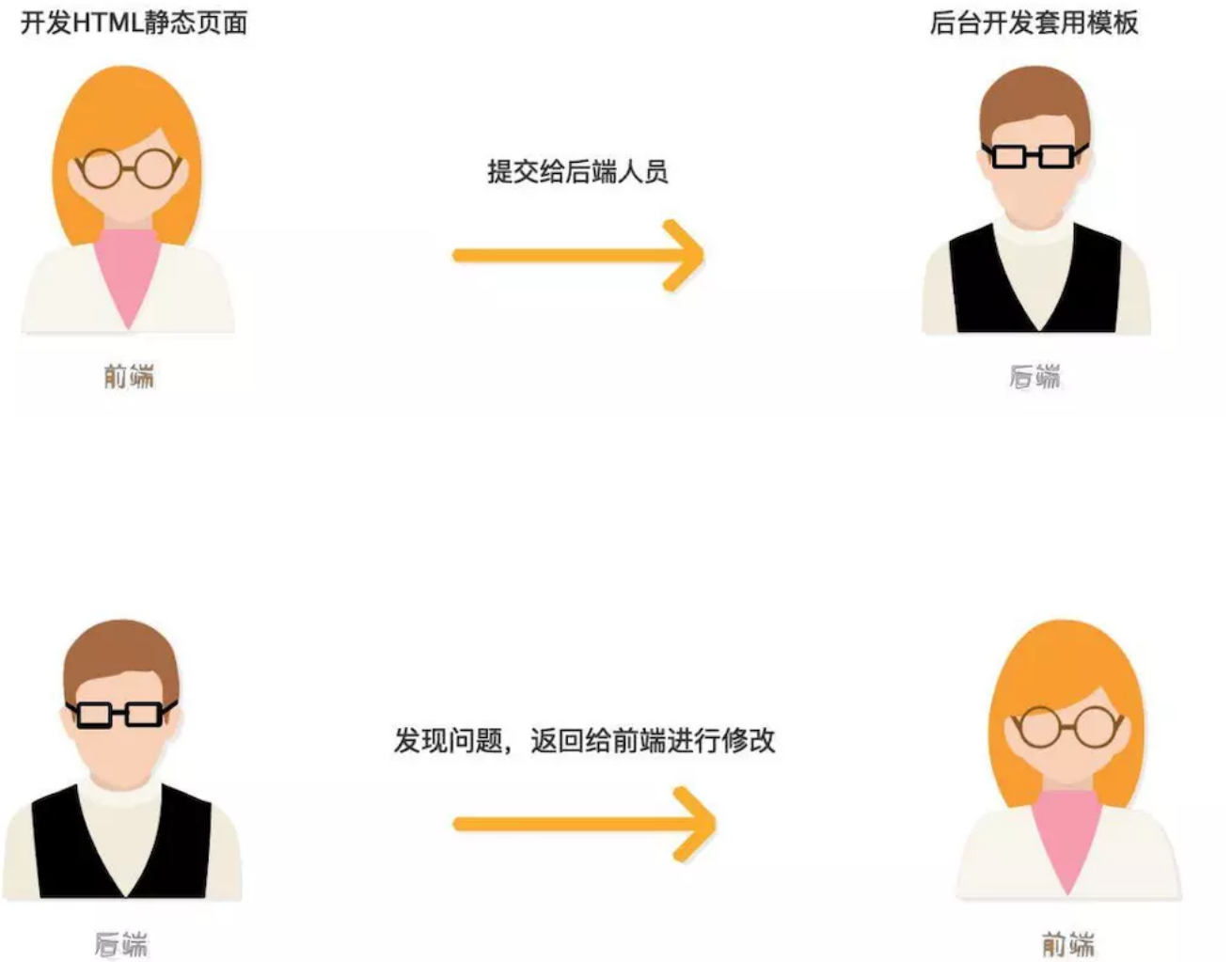
从目前应用软件开发的发展趋势来看

- - 越来越注重用户体验，随着互联网的发展，开始多终端化。
- - 大型应用架构模式正在向云化、微服务化发展。

传统的开发模式

前后端分离前我们的开发协作模式一般是这样的：



前端写好静态的HTML页面交付给后端开发。静态页面可以本地开发，也无需考虑业务逻辑只需要实现View即可。

后端使用模板引擎去套模板，当年使用最广泛的就是jsp，freemarker等等，同时内嵌一些后端提供的模板变量和一些逻辑操作。

然后前后端集成对接，遇到问题，前台返工，后台返工。

然后在集成，直至集成成功。

这种模式的问题：

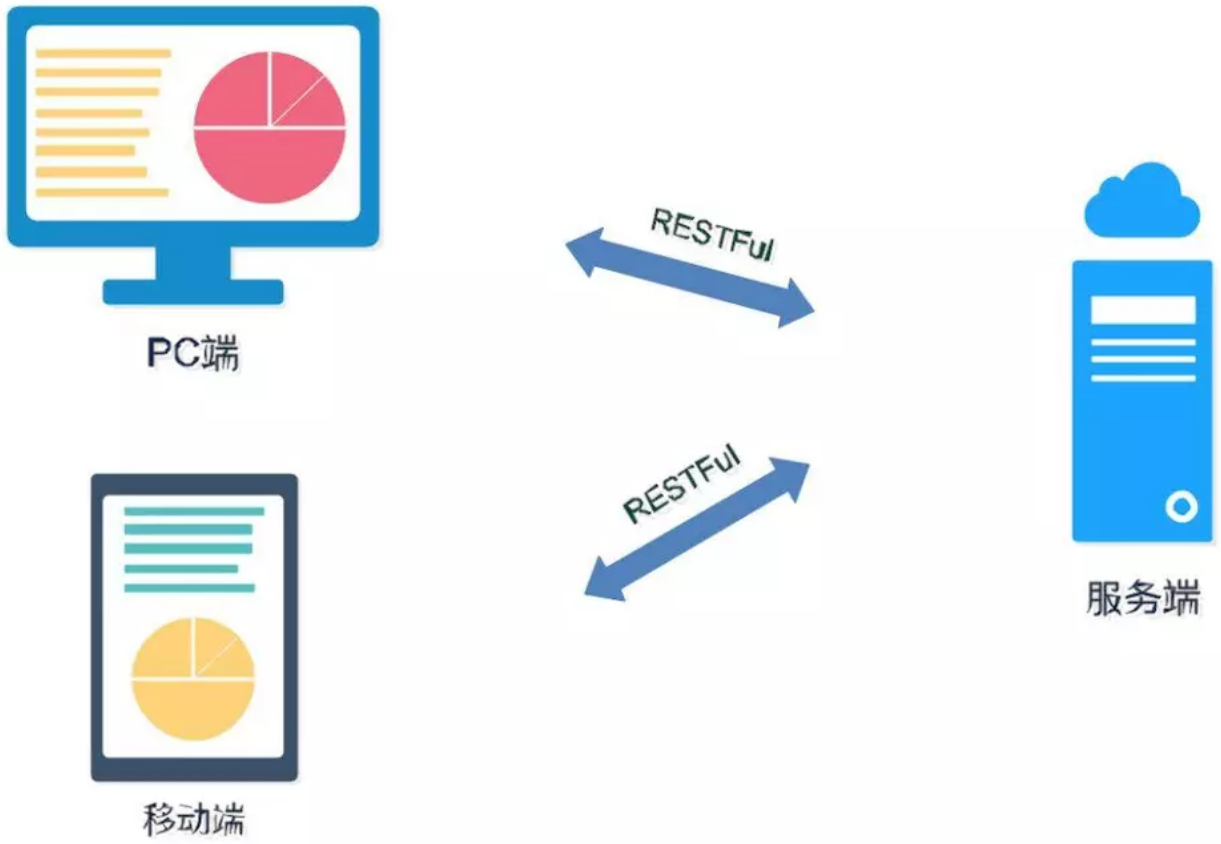
