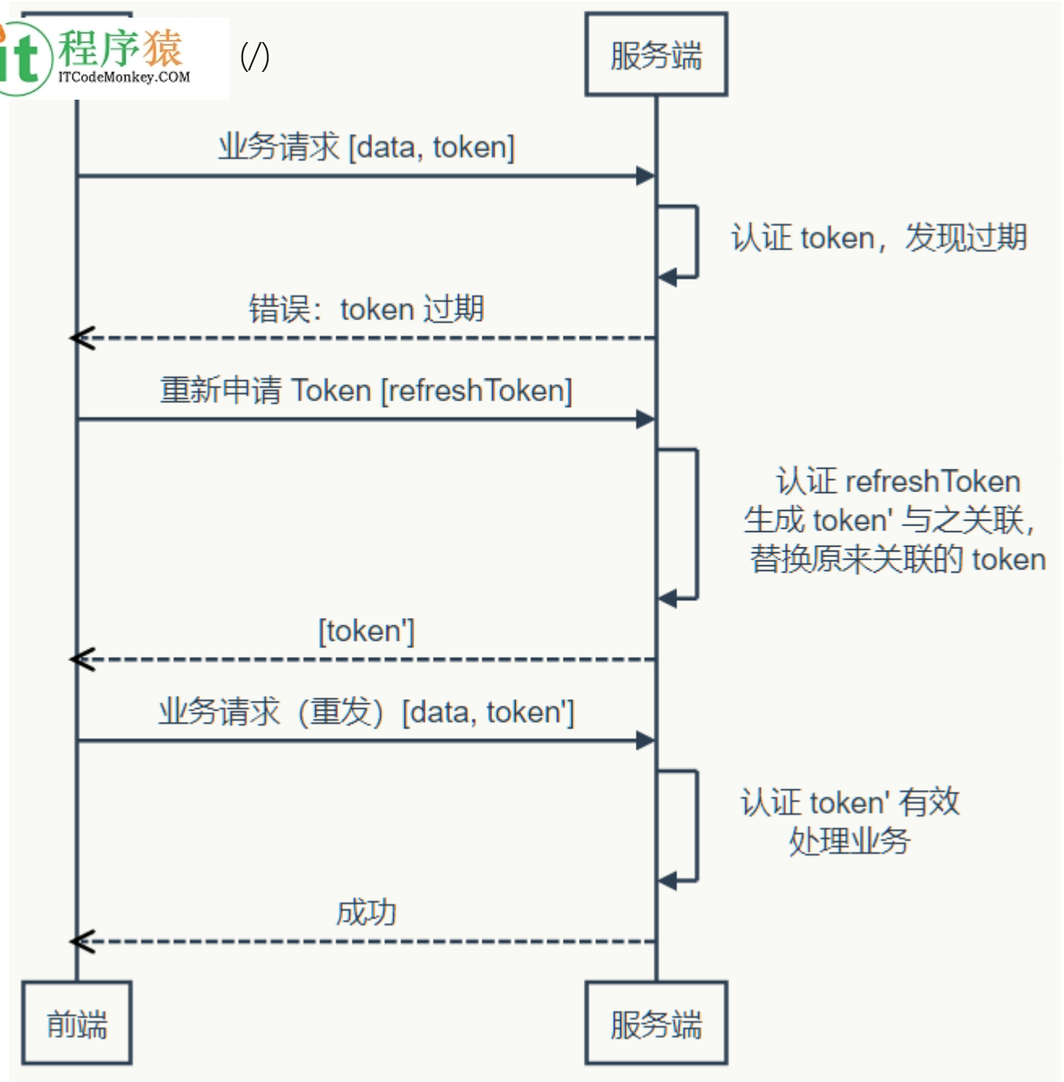
1) 登录



2) 业务请求



3) Token 过期，刷新 Token



上面的时序图中并未提到 Refresh Token 过期怎么办。不过很显然，Refresh Token 既然已经过期，就该要求用户重新登录了。

当然还可以把这个机制设计得更复杂一些，比如，Refresh Token 每次使用的时候，都更新它的过期时间，直到与它的创建时间相比，已经超过了非常长的一段时间（比如三个月），这等于是在相当长一段时间内允许 Refresh Token 自动续期。

到目前为止，Token 都是有状态的，即在服务端需要保存并记录相关属性。那说好的无状态呢，怎么实现？

无状态 Token

息都附加在 Token 上，服务器就可以不保存。但是服务端仍然需要认证 Token 有确认是自己签发的 Token，而且其信息未被改动过，那就可以认为 Token 有效——“签名”可以作此保证。平时常说的签名都存在一方签发，另一方验证的情况，所以要使用非对称加密算法。但是在这里，签发和验证都是同一方，所以对称加密算法就能达到要求，而对称算法比非对称算法要快得多（可达数十倍差距）。更进一步思考，对称加密算法除了加密，还带有还原加密内容的功能，而这一功能在对 Token 签名时并无必要——既然不需要解密，摘要（散列）算法就会更快。可以指定密码的散列算法，自然是 HMAC。

