一文读懂 JWT!

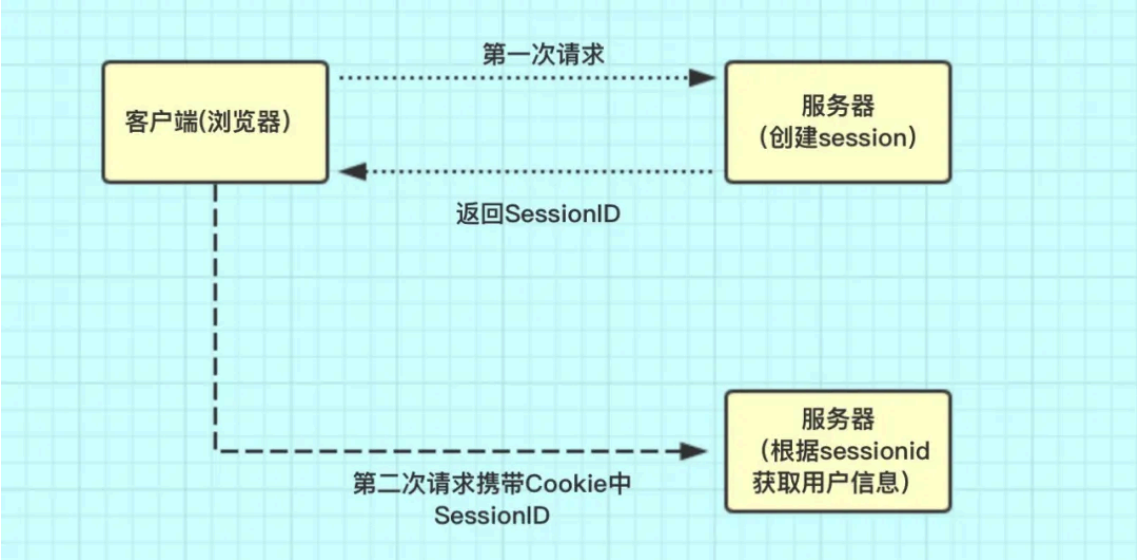
原创 binron 后端元宇宙 2022年12月14日 07:30 浙江

在JWT之前我们在做用户认证的时候,基本上会考虑 session 或者 token ,所以在讲JWT之前，我们先来回顾下这两个。

一、传统的session认证流程

1、session认证流程

session 是基于 cookie 实现的， session存储在服务器端 ，sessionId会被存储到客户端的cookie 中,具体流程如下



1. 用户第一次请求服务器的时候，服务器根据用户提交的相关信息，创建对应的 Session 。

2. 请求返回时将此 Session 的唯一标识信息 SessionID 返回给浏览器。

3. 浏览器接收到服务器返回的 SessionID 信息后，会将此信息存入到 Cookie 中，同时 Cookie 记录此 SessionID 属于该 域名 。

4. 当用户第二次访问服务器的时候，请求会自动将该域名下 Cookie 信息也发送给服务端，服务端会 从 Cookie 中获取 SessionID。

5. 再根据 SessionID 查找对应的 Session 信息，如果找到用户信息则可以执行后面操作，没找到说明没有登录或者登录失效。

总结 根据以上流程可知，SessionID 是连接 Cookie 和 Session 的一道桥梁，大部分系统也是根据此原理来验证用户登录状态。

2、使用session需要考虑的问题

1. 将 session 存储在服务器里面，当用户同时在线量比较多时，这些 session 会占据较多的内存。

2. 当系统采用集群部署的时候，多台 web 服务器之间如何做 session 共享的问题 。