在前端调试的时候要安装完整的一套后端开发工具，要把后端程序完全启动起来。遇到问题需要后端开发来帮忙调试。我们发现前后端严重耦合，还要要求后端人员会一些HTML，JS等前端语言。前端页面里还嵌入了很多后端的代码。一旦后端换了一种语言开发，简直就要重做。

像这种增加了大量的沟通成本，调试成本等，而且前后端的开发进度相互影响，从而大大降低了开发效率。

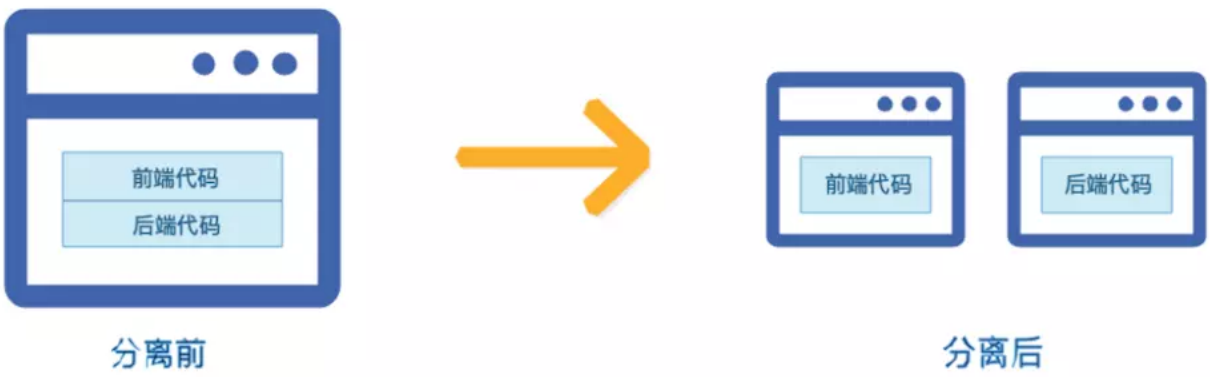
前后端分离

前后端分离并不只是开发模式，而是web应用的一种架构模式。在开发阶段，前后端工程师约定好数据交互接口，实现并行开发和测试；在运行阶段前后端分离模式需要对web应用进行分离部署，前后端之前使用HTTP或者其他协议进行交互请求。

