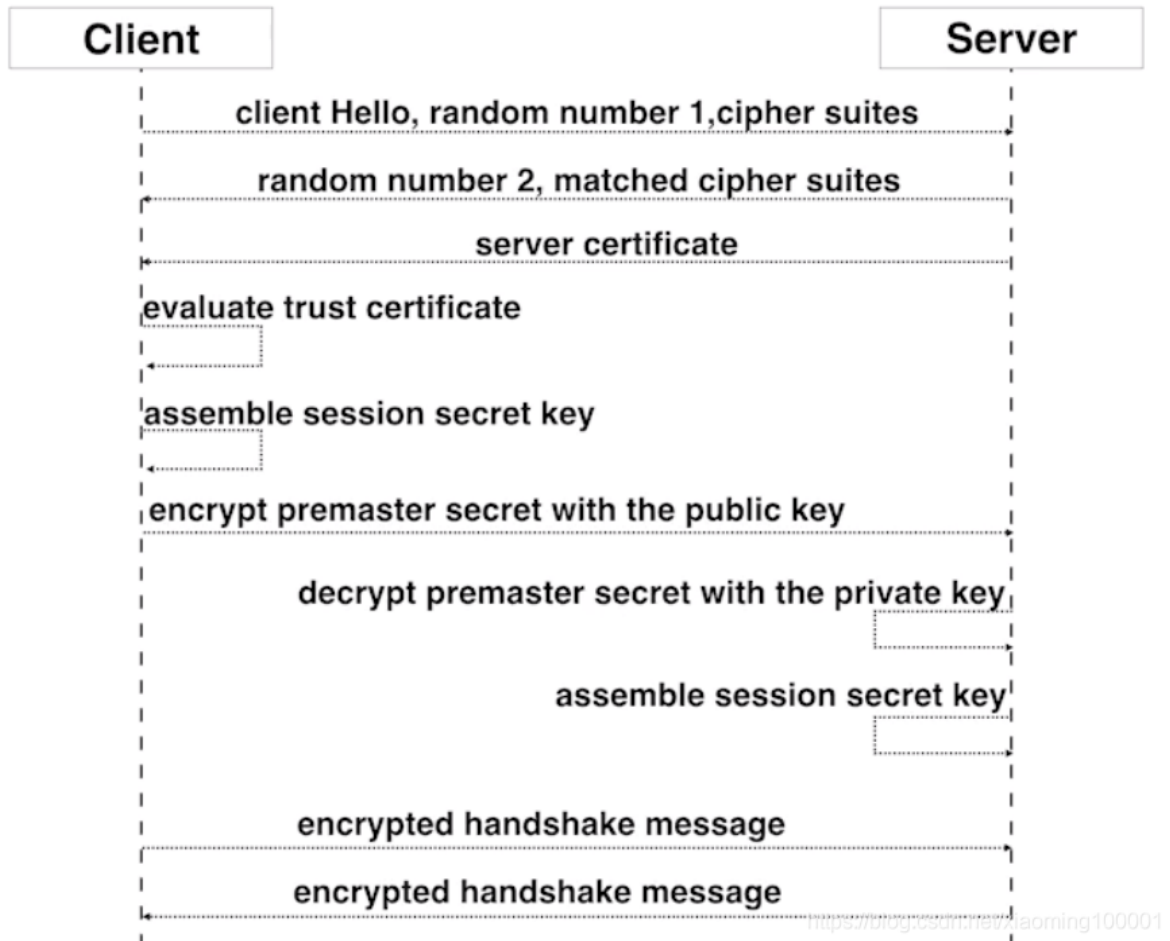


安全

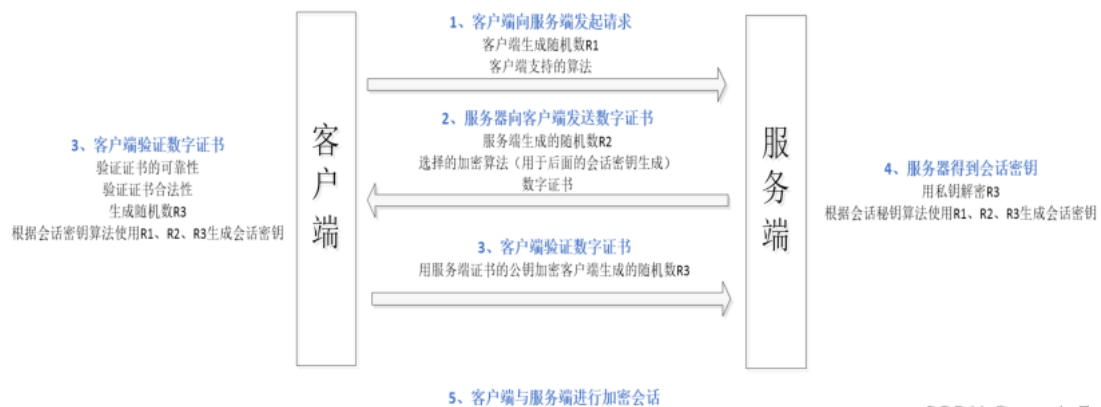
http 与 https 的区别

http 是明文传输的可以针对一些敏感信息，主要中间抓包就可以拿到这些信息

而 https 则是将发送的数据进行加密，加密流程如下：



Https 的整个流程



CSRF

攻击案例

受害者 Bob 在银行有一笔存款，通过对银行的网站发送请求 <http://bank.example/withdraw?account=bob&amount=1000000&for=bob2> 可以使 Bob 把 1000000 的存款转到 bob2 的账号下。通常情况下，该请求发送到网站后，服务器会先验证该请求是否来自一个合法的 session，并且该 session 的用户 Bob 已经成功登陆。

黑客 Mallory 自己在该银行也有账户，他知道上文中的 URL 可以把钱进行转账操作。Mallory 可以自己发送一个请求给银行：<http://bank.example/withdraw?account=bob&amount=1000000&for=Mallory>。但是这个请求来自 Mallory 而非 Bob，他不能通过安全认证，因此该请求不会起作用。

这时，Mallory 想到使用 CSRF 的攻击方式，他先自己做一个网站，在网站中放入如下代码：`src="http://bank.example/withdraw?account=bob&amount=1000000&for=Mallory"`，并且通过广告等诱使 Bob 来访问他的网站。当 Bob 访问该网站时，上述 url 就会从 Bob 的浏览器发向银行，而这个请求会附带 Bob 浏览器中的 cookie 一起发向银行服务器。大多数情况下，该请求会失败，因为他要求 Bob 的认证信息。