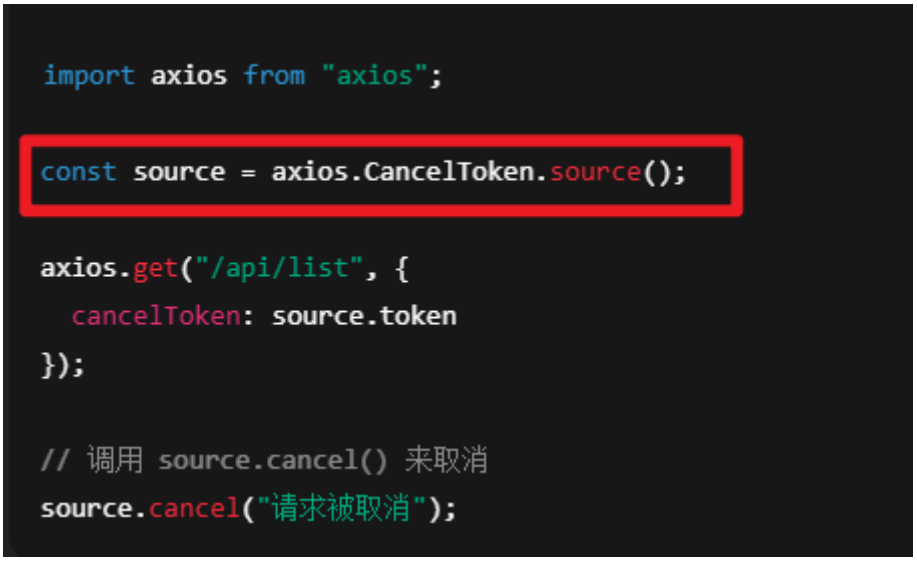
取消请求

Axios的取消请求机制有两版本

cancelToken机制

CancelToken 基本用法



核心：`CancelToken.source()` 生成一个 `token`（给请求配置）和一个 `cancel`（外部触发取消）。

给Axios的config设置一个cancelToken

是的，`{ cancelToken: source.token }` 是 Axios 的配置对象（config）。

当你使用 Axios 发送请求时，你可以传递一个配置对象作为参数，这个对象允许你控制请求的各种行为，比如设置请求头、传递参数，或者像你代码中那样，添加一个 `cancelToken`。

`cancelToken` 的作用就是将这个请求和一个“取消源”（`source`）关联起来。当你调用 `source.cancel()` 方法时，Axios 就会检查所有正在进行的、并且使用了这个特定 token 的请求，然后取消它们。

这在很多场景下非常有用，比如当用户在请求完成前切换页面，或者在搜索框中快速输入时，你可以取消旧的请求，只保留最新的请求，以避免不必要的网络开销。

在封装的Axios文件里面可以这样做

```
import axios from "axios";

const instance = axios.create({
  baseURL: "/backend",
  timeout: 5000,
});

// 封装 API
function getList(params, cancelExecutor) {
  return instance.get("/url", {
    params,
    cancelToken: new axios.CancelToken(cancelExecutor)
  });
}

export default {
  getList
};
```

`new axios.CancelToken(cancelExecutor)` 是 Axios 中用于创建**取消令牌（CancelToken）**的语法。**它允许你通过一个函数来捕获并使用取消请求的方法。**

它的工作原理

1. **创建令牌:** 当 `new axios.CancelToken(cancelExecutor)` 被调用时，Axios 会创建一个新的取消令牌。
2. **执行函数:** Axios 立即调用你提供的 `cancelExecutor` 函数。
3. **传递取消函数:** 在这个执行函数内部，Axios 会把一个可以**取消请求的函数**作为参数传递给你。
4. **捕获并保存:** 你的代码可以在 `cancelExecutor` 函数内部，将这个取消函数（例如你代码中的 `c`）保存到一个外部变量中。
5. **取消请求:** 当你需要取消请求时，你只需调用你之前保存的那个函数即可。

cancelExecutor是一个立即执行函数

Axios会 立即调用你提供的 `cancelExecutor` 函数。

然后你在这个函数里面里面，Axios会把这个cancel的取消函数作为参数传递给你

具体使用的时候

```js
import api from "./api";

let cancelFn = null;

api.getList({ page: 1 }, function executor(c) {
  cancelFn = c; // 把 cancel 函数存起来
});

// 需要时取消请求
if (cancelFn) {
  cancelFn("手动取消 getList 请求");
}
```