上面说了这么多，还需要自己去实现吗？不用！JWT (https://jwt.io/) 已经定义了详细的规范，而且有各种语言的若干实现。

不过在使用无状态 Token 的时候在服务端会有一些变化，服务端虽然不保存有效的 Token 了，却需要保存未到期却已注销的 Token。如果一个 Token 未到期就被用户主动注销，那么服务器需要保存这个被注销的 Token，以便下次收到使用这个仍在有效期内的 Token 时判其无效。有没有感到一点沮丧？

在前端可控的情况下（比如前端和服务端在同一个项目组内），可以协商：前端一但注销成功，就丢掉本地保存（比如保存在内存、LocalStorage 等）的 Token 和 Refresh Token。基于这样的约定，服务器就可以**假设**收到的 Token 一定是没注销的（因为注销之后前端就不会再使用了）。

如果前端不可控的情况，仍然可以进行上面的假设，但是这种情况下，需要尽量缩短 Token 的有效期，而且必须在用户主动注销的情况下让 Refresh Token 无效。这个操作**存在一定的安全漏洞**，因为用户会认为已经注销了，实际上在较短的一段时间内并没有注销。如果应用设计中，这点漏洞并不会造成什么损失，那采用这种策略就是可行的。

在使用无状态 Token 的时候，有两点需要注意：

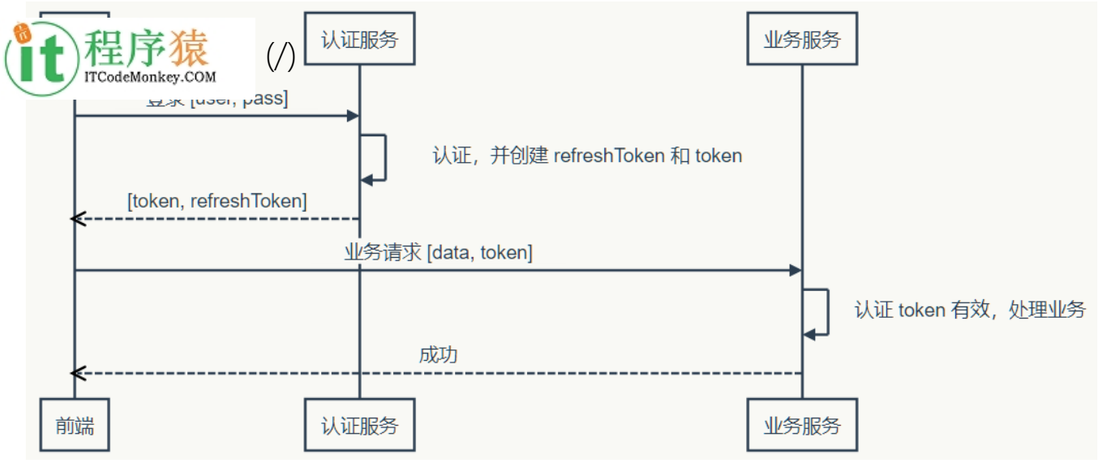
1、Refresh Token 有效时间较长，所以它应该在服务器端有状态，以增强安全性，确保用户注销时可控

2、应该考虑使用二次认证来增强敏感操作的安全性

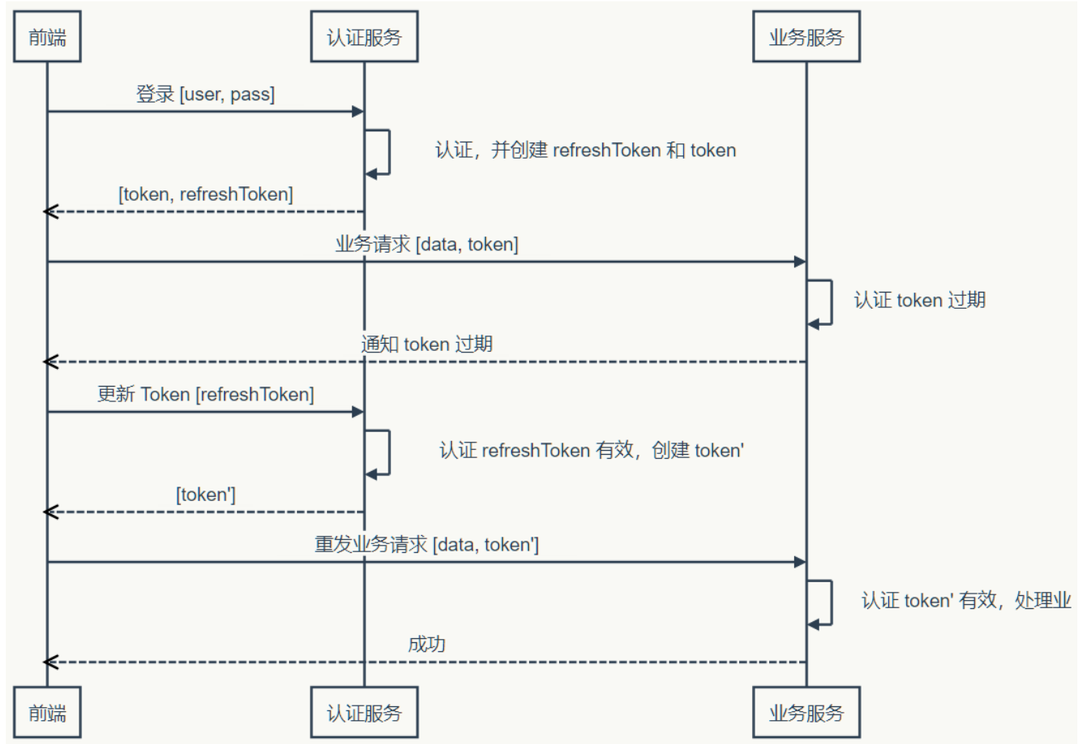
到此，关于 Token 的话题似乎差不多了——然而并没有，上面说的只是认证服务和业务服务集成在一起的情况，如果是分离的情况呢？