但是，如果 Bob 当时恰巧刚访问他的银行后不久，他的浏览器与银行网站之间的 session 尚未过期，浏览器的 cookie 之中含有 Bob 的认证信息。这时，悲剧发生了，这个 url 请求就会得到响应，钱将从 Bob 的账号转移到 Mallory 的账号，而 Bob 当时毫不知情。等以后 Bob 发现账户钱少了，即使他去银行查询日志，他也只能发现确实有一个来自于他本人的合法请求转移了资金，没有任何被攻击的痕迹。而 Mallory 则可以拿到钱后逍遥法外。

防御方法

防御CSRF攻击：

目前防御 CSRF 攻击主要有三种策略：验证 HTTP Referer 字段；在请求地址中添加 token 并验证；在 HTTP 头中自定义属性并验证。

(1) 验证 HTTP Referer 字段

根据 HTTP 协议，在 HTTP 头中有一个字段叫 Referer，它记录了该 HTTP 请求的来源地址。在通常情况下，访问一个安全受限页面的请求来自于同一个网站，比如需要访问 <http://bank.example/withdraw?account=bob&amount=1000000&for=Mallory>，用户必须先登陆 bank.example，然后通过点击页面上的按钮来触发转账事件。这时，该转账请求的 Referer 值就会是转账按钮所在的页面的 URL，通常是以 bank.example 域名开头的地址。而如果黑客要对银行网站实施 CSRF 攻击，他只能在他自己的网站构造请求，当用户通过黑客的网站发送请求到银行时，该请求的 Referer 是指向黑客自己的网站。因此，要防御 CSRF 攻击，银行网站只需要对于每一个转账请求验证其 Referer 值，如果是 bank.example 开头的域名，则说明该请求是来自银行网站自己的请求，是合法的。如果 Referer 是其他网站的话，则有可能是黑客的 CSRF 攻击，拒绝该请求。