进阶

进阶版可以让API函数自动封装取消函数

```js
function getList(params) {
  let cancelFn;
  const request = instance.get("/url", {
    params,
    cancelToken: new axios.CancelToken(c => {
      cancelFn = c;
    })
  });
  return { request, cancel: cancelFn };
}
```

使用时：

```js
const { request, cancel } = api.getList({ page: 1 });

request.then(res => {
  console.log("数据：", res.data);
});

// 取消请求
cancel("用户切换页面，取消请求");
```

CancelToken 已废弃，如果你用的是 `axios@0.22+`，建议换成 `AbortController`。

但如果你的项目里还在用老代码，`CancelToken` 这样封装完全没问题。

如果要做成全局管理（比如一个请求池，页面切换时批量取消），建议把 `cancelFn` 存到一个 Map 里，key 可以用请求的唯一标识。

CancelToken 的缺点

1. 不是标准 API

- `CancelToken` 只是 Axios 自己实现的一套取消机制，脱离 Axios 你没法用。
- 不能像 `AbortController` 一样通用于 `fetch`、`XMLHttpRequest` 或其他库。

2. 维护成本高

- Axios 团队要自己维护一套 `CancelToken` 实现，实际上和社区标准 `AbortController` 重复。
- 这增加了库的复杂度，所以后来直接弃用了，推荐走标准。

3. 容易内存泄漏

- 如果创建了 `CancelToken` 但没有调用 `cancel()`，内部的 **回调引用会一直存在。**
- 在请求很多的场景（比如频繁切换搜索框的关键词请求），这可能导致内存堆积。

4. 语法笨重

- `CancelToken` 必须写成 `new axios.CancelToken(executor)`，需要显式传入 `c => {}`，使用不直观。
- 相比之下，`AbortController` 的写法要简洁得多：

```js
const controller = new AbortController();
axios.get('/url', { signal: controller.signal });
controller.abort();
```

5. 功能单一

- `CancelToken` 只能"取消请求"，没法像 `AbortController` 那样附带更多语义（如传递给多个异步操作）。
- `AbortController` 是事件驱动的，可以同时影响多个并发任务。

总结

- `CancelToken` 的最大问题是：**不标准** + **容易造成内存泄漏** + **语法繁琐。**
- 所以从 **Axios v0.22+** 开始，它就被 **标记为废弃**，官方推荐完全用 `AbortController`。

基于Axios封装

```
import axios from 'axios';

//创建实例
const API=axios.create({
    baseURL:'http://localhost:5000/api',
    timeout:10000,
    cors:true,
    withCredentials:true,

    validateStatus:(status)=>{
      if(status>400){
        console.error('请求失败')
      }
        return status>=200 && status<500
    }
})

//存储需求函数
let cancel;

//请求拦截器
API.interecpotors.request.use(

  (config)=>{
    config.cancelToken= new axios.CancelToken((c)=>{
      cancel=c
    })
    return config
  },
  //请求失败 参数错误 直接失败
  error=>{
  return Promise.reject(error)
})


//响应拦截器
API.interceptors.response.use(

  response=>{return response},

  error=>{
  //响应失败
  return Promise.reject(error)
  }

)
```

```
//导出暴露这个API实例 和cancel取消函数
export {API,cancel}
```

这个是一个最简单的基于cancel封装的取消机制

使用的时候，只支持手动取消

```
import API, { cancelRequest } from './api';

// 发起请求
API.get('/users')
  .then(res => console.log(res))
  .catch(err => console.error('错误:', err.message));

// 手动取消请求
setTimeout(() => {
  cancelRequest('我点了取消');
}, 1000);
```

