axios 学习笔记

axios 是一个用于接收、发出请求的 js 库。它可以出现在普通的 html 页面中，通过浏览器发出请求、接收返回的响应；它还可以出现在以 nodejs 提供服务的后端场景中。一个库同时提供了前端与后端接发请求的解决方案，这大大方便了前端人员的开发效率以及学习成本。

> 一、axios 在浏览器 与 nodejs 收发请求的分别实现

首先，浏览器 与 nodejs 进行收发请求的实现方式并不相同，各有个的招式。

- 1、浏览器是通过其内部提供的 XMLHttpRequest 接口提供给用户使用
- 2、nodejs 中，是通过其内部提供的 http 库，提供给用户使用的

两种环境（浏览器、nodejs）使用了两种不同方式去解决了相同收发请求的问题，虽然它俩（浏览器、nodejs）各自用自己的喜好完成了同一件事，但是也是有共通点的，就是它们都看得懂 javascript 。

那么作为 js 库的 axios 想要在这两种环境中都能实现同样的收发请求功能，要做事就是“见人说人话，见鬼说鬼话”这一件事了。就是要以浏览器、nodejs 它俩各自不同的方式进行对接。当 axios 发现自己运行在浏览器环境中，就使用 XMLHttpRequest ，遇到 nodejs 时就换成 nodejs 喜欢 http 方式。原理与思路就是这样并不难理解，axios 其内部的源码实现也不难(adapters 加起来都四百多行了...) ……

经过阅读 axios 没有打包的 lib 文件夹的源码发现，（其主入口 index.js 文件就是直接导出 引入后的 './lib/axios'）在 axios.js 文件中，引入了 default 模块，这个模块就完成了一件事：在 XMLHttpRequest 与 http 中选择好，使用那一个才能让自己所在环境能够“听得懂”。 default.js 中有一个 getDefaultAdapter 的函数来完成这个选择。对于是否是浏览器环境的判读比较好理解，直接环境中是否有 XMLHttpRequest 这个对象，如果为 true，自然就是了（浏览器环境）；而对于是否是 nodejs 环境的判断， axios 的判断更周全，这点有向它学习到了。一般来说经过上面那步，已经判知了不是浏览器环境，那么axios 又只打算兼容浏览器与 nodejs 这两个环境，那不是浏览器，那就必须只能是 nodejs 环境了呗，要我肯定就直接简单的走 if..else 了，但发现其并没有这么简单粗暴。进行 else if 的再判断，判断环境下是否有 process （nodejs下进程事件）并且还对这个环境下的 Object 的原型进行判读（将其this指向到了 process 上，又将结果转成了字符串），判断是否是全等于字符串 '[object process]' 。这步操作可见作者的功力，其用意虽然是理解的了，但本人目前尚不了解其他后端语言，不知道其他语言会将其 Object 打印成什么东东给我们看。但可以看出经过这一层的“与”判断就能确定是 nodejs 环境的了。断定了所在环境，剩下的就是将封装好的 XMLHttpRequest 或 http 模块引入，用变量保存，并将这个变量 return 出去。

```
function getDefaultAdapter() {
  var adapter;
  if (typeof XMLHttpRequest !== 'undefined') {
    // For browsers use XHR adapter
    adapter = require('./adapters/xhr');
  } else if (typeof process !== 'undefined' &&
Object.prototype.toString.call(process) === '[object process]') {
    // For node use HTTP adapter
    adapter = require('./adapters/http');
  }
  return adapter;
}
```

经过判断环境后， default.js 文件里声明了一个变量，最终导出的也是它。可知在这个 default.js 文件中，判读环境后的操作，都是以不断地修饰这个最终要导出的变量而进行的。要进行的内容主要是 request（请求）、response（响应）的处理，当然它还对网络通信的状态进行的判断、XSRF 的问题。

- Http 的状态码要保证在 200 以内，即成功的状态。
    - 100 、101 服务器收到请求一类
    - 300 重定向一类
    - 300 以上全是错误的类型了
- 对于 request 与 response 装饰过程中，发现 axios 是将对应的函数存到了数组中，可能是为了方便再添加、扩展等操作
- request 的装饰方法中接收2个参数，1、请求携带的数据，2、请求头。根据传递数据的不同复写请求头
- response 装饰方法里发现了一个它的小操作。如果响应返回的数据是 string 格式，会将其转为 JSON 格式再返回前端，这步操作使用了 try/catch 但是 catch 部分是不打印错误的。
- 最后还设置了通用、默认的 header ，为：json 的格式

