axios库的使用

王红元 coderwhy

实力IT教育

前端网络请求的选择一

- 目前前端中发送网络请求的方式有很多种：
- 选择一: 传统的Ajax是基于XMLHttpRequest(XHR)
- 为什么不用它呢?
  - 非常好解释, 配置和调用方式等非常混乱.
  - 编码起来看起来就非常蛋疼.
  - 所以真实开发中很少直接使用, 而是使用jQuery-Ajax
- 选择二: 在前面的学习中, 我们经常会使用jQuery-Ajax
  - 相对于传统的Ajax非常好用.
- 为什么不选择它呢？
  - jQuery整个项目太大，单纯使用ajax却要引入整个JQuery非常的不合理（采取个性化打包的方案又不能享受CDN服务）
  - 基于原生的XHR开发，XHR本身的架构不清晰，已经有了fetch的替代方案；
  - 尽管JQuery对我们前端的开发工作曾有着深远的影响，但是的确正在推出历史舞台；

前端网络请求的选择二

- 选择三: Fetch API
- 选择或者不选择它?
  - Fetch是AJAX的替换方案，基于Promise设计，很好的进行了关注分离，有很大一批人喜欢使用fetch进行项目开发；
  - 但是Fetch的缺点也很明显，首先需要明确的是Fetch是一个 low-level（底层）的API，没有帮助你封装好各种各样的功能和实现；
  - 比如发送网络请求需要自己来配置Header的Content-Type，不会默认携带cookie等；
  - 比如错误处理相对麻烦（只有网络错误才会reject，HTTP状态码404或者500不会被标记为reject）；
  - 比如不支持取消一个请求，不能查看一个请求的进度等等；
  - MDN Fetch学习地址：https://developer.mozilla.org/zh-CN/docs/Web/API/Fetch_API/Using_Fetch
- 选择四: axios
  - axios是目前前端使用非常广泛的网络请求库，包括Vue作者也是推荐在vue中使用axios；
  - 主要特点包括：在浏览器中发送 XMLHttpRequests 请求、在 node.js 中发送 http请求、支持 Promise API、拦截请求和响应、转换请求和响应数据等等；
  - axios: ajax i/o system.

Axios的基本使用

- 支持多种请求方式:
  - axios(config)
  - axios.request(config)
  - axios.get(url[, config])
  - axios.delete(url[, config])
  - axios.head(url[, config])
  - axios.post(url[, data[, config]])
  - axios.put(url[, data[, config]])
  - axios.patch(url[, data[, config]])

- 如何发送请求呢?
  - **下面的测试我都会使用httpbin.org这个网站来测试，是我个人非常喜欢的一个网站；**

axios发送请求

- axios发送请求：
  - 直接通过axios函数发送请求
  - 发送get请求
  - 发送post请求
  - 多个请求的合并
  - 使用async、await发送请求
- axios函数、get、post请求本质上都是request请求

axios的配置信息

- 1.请求配置选项
- 2.响应结构信息
- 3.全局默认配置

```
axios.defaults.baseURL = 'https://api.example.com';
axios.defaults.headers.common['Authorization'] = AUTH_TOKEN;
axios.defaults.headers.post['Content-Type'] = 'application/x-www-form-urlencoded';
```

- 4.自定义实例默认配置：

```
const instance = axios.create({
  baseURL: 'https://api.example.com'
});

instance.defaults.headers.common['Authorization'] = AUTH_TOKEN;
```

  - 优先是请求的config参数配置；
  - 其次是实例的default中的配置；
  - 最后是创建实例时的配置；

axios拦截器

- axios库有一个非常好用的特性是可以添加拦截器：
- 请求拦截器：在发送请求时，请求被拦截；
  - 发送网络请求时，在页面中添加一个loading组件作为动画；
  - 某些网络请求要求用户必须登录，可以在请求中判断是否携带了token，没有携带token直接跳转到login页面；
  - 对某些请求参数进行序列化；

