前端请求库

AJAX

AJAX 是 **Asynchronous JavaScript and XML** 的缩写，它并不是一门新的技术，而是一种**技术思想或方法论。**

它指的是利用 JavaScript 和一些现有技术（例如 **XMLHttpRequest** 对象）**在不重新加载整个页面的情况下，与服务器进行异步通信。**

在 AJAX 出现之前，网页是“同步”的。

如果您想获取服务器上的新数据（比如更新一条新闻、提交一个表单），**浏览器会刷新整个页面。**

这不仅用户体验差（页面会闪烁或短暂空白），而且效率很低，因为您需要重新加载所有资源（HTML、CSS、JS 等）。

AJAX 的核心思想就是：**“局部刷新”。**它使得我们能够：

1.通过 **JavaScript 在后台向服务器发送请求。**

2.在**不中断用户当前操作**的情况下，**接收服务器返回的数据。**

3.**用 JavaScript 操作 DOM（Document Object Model），将新数据动态地插入到页面中，只更新需要变化的那部分内容。**

简而言之，**AJAX 就是让网页变得更像一个桌面应用程序**，用户可以流畅地进行交互，而不需要等待页面刷新。

XMLHttpRequest对象

XMLHttpRequest（简称 XHR）是 AJAX 的关键

是一个内置在浏览器中的 JavaScript 对象，**专门用来在后台与服务器进行通信。**

基本工作流程

1. **创建实例：** 实例化 `XMLHttpRequest` 对象。
2. **监听事件：** 注册事件监听器来处理请求过程中的不同状态。
3. **配置请求：** 使用 `open()` 方法配置请求的类型、URL、是否异步等。
4. **发送请求：** 使用 `send()` 方法发送请求。
5. **处理响应：** 在事件监听器中处理服务器返回的响应。

1.创建对象

```
const xhr = new XMLHttpRequest();
```

2.配置请求

在发送请求之前，您需要使用 **`open()` 方法**来配置请求的细节。

语法： `xhr.open(method, url, [async], [user], [password])`

```
let xhr = new XMLHttpRequest();
xhr.open(method, url, async, user, password);
```

参数解释

method请求类型

method（必需）：请求的类型，比如：

- `"GET"`：从服务器获取数据（像下载文件）。
- `"POST"`：向服务器发送数据（像提交表单）。
- 其他类型：`"PUT"`、`"DELETE"` 等（不常用）。

url请求的地址

- 一个 XML 文件的路径：`"data.xml"`
- 一个服务器接口：`"https://example.com/api"`

async（可选，布尔值）：请求是否异步，默认是 true。

- `true`：异步请求，代码不会停下来等数据返回（推荐）。
- `false`：同步请求，代码会暂停直到请求完成（不常用，可能卡住页面）。

user（可选）：如果服务器需要认证，填用户名。

password（可选）：如果服务器需要认证，填密码。

```
// 创建 XMLHttpRequest 对象
let xhr = new XMLHttpRequest();

// 初始化请求：用 GET 方法获取 data.xml，异步请求
xhr.open("GET", "data.xml", true);

// 发送请求
xhr.send();
```

3.设置请求头

xhr.setRequestHeader()

如果您**需要发送 POST 请求或设置自定义请求头，可以使用** `setRequestHeader()` **方法。**

语法： `xhr.setRequestHeader(header, value)`

`header`：请求头的名称，如 `'Content-Type'`。

`value`：请求头的值。

```
JavaScript

// 设置请求头，告诉服务器发送的数据是 JSON 格式
xhr.setRequestHeader('Content-Type', 'application/json');
```

4.发送请求

xhr.send(body)

一旦配置完成，**就可以调用 `send()` 方法发送请求了。**

对于 **GET** 请求，`send()` 方法不接受任何参数，因为数据会附加在 URL 上。

```
xhr.send();
```

对于 **POST** 或 **PUT** 请求，**`send()` 方法可以传入请求体（如 JSON 字符串、表单数据等）。**

```
const data = { name: 'John', age: 30 };
xhr.send(JSON.stringify(data));
```

5.监听事件和处理响应

这是 XHR 最复杂的部分。**XHR 有一个 `readyState` 属性，它会随着请求的进展而变化，从 0 到 4。**

我们可以通过监听 `onreadystatechange` 事件来跟踪这个状态。

`xhr.readyState` 的值：

值	状态	描述
0	UNSENT	客户端已创建 XMLHttpRequest 对象，但尚未调用 open() 方法。
1	OPENED	已经调用 open() 方法，但尚未调用 send() 方法。
2	HEADERS_RECEIVED	send() 方法已经被调用，并且已经收到服务器的响应头。
3	LOADING	正在下载响应体。此时 responseText 属性可能包含部分数据。
4	DONE	请求已完成，并且响应数据已经完全接收。