Axios封装进阶版本

（支持自动取消重复请求）

```
import axios from "axios";

export const API=axios.create({


    baseURL:'http://localhost:5000/api',
    timeout:10000,
    cors:true,
    withCredentials:true,

    validateStatus:(status)=>{
      if(status>400){
        console.error('请求失败')
      }

      return status>=200 && status<500
    }
})

//用map来存储取消取消函数 ， key是请求的url+method
const pendingRequests=new Map()

//生成唯一的请求key
const getPendingKey=(config)=>{

  return `${config.url}-${config.method}`
}

//请求拦截器
API.interceptors.request.use(

  (config)=>{
    //生成唯一的取消请求key
    const key=getPendingKey(config)

    config.cancelToken=new axios.CancelToken((c)=>{
      if(pendingReuqests.has(key)){
        //如果已经有这个请求了，重复请求取消掉
        const cancel=pendingRequests.get(key)
        cancel('取消重复请求')
      }else{
        //没有就存储这个请求的取消函数
        pendingRequests.set(key,c)
      }
```

```
    })
    return config
  },

  //错误处理
  (error)=>{
    return Promise.reject(error)
  }

)

//响应拦截器
API.interceptors.response.use(
  response=>{
    //响应回来后删除这个请求
    const key=getPendingKey(response.config)
    pendingRequests.delete(key)

    //正常返回数据
    return response
  },

  //响应失败
  error=>{
  return Promise.reject(error)
  }

)


//取消对应请求
export const cancelRequest=(url,method)=>{
  const key=`${url}-${method}`
  if(pendingRequests.has(key)){
    const cancel=pendingRequests.get(key)
    cancel('手动取消请求')
    pendingRequests.delete(key)
  }
}

//取消全部请求
export const cancelAllRequests=()=>{
    for(const [key,cancel] of pendingRequests){
      cancel('手动取消全部请求')
```

```
    }
    //清空队列map
    pendingRequests.clear()

}

//默认导出实例
export default API
```

我们在请求拦截器里面，用一个函数，通过URL+method生成一个唯一KEY，然后把这个KEY对应的canceltoken放入map，如果已经map中has 那么就取消这个请求

响应拦截器里面，我们在response里面把已经完成的请求从队列中删除

然后我们在设置两个方法，一个全部请求取消，一个是对应key的请求取消

AbortController

`AbortController`，它就是现代浏览器和 Node.js 里的 **标准请求取消方案**，Axios 在 v0.22+ 以后也直接支持。

AbortController 是什么？

- `AbortController` 是 **Web 标准 API**，用来 **中止异步操作。**
- 它的核心思想：
  - `controller`：**控制器，能发出“中止”信号。**
  - `controller.signal`：**信号对象，传给异步任务。**
  - `controller.abort()`：**触发中止，所有用到这个 `signal` 的任务都会被终止。**

fetch中的使用

Axios中的使用

直接在 封装的API 里返回 controller

```
import axios from "axios";

function getList(params) {
  const controller = new AbortController();

  const request = axios.get("/url/api", {
    params,
    signal: controller.signal
  });

  return { request, controller };
}

export default { getList };
```

使用

```
import api from "./api";

// 发请求
const { request, controller } = api.getList({ page: 1 });

// 处理结果
request.then(res => {
  console.log("数据: ", res.data);
}).catch(err => {
  if (err.name === "CanceledError") {
    console.log("请求被取消");
  } else {
    console.error("请求出错", err);
  }
});

// 需要时取消
controller.abort();
```

或者是直接把整个AbortController传递进去

如果调用方已经有 `AbortController`，你可以在 `getList` 接口里接受它。

```js
function getList(params, controller) {
  return axios.get("/url/api", {
    params,
    signal: controller?.signal  // 可选链，避免没传时报错
  });
}
```

使用：

```js
const controller = new AbortController();

// 传 controller 进去
api.getList({ page: 1 }, controller)
  .then(res => console.log("数据：", res.data))
  .catch(err => {
    if (err.name === "CanceledError") {
      console.log("请求被取消");
    }
  });

// 取消请求
controller.abort();
```