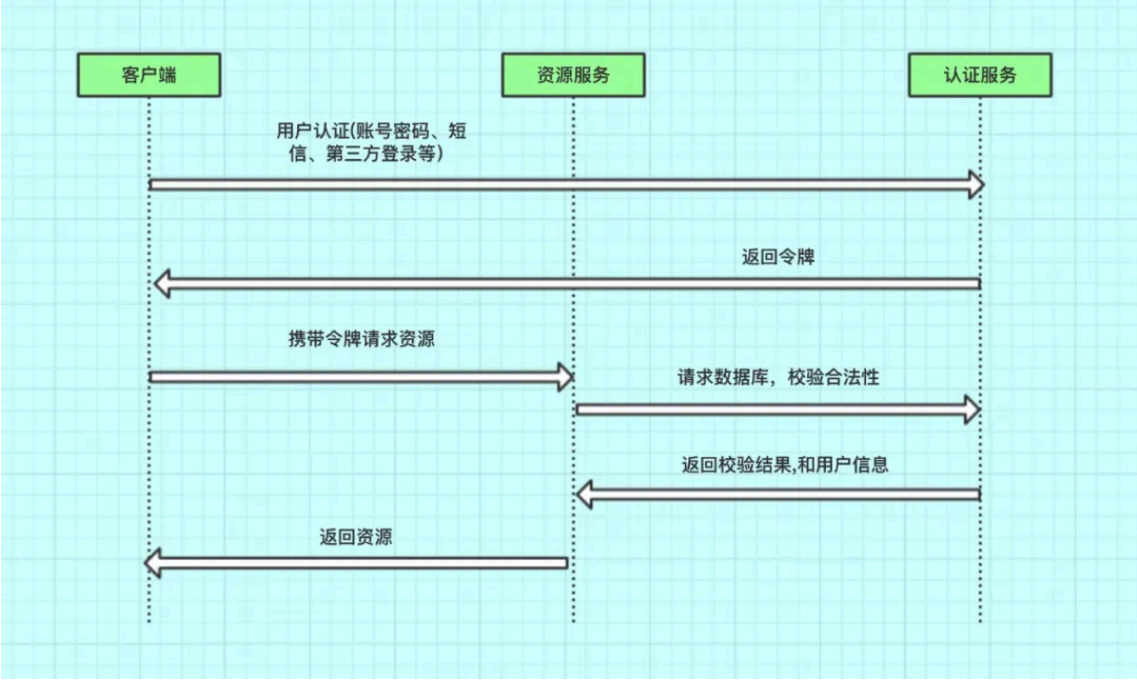
3. sessionId 是存储在 cookie 中的，假如浏览器禁止 cookie 或不支持 cookie 怎么办？

4. CSRF攻击：因为是基于cookie来进行用户识别的, cookie如果被截获，用户就会很容易受到跨站请求伪造的攻击。

二、常用的Token(令牌)认证流程

1、Token认证流程

它不需要在服务端去保留用户的认证信息或者会话信息，而是借助数据库(一般为redis保存认证信息）。



这个不是标准时序图，能看懂大致意思就行。

token认证流程

1. 客户端使用username跟password请求登录接口。

2. 服务端收到请求，去验证username跟password。

3. 验证成功后，服务端会签发一个 token 同时把这个token值作为key，用户信息为value存入redis中,并把这个 token 发送给客户端。

4. 客户端收到 token 以后，会把它存储起来，比如放在 cookie 里或者 localStorage 里。

5. 客户端每次向服务端请求资源的时候需要带上这个token。

6. 服务端收到请求，首先拿着这个token去redis查询用户信息，如果能查到用户信息说明已经登录过。如果查不到数据说明未登录或者登录失效。

2、token特点

1. 如果你认为用数据库来存储 token 会导致查询时间太长，可以选择放在内存当中。比如 redis 很适合你对 token 查询的需求。

2. token 完全由应用管理，所以它可以避开同源策略。

3. token 可以避免 CSRF 攻击(因为可以不需要 cookie 了)

