1. 专题
   1. 模块化
      1. CommonJS规范及nodeJS的实现

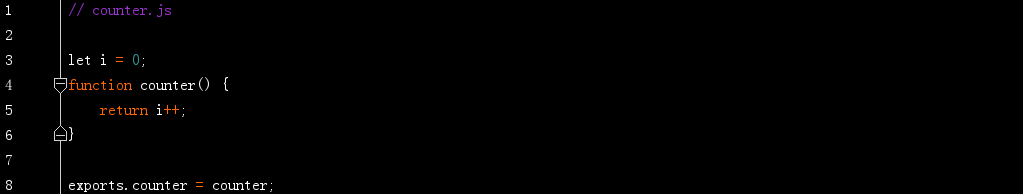
**1.规范和实现原理**

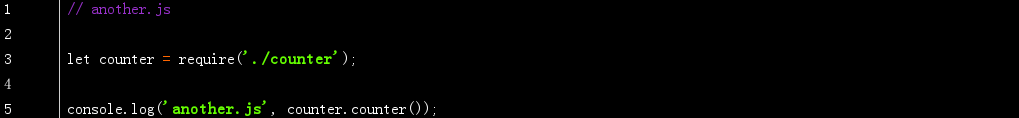
CommonJS是1个模块化规范，规定使用require引入模块，使用module.exports或exports导出模块。

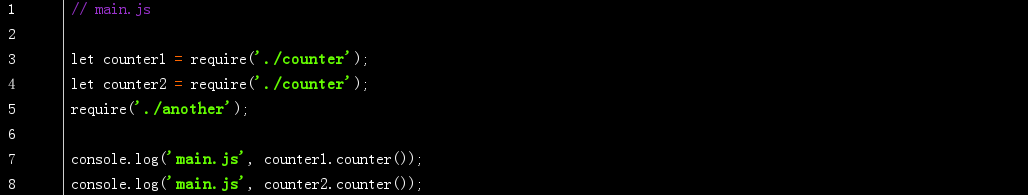
nodeJS的模块化方案就严格遵守CommonJS规范。在nodeJS环境下，使用require方法加载文件时，会读取被加载模块代码，写入1个上下文中。如在下面的counter.js模块文件为：

**2.实际使用注意事项**

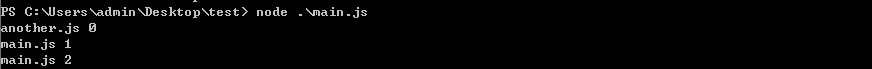
（1）被加载模块仅在第一次被使用时执行一次（貌似使用save函数保存到1个数据结构中），并在执行过程中初始化模块的导出对象。之后，缓存起来的导出对象被重复利用。



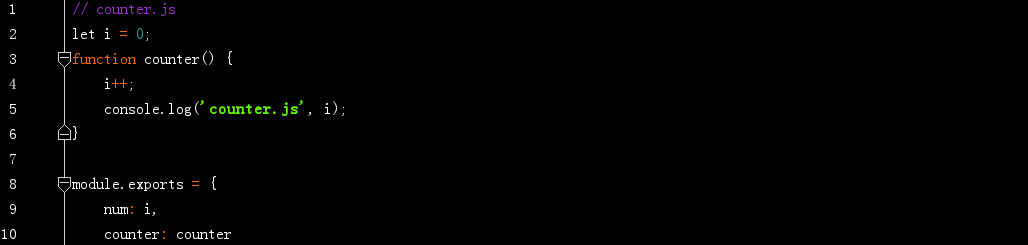


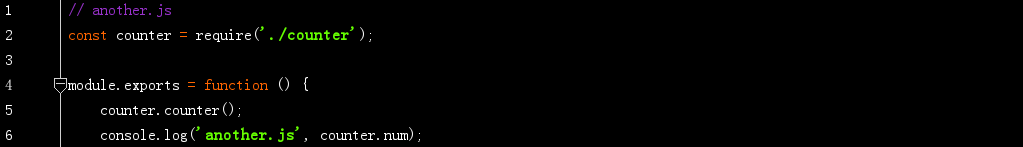


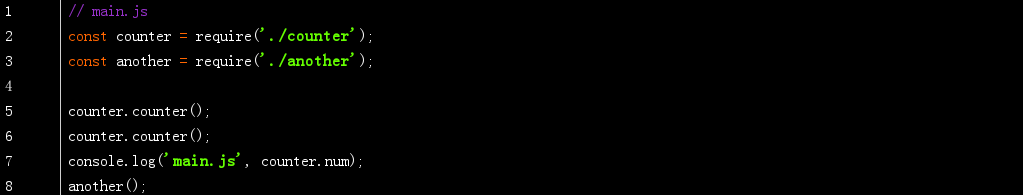
在main.js中引入counter.js两次，引入的another.js中又引入counter.js一次。但counter.js只会在第1次引入时执行并保持为导出对象。



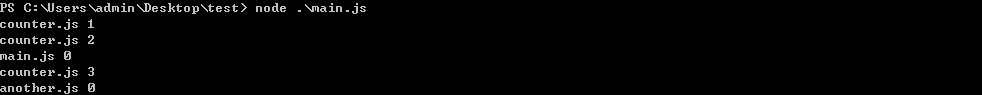
（2）exports导出的是原模块中值的拷贝。这个值与原模块的值不同，且不同引入模块内也不同。







执行main.js，结果如下：



可见，main.js和another.js引入的counter函数是1个，原理见（1）。但couter.js输出的num所用到的i是1个新的拷贝，和counter用到的i不是一回事。

（3）module.exports和exports

因为exports是包装时传入的module.exports，可以对module.exports直接赋值，也可以对exports添加属性，但不能对exports直接赋值，否则将导致exports不再是module的属性。

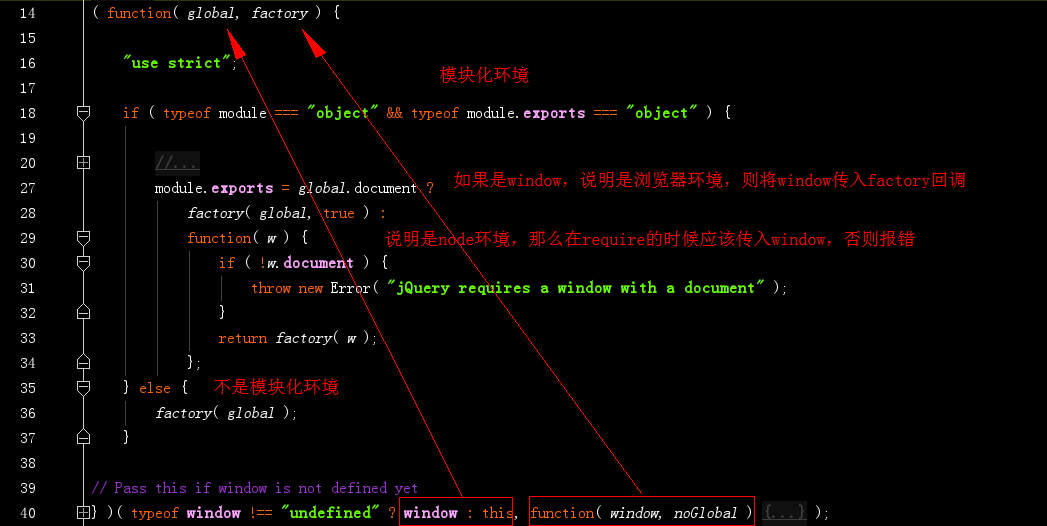


（4）直接输出数组、变量、函数

直接将值赋值给module.exports，就成为1个可以直接使用的数组、变量、函数。如下：



在jQuery源码中，有：

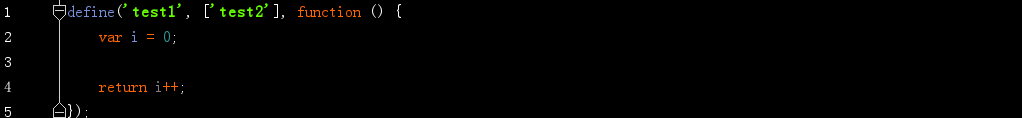


* + 1. 浏览器环境的实现

CommonJS规范中的写法在服务端没有任何问题，因为服务器读取模块都是在本地磁盘，加载速度很快。但是如果在客户端，加载模块的时候有可能出现假死状况。因此出现AMD（AsynchronousModuleDefinition）规范，规定通过异步方式加载模块。require.js就是实现AMD规范的1个库。