进阶用法

多个请求同时取消

```
const controller = new AbortController();

axios.get("/api/list", { signal: controller.signal });
axios.get("/api/detail", { signal: controller.signal });

// 一次性取消两个请求
controller.abort();
```

超时自动取消

```
function fetchWithTimeout(url, ms) {
  const controller = new AbortController();
  const timeoutId = setTimeout(() => controller.abort(), ms);

  return fetch(url, { signal: controller.signal })
    .finally(() => clearTimeout(timeoutId));
}

fetchWithTimeout("/api/list", 3000)
  .catch(err => console.log("请求超时或被取消", err));
```

基于Axios封装的取消

```
import axios from "axios";

//创建实例
const API=axios.created({

  baseUrl:"http://localhost:5000/api",
  timeout:10000,
  cors:true,
  withCredentials:true,

  ValidityStateus:(status)=>{
      if(status>400){
        console.error('请求失败')
      }
      return status>=200 && status<500
  }
})

//用map来存储取消函数，key是请求的url+method
const pendingRequests=new Map()

//生成唯一的请求key
const getPendingKey=(config)=>{
  return `${config.url}-${config.method}`
}


//请求拦截器
API.interceptores.request.use(
  config=>{
    //生成唯一的请求key
    const key=getPendingKey(config)
    //设置请求函数
    const controller =new AbortController()

    config.signal=controller.signal

    //如果已经有这个请求了，重复请求取消掉
    if(pendingRequests.has(key)){
      const cancel=pendingRequests.get(key)
      cancel('取消重复请求')
    }else{
      //没有就存储这个请求的取消函数
      pendingRequests.set(key,controller.abort)
```

```
    }

    return config

  },

  error=>{
    return Promise.reject(error)
  }

)

//响应拦截器
API.interceptores.response.use(
  response=>{
    //响应回来之后删除这个请求

    //先计算key
    const key=getPendingKey(response.config)

    pendingRequests.delete(key)

    //正常返回数据
    return response
  },

  error=>{
    return Promise.reject(error)
  }
)

//取消单个请求
export const cancelRequest=(url,method)=>{
  const key=`${url}-${method}`
  if(pendingRequests.has(key)){
    const cancel=pendingRequests.get(key)
    cancel('手动取消请求')
    pendingRequests.delete(key)
  }
}

//取消全部队列的请求
export const cancelAllRequests=()=>{
```

```
  for( const [key,cancel] of pendingRequests){
    cancel('手动取消全部请求')
  }
  //清空队列map
    pendingRequests.clear()
}

export default API
```

拦截器

`axios.isCancel(error)` 其实是 **Axios 内置的工具方法**，专门用来判断当前 `error` 是否是由 **取消请求**导致的。

这是 **配合 axios 自带的 CancelToken**（老版本用法）时才完全生效。

你现在是用 **AbortController** 的方式，**这时候 `axios.isCancel(error)` 有时候不会返回 `true`，**因为 Axios 直接把浏览器的 **`DOMException` 或 `CanceledError` 往外抛了。**

对，你说的完全对 😄

- `axios.isCancel(error)` **只能判断旧的 CancelToken** 取消请求。
- **AbortController** 取消请求时，Axios 会抛出一个 `CanceledError`（Axios 1.x+ 用的）或者浏览器的 `DOMException`（name 是 `"AbortError"`）。

所以如果你用 AbortController，就不能依赖 `axios.isCancel`，而是需要判断 **error 对象的属性。**

```js
if (
  axios.isCancel(error) ||              // CancelToken 取消
  error.name === "CanceledError" ||     // Axios 1.x AbortController
  error.name === "AbortError" ||        // 浏览器原生 AbortController
  error.message.includes("取消请求")    // 你自定义 cancel 的 message
) {
  console.warn("请求被手动取消: ", error.message)
  return Promise.reject({ canceled: true, message: error.message })
}
```

复制代码

解释:

1. `CanceledError` → Axios 内部包装的 AbortController 错误（1.x+）
2. `AbortError` → 浏览器原生 AbortController 触发的错误
3. `axios.isCancel(error)` → 旧 CancelToken
4. `error.message.includes("取消请求")` → 你手动传入的字符串