4. 移动端对 cookie 的支持不是很好，而 session 需要基于 cookie 实现，所以移动端常用的是token

3、Token 和 Session 的区别

Session 是一种记录服务器和客户端会话状态的机制，使服务端有状态化，可以记录会话信息。

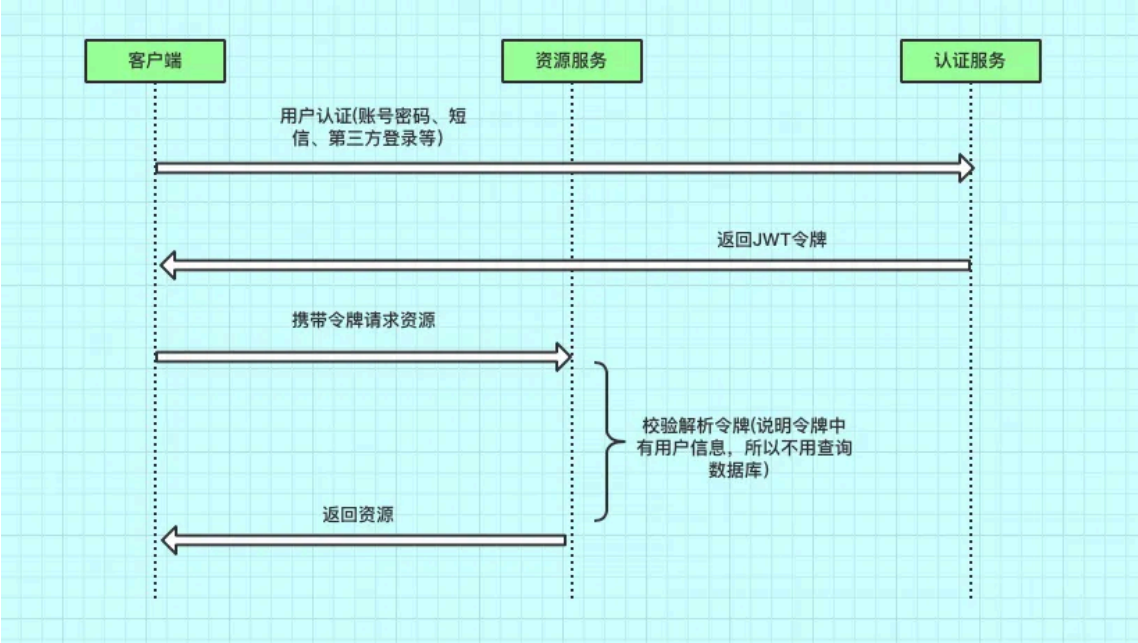
而 Token 是令牌，访问资源接口（API）时所需要的资源凭证。

三、聊一聊JWT

我们在使用session或者使用redis,前端cookie其实只是存了个key,我们还需要拿着这个key到服务端的session，或者redis或者MySQL，总之都需呀查一遍。

但如果是JWT，它最大的特点就是这个JWT本身就含有用户信息，服务端只要解析这个成功JWT，就可以获取用户信息。

1、JWT认证流程



1. 用户输入用户名/密码登录，服务端认证成功后，会返回给客户端一个 JWT。

2. 客户端将 jwt 保存到本地（通常使用 localstorage，也可以使用 cookie）。

3. 当用户希望访问一个受保护的路由或者资源的时候，需要请求头的 Authorization 字段中使用Bearer 模式添加 JWT，其内容看起来是下面这样 Authorization: Bearer jwt

4. 服务端的保护路由将会检查请求头 Authorization 中的 JWT 信息，如果合法，则允许用户的行为。

因为 JWT 是自身包含用户信息，因此减少了需要查询数据库的需要，用户的状态也不再存储在服务端的内存中，所以这是一种无状态的认证机制。

2、Token 和 JWT 的区别

相同

- 都是访问资源的令牌
- 都可以记录用户的信息
- 都是使服务端无状态化
- 都是只有验证成功后，客户端才能访问服务端上受保护的资源

区别

「**Token**」 服务端验证客户端发送过来的 Token 时，还需要查询数据库获取用户信息，然后验证 Token 是否有效。

「**JWT**」 服务端只需要使用密钥解密进行校验（校验也是 JWT 自己实现的）即可，不需要查询或者减少查询数据库，因为 JWT 自包含了用户信息和加密的数据。