分离认证服务

当 Token 无状态之后，单点登录就变得容易了。前端拿到一个有效的 Token，它就可以在任何同一体系的服务上认证通过——只要它们使用同样的密钥和算法来认证 Token 的有效性。就样这样：



当然，如果 Token 过期了，前端仍然需要去认证服务更新 Token：



可见，虽然认证和业务分离了，实际即并没产生多大的差异。当然，这是建立在**认证服务器信任业务服务器的前提**下，因为认证服务器产生 Token 的密钥和业务服务器认证 Token 的密钥和算法相同。换句话说，业务服务器同样可以创建有效的 Token。

如果业务服务器不能被信任，该怎么办？

不受信的业务服务器

调到不受信任的业务服务器时，很容易想到的办法是使用不同的密钥。认证服务器使用密钥1签发，业务服务器使用密钥2验证。这是典型非对称加密签名的应用场景。认证服务器自己使用私钥对 Token 签名，公开公钥。信任这个认证服务器的业务服务器保存公钥，用于验证签名。幸好，JWT 不仅可以使用 HMAC 签名，也可以使用 RSA（一种非对称加密算法）签名。

不过，当业务服务器已经不受信任的时候，**多个业务服务器之间使用相同的 Token 对用户来说是不安全的**。因为任何一个服务器拿到 Token 都可以仿冒用户去另一个服务器处理业务……悲剧随时可能发生。

为了防止这种情况发生，就需要在认证服务器产生 Token 的时候，把使用该 Token 的业务服务器的信息记录在 Token 中，这样当另一个业务服务器拿到这个 Token 的时候，发现它并不是自己应该验证的 Token，就可以直接拒绝。

现在，认证服务器不信任业务服务器，业务服务器相互也不信任，但前端是信任这些服务器的——如果前端不信任，就不会拿 Token 去请求验证。那么为什么会信任？可能是因为这些是同一家公司或者同一个项目中提供的若干服务构成的服务体系。

但是，前端信任不代表用户信任。如果 Token 不没有携带用户隐私（比如姓名），那么用户不会关心信任问题。但如果 Token 含有用户隐私的时候，用户得关心信任问题了。这时候认证服务就不得不再啰嗦一些，当用户请求 Token 的时候，问上一句，你真的要授权给某某某业务服务吗？而这个“某某某”，用户怎么知道它是不是真的“某某某”呢？用户当然不知道，甚至认证服务也不知道，因为公钥已经公开了，任何一个业务都可以声明自己是“某某某”。

为了得到用户的信任，认证服务就不得不帮助用户来甄别业务服务。所以，认证服器决定不公开公钥，而是要求业务服务先申请注册并通过审核。只有通过审核的业务服务器才能得到认证服务为它创建的，仅供它使用的公钥。如果该业务服务泄漏公钥带来风险，由该业务服务自行承担。现在认证服务可以清楚的告诉用户，“某某某”服务是什么了。如果用户还是不够信任，认证服务甚至可以问，某某某业务服务需要请求 A、B、C 三项个人数据，其中 A 是必须的，不然它不工作，是否允许授权？如果你授权，我就把你授权的几项数据加密放在 Token 中……

废话了这么多，有没有似曾相识……对了，这类似开放式 API 的认证过程。开发式 API 多采用 OAuth 认证，而关于 OAuth 的探讨资源非常丰富，这里就不深究了。

上一篇：聊聊分布式事务，再说说解决方案 (/article/2163.html)

下一篇：PHP代码安全杂谈 (/article/2185.html)