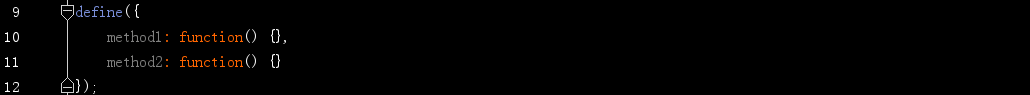
**1.定义模块**

（1）模块应定义在define中。第1个参数为模块名称，字符串格式。第2个参数为模块依赖，为数组。第3个参数为1个回调函数。所有依赖这个模块的语句，都定义在这个回调函数中。等到加载完成之后，这个回调函数才会运行。



模块依赖参数不是模块名称，而是模块路径。要不根本找不到这个模块。

（2）模块名称可以省略，则为匿名模块。如果没有任何依赖，则为独立模块，可以为如上1个匿名函数，也可以为：



（3）require、define和模块返回值

define定义模块，require加载使用模块。两者区别为：用define定义模块时，如果模块有返回值，会将返回值传入依赖模块。而require定义的模块不能被其他模块调用，否则报错。

如下，因jQuery没有返回值，test1返回0，依赖模块参数为undefined和0。

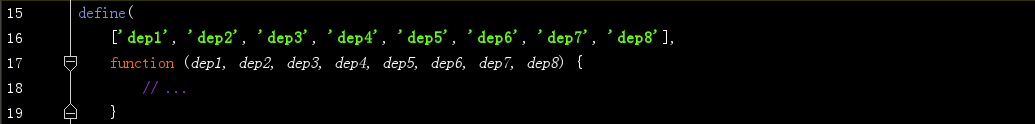




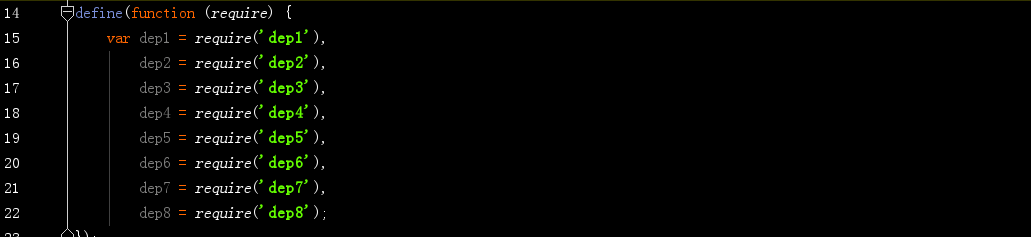
另外，虽然jQuery模块没有返回值，这里在依赖模块中也可以使用jQuery，原因是jQuery直接把自己作为window的属性了。

所以，define能代替require，但估计性能原因不这样做。

（4）如果依赖的模块很多，参数与模块一一对应的写法非常麻烦，可以使用require参数简化。



可以简化为：



这里的require参数必须写，否则不会传任何值给依赖模块。

**2.require.js文件的加载**

（1）引入

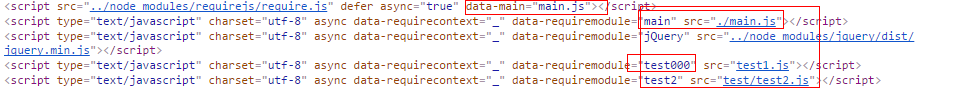
可以直接使用npm下载（requirejs），或官网直接下载。在html页面中如下引入。



main.js是主模块，里面引入的代码会被自动加载。



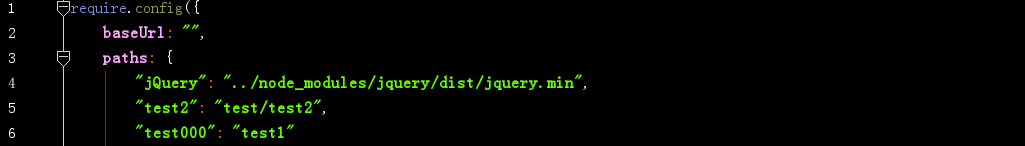
这里最好使用require函数。



这里引入的模块不仅仅是主模块依赖的，还包括主模块依赖模块的进一步依赖，都会自动引入到网页中。

（2）require.config()

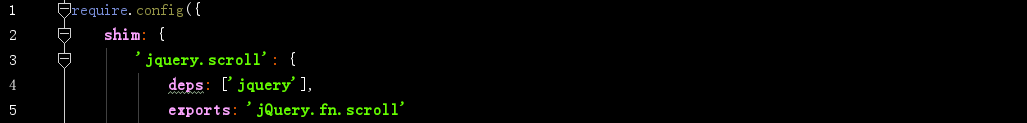
参数为1个对象。可以通过paths属性统一定义模块的路径，还可以通过baseUrl定义参考目录。



说明：这里只是1个路径映射，和模块名应该关系不大。模块名应该没啥用处，可能在报错方面会用模块名代替路径。

（3）加载非规范的模块

如果依赖的库没有安装AMD规范写，可以在require.config.shim中通过定义：1）exports值（输出的变量名），表明这个模块外部调用时的名称；2）deps数组，表明该模块的依赖性。



（4）require.js插件

可以使用一些插件（前提还是本地有这个插件），写入依赖数组中（路径）。如domready可以使在页面DOM结构加载完成后再运行。



**3.ES6中的模块化**

直接使用export和import命令实现。

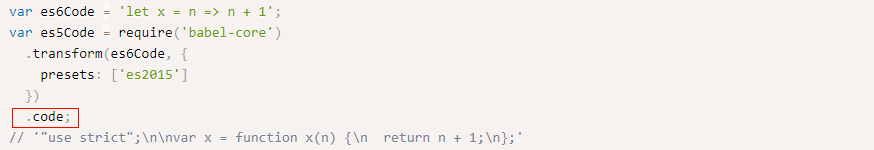
* 1. babel
     1. 各种包

**1.babel-core**

（1）babel的编译器。babel自身的核心api都在里面，如transform，用于将js代码抽象成ast（abstract syntax tree）。ast可以理解为用于分析js代码的一种树状结构。

（2）能转换es5语法，不能转换api。





**2.babel-cli**

命令行工具（二进制），用于在命令行中转码。如下，将script.js转码后输出为script-compiled.js。



**3.babel-node**

也是babel-cli下的一个命令，随babel-cli一起安装，不用单独安装，用于提供一个支持ES6的REPL环境。执行babel-node就进入PEPL环境。



**4.babel-register**

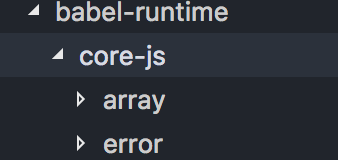
用于改写require命令，为require加上一个钩子。此后，每当使用require加载.js、.jsx、.es和.es6后缀名的文件，就会先用Babel进行转码。



**5.babel-runtime**

（1）babel-core只转换es6句法，不转换api。为此，可以使用core-js和regenerator。其中，core-js几乎包含了所有JavaScript最新标准的垫片，但不包含generator。regenerator是facebook的一个库，主要就是实现了generator/yeild和async/await。

（2）babel-runtime就是引用了core-js和regenerator，放在放到一个文件夹里作为运行时的helper。如果代码中用到了1个api，手动引入这个api对应的垫片。





（3）如果手动引入就太麻烦了，可以使用babel-plugin-transform-runtime插件自动加载代码中用到的垫片。

**6.babel-polyfill**

（1）同样是引用core-js和regenerator。

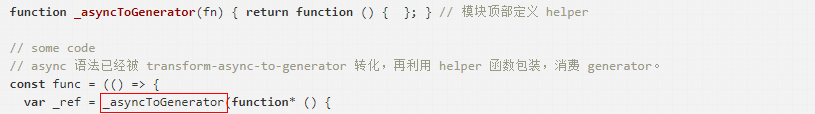
（2）使用babel-node时，babel-polyfill会自动加载。

（3）作用机制是重载全局变量，以及原型和类上的静态方法，从而实现对es6+的支持。因一次性引入项目中，无论是否用得着就编译到项目代码内，会大大增加体积。

**7.babel-external-helpers**

（1）babel-cli中的1个命令，用于生成babel转换所需要的所有helper函数的代码。

（2）babel transform执行过程中，需要用到很多帮助函数。这些帮助函数都在babel-helpers包中。如果babel编译的时候检测到某个文件需要这些helpers，就会在编译成模块的时候，将这些helper会放到模块的顶部。如下：



（3）如果多个文件都需要这些helper函数，会导致每个模块都定义一份，代码冗余。为此，babel提供babel-external-helpers命令，用于生成一个包含了所有helpers函数的helpers.js文件。然后在入口文件处引入这个文件，再通过一个plugin，去检测全局下是否存在这个模块，存在就不需要重新定义了。