1. 客户端和服务端采用RESTFul API的交互方式进行交互



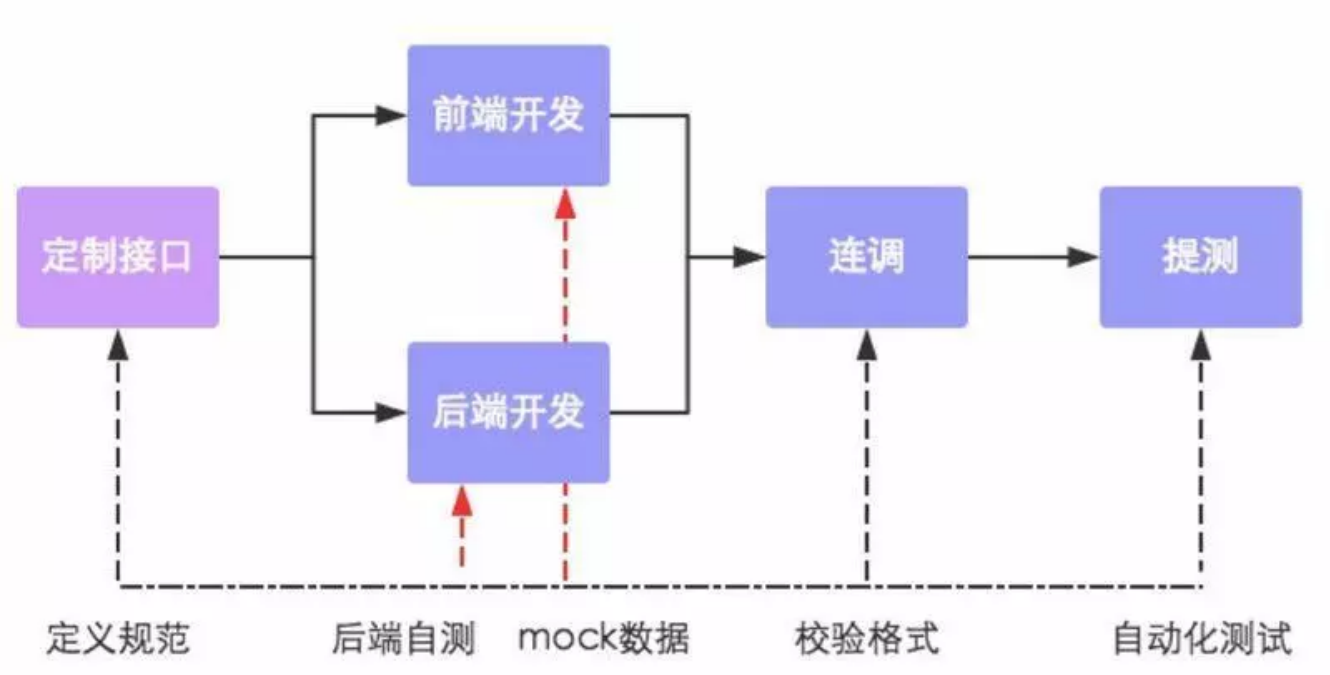
2. 前后端代码库分离



在传统架构模式中，前后端代码存放于同一个代码库中，甚至是同一工程目录下。页面中还夹杂着后端代码。前后端工程师进行开发时，都必须把整个项目导入到开发工具中。

前后端代码库分离，前端代码中有可以进行Mock测试(通过构造虚拟测试对 象以简化测试环境的方法)的伪后端，能支持前端的独立开发和测试。而后端代码中除了功能实现外，还有着详细的测试用例，以保证API的可用性，降低集成风险。

3. 并行开发



在开发期间前后端共同商定好数据接口的交互形式和数据格式。然后实现前后端的并行开发，其中前端工程师在开发完成之后可以独自进行mock测试，而后端也可以使用Postman等接口测试软件进行接口自测，然后前后端一起进行功能联调并校验格式，最终进行自动化测试。