这种方法的显而易见的好处就是简单易行，网站的普通开发人员不需要操心 CSRF 的漏洞，只需要在最后给所有安全敏感的请求统一增加一个拦截器来检查 Referer 的值就可以。特别是对于当前现有的系统，不需要改变当前系统的任何已有代码和逻辑，没有风险，非常便捷。

然而，这种方法并非万无一失。Referer 的值是由浏览器提供的，虽然 HTTP 协议上有明确的要求，但是每个浏览器对于 Referer 的具体实现可能有差别，并不能保证浏览器自身没有安全漏洞。使用验证 Referer 值的方法，就是把安全性都依赖于第三方（即浏览器）来保障，从理论上讲，这样并不安全。事实上，对于某些浏览器，比如 IE6 或 FF2，目前已经有一些方法可以篡改 Referer 值。如果

bank.example 网站支持 IE6 浏览器，黑客完全可以把用户浏览器的 Referer 值设为以 bank.example 域名开头的地址，这样就可以通过验证，从而进行 CSRF 攻击。

即便是使用最新的浏览器，黑客无法篡改 Referer 值，这种方法仍然有问题。因为 Referer 值会记录下用户的访问来源，有些用户认为这样会侵犯到他们自己的隐私权，特别是有些组织担心 Referer 值会把组织内网中的某些信息泄露到外网中。因此，用户自己可以设置浏览器使其在发送请求时不再提供 Referer。当他们正常访问银行网站时，网站会因为请求没有 Referer 值而认为是 CSRF 攻击，拒绝合法用户的访问。

(2) 在请求地址中添加 token 并验证

CSRF 攻击之所以能够成功，是因为黑客可以完全伪造用户的请求，该请求中所有的用户验证信息都是存在于 cookie 中，因此黑客可以在不知道这些验证信息的情况下直接利用用户自己的 cookie 来通过安全验证。要抵御 CSRF，关键在于在请求中放入黑客所不能伪造的信息，并且该信息不存在于 cookie 之中。可以在 HTTP 请求中以参数的形式加入一个随机产生的 token，并在服务器端建立一个拦截器来验证这个 token，如果请求中没有 token 或者 token 内容不正确，则认为可能是 CSRF 攻击而拒绝该请求。

这种方法要比检查 Referer 要安全一些，token 可以在用户登陆后产生并放于 session 之中，然后在每次请求时把 token 从 session 中拿出，与请求中的 token 进行比对，但这种方法的难点在于如何把 token 以参数的形式加入请求。对于 GET 请求，token 将附在请求地址之后，这样 URL 就变成 <http://url?csrftoken=tokenvalue>。而对于 POST 请求来说，要在 form 的最后加上，这样就把 token 以参数的形式加入请求了。但是，在一个网站中，可以接受请求的地方非常多，要对于每一个请求都加上 token 是很麻烦的，并且很容易漏掉，通常使用的方法就是在每次页面加载时，使用 javascript 遍历整个 dom 树，对于 dom 中所有的 a 和 form 标签后加入 token。这样可以解决大部分的请求，但是对于在页面加载之后动态生成的 html 代码，这种方法就没有作用，还需要程序员在编码时手动添加 token。

该方法还有一个缺点是难以保证 token 本身的安全。特别是在一些论坛之类支持用户自己发表内容的网站，黑客可以在上面发布自己个人网站的地址。由于系统也会在这个地址后面加上 token，黑客可以在自己的网站上得到这个 token，并马上就可以发动 CSRF 攻击。为了避免这一点，系统可以在添加 token 的时候增加一个判断，如果这个链接是链到自己本站的，就在后面添加 token，如果是通向外网则不加。不过，即使这个 csrftoken 不以参数的形式附加在请求之中，黑客的网站也同样可以通过 Referer 来得到这个 token 值以发动 CSRF 攻击。这也是一些用户喜欢手动关闭浏览器 Referer 功能的原因。

xss攻击

XSS 攻击既为 SQL注入脚本攻击，我们可以针对这种攻击添加filter 进行拦截，

oauth2协议

OAuth 2.0 规定了四种获得令牌的流程。你可以选择最适合自己的那一种，向第三方应用颁发令牌

下面就是这四种授权方式。

- 授权码 (authorization-code)
- 隐藏式 (implicit)
- 密码式 (password) :
- 客户端凭证 (client credentials)