生成helpers.js文件



安装、注册插件。





入口文件引入 helpers.js

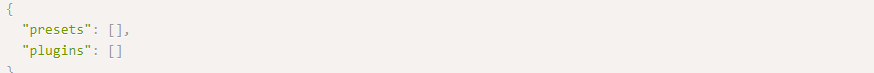


（4）如果使用了transform-runtime就不需要生成helpers.js文件了。

* + 1. 配置文件.babelrc

（1）官方推荐是写到.babelrc文件中，并存放在项目的根目录下。也可以在package.json里面添加babel字段。最后都会被babel-cli读取。

（2）配置文件格式：



（3）babel解析过程：1）通过babylon解析成AST。2）使用plugins/presets进一步编译转换，结果仍然是AST。3）使用babel-generator生成输出。

（4）plugins就是在第2步加强编译的。原理是遍历AST并在上面操作。

（5）如果只使用插件机制，要想将es5转换为es6，需要多达几十个不同的插件。为此，Babel将同属于es2015的插件集合到es2015这1个preset中。其他env、react、lastet、stage-x等preset类似。

（6）stage的含义如下，因此，stage2基本上就确定要发布了，所以一般也就支持到stage2。

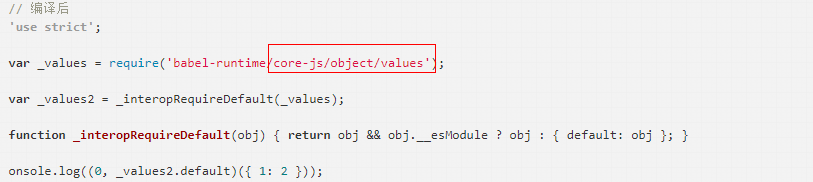


* + 1. 插件和presets使用

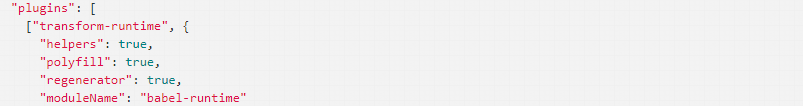
**1.babel-plugin-transform-runtime**

（1）babel-plugin-transform-runtime分析代码是否使用垫片，在在当前模块顶部插入需要的垫片。如：





（2）配置上，helpers设为false，就相当于没有启用babel-plugin-external-helpers。regenerator和polyfill分别对应regenerator和core-js。



（3）安装babel-plugin-transform-runtime会自动安装babel-runtime，然后从babel-runtime目录下获取垫片。

**2.babel-preset-env**

（1）官方推荐使用babel-preset-env，在没有其他任何配置项时，同babel-preset-latest，即等价于babel-preset-es2015、babel-preset-es2016、babel-preset-es2017结合一起。

（2）能能根据当前的运行环境，自动确定需要的plugins和polyfills。原理是通过各个es标准的feature在不同浏览器以及node版本的支持情况，维护一个feature跟plugins之间的映射关系，最终确定需要的plugins。

（3）上面的意思是，一部分node和浏览器以及实现了部分es6语法，因此不必全部转换，根据设置的目标进行转换。

（4）配置如下：其中useBuiltIns为true会自动判断需要什么样的polyfill，然后全部引入对应的垫片（不管用到没有，但浏览器支持的就不引入）。显然，使用transform-runtime代码更少。



* + 1. 使用

（1）如前所述，babel编译是babel-cli（貌似安装babel-core也行，安装babel-cli会自动安装babel-core）通过各种插件实现的，而babel-preset-env是各种插件的集合，因此安装这babel-cli和preset。



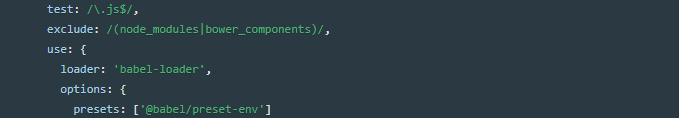
（2）因需要编译各种api，因此还需要安装babel-plugin-transform-runtime。

（3）webpack中使用babel：

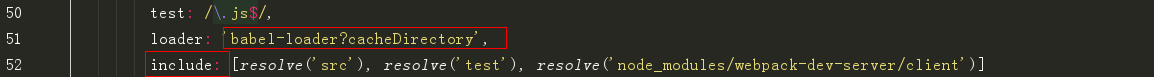
鉴于babel的node api转到babel-core中，所以安装babel-core，不安装babel-cli（不需要命令行）。



babel-loader在编译的时候，会自动调用babel-core（node命令），因此不需要配置其他。



options属性中的presets和plugins可以覆盖.babelrc配置。如果再.babelrc配置好了，这里就不用写这些配置了。



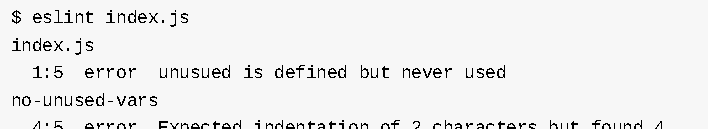
上面的配置都是babel-loader的配置，和babel没有关系。

* 1. ESLint

ESLint是一个语法规则和代码风格的检查工具，可以用来保证写出语法正确、风格统一的代码。

**1.使用**

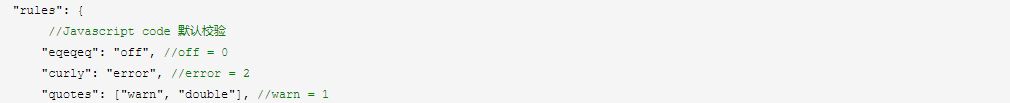
（1）安装ESLint后，在bin目录下便有其cmd文件，内部调用node执行一些代码，根据配置文件.eslintrc执行检查。可以直接在命令行中校验。



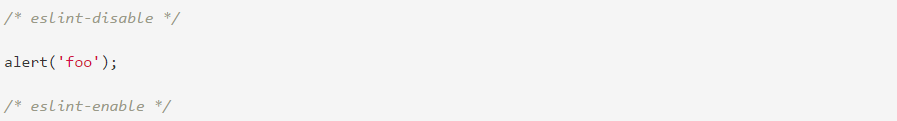
（2）运行eslint --init，项目根目录下将新建一个.eslintrc文件。

（3）parserOptions配置校验的ecma的版本；Parser配置校验解析器，默认为esprima，可以修改为babel-eslint；eslint默认已配置好一些环境，可以使用env定义脚本将要运行在什么环境中，如brower、node等，可以为多个；使用plugins选项可以包含第三方插件；Rules为定义开启规则和发生错误时报告的等级，其中错误等级有三种：

* 0或'off'：关闭规则。
* 1或'warn'：打开规则，并且作为一个警告（并不会导致检查不通过）。
* 2或'error'：打开规则，并且作为一个错误 (退出码为1，检查不通过)。



除Rules外，还可以直接在js文件注释中，临时开启/关闭规则检查：



**2.规则**

可能的逻辑错误

|  |  |
| --- | --- |
| no-cond-assign | 禁止条件表达式中出现赋值操作符 |
| no-control-regex | 禁止在正则表达式中使用控制字符 |
| no-dupe-args | 禁止function定义中出现重名参数 |
| no-empty-character-class | 禁止在正则表达式中使用空字符集 |
| no-func-assign | 禁止对function声明重新赋值 |
| no-inner-declarations | 禁止在嵌套的块中出现变量声明或function声明 |
| no-sparse-arrays | 禁用稀疏数组 |

* 1. 前端通信技术
     1. http

HTTP是基于TCP/IP协议的应用层协议，主要规定了客户端和服务器之间的通信格式，默认使用80端口。

**1.HTTP/0.9**

最早版本是0.9版，只有一个GET命令，只能传输字符串。

**2.****HTTP/1.0**

（1）概述

1996年发布HTTP/1.0版本，有如下功能：

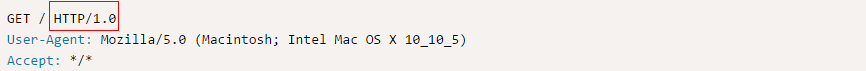
1）不仅可以传输文字，还能传输图像、视频、二进制文件。

2）除了GET命令，还引入了POST和HEAD命令，丰富了浏览器与服务器的互动手段。

3）请求和回应的格式也变了。除了数据部分，每次通信都必须包括头信息，用来描述一些元数据。

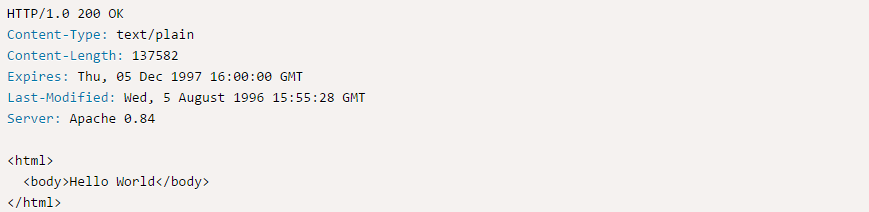
4）除此之外，还增加状态码、多字符集、多部分发送、权限、缓存、内容编码等功能。

（2）请求格式



第1行是请求命令，必须在尾部添加协议版本（HTTP/1.0）。后面就是多行头信息。

（3）回应格式



回应格式是“头信息”、“1个空行（\r\n）”、“数据”，其中第1行为“协议版本状态码状态描述”。

（4）Content-Type

1.0版规定，头信息必须是ASCII码，后面的数据可以是任何格式。因此需要在头信息中使用Content-Type说明后面的数据格式，如Content-Type、image/png、application/javascript等，即MIMEtype。