分离后，开发模式是这样的：

1. 接收数据，返回数据
2. 处理渲染逻辑
3. Client-side MV* 架构
4. 代码跑在浏览器上
5. 独立开发

前端

模拟测试

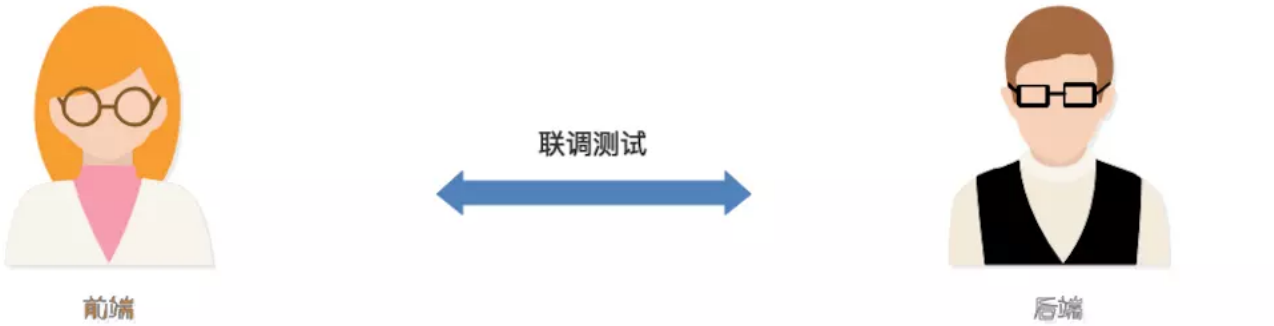
Mock数据

1. 提供数据
2. 处理业务逻辑
3. Server-side MVC架构
4. 代码跑在服务器上
5. 独立开发

Postman

模拟测试

后端



4. 前后端分离架构后的优点：

为优质产品打造精益团队

通过将开发团队前后端分离化，让前后端工程师只需要专注于前端或后端的开发工作，是的前后端工程师实现自治，培养其独特的技术特性，然后构建出一个全栈式的精益开发团队。

提升开发效率

前后端分离以后，可以实现前后端代码的解耦，只要前后端沟通约定好应用所需接口以及接口参数，便可以开始并行开发，无需等待对方的开发工作结束。与此同时，即使需求发生变更，只要接口与数据格式不变，后端开发人员就不需要修改代码，只要前端进行变动即可。如此一来整个应用的开发效率必然会有质的提升。

完美应对复杂多变的前端需求

如果开发团队能完成前后端分离的转型，打造优秀的前后端团队，开发独立化，让开发人员做到专注专精，开发能力必然会有所提升，能够完美应对各种复杂多变的前端需求。

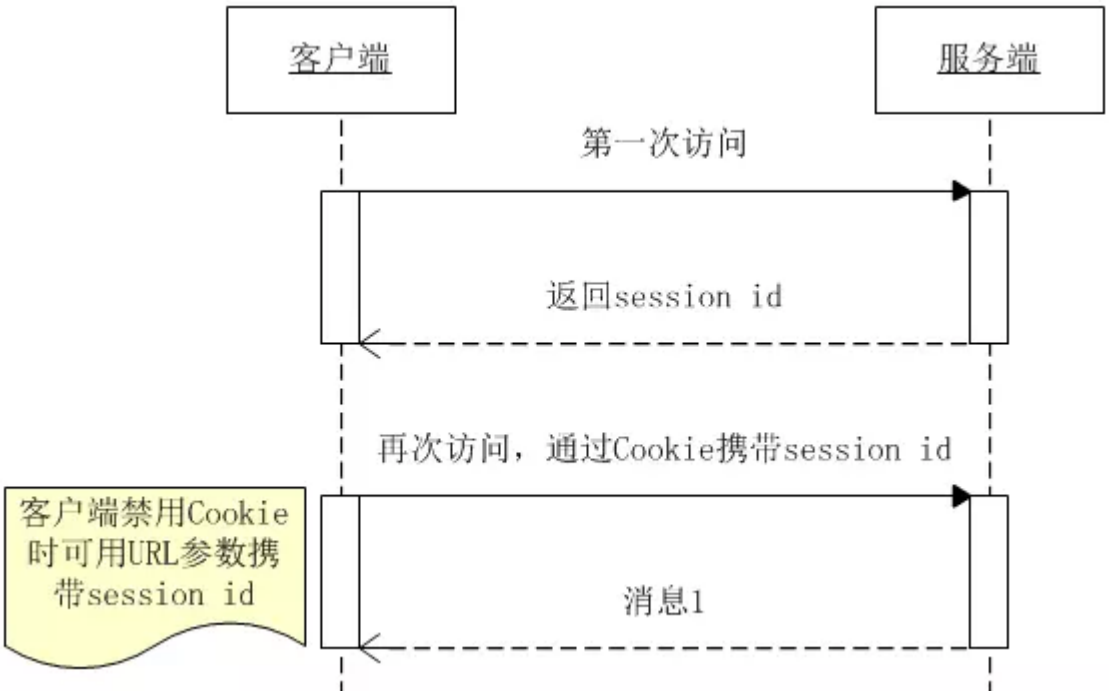
增强代码可维护性

前后端分离后，应用的代码不再是前后端混合，只有在运行期才会有调用依赖关系。应用代码将会变得整洁清晰，不论是代码阅读还是代码维护都会比以前轻松。

使用了前后端分离架构后，除了开发模式的变更，我们在开发的过程中还会遇到哪些问题呢？我们接着往下看。

用户认证

我们先来看看传统开发，我们是如何进行用户认证的：



前端登录，后端根据用户信息生成一个jsessionid，并保存这个 jsessionid 和对应的用户id到Session中，接着把 jsessionid 传给用户，存入浏览器 cookie，之后浏览器请求带上这个cookie，后端根据这个cookie值来查询用户，验证是否过期。