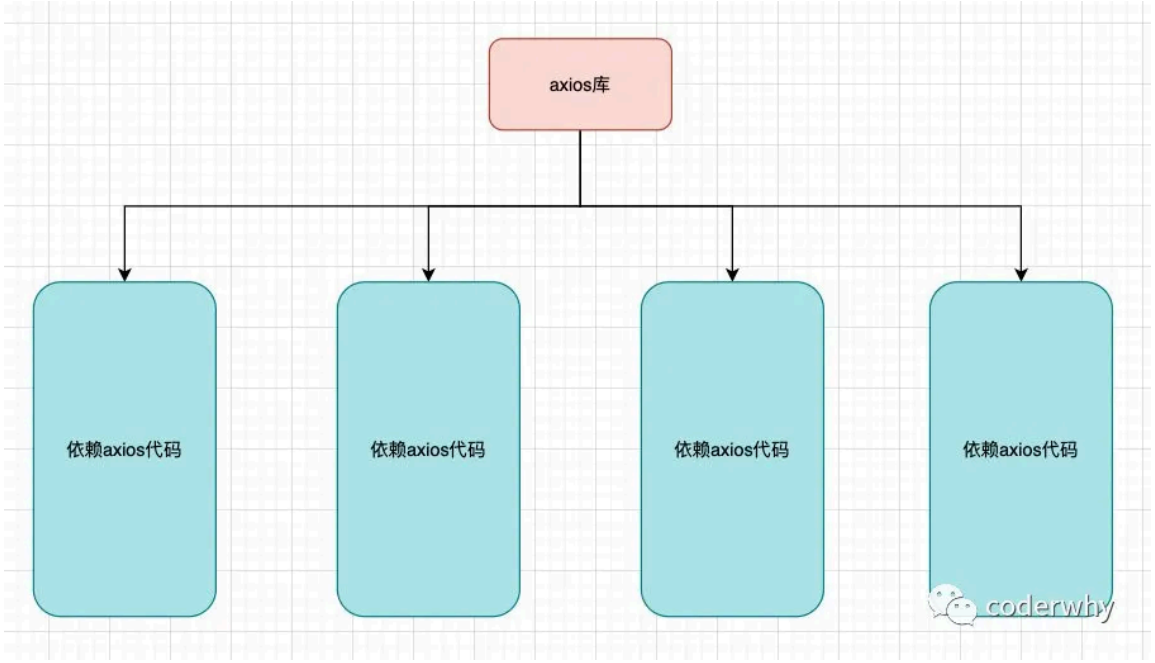
```
axios.interceptors.request.use(函调函数1, 回调函数2);
```

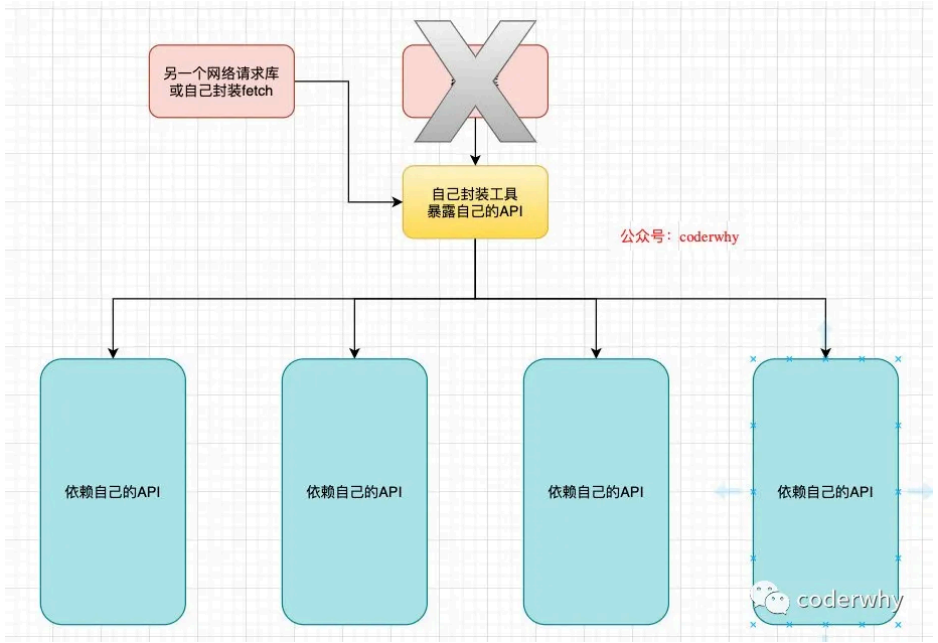
- 响应拦截器：在响应结果中，结果被拦截；
  - 响应拦截中可以对结果进行二次处理（比如服务器真正返回的数据其实是在response的data中）；
  - 对于错误信息进行判断，根据不同的状态进行不同的处理；

```
axios.interceptors.response.use(函调函数1, 回调函数2);
```

为什么要二次封装？

- 为什么我们要对axios进行二次封装呢？
  - 默认情况下我们是可以直接使用axios来进行开发的；
  - 但是我们考虑一个问题，假如有100多处中都直接依赖axios，突然间有一天axios出现了重大bug，并且该库已经不再维护，这个时候你如何处理呢？
  - 大多数情况下我们会寻找一个新的网络请求库或者自己进行二次封装；
  - 但是有100多处都依赖了axios，方便我们进行修改吗？我们所有依赖axios库的地方都需要进行修改；





axios二次封装

```
const devBaseURL = "https://httpbin.org";
const proBaseURL = "https://production.org";
export const baseURL = process.env.NODE_ENV === 'development' ? devBaseURL: proBaseURL;
```

```
const instance = axios.create({
  timeout: TIMEOUT,
  baseURL: baseURL
})

axios.interceptors.request.use(config => {
  // 1.发送网络请求时，在页面中添加一个loading组件作为动画；
  // 2.某些网络请求要求用户必须登录，可以在请求中判断是否携带了token，
  // 3.对某些请求参数进行序列化；
  return config;
}, err => {
  return err;
})
```

```
instance.interceptors.response.use(response => {
  return response.data;
}, err => {
  if (err && err.response) {
    switch (err.response.status) {
      case 400:
        err.message = "请求错误";
        break;
      case 401:
        err.message = "未授权访问";
        break;
    }
  }
  return err;
})

export default instance;
```