每个MIMEtype包括一级类型和二级类型，之间用斜杠分隔。除了预定义的类型，厂商也可以自定义类型。尾部使用分号，添加参数。



客户端请求的时候，可以使用Accept字段声明自己可以接受哪些数据格式。



MIMEtype不仅用在HTTP协议，还可以用在其他地方，比如HTML网页中。



（5）Content-Encoding

说明数据的压缩方法，可以为如gzip、deflate等。客户端在请求时，用Accept-Encoding字段说明自己可以接受哪些压缩方法。

（6）HTTP/1.0的缺点

主要缺点是每个TCP连接只能发送一个请求。发送数据完毕，连接就关闭，如果还要请求其他资源，就必须再新建一个连接。因TCP连接的新建成本很高，需要客户端和服务器3次握手，并且开始时发送速率较慢。为此，有些浏览器在请求时，用了一个非标准的Connection字段。



客户端请求和服务器端回应如果都带着个头信息，则不会关闭TCP连接，以便其他请求复用。

**3.HTTP/1.1**

（1）持久连接

即TCP连接默认不关闭，可以被多个请求复用，不用声明Connection为keep-alive。当客户端和服务器发现对方一段时间没有活动，就可以主动关闭连接。不过，规范的做法是，客户端在最后一个请求时，发送Connection:close，明确要求服务器关闭TCP连接。

对于同一个域名，大多数浏览器允许同时建立6个持久连接。

（2）管道机制

即在同一个TCP连接里面，客户端可以同时发送多个请求。如，客户端需要请求两个资源。以前的做法是，在同一个TCP连接里面，先发送A请求，然后等待服务器做出回应，收到后再发出B请求。管道机制则是允许浏览器同时发出A请求和B请求，但是服务器还是按照顺序，先回应A请求，完成后再回应B请求。

（3）Content-Length字段

因1个TCP连接现在可以传送多个回应，势必就需要有一种机制，区分数据包是属于哪一个回应的。在1.0版中，Content-Length字段不是必需的，因为浏览器发现服务器关闭了TCP连接，就表明收到的数据包已经全了。

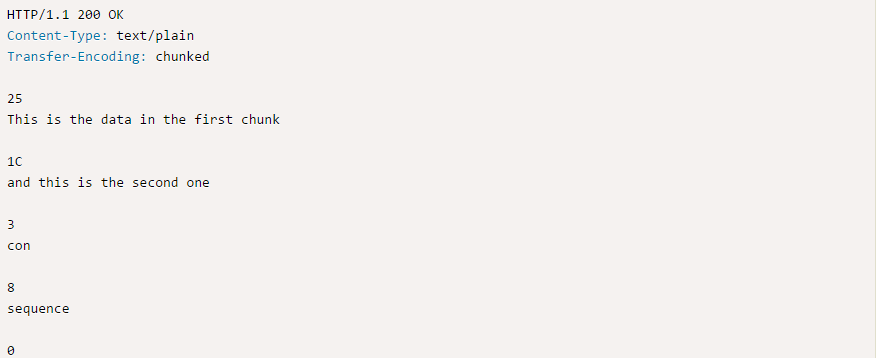
（4）分块传输编码

使用Content-Length字段的前提条件是，服务器发送回应之前，必须知道回应的数据长度。而对于一些很耗时的动态操作来说，这意味着，服务器要等到所有操作完成，才能发送数据，显然这样的效率不高。解决方法是，产生一块数据，就发送一块，采用流模式（stream）取代缓存模式（buffer）。

1.1版中规定可以不使用Content-Length字段，而使用分块传输编码。如果请求或回应的头信息有Transfer-Encoding字段，就表明回应将由数量未定的数据块组成。



每个非空的数据块之前，会有一个16进制的数值，表示这个块的长度。最后是一个大小为0的块，就表示本次回应的数据发送完了。



（5）其他

1.1版还新增了PUT、PATCH、HEAD、OPTIONS、DELETE等请求方法。客户端请求的头信息新增了Host字段，用来指定服务器的域名。为虚拟主机的兴起打下了基础。

* + 1. https

HTTPS安全超文本传输协议是HTTP的安全版，是使用TLS/SSL加密的HTTP协议。TLS/SSL安全传输层协议是介于TCP和HTTP之间的1层安全协议，不影响原有的TCP协议和HTTP协议，所以使用HTTPS基本上不需要对HTTP页面进行改造。

**1.****TLS/SSL的功能**

不加密的通信的风险：

* 窃听风险：第三方可以获知通信内容。
* 篡改风险：第三方可以修改通信内容。
* 冒充风险：第三方可以冒充他人身份参与通信（中间人劫持）。

TLS/SSL针对上面的风险需要实现如下功能：

* 所有信息都是加密传播，第三方无法窃听。
* 具有校验机制，一旦被篡改，通信双方会立刻发现。
* 配备身份证书，防止身份被冒充。

**2.TLS/SSL实现原理**

加密可区分为对称加密和非对称加密。两者区别为：对称加密只有一个密钥，加密和解密都用这个密钥。非对称加密有公钥和私钥，私钥加密后的内容只有公钥才能解密，公钥加密的内容只有私钥才能解密。

SSL/TLS协议采用非对称加密。但有如下问题：

（1）如何保证公钥不被篡改？

将公钥放在数字证书中。只要证书是可信的，公钥就是可信的。

（2）公钥加密计算量太大，如何减少耗用的时间？

每一次对话（session），客户端和服务器端都生成一个对话密钥（sessionkey），用来加密信息。由于对话密钥是对称加密，所以运算速度非常快。

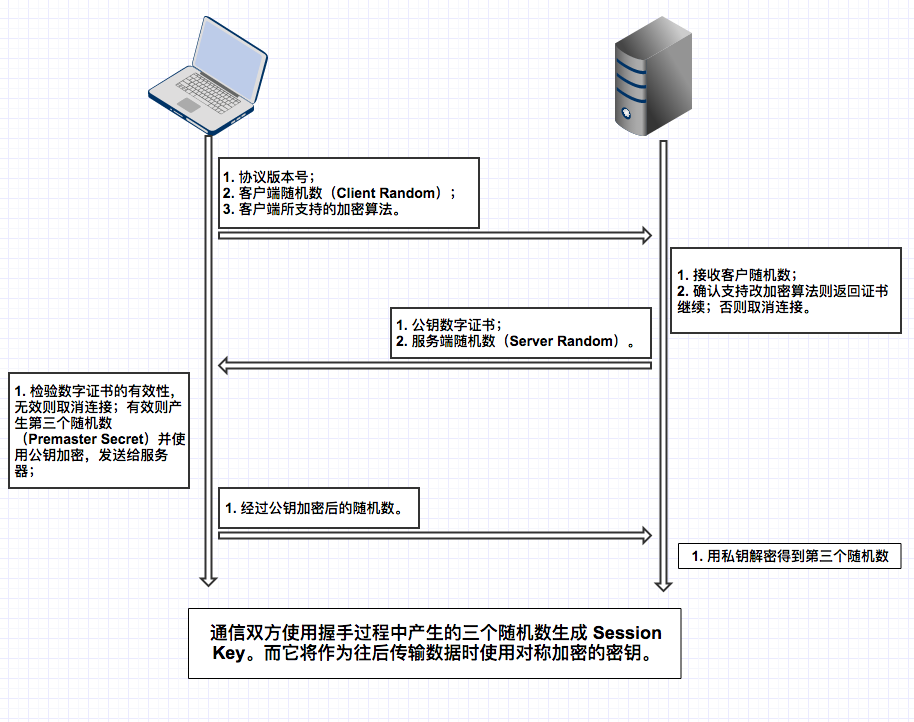
因此，SSL/TLS协议的基本过程是这样的：

1）客户端向服务器端索要并验证公钥。

2）双方协商生成对话密钥。

3）双方采用对话密钥进行加密通信。

**3.https握手阶段的详细过程**



（1）客户端（通常是浏览器）先向服务器发出加密通信的请求。

提供以下信息：1）支持的协议版本；2）一个客户端生成的随机数（用于稍后生成对话密钥）；3）支持的加密方法（如RSA公钥加密）；4）支持的压缩方法。

因客户端发送的信息之中不包括服务器的域名，理论上服务器只能包含一个网站。

（2）服务器收到客户端请求后，向客户端发出回应。

提供以下信息：1）确认使用的加密通信协议版本，如果支持的版本不一致，服务器关闭加密通信；2）一个服务器生成的随机数（用于稍后生成对话密钥）；3）确认使用的加密方法；4）服务器证书。