第一种授权方式：授权码

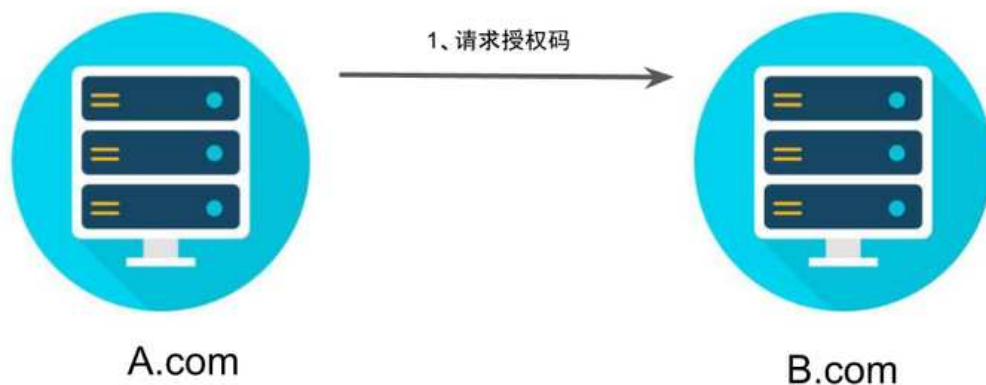
授权码 (authorization code) 方式，指的是第三方应用先申请一个授权码，然后再用该码获取令牌。

这种方式是最常用的流程，安全性也最高，它适用于那些有后端的 Web 应用。授权码通过前端传送，令牌则是储存在后端，而且所有与资源服务器的通信都在后端完成。这样的前后端分离，可以避免令牌泄漏。

第一步，A 网站提供一个链接，用户点击后就会跳转到 B 网站，授权用户数据给 A 网站使用。下面就是 A 网站跳转 B 网站的一个示意链接。

```
https://b.com/oauth/authorize?  
response_type=code&  
client_id=CLIENT_ID&  
redirect_uri=CALLBACK_URL&  
scope=read
```

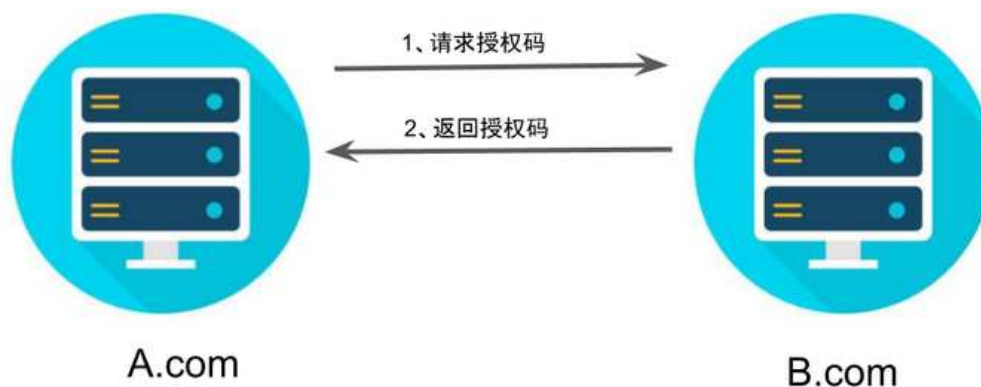
上面 URL 中，`response_type` 参数表示要求返回授权码 (`code`)，`client_id` 参数让 B 知道是谁在请求，`redirect_uri` 参数是 B 接受或拒绝请求后的跳转网址，`scope` 参数表示要求的授权范围（这里是只读）。



第二步，用户跳转后，B 网站会要求用户登录，然后询问是否同意给予 A 网站授权。用户表示同意，这时 B 网站就会跳回 `redirect_uri` 参数指定的网址。跳转时，会传回一个授权码，就像下面这样。

```
https://a.com/callback?code=AUTHORIZATION_CODE
```