HTTP有一个特性：无状态的，就是前后两个HTTP事务它们并不知道对方的信息。而为了维护会话信息或用户信息，一般可用Cookie和Session技术缓存信息。

- Cookie是存储在客户端的

- Session是存储在服务端的

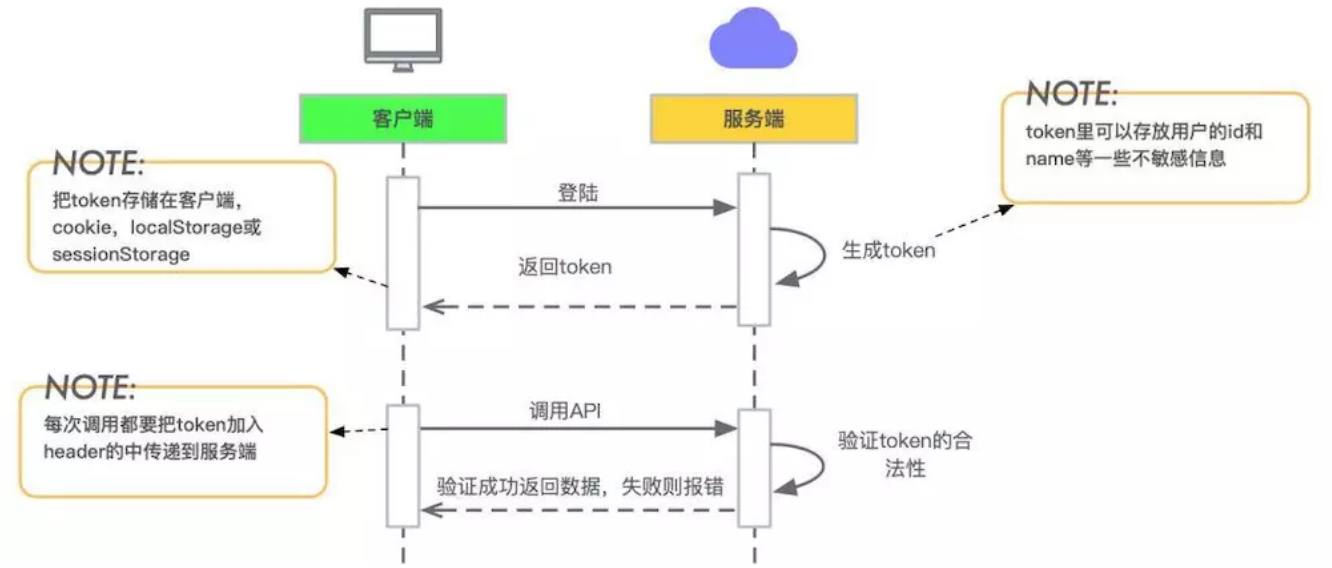
但这样做问题就很多，如果我们的页面出现了 XSS 漏洞，由于 cookie 可以被 JavaScript 读取，XSS 漏洞会导致用户 token 泄露，而作为后端识别用户的标识，cookie 的泄露意味着用户信息不再安全。尽管我们通过转义输出内容，使用 CDN 等可以尽量避免 XSS 注入，但谁也不能保证在大型的项目中不会出现这个问题。

在设置 cookie 的时候，其实你还可以设置 httpOnly 以及 secure 项。设置 httpOnly 后 cookie 将不能被 JS 读取，浏览器会自动的把它加在请求的 header 当中，设置 secure 的话，cookie 就只允许通过 HTTPS 传输。secure 选项可以过滤掉一些使用 HTTP 协议的 XSS 注入，但并不能完全阻止。

httpOnly 选项使得 JS 不能读取到 cookie，那么 XSS 注入的问题也基本不用担心了。但设置 httpOnly 就带来了另一个问题，就是很容易的被 XSRF，即跨站请求伪造。当你浏览器开着这个页面的时候，另一个页面可以很容易的跨站请求这个页面的内容。因为 cookie 默认被发了出去。