如果服务器需要确认客户端的身份，就会再包含一项请求，要求客户端提供客户端证书。如金融机构往往只允许认证客户连入自己的网络，就会向正式客户提供USB密钥，里面就包含了一张客户端证书。

（3）客户端回应。

客户端收到服务器回应以后，首先验证服务器证书。如果证书有问题，会向访问者显示一个警告，由其选择是否还要继续通信。如果证书没有问题，客户端就会从证书中取出服务端的公钥。然后向服务器发送：1）一个随机数。该随机数用服务器公钥加密，防止被窃听。2）编码改变通知，表示随后的信息都将用双方商定的加密方法和密钥发送；3）客户端握手结束通知，表示客户端的握手阶段已经结束。

（4）服务器的最后回应。

向客户端最后发送：1）编码改变通知；2）服务器握手结束通知。

握手结束。之后会用3个随机数合并作为本次会话密钥。

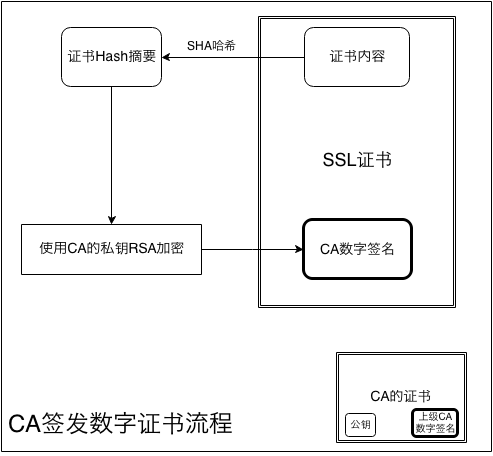
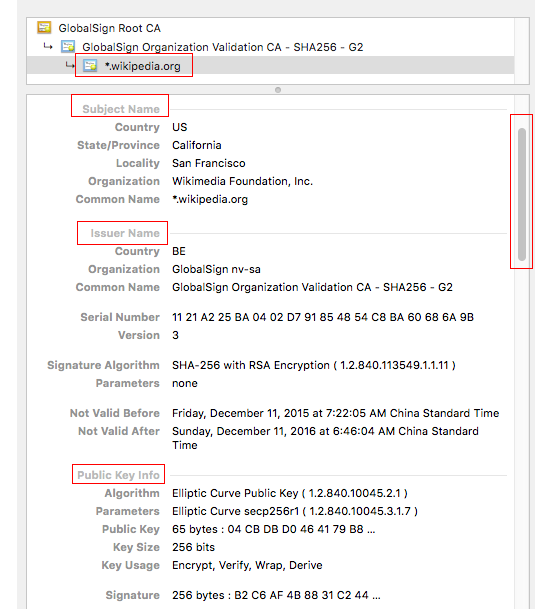
**4.session的恢复**

如果出于某种原因，对话中断，就需要重新握手。但也可以使用sessionID恢复原来的session。原理是每一次对话都有一个编号（sessionID）。如果对话中断，下次重连的时候，只要客户端给出这个编号，且服务器有这个编号的记录，双方就可以重新使用已有的"对话密钥"，而不必重新生成一遍。

**5.数字证书**

（1）证书内容

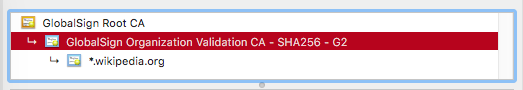
证书客户对象如域名、地址、国家等相关信息。证书颁发者对象相关信息。证书相关信息，如公钥、签名等信息。如Wikipedia的公钥证书如下：



（2）数字证书的生成及验证

数字证书是分层级生成的。即，下一级的证书需要其上一级证书的私钥签名（所以可以用公钥解密）。将证书客户对象信息通过SHA256计算出摘要，然后通过公钥解密，再与证书数字签名比对即可判断该证书是否合法。

问题1：怎么验证证书是有没被篡改？



当客户端通过HTTPS访问站点时，服务器会返回整个证书链。即，可以获取每1个证书环节的证书信息，然后由此一直往上验证到受信任的根证书。而操作系统和浏览器证书数据库保存受信任的根证书。

问题2：生成的数字证书都可用来签署新的证书吗？

不一定。证书中有1个字段记录路径长度约束，如果为0则不能继续签发。

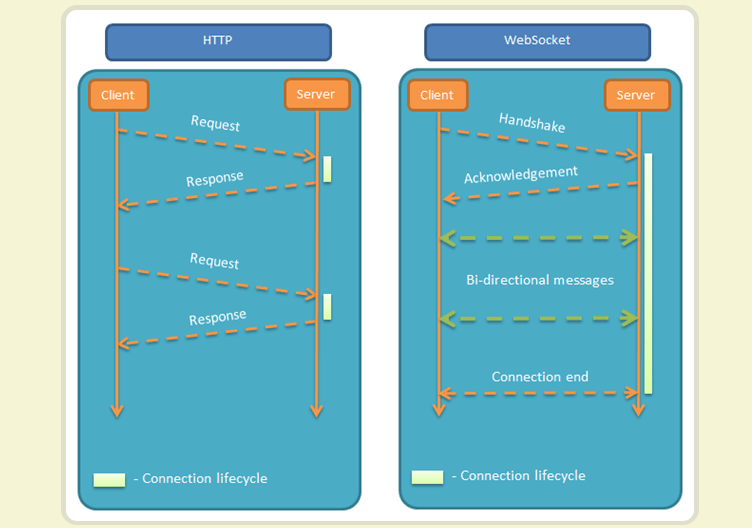
* + 1. WebSocket

WebSocket属于服务器推送技术的一种。最大特点就是服务器可以主动向客户端推送信息，是真正的双向平等对话。客户端ie10以上浏览器都支持。

**1.主要特点**

（1）建立在TCP协议之上，服务器端比较容易实现。

（2）与HTTP协议有着良好的兼容性。默认端口也是80和443，并且握手阶段采用HTTP协议，因此握手时不容易屏蔽，能通过各种HTTP代理服务器。



（3）可以发送文本，也可以发送二进制数据。

（4）没有同源限制，客户端可以与任意服务器通信。

（5）协议标识符是ws（如果加密，则为wss），服务器网址就是URL。

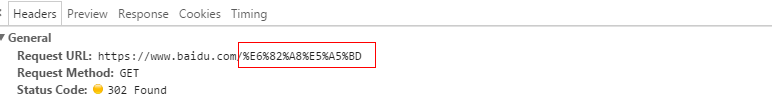


* + 1. js编码

**1.需要浏览器编码的几种情况**

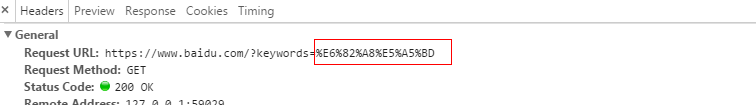
（1）url路径中包含汉字

如https://www.baidu.com/你好，将会转为：



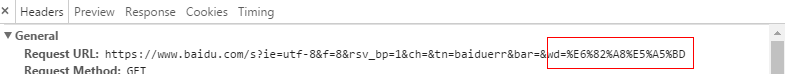
（2）查询字符串中包括汉字

如https://www.baidu.com/?keywords=%E6%82%A8%E5%A5%BD，将会转为：



（3）表单提交中包含汉字（get方法）





（4）ajax调用的URL包含汉字

如：$.ajax("url":"")中包含汉字。

（1）-（3）编码字符集由HTML源码中字符集的设定决定。



（4）由浏览器自行决定。不同浏览器可能不同。

**2.RFC-2396中关于URI字符的分类**

（1）基本字符

ASCII码中的字母和数字。

（2）保留字符

是URI中的保留关键字符，用于分割URI中的各个部分，有;/?:@&=+$,#。

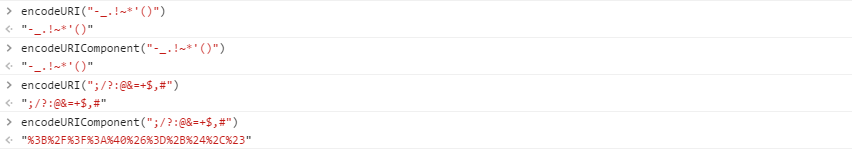
（3）Mark字符

特别定义，但又没有特别说明用途。

另外，在对url编码时，会将空格转为+（%20）。

**3.encodeURI和encodeURIComponent**

encodeURI对所有基本字符、保留字符、Mark字符外的字符编码，encodeURIComponent还会对保留字符编码。编码后输出形式utf-8，并且在每个字节前加上%。



