axios 学习笔记

介绍

axios 是基于 Promise 的 HTTP 客户端,可以运行在浏览器也可以运行在 node 环境下。

特性

1. axios 会根据环境判断做适配
   (1) 如果是浏览器则调用 XMLHttpRequests 对象
   (2) 如果是 node 则调用 http 模块
2. 支持 promise 的 api
3. 会拦截请求和响应（Intercept）
4. 对请求数据和响应数据的转换
5. 取消请求
6. 自动转换 json 数据
7. 支持 XSRF 的安全性的处理

基本使用

```
axios({
      method: 'post',
      url: '/user/12345',
      data: {
          firstName: 'Fred',
          lastName: 'Flintstone'
      }
});
```

参数含义：

method 请求的方式

url 请求的地址

data 请求携带在请求正文里的数据（一般用于'PUT', 'POST', 'DELETE , 'PATCH'）

其他参数：

baseURL 设置基准 url,会被加到请求 url 之前

transformRequest 对请求的数据进行数据转换

transformResponse 对响应的数据进行转换

params 带到 url 上的参数(URLSearchParams object) 也就是我们说的 queryString

paramsSerializer 对 params 的解析处理,负责序列化 params

data 也可以是字符串的写法，例如'Country=Brasil&City=Belo Horizonte'

withCredentials 做跨域设置

adapte 适配器,对浏览器和 node 环境的适配处理，如果你想用 fetch 也可以在这里做处理

responseType 返回的数据类型

responseEncoding 返回数据的编码格式

onUploadProgress 监控上传的进度

onDownloadProgress 监控下载的进度

maxContentLength 最大响应的内容大小

maxBodyLength 最大请求内容的大小

创建一个基础配置的 axios

```
const instance = axios.create({
  baseURL: https://api.example.com',
  timeout: 1000,
  headers: {'X-Custom-Header': 'foobar'}
});
```

我们可以在基础配置的 axios 上继续添加我们的参数，简化了代码；
或者我们可以通过 Global axios defaults 来对 axios 进行基础的配置。

```
axios.defaults.baseURL = 'https://api.example.com';
```

Interceptors 拦截器

拦截器是在 axios 请求之前和响应之后部分注入的外部动作，而我们在 koa 中使用的中间件是向框架的中间层部分注入的外部动作。

```
axios.interceptors.request.use(function(config) {
     config.headers.a = 'kkb';
     return config;
}, function() {

});
```

上述的例子是在请求前在请求头中添加一个 a 属性，axios 在请求时会携带这个属性。

```
axios.interceptors.response.use(function(response) {
    if ([400,401].find(response.status)) {
         // alert('....');
    }
```

```
    return response;
}, function() {
```

```
});
```

上述的例子是在响应前对数据进行处理，若状态码出现 400 或 401 则进行一些处理。

源码分析

```
var axios = createInstance(defaults);
axios.create = function create(instanceConfig) {
  return createInstance(mergeConfig(axios.defaults, instanceConfig));
};
```

axios 是通过 createInstance 函数创建而来，并且 axios 向外暴露了一个 create 方法，这个方法同 axios 一样都是通过 createInstance 方法创建而来，不同的是 create 方法将传入的配置和默认配置做了合并处理。

```
function createInstance(defaultConfig) {
  var context = new Axios(defaultConfig);
  var instance = bind(Axios.prototype.request, context);
  utils.extend(instance, Axios.prototype, context);
  utils.extend(instance, context);
  return instance;
}
```

createInstance 这个函数返回了一个 instance,instance 是 Axios.prototype.request，而它的 this 被绑定到 Axios 实例对象上，并且将 Axios.prototype 和 Axios 实例上的属性和方法代理到 instance 上。那么 instance 就可以使用 Axios.prototype.request 等 Axios 原型上的方法。

```
function Axios(instanceConfig) {
  this.defaults = instanceConfig;
  this.interceptors = {
    request: new InterceptorManager(),
    response: new InterceptorManager()
  };
}
```

Axios 对象是通过 es5 的构造函数写的，这个函数初始化定义了两个事情，一个是定义默认配置，一个是定义了拦截器（请求和响应），在他的原型上定义了方法，例如 request，getUri 以及请求的一些方法。

interceptors 拦截器，定义了请求拦截和响应拦截，不过均使用的是 InterceptorManager 对象，这个对象内部维护了一个数组，这个数组存入的是对象需要包含两个函数，一个成功的（resolve），一个失败的(reject)。

Axios 原型上的 request 方法中：

```
var chain = [dispatchRequest, undefined];
var promise = Promise.resolve(config);
```

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

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

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

dispatchRequest 是发请求的，this.interceptors.request.forEach 将请求的内容拦截并做处理，把拦截的内容向前添加到 chain 中，this.interceptors.response.forEach 将响应的内容拦截也做处理，把拦截的内容向后添加到 chain 中。例子如下：

```
chain = [dispatchRequest, undefined]
axios.interceptors.request.use(rs1, rf1);
axios.interceptors.request.use(rs2, rf2);
axios.interceptors.response.use(ps1, pf1);
axios.interceptors.response.use(ps2, pf2);
```

chain 结果就成了

```
chain = [rs2, rf2, rs1, rf1, dispatchRequest, undefined, ps1, pf1, ps2,
pf2]
```

然后循环 chain，执行顺序如下所示：

```
promise.then(rs2, rf2).then(rs1, rf1).then(dispatchRequest, undefined).
then(ps1, pf1).then(ps2, pf2)
```

这样就实现了在 dispatchRequest 请求之前处理 axios.interceptors.request 的请求，在 dispatchRequest 响应后处理了 axios.interceptors.response 的请求，达到了拦截的效果。