JWT解决方案

JWT（Json Web Token）



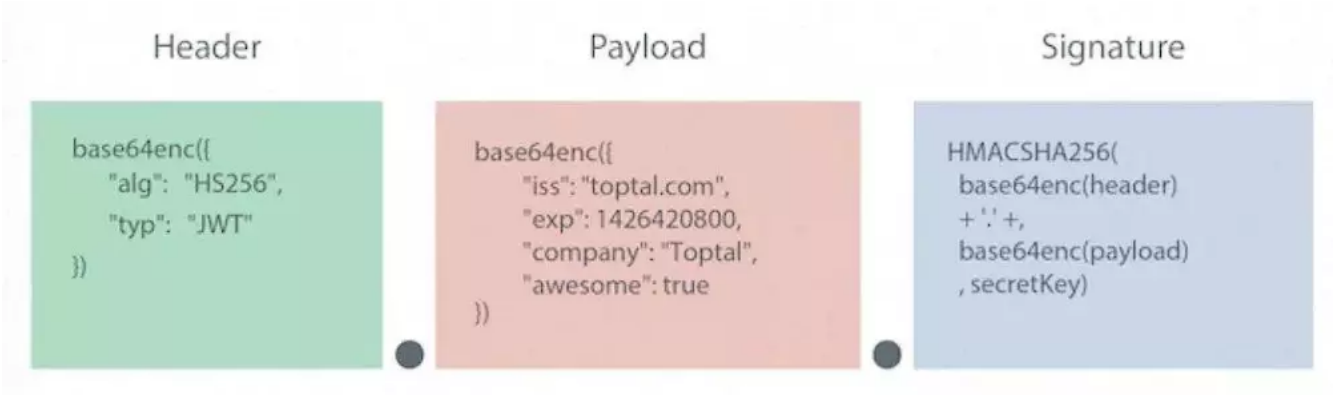
JWT 是一个开放标准(RFC 7519)，它定义了一种用于简洁，自包含的用于通信双方之间以 JSON 对象的形式安全传递信息的方法。该信息可以被验证和信任，因为它是数字签名的。JWTS可以使用秘密（使用HMAC算法）或公钥/私钥对使用RSA或ECDSA来签名。

- 简洁(Compact)：可以通过URL, POST 参数或者在 HTTP header 发送，因为数据量小，传输速度快。

- 自包含(Self-contained)：负载中包含了所有用户所需要的信息，避免了多次查询数据库。

JWT 组成

JWT由3个子字符串组成，分别为Header，Payload以及Signature，结合JWT的格式即：Header.Payload.Signature。（Claim是描述Json的信息的一个Json，将Claim转码之后生成Payload）。



Header

Header是由下面这个格式的Json通过Base64编码（编码不是加密，是可以通过反编码的方式获取到这个原来的Json，所以JWT中存放的一般是不敏感的信息）生成的字符串，Header中存放的内容是说明编码对象是一个JWT以及使用“SHA-256”的算法进行加密（加密用于生成Signature）

{

"typ":"JWT",

"alg":"HS256"

}

- 头部包含了两部分，token 类型和采用的加密算法

- Base64是一种编码，也就是说，它是可以被翻译回原来的样子来的。它并不是一种加密过程。

JWS	算法名称	描述
HS256	HMAC256	HMAC with SHA-256
HS384	HMAC384	HMAC with SHA-384
HS512	HMAC512	HMAC with SHA-512
RS256	RSA256	RSASSA-PKCS1-v1_5 with SHA-256
RS384	RSA384	RSASSA-PKCS1-v1_5 with SHA-384
RS512	RSA512	RSASSA-PKCS1-v1_5 with SHA-512
ES256	ECDSA256	ECDSA with curve P-256 and SHA-256
ES384	ECDSA384	ECDSA with curve P-384 and SHA-384
ES512	ECDSA512	ECDSA with curve P-521 and SHA-512

Payload

Payload是通过Claim进行Base64转码之后生成的一串字符串，Claim是一个Json，Claim中存放的内容是JWT自身的标准属性，所有的标准属性都是可选的，可以自行添加，比如：JWT的签发者、JWT的接收者、JWT的持续时间等；同时Claim中也可以存放一些自定义的属性，这个自定义的属性就是在用户认证中用于标明用户身份的一个属性，比如用户存放在数据库中的id，为了安全起见，一般不会将用户名及密码这类敏感的信息存放在Claim中。将Claim通过Base64转码之后生成的一串字符串称作**Payload**。

{

"iss":"Issuer —— 用于说明该JWT是由谁签发的",

"sub":"Subject —— 用于说明该JWT面向的对象",

"aud":"Audience —— 用于说明该JWT发送给的用户",

"exp":"Expiration Time —— 数字类型，说明该JWT过期的时间",

"nbf":"Not Before —— 数字类型，说明在该时间之前JWT不能被接受与处理",

"iat":"Issued At —— 数字类型，说明该JWT何时被签发",

"jti":"JWT ID —— 说明标明JWT的唯一ID",

"user-definde1":"自定义属性举例",

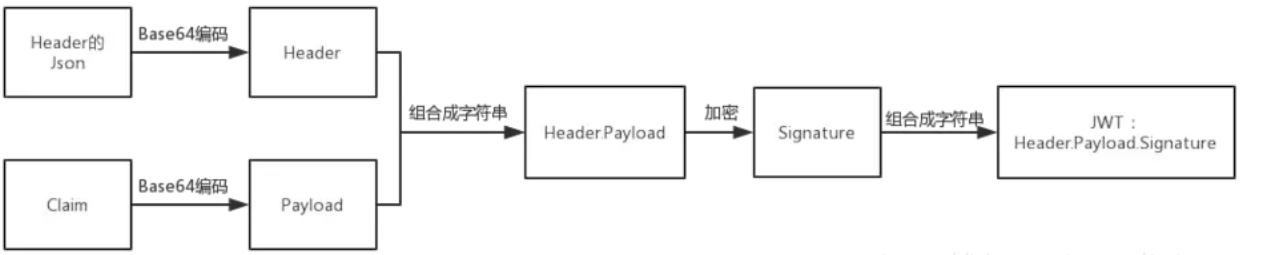
"user-definde2":"自定义属性举例"