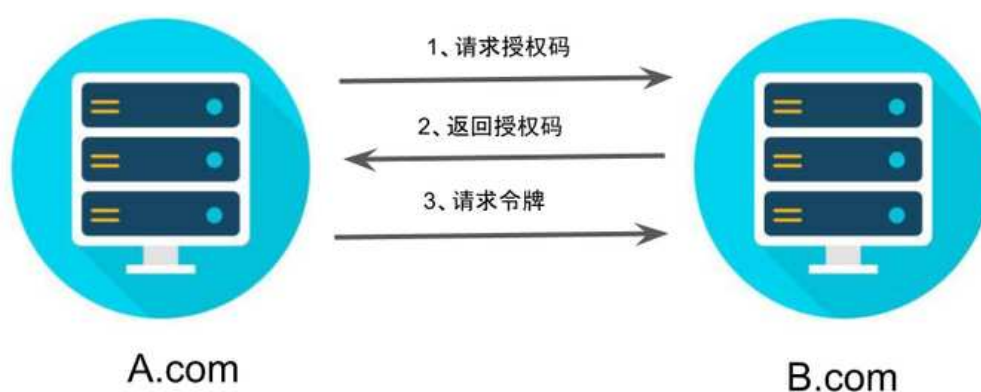
上面 URL 中，`code` 参数就是授权码。



第三步，A 网站拿到授权码以后，就可以在后端，向 B 网站请求令牌。

```
https://b.com/oauth/token?
client_id=CLIENT_ID&
client_secret=CLIENT_SECRET&
grant_type=authorization_code&
code=AUTHORIZATION_CODE&
redirect_uri=CALLBACK_URL
```

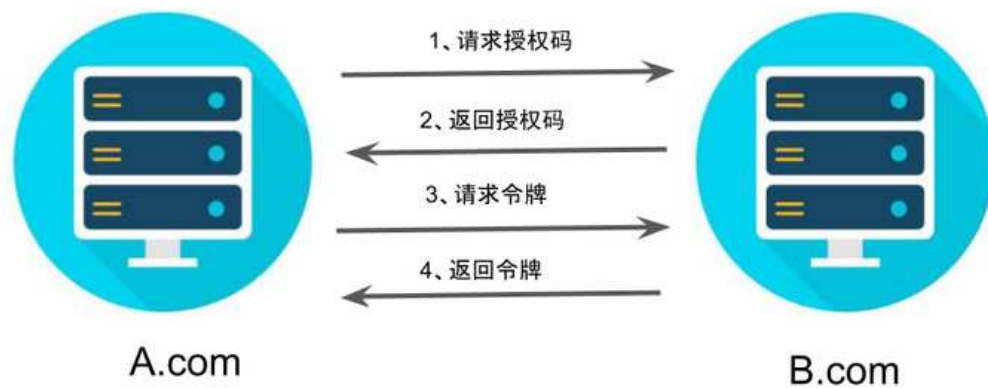
上面 URL 中，`client_id` 参数和 `client_secret` 参数用来让 B 确认 A 的身份（`client_secret` 参数是保密的，因此只能在后端发请求），`grant_type` 参数的值是 `AUTHORIZATION_CODE`，表示采用的授权方式是授权码，`code` 参数是上一步拿到的授权码，`redirect_uri` 参数是令牌颁发后的回调网址。



第四步，B 网站收到请求以后，就会颁发令牌。具体做法是向 `redirect_uri` 指定的网址，发送一段 JSON 数据。

```
{
  "access_token": "ACCESS_TOKEN",
  "token_type": "bearer",
  "expires_in": 2592000,
  "refresh_token": "REFRESH_TOKEN",
  "scope": "read",
  "uid": 100101,
  "info": {...}
}
```

上面 JSON 数据中，`access_token` 字段就是令牌，A 网站在后端拿到了。



第二种方式：隐藏式

有些 Web 应用是纯前端应用，没有后端。这时就不能用上面的方式了，必须将令牌储存在前端。RFC 6749 就规定了第二种方式，允许直接向前端颁发令牌。这种方式没有授权码这个中间步骤，所以称为（授权码）“隐藏式”（implicit）。

第一步，A 网站提供一个链接，要求用户跳转到 B 网站，授权用户数据给 A 网站使用。

```
https://b.com/oauth/authorize?
response_type=token&
client_id=CLIENT_ID&
redirect_uri=CALLBACK_URL&
scope=read
```