两者各用于不同场景。如传递的参数为url时需要使用encodeURIComponent，这样参数url才不会被#等特殊字符截断。



如果只是url跳转，使用encodeURI即可。如Location.href="/encodeURI"。

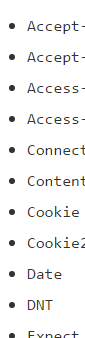
**4.escape**

ES3已不推荐使用。不能直接用于URL编码，可以返回1个字符的Unicode编码值。

* + 1. cookie

（1）禁止修改的消息首部

禁止修改的消息首部指的是不能通过编码的方式（直接修改header）进行修改的HTTP协议消息首部。禁止修改的消息首部包括以Proxy-和Sec-开头的消息首部。所以用户代理对消息首部拥有完全的掌控权。



（2）HttpOnly

cookie有个属性叫HttpOnly，当这个值设为true时，浏览器是无法读取这个cookie值的，只有服务端能够读取（包括document.cookie）。

* 1. CORS

http://www.ruanyifeng.com/blog/2016/04/cors.html

全称为跨域资源共享，允许浏览器向跨源服务器，发出XMLHttpRequest请求，从而克服了AJAX只能同源使用的限制。CORS需要浏览器和服务器同时支持。目前，IE10以上浏览器均支持。

浏览器将CORS请求分成简单请求和非简单请求。非简单请求是对服务器有特殊要求的请求，如请求方法是PUT或DELETE，或者Content-Type字段的类型是application/json。非简单请求的CORS请求，会在正式通信之前，增加一次HTTP查询请求，称为预检请求。如果服务器肯定回复，之后则和简单请求一样。

简单请求头会添加Origin字段，说明源。服务器返回的响应会增加Access-Control-Allow-Origin、Access-Control-Allow-Credentials、Access-Control-Expose-Headers头字段。

CORS请求默认不发送cookie，需要设置withCredentials属性。



* 1. fetch
     1. 语法

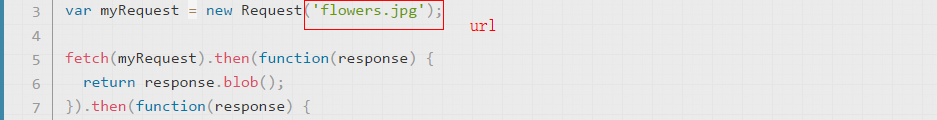
**1.fetch**

fetch用于发起获取资源的请求。



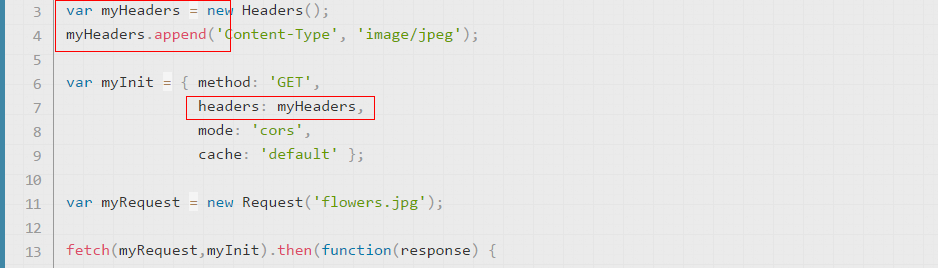
（1）接收参数

1）input：要获取的资源，可以为1个url（string）或Request对象。



2）init：配置项对象，包括所有对请求的设置。可选的参数有：

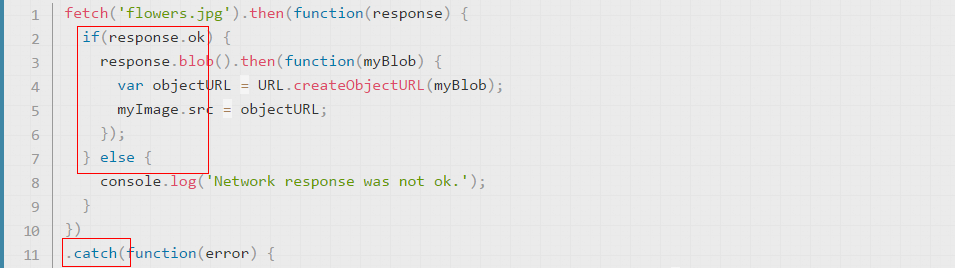
* method：请求使用的方法，如GET、POST。
* headers：请求的头信息，形式为对象字面量。



* body：请求的body信息。GET或HEAD方法的请求不能包含body信息。
* mode：请求的模式，如cors、no-cors或者same-origin。
* credentials：请求的credentials。为了在当前域名内自动发送cookie，必须提供这个选项。
* cache：请求的cache模式：default、no-store、reload、no-cache、force-cache或者only-if-cached。
* redirect：可用的redirect模式：follow(自动重定向),error(如果产生重定向将自动终止并且抛出一个错误),或者manual(手动处理重定向)。
* referrer：可以是no-referrer、client或URL，默认是client。

（2）返回值

返回一个promise。promise会在请求响应后被resolve，并传回Response对象。与ajax等不同的是：只有遇到网络错误（也可能是权限或其它问题导致）时，fetch返回的promise才会被reject，并传回TypeError。包括404等都不被认为是网络故障。因此，请求是否成功还需要检查Response.ok属性为true。



**2.Request**

用于创建一个新的Request对象。



参数同fetch，也可以接受init。



**3.Headers**

创建一个新的Headers对象。



可以使用Headers.append方法添加一个header并赋值：



也可以创建时添加多个header：



**4.Response**

在fetch处理完promises之后返回，也通过JavaScript来创建。常见属性有：

status：response的状态码。

statusText：与HTTP状态码消息对应。

ok：检查response的状态是否在200-299范围内，返回一个Boolean值。

**5.Body**

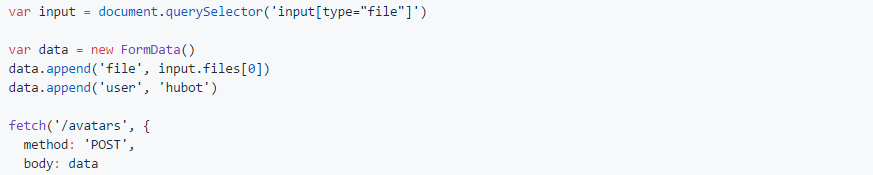
为响应/请求的正文。可以是string、URLSearchParams、FormData、ArrayBuffer、ArrayBufferView、Blob/File等类型的实例。有属性bodyUsed表明是否读取过。定义了text()、json()、formData()、blob()、arrayBuffer()等方法，以获取body内容。

* + 1. 使用

（1）提交表单数据







（2）处理Text/HTML、json响应。





（3）发送cookie

自动附带，不跨域使用same-origin，跨域使用include。



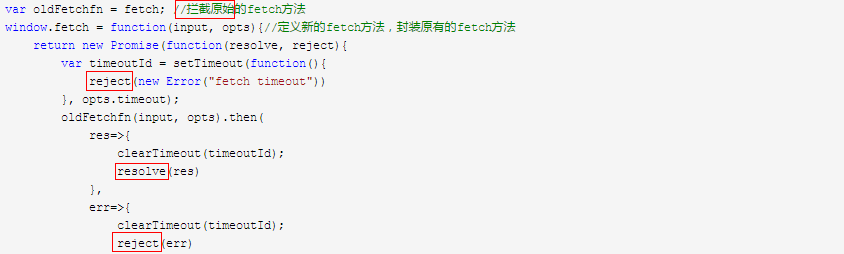
（4）接受cookie

与XMLHttpRequest一样，不能直接从header中读取cookie，而是由浏览器负责处理，可以通过document.cookie读取（也可以设置）。

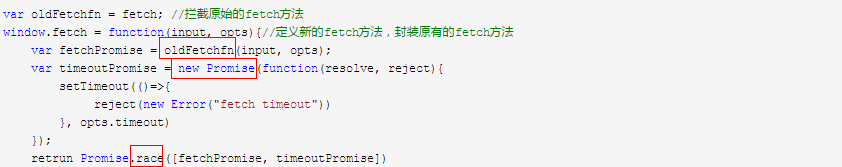
（5）fetch不支持超时timeout的处理

由于fetch返回的promise实例，根据Promise指导规范标准，fetch不能被abort，也不能手动改变promise实例状态，只能由内部来根据请求结果来改变promise的状态。对此，目前有2种不同的方法处理：