}

Signature

Signature是由Header和Payload组合而成，将Header和Claim这两个Json分别使用Base64方式进行编码，生成字符串Header和Payload，然后将Header和Payload以Header.Payload的格式组合在一起形成一个字符串，然后使用上面定义好的加密算法和一个密匙（这个密匙存放在服务器上，用于进行验证）对这个字符串进行加密，形成一个新的字符串，这个字符串就是Signature。

签名的目的：最后一步签名的过程，实际上是对头部以及负载内容进行签名，防止内容被窜改。如果有人对头部以及负载的内容解码之后进行修改，再进行编码，最后加上之前的签名组合形成新的JWT的话，那么服务器端会判断出新的头部和负载形成的签名和JWT附带上的签名是不一样的。如果要对新的头部和负载进行签名，在不知道服务器加密时用的密钥的话，得出来的签名也是不一样的。



三个部分通过.连接在一起就是我们的 JWT 了：

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjU3ZmVmMTY0ZTU0YWY2NGZmYzUzZGJkNSIsInhzcmYiOiI0ZWE1YzUwOGE2NTY2ZTc2MjQwNTQzZjhmZWIwNmZkNDU3Nzc3YmUzOTU0OWM0MDE2NDM2YWZkYTY1ZDIzMzBlIiwiaWF0IjoxNDc2NDI3OTMzfQ.PA3QjeyZSUh7H0GfE0vJaKW4LjKJuC3dVLQiY4hii8s

JWT认证

服务器在生成一个JWT之后会将这个token发送到客户端机器，在客户端再次访问受到JWT保护的资源URL链接的时候，服务器会获取到这个token信息，首先将Header进行反编码获取到加密的算法，在通过存放在服务器上的密匙对Header.Payload 这个字符串进行加密，比对token中的Signature和实际加密出来的结果是否一致，如果一致那么说明该token是合法有效的，认证成功，否则认证失败。

JWT使用总结

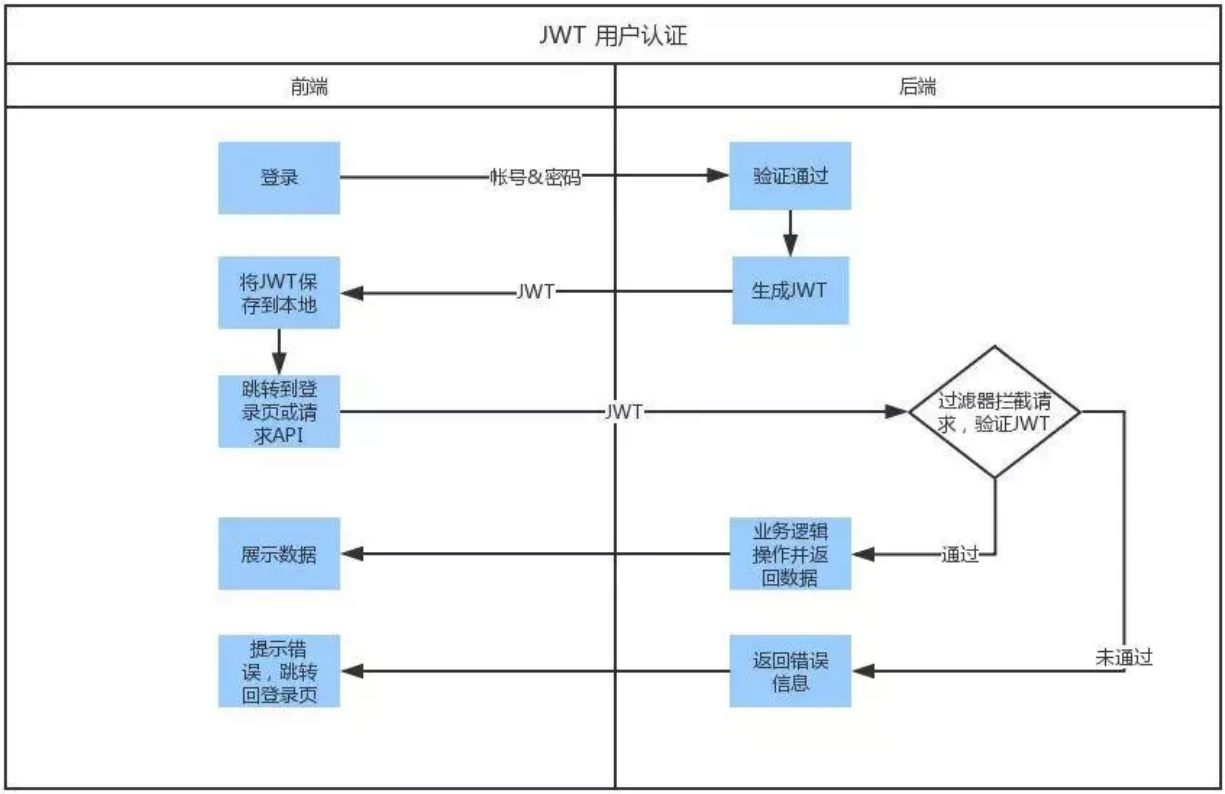
1. 首先，前端通过Web表单将自己的用户名和密码发送到后端的接口。这一过程一般是一个HTTP POST请求。建议的方式是通过SSL加密的传输（https协议），从而避免敏感信息被嗅探。
2. 后端核对用户名和密码成功后，将用户的id等其他信息作为JWT Payload（负载），将其与头部分别进行Base64编码拼接后签名，形成一个JWT。形成的JWT就是一个形同lll.zzz.xxx的字符串。
3. 后端将JWT字符串作为登录成功的返回结果返回给前端。前端可以将返回的结果保存在Cookie或localStorage或sessionStorage上，退出登录时前端删除保存的JWT即可。