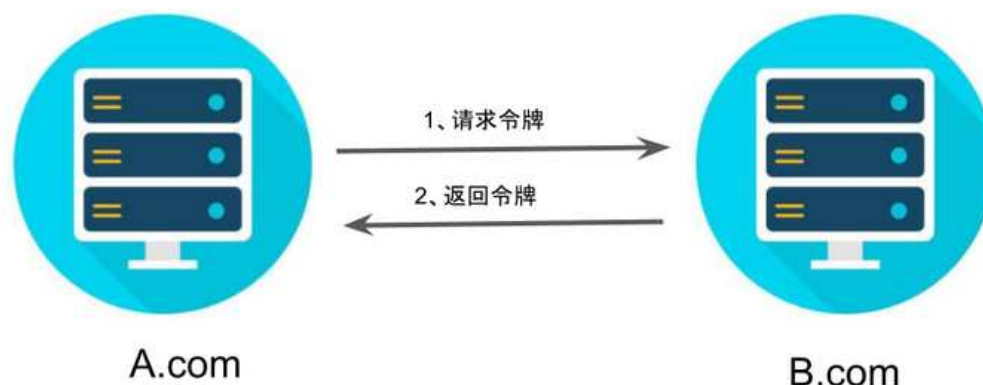
上面 URL 中，`response_type` 参数为 `token`，表示要求直接返回令牌。

第二步，用户跳转到 B 网站，登录后同意给予 A 网站授权。这时，B 网站就会跳回 `redirect_uri` 参数指定的跳转网址，并且把令牌作为 URL 参数，传给 A 网站。

```
https://a.com/callback#token=ACCESS_TOKEN
```

上面 URL 中，`token` 参数就是令牌，A 网站因此直接在前端拿到令牌。

注意，令牌的位置是 URL 锚点（fragment），而不是查询字符串（querystring），这是因为 OAuth 2.0 允许跳转网址是 HTTP 协议，因此存在“中间人攻击”的风险，而浏览器跳转时，锚点不会发到服务器，就减少了泄漏令牌的风险。



这种方式把令牌直接传给前端，是很不安全的。因此，只能用于一些安全要求不高的场景，并且令牌的有效期必须非常短，通常就是会话期间（session）有效，浏览器关掉，令牌就失效了。

第三种方式：密码式

如果你高度信任某个应用，RFC 6749 也允许用户把用户名和密码，直接告诉该应用。该应用就使用你的密码，申请令牌，这种方式称为“密码式”（password）。

第一步，A 网站要求用户提供 B 网站的用户名和密码。拿到以后，A 就直接向 B 请求令牌。

```
https://oauth.b.com/token?
grant_type=password&
username=USERNAME&
password=PASSWORD&
client_id=CLIENT_ID
```

上面 URL 中，`grant_type` 参数是授权方式，这里的 `password` 表示“密码式”，`username` 和 `password` 是 B 的用户名和密码。

第二步，B 网站验证身份通过后，直接给出令牌。注意，这时不需要跳转，而是把令牌放在 JSON 数据里面，作为 HTTP 回应，A 因此拿到令牌。

这种方式需要用户给出自己的用户名/密码，显然风险很大，因此只适用于其他授权方式都无法采用的情况，而且必须是用户高度信任的应用。

第四种方式：凭证式

最后一种方式是凭证式 (client credentials) , 适用于没有前端的命令行应用, 即在命令行下请求令牌。

第一步, A 应用在命令行向 B 发出请求。

```
https://oauth.b.com/token?
grant_type=client_credentials&
client_id=CLIENT_ID&
client_secret=CLIENT_SECRET
```

上面 URL 中, `grant_type` 参数等于 `client_credentials` 表示采用凭证式, `client_id` 和 `client_secret` 用来让 B 确认 A 的身份。

第二步, B 网站验证通过以后, 直接返回令牌。

这种方式给出的令牌, 是针对第三方应用的, 而不是针对用户的, 即有可能多个用户共享同一个令牌。

令牌的使用

A 网站拿到令牌以后, 就可以向 B 网站的 API 请求数据了。

此时, 每个发到 API 的请求, 都必须带有令牌。具体做法是在请求的头信息, 加上一个 `Authorization` 字段, 令牌就放在这个字段里面。

```
curl -H "Authorization: Bearer ACCESS_TOKEN" \
"https://api.b.com"
```