1）单纯setTimeout方式



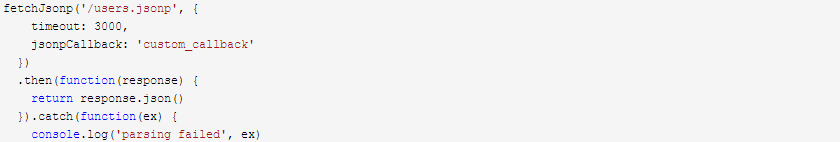
2）利用Promise.race方法



但是，即使fetch的timeout超时发生了，本次请求也不会被abort丢弃掉。后台仍然会发送到服务器端。只是本次请求的响应内容被丢弃而已。

（6）fetch不支持JSONP

使用fetch-jsonp模块。



（7）fetch不支持progress事件

response.body是一个可读字节流对象，其实现了一个getRender方法，可以利用这点模拟progress。

（8）跨域问题

XHR2支持跨域请求的，需要满足浏览器端支持CORS，且服务器通过Access-Control-Allow-Origin允许指定的源实现。fetch除支持这种方式外，还支持一种跨域，使用CORS方式，不需要服务器支持，通过设置mode配置项实现。

* same-origin：不允许跨域。
* cors：支持跨域请求，对应的responsetype为cors。
* no-cors：用于跨域请求但服务器不带CORS响应头，也就是服务端不支持CORS，是fetch的特殊跨域请求方式。其对应的responsetype为opaque。

（9）浏览器支持

引入如下polyfill后可以完美支持IE8+。基本原理是探测是否存在window.fetch方法，如果没有则用XHR实现。

* 由于IE8是ES3，需要引入ES5的polyfill：es5-shim、es5-sham。
* 引入Promise的polyfill：es6-promise。
* 引入fetch探测库：fetch-detector。
* 引入fetch的polyfill：fetch-ie8。
* 可选：如果你还使用了jsonp，引入fetch-jsonp。
* 可选：开启Babel的runtime模式，现在就使用async/await。

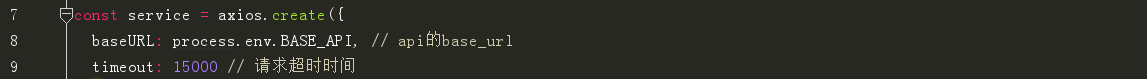
（10）其他问题

fetch不支持同步请求。

* 1. axios
     1. api

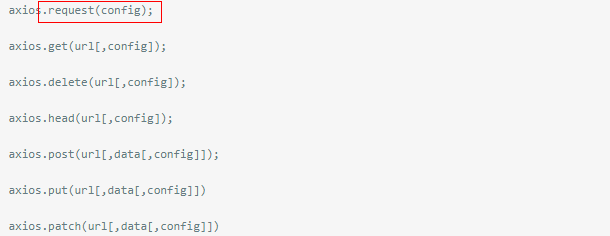
**1.axios(config)和axios(url[,config])**

貌似实例也可以。

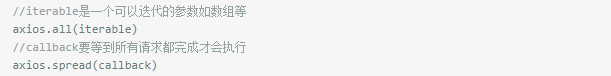




**2.请求方式别名**



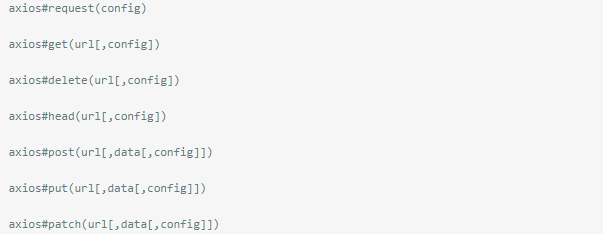
**3.并发请求**



**4.创建一个axios实例**



**5.实例方法**

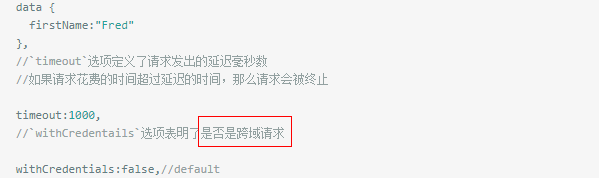


* + 1. 配置

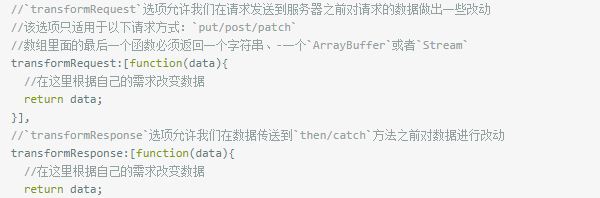
只有url选项是必须的，如果method选项未定义，默认为GET。





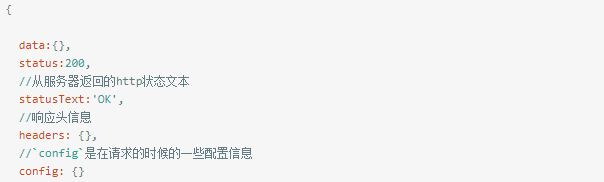




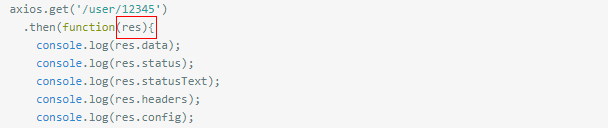


* + 1. 获取响应

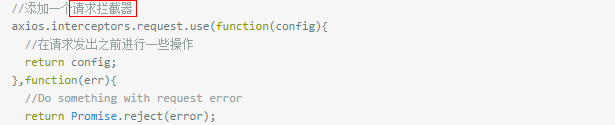
如果服务端返回的数据为：

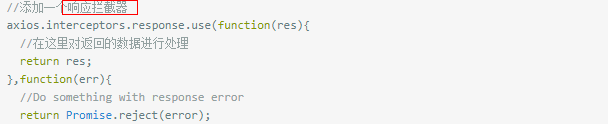


则可以从Promise中如下获取：



* + 1. 拦截器





取消拦截器：



给实例添加拦截器：



* + 1. 取消请求

可以通过一个cancel token来取消一个请求。可以通过CancelToken.source工厂函数来创建一个cancel token。



也可以给cancelToken构造函数传递一个executor function来创建一个cancel token。



* 1. 认证
     1. Cookie Auth

最传统的方式：认证之后，在服务端创建一个Session对象，同时在浏览器端创建了一个Cookie对象。通过客户端带上来Cookie对象来与服务器端的session对象匹配来实现状态管理的。

* + 1. HTTP Basic Auth

每次请求API时都提供用户的username和password。虽然简单，但存在将用户名密码暴露给第三方客户端的风险，使用越来越少。

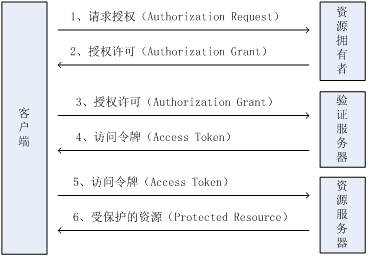
* + 1. OAuth

（1）OAuth（开放授权）是一个开放的授权标准，允许用户让第三方应用访问该用户在某一web服务上存储的私密的资源（如照片，视频，联系人列表），而无需将用户名和密码提供给第三方应用。

（2）OAuth允许用户提供一个令牌，而不是用户名和密码来访问他们存放在特定服务提供者的数据。

（3）每一个令牌授权一个特定的第三方系统（例如，视频编辑网站)在特定的时段（例如，接下来的2小时内）内访问特定的资源（例如仅仅是某一相册中的视频）。即，只让用户授权给第三方网站访问他们存储在另外服务提供者的某些特定信息，而非所有内容。

（4）流程：



请求授权：用户打开第三方网站，第三方网站要求用户给予授权。

授权许可：用户同意给予客户端授权。

授权许可：第三方网站使用上一步获得的授权，向认证服务器申请令牌。

访问令牌：认证服务器对第三方网站进行认证以后，确认无误，同意发放令牌。

访问令牌：第三方网站使用令牌，向资源服务器申请获取资源。

受保护资源：资源服务器确认令牌无误，同意向第三方网站开放资源。

* + 1. Token Auth

**1.相对cookie方式的优势**

（1）支持跨域访问：Cookie不允许垮域访问。如果用户认证信息通过HTTP头传输，Token机制可以跨域。

（2）无状态（即，服务端可扩展行）：服务端不需要存储session信息，因为Token自身包含了所有登录用户的信息，只需在客户端的cookie或本地介质存储状态信息即可。