特性	Cookie	LocalStorage	sessionStorage
数据的声明周期	一般由服务器生成，可设置失效时间。如果在浏览器生成，默认是关闭浏览器之后失效	除非被清楚，否则永久保存	仅在当前会话有效，关闭页面或浏览器后被清除
存放数据大小	4KB	*一般 5MB*	*一般 5MB*
与服务端通信	*每次都会携带在HTTP头中，如果使用 cookie 保存过多数据会带来性能问题*	仅在客户端中保存，不参与和服务器的通信。~~也可有脚本选择性提交到服务器端?~~	*同 LocalStorage*
用途	一般由服务器端生成，用于标识用户身份	用于浏览器端缓存数据	*同 LocalStorage*
共享			

4. 前端在每次请求时将JWT放入HTTP Header中的Authorization位。(解决XSS和XSRF问题)

5. 后端检查是否存在，如存在验证JWT的有效性。例如，检查签名是否正确；检查Token是否过期；检查Token的接收方是否是自己（可选）。

6. 验证通过后后端使用JWT中包含的用户信息进行其他逻辑操作，返回相应结果。



JWT和Session方式存储id的差异

Session方式存储用户id的最大弊病在于Session是存储在服务器端的，所以需要占用大量服务器内存，对于较大型应用而言可能还要保存许多的状态。一般而言，大型应用还需要借助一些KV数据库和一系列缓存机制来实现Session的存储。

而JWT方式将用户状态分散到了客户端中，可以明显减轻服务端的内存压力。除了用户id之外，还可以存储其他的和用户相关的信息，例如该用户是否是管理员、用户所在的分组等。虽说JWT方式让服务器有一些计算压力（例如加密、编码和解码），但是这些压力相比磁盘存储而言可能就不算什么了。

单点登录

Session方式来存储用户id，一开始用户的Session只会存储在一台服务器上。对于有多个子域名的站点，每个子域名至少会对应一台不同的服务器，例如：

www.taobao.com，nv.taobao.com，nz.taobao.com，login.taobao.com。所以如果要实现在login.taobao.com登录后，在其他的子域名下依然可以取到Session，这要求我们在多台服务器上同步Session。使用JWT的方式则没有这个问题的存在，因为用户的状态已经被传送到了客户端。

跨域问题

当客户端和服务端分开部署到不同服务器的时候，就会遇到跨域访问的问题，是由浏览器同源策略限制的一类请求场景。

URL	说明	是否允许通信
http://www.a.com/a.js http://www.a.com/b.js	同一域名下	允许
http://www.a.com/lab/a.js http://www.a.com/script/b.js	同一域名下不同文件夹	允许
http://www.a.com:8000/a.js http://www.a.com/b.js	同一域名，不同端口	不允许
http://www.a.com/a.js https://www.a.com/b.js	同一域名，不同协议	不允许
http://www.a.com/a.js http://70.32.92.74/b.js	域名和域名对应ip	不允许
http://www.a.com/a.js http://script.a.com/b.js	主域相同，子域不同	不允许
http://www.a.com/a.js http://a.com/b.js	同一域名，不同二级域名（同上）	不允许（cookie这种情况下也不允许访问）
http://www.cnblogs.com/a.js http://www.a.com/b.js	不同域名	不允许

跨域解决方案

推荐使用Nginx反向代理的方案：

反向代理

代理访问其实在实际应用中有很多场景，在跨域中应用的原理做法为：通过反向代理服务器监听同端口，同域名的访问，不同路径映射到不同的地址，比如，在nginx服务器中，监听同一个域名和端口，不同路径转发到客户端和服务器，把不同端口和域名的限制通过反向代理，来解决跨域的问题：

server {

listen 80;

server_name domain.com;

```
#charset koi8-r;

#access_log logs/host.access.log main;

location /client { #访问客户端路径

proxy_pass http://localhost:81;

proxy_redirect default;

}

location /apis { #访问服务器路径

rewrite ^/apis/(.*)/1 break;

proxy_pass http://localhost:82;

}

}
```