3、JWT一些缺点

前面说了它最大的优点就是，当服务端拿到JWT之后,我们不需要向token样还需去查询数据库校验信息，因为JWT中就包含用户信息,所以减少一次数据的查询，但这样做也会带来很明显的问题

1、无法满足修改密码场景

因为上面说过，服务端拿到jwt是不会在去查询数据库的，所以就算你改了密码，服务端还是未知的。

盗号者在原 jwt 有效期之内依旧可以继续访问系统。

2、无法满足注销场景

传统的 session+cookie 方案用户点击注销，服务端清空 session 即可，因为状态保存在服务端。

但 jwt 的方案就比较难办了，因为 jwt 是无状态的，服务端通过计算来校验有效性。没有存储起来，所以即使客户端删除了 jwt，但是该 jwt 还是在有效期内，只不过处于一个游离状态。

3、无法满足token续签场景

我们知道微信只要你每天使用是不需要重新登录的，因为有token续签，因为传统的 cookie 续签方案一般都是框架自带的，session 有效期 30 分钟，30 分钟内如果有访问，session 有效期被刷新至 30 分钟。

但是 jwt 本身的 payload 之中也有一个 exp 过期时间参数，来代表一个 jwt 的时效性，而 jwt 想延期这个 exp 就有点身不由己了，因为 payload 是参与签名的，一旦过期时间被修改，整个 jwt 串就变了，jwt 的特性天然不支持续签！

四、JWT到底长什么样呢？

JWT由三部分组成，分别是 头信息 ， 有效载荷 , 签名 中间以 点(.) 分隔，具体如下

```
xxxxx.yyyyy.zzzzz
```

1、header(头信息)

由两部分组成，「令牌类型」（即：JWT）、「散列算法」（HMAC、RSASSA、RSASSA-PSS等），例如：

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

然后，这个JSON被编码为Base64Url，形成JWT的第一部分。

2、Payload（有效载荷）

JWT的第二部分是payload，其中包含claims。

claims是关于实体（常用的是用户信息）和其他数据的声明，claims有三种类型：registered, public, and private claims。

「**Registered claims**」：这些是一组预定义的claims，非强制性的，但是推荐使用， iss（发行人）， exp（到期时间）， sub（主题）， aud（观众）等；

「**Public claims**」：自定义claims，注意不要和JWT注册表中属性冲突，这里可以查看JWT注册表。

「**Private claims**」：这些是自定义的claims，用于在同意使用这些claims的各方之间共享信息，它们既不是Registered claims，也不是Public claims。

示例

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

然后，再经过Base64Url编码，形成JWT的第二部分；

注意：对于签名令牌，此信息虽然可以防止篡改，但任何人都可以读取。除非加密，否则不要将敏感信息放入到Payload或Header元素中。

3、Signature(签名)

jwt的第三部分是一个签证信息，这个签证信息由三部分组成： `header (base64后的)` ， `payload (base64后的)`， `secret` 这个部分需要base64加密后的header和base64加密后的payload使用.连接组成的字符串，然后通过header中声明的加密方式进行加盐secret组合加密，然后就构成了jwt的第三部分。

```
var encodedString = base64UrlEncode(header) + '.' + base64UrlEncode(payload);

var signature = HMACSHA256(encodedString, 'secret'); // TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7H
```

将这三部分用.连接成一个完整的字符串,构成了最终的jwt:

```
 eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9
.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

注意 secret是保存在服务器端的，jwt的签发生成也是在服务器端的，secret就是用来进行jwt的签发和jwt的验证，所以，它就是你服务端的私钥，在任何场景都不应该流露出去。

一旦客户端得知这个secret, 那就意味着客户端是可以自我签发jwt了。

后端元宇宙

专注后端技术栈，热爱分享，热爱工作总结，更新或许不是很频繁，但会一直坚持原创...

58篇原创内容

公众号

求一键三连：点赞、转发、在看。

单点登录 3

单点登录 · 目录

下一篇 · 看完这篇不能再说不懂SSO原理了！