> 二、Axios 是一个基于 promise 的 请求与响应的库

当判定了所在环境后， defaults.adapter 会引入对应的 xhr 或 http 模块，但无论是哪一个，返回的都是一个 promise 。这个 promise 的运用，使我们后续的编写更加优雅。如我们在前端使用 ajax 技术，浏览器的 XMLHttpRequest api 发请求，这是一个异步过程。收到响应以后的操作要使用回调的方式进行，即在 xhr.onload 给它一个回调函数。简单的使用回调的方式进行，会降低代码的可读性。使用 promise 进行包裹后，使得我们在前端可以配合 async/await 的写法，变得像同步操作一样。而且 axios 内部使用了 promise 后，处理请求与响应的任务也更加清晰。在 ./lib/core/Axios.js 文件中 声明了一个 chain 的变量，对任务进行了管理，并且以2个为一组的形式进行存放任务(1个是任务成功回调，1个是任务失败回调)，通过不断调用 promise.then 方法形成一个 promise 链，在执行的过程中对 chain 任务数组使用了 shift 方法，并且每次 .then 时都是以一对（成功的回调、失败的回调）的方式进行，直至全部任务被执行完毕。这样不断地 promise.then 的方式将所有任务的执行串联了起来。

```
  // Hook up interceptors middleware
  var chain = [dispatchRequest, undefined];
  var promise = Promise.resolve(config);

  this.interceptors.request.forEach(function
unshiftRequestInterceptors(interceptor) {
    chain.unshift(interceptor.fulfilled, interceptor.rejected);
  });

  this.interceptors.response.forEach(function
pushResponseInterceptors(interceptor) {
    chain.push(interceptor.fulfilled, interceptor.rejected);
  });

  while (chain.length) {
    promise = promise.then(chain.shift(), chain.shift());
  }
```

- 在这段代码的上面，还发现了：原来使用 axios 是可以不填请求的方法的（如果想用的方式是 'get' 的方式），它对请求方法这个环节也做了处理。

- 全部转为小写
- 默认是 get 请求

三、拦截请求和响应

拦截请求和响应这个操作，感觉是现在很多库必备的一个环节了，就像 vue.js 提供了很多的钩子函数，方便我们在使用时，可以对执行过程中一些特定环节添加想要执行的内容。

在 ./lib/core/Axios.js 文件里引入了一个 InterceptorManager （拦截管理者）的模块，并在构造函数里分别为 request、response 添加了拦截器管理。

InterceptorManager 这个类的原型中也有一个 use 函数，其用意同 koa 库中的 use；用来注册中间件的。只不过 InterceptorManager 的 use 方法接收的不是 ctx 与 next，而是 fulfilled（成功函数），rejected（失败函数）。

```
InterceptorManager.prototype.use = function use(fulfilled, rejected) {
  //添加一个新的拦截器
  this.handlers.push({        // push 到 this.handlers 数组里面，向后添加
    fulfilled: fulfilled,     // 成功函数
    rejected: rejected        // 失败函数
  });
  return this.handlers.length - 1;   // 返回索引
};
```

四、取消请求

axios 的取消机制分为 3 个模块，1、Cancel.js，2、CancelToken.js，3、isCancel.js，其中 CanelToken.js 是主模块。Cancel.js 导出的是一个 Object ，并在执行取消操作的时候被实例化，且用于对是否是进行了多次取消操作的辨别，在取消操作的时候它也会被抛出。isCancel.js 返回的是一个布尔值，判断是否取消了。在调用时它接收的应该是一个 Cancel 实例，然后内部判断是否传了这个参数，如果进行了传参，进行 与 运算，判断该参数的 **CANCEL** 属性，最终将结果返回。

```
// Cancel.js
function Cancel(message) {
  this.message = message;
}

Cancel.prototype.toString = function toString() {
  return 'Cancel' + (this.message ? ': ' + this.message : '');
};

Cancel.prototype.__CANCEL__ = true;
```

学习总结

axios 的特性除了上面简述的 4 项内容以外，还有【转换请求数据和响应数据】、【自动转换 JSON 数据】、【客户端支持防御 XSRF】。【转换请求数据和响应数据】、【自动转换 JSON 数据】这两个特性，在前面的源码阅读中已经读到了，并没看到对这两项有单独提取出去进行处理，是与其他特性的解决方案中并行处理了。

经过此次 axios 的课程及课后阅读源码的过程，感觉很有收获，也提升了阅读源码的兴趣。虽然在此次阅读源码的经历，axios 中的 CancelToken.js 中还有小部分内容没有看懂，还需再对其多进行几回拆分测试、理解等等。但已然是开启了阅读源码的兴趣大门。

暗号：axios