上面命令中, `ACCESS_TOKEN` 就是拿到的令牌。

更新令牌

令牌的有效期到了, 如果让用户重新走一遍上面的流程, 再申请一个新的令牌, 很可能体验不好, 而且也没有必要。OAuth 2.0 允许用户自动更新令牌。

具体方法是, B 网站颁发令牌的时候, 一次性颁发两个令牌, 一个用于获取数据, 另一个用于获取新的令牌 (refresh token 字段)。令牌到期前, 用户使用 refresh token 发一个请求, 去更新令牌。

```
https://b.com/oauth/token?
grant_type=refresh_token&
client_id=CLIENT_ID&
client_secret=CLIENT_SECRET&
refresh_token=REFRESH_TOKEN
```

上面 URL 中, `grant_type` 参数为 `refresh_token` 表示要求更新令牌, `client_id` 参数和 `client_secret` 参数用于确认身份, `refresh_token` 参数就是用于更新令牌的令牌。

B 网站验证通过以后, 就会颁发新的令牌。

Jwt 既json web token 主要是为了在不支持cookie的场景下使用，而且jwt 中也可以携带一部分自定义信息，主要用于替代session

基于session认证所显露的问题

Session: 每个用户经过我们的应用认证之后，我们的应用都要在服务端做一次记录，以方便用户下次请求的鉴别，通常而言session都是保存在内存中，而随着认证用户的增多，服务端的开销会明显增大。

扩展性: 用户认证之后，服务端做认证记录，如果认证的记录被保存在内存中的话，这意味着用户下次请求还必须要请求在这台服务器上,这样才能拿到授权的资源，这样在分布式的应用上，相应的限制了负载均衡器的能力。这也意味着限制了应用的扩展能力。

CSRF: 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。

基于token的鉴权机制

基于token的鉴权机制类似于http协议也是无状态的，它不需要在服务端去保留用户的认证信息或者会话信息。这就意味着基于token认证机制的应用不需要去考虑用户在哪一台服务器登录了，这就为应用的扩展提供了便利。

流程上是这样的：

- 用户使用用户名密码来请求服务器
- 服务器进行验证用户的信息
- 服务器通过验证发送给用户一个token
- 客户端存储token，并在每次请求时附上这个token值
- 服务端验证token值，并返回数据

这个token必须要在每次请求时传递给服务端，它应该保存在请求头里，另外，服务端要支持 CORS(跨来源资源共享) 策略，一般我们在服务端这么做就可以了 `Access-Control-Allow-Origin: *`。

JWT的构成

第一部分我们称它为头部 (header),第二部分我们称其为载荷 (payload, 类似于飞机上承载的物品)，第三部分是签证 (signature).

header

jwt的头部承载两部分信息：

- 声明类型，这里是jwt
- 声明加密的算法 通常直接使用 HMAC SHA256

完整的头部就像下面这样的JSON：

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

然后将头部进行base64加密 (该加密是可以对称解密的),构成了第一部分.

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9
```

payload

载荷就是存放有效信息的地方。这个名字像是特指飞机上承载的货品, 这些有效信息包含三个部分

- 标准中注册的声明
- 公共的声明
- 私有的声明

标准中注册的声明 (建议但不强制使用) :

- **iss**: jwt签发者
- **sub**: jwt所面向的用户
- **aud**: 接收jwt的一方
- **exp**: jwt的过期时间, 这个过期时间必须要大于签发时间
- **nbf**: 定义在什么时间之前, 该jwt都是不可用的.
- **iat**: jwt的签发时间
- **jti**: jwt的唯一身份标识, 主要用来作为一次性token,从而回避重放攻击。

公共的声明 :

公共的声明可以添加任何的信息, 一般添加用户的相关信息或其他业务需要的必要信息.但不建议添加敏感信息, 因为该部分在客户端可解密.

私有的声明 :

私有声明是提供者和消费者所共同定义的声明, 一般不建议存放敏感信息, 因为base64是对称解密的, 意味着该部分信息可以归类为明文信息。

定义一个payload:

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

然后将其进行base64加密, 得到Jwt的第二部分。