（3）去耦： Token可以在任何地方生成，不需要绑定到一个特定的身份验证方案。

（4）更适用于移动应用：原生平台（iOS、Android等）不支持Cookie。

（5）CSRF：因为不再依赖于Cookie，所以不存在CSRF（跨站请求伪造）。

**2.基于JWT的Token认证机制实现**

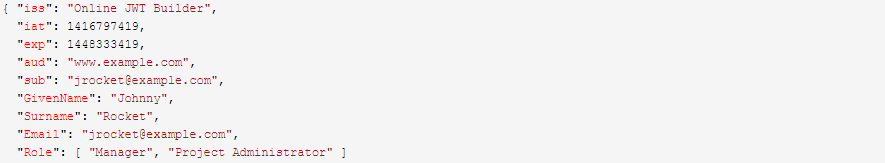
（1）一个JWT是一个字符串，由三部分组成：头部、载荷与签名。

（2）头部：用于描述JWT的最基本的信息，如类型、签名算法等。这些信息是一个JSON对象，然后进行BASE64编码。





（3）载荷：包含了如JWT签发者（服务端签发的）、接收方（客户端）、有效期等信息。也是1个json对象，然后在BASE64编码。





（4）签名：上面的两个编码后的字符串用句号.连接在一起（头部在前），为：



然后用头部设置的签名算法加密（还需要1个salt），然后拼接在上面的字符串后面：



**3.认证过程**

（1）登录：第一次认证，用户在浏览器输入用户名/密码，提交后到服务器，服务器进行认证，生成JWT（都有第三方库），设置到cookie中返回客户端。

（2）之后的请求：每次客户端请求都会带上这个token，可能在cookie中，也可以在HTTP的Authorization头中。服务器先检查Authorization头，在从cookie中获取。然后服务端调用库对token解密解码，并对token中的签发者、接收方、有效期等进行验证，并与签名信息匹配。

（3）注意token中并没有用户的用户名和密码，只是本次登录后双方的一些信息。

* 1. Mock

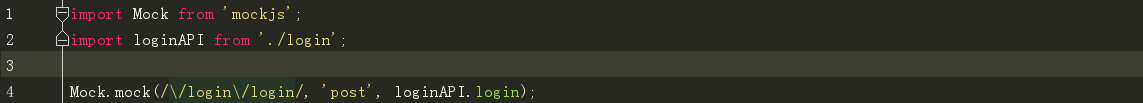
**1.原理**

并非拦截请求，而是将发送请求的函数（如ajax）替换为mock.js设置的函数，导致请求的是mock的数据。

**2.使用**

如下，请求login/login，如果mock中已经注册过，就到了mock的逻辑中。

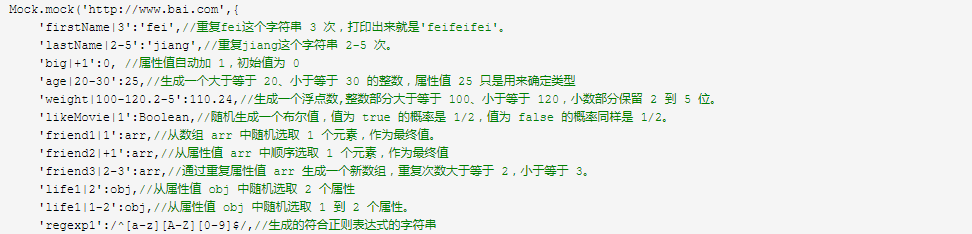




**3.数据生成**

（1）可以使用mock提供的数据模板自动生成数据。

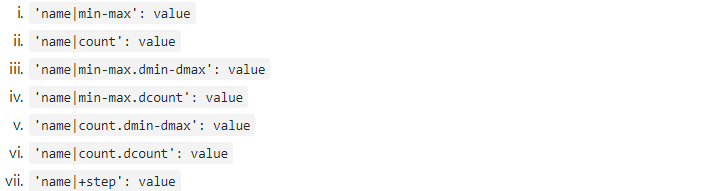
（2）数据模板中的每个属性由3部分构成：属性名、生成规则、属性值。属性名和生成规则之间用竖线|分隔。生成规则是可选的。



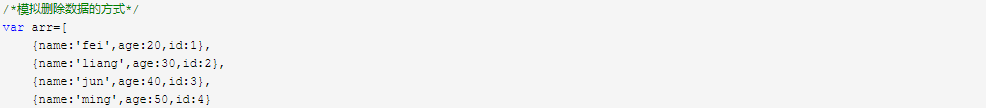


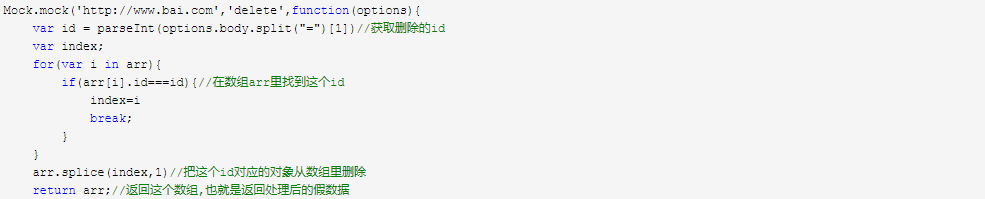
返回。

（3）生成规则有7种格式：



（4）实现增删改查模拟接口：





* 1. Markdown
  2. typescript
  3. 拖拽

**1.普通拖拽**

（1）除文本、图片、超链接外，其它元素默认不可拖拽。让其他HTML元素也能被拖拽，必须：1）给拖拽元素设置draggable属性；2）为拖拽元素添加dragstart事件侦听；3）在侦听中设置拖拽数据。

（2）一旦元素设置为可拖拽，则无法再直接选中元素中的文本，需要使用alt。

（3）支持的事件：dragStart、dragend和drag，分别对应拖拽开始、结束和拖拽过程中（不断触发）。

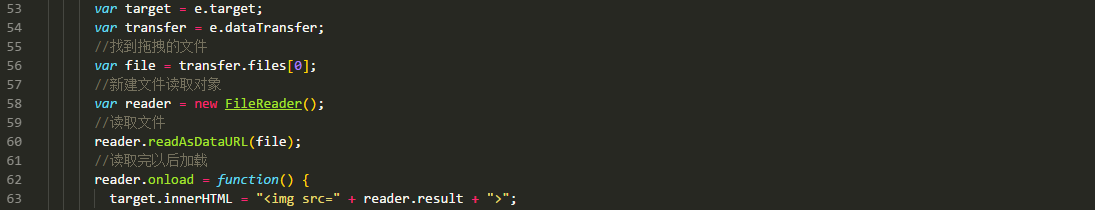


（4）貌似任何元素都可能成为目标元素，只要拖拽经过该目标元素，都会触发对应事件。目标元素支持的事件有：dragenter（进入时触发）、dragleave（离开时触发）、dragover（到达后不断触发）、drop（鼠标发下完成拖拽时触发）。

因元素默认不允许拖放，需要在dragover的事件中组织默认行为。

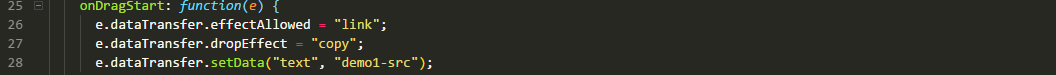
**2.文件拖拽上传**

貌似只是可以从DataTransfer对象的files获取上传文件，其他没有任何区别。



**3.DataTransfer对象**

（1）只要在拖拽过程中，在任何拖拽事件侦听函数的事件对象上，都可以获取DataTransfer对象。





（2）对象最重要的方法是getData()和setData()。

支持如下format：text/html（HTML代码格式）、text/plain（文本文字格式）、text/xml（XML字符格式）、text/url-list（URL格式列表），另外为了兼容，也支持text和URL格式。

通过setData()保存在dataTransfer对象中的数据，只能在drop事件处理程序中读取。如果没有读取，那将随着dataTransfer对象销毁而丢失。

（3）effectAllowed和dropEffect用于设置交互类型。经测，貌似除了设为none会阻止拖拽外，其他设置都没有附加的动作，要想复制、移动等操作，还是自己操作dom。但是，设置不同的值，在进入目标对象时，显示的鼠标光标不同。