```
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjYwRtaW4iOnRydWV9
```

signature

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。

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

var signature = HMACSHA256(encodedString, 'secret'); //
TJVA95OrM7E2cBab30RMHrHDCEfXjoYZgeFONFh7HgQ
```

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.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0Ij0iNjY2OTY0MDE5IiwiaXNja30RMHrHDCEfXjoYZgeFONFh7HgQ
```

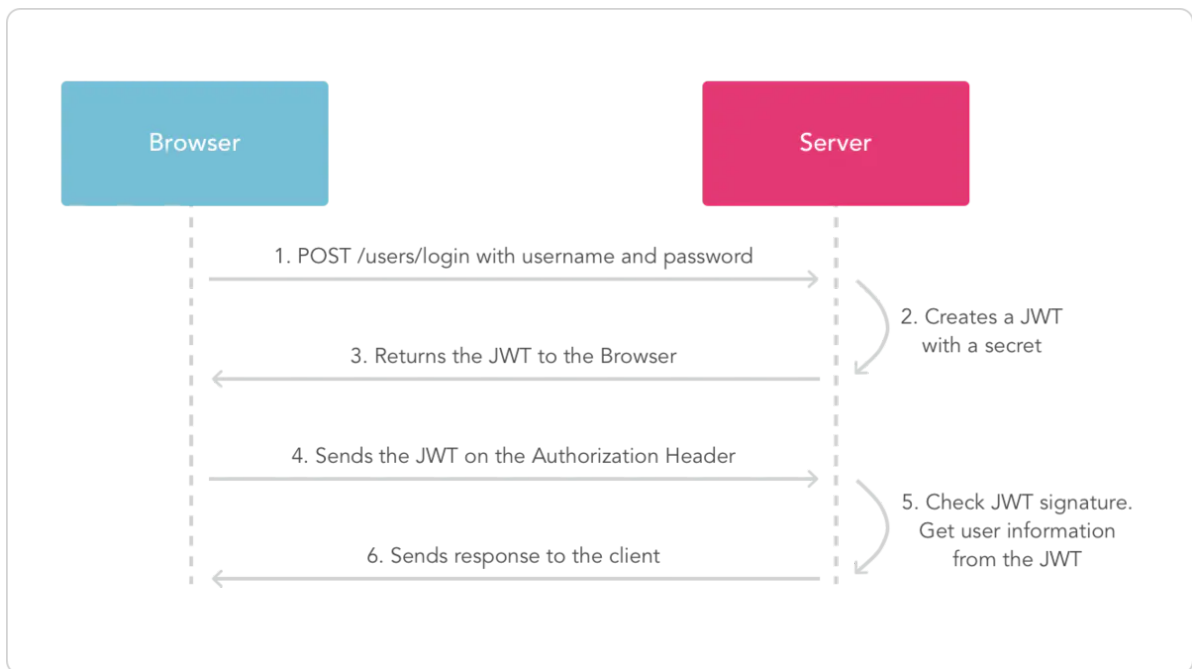
注意：secret是保存在服务器端的，jwt的签发生成也是在服务器端的，secret就是用来进行jwt的签发和jwt的验证，所以，它就是你服务端的私钥，在任何场景都不应该流露出去。一旦客户端得知这个secret, 那就意味着客户端是可以自我签发jwt了。

如何应用

一般是在请求头里加入`Authorization`，并加上`Bearer` 标注：

```
fetch('api/user/1', {
  headers: {
    'Authorization': 'Bearer ' + token
  }
})
```

服务端会验证token，如果验证通过就会返回相应的资源。整个流程就是这样的：



jwt-diagram

总结

优点

- 因为json的通用性，所以JWT是可以进行跨语言支持的，像JAVA,JavaScript,NodeJS,PHP等很多语言都可以使用。
- 因为有了payload部分，所以JWT可以在自身存储一些其他业务逻辑所必要的非敏感信息。
- 便于传输，jwt的构成非常简单，字节占用很小，所以它是非常便于传输的。
- 它不需要在服务端保存会话信息, 所以它易于应用的扩展

安全相关

- 不应该在jwt的payload部分存放敏感信息，因为该部分是客户端可解密的部分。
- 保护好secret私钥，该私钥非常重要。
- 如果可以，请使用https协议

tcp 为什么需要三次握手而不是两次