在 onreadystatechange 事件处理函数中，**我们主要关注 readyState 值为 4 的情况，因为这表示请求已经完成。**

一个完整的流程

```
// 1. 创建请求对象
const xhr = new XMLHttpRequest()

// 2. 配置请求
xhr.open('GET', 'https://jsonplaceholder.typicode.com/posts/1', true)

// 3. 监听状态变化
xhr.onreadystatechange = function () {
  if (xhr.readyState === 4 && xhr.status === 200) {
    console.log('响应结果:', xhr.responseText)
  }
}

// 4. 发送请求
xhr.send()
```

关键点：

- `readyState`：表示请求的状态，`4` 意味着请求完成了。
- `status`：HTTP 状态码，`200` 表示成功。
- `responseXML`：如果是 XML 数据，服务器会返回解析好的 XML 文档对象。

6.额外API

onload、onerror 和 onprogress 这三个事件。这些事件是 **XMLHttpRequest 用来处理请求的不同阶段的，特别适合小白理解 HTTP 请求的生命周期。**

onload 事件

什么是 onload?

onload 事件在 **请求成功完成** 时触发。具体来说：

- XMLHttpRequest 的 **readyState 达到 4（请求完成，数据已接收）**。
- HTTP 状态码是 **2xx**（通常是 200，表示成功）

你可以用 **onload 来处理服务器返回的数据，替代传统的 onreadystatechange（更简洁）**。

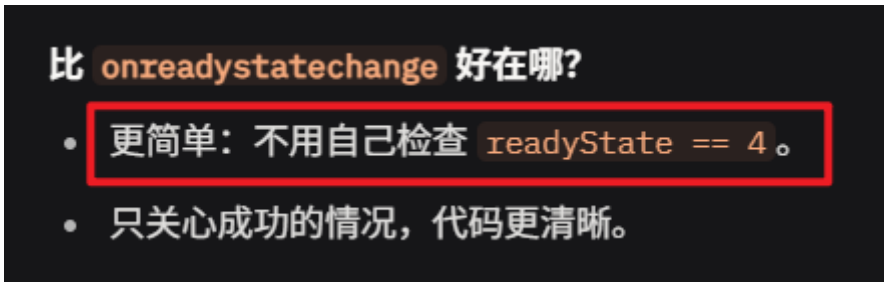
```
let xhr = new XMLHttpRequest();
xhr.open("GET", "data.xml", true);

// 当请求成功完成时触发
xhr.onload = function () {
  if (xhr.status >= 200 && xhr.status < 300) { // 2xx 状态码
    let xmlData = xhr.responseXML; // 获取 XML 数据
    console.log(xmlData); // 打印 XML 数据
  }
};

// 发送请求
xhr.send();
```

关键点：

- `onload` 只在请求成功（2xx 状态码）时触发。
- 你还是需要检查 `xhr.status` 来确认是否真的成功（比如 `200`）。
- 如果是 XML 数据，用 `xhr.responseXML` 获取解析好的 XML 文档对象。



onerror 事件

什么是 onerror?

onerror 事件在 **请求失败** 时触发，比如：

- 网络断开或不稳定。
- 服务器地址错误（比如 URL 写错了）。
- 跨域请求被浏览器阻止（CORS 问题）。

用 onerror 来捕获错误并提示用户或记录问题。

```
let xhr = new XMLHttpRequest();
xhr.open("GET", "wrong-url.xml", true);

// 当请求失败时触发
xhr.onerror = function () {
  console.log("请求失败了！可能是网络问题或文件不存在。");
};

// 发送请求
xhr.send();
```

onerror 不提供具体的错误原因（比如是 404 还是网络断开），需要额外检查。

如果需要具体错误信息，可以结合 **xhr.status 或其他方式（但 onerror 里 status 可能不可靠）。**

常用于用户提示或日志记录。

onprogress事件

什么是 onprogress?

onprogress 事件在 **下载响应数据时周期性触发**，可以用来监控数据传输的进度。通常用于：

- **显示下载进度条（比如加载大文件时）。**
- 跟踪数据接收的字节数。

onprogress 事件会提供一个 event 对象，包含：

- event.loaded：已下载的字节数。
- event.total：总字节数（如果服务器提供）。
- event.lengthComputable：是否能计算进度（true 表示 total 有效）。

```
let xhr = new XMLHttpRequest();
xhr.open("GET", "large-data.xml", true);

// 监控下载进度
xhr.onprogress = function (event) {
  if (event.lengthComputable) {
    let percent = (event.loaded / event.total) * 100;
    console.log(`已下载: ${percent.toFixed(2)}%`);
  } else {
    console.log(`已下载 ${event.loaded} 字节，总体大小未知`);
  }
};

// 请求成功时
xhr.onload = function () {
  if (xhr.status === 200) {
    console.log("下载完成！", xhr.responseXML);
  }
};

// 发送请求
xhr.send();
```

event.total 不一定总有值（服务器可能不提供文件大小）。

如果 event.lengthComputable 是 false，就无法算百分比，只能知道已下载的字节数。

适合大文件下载或需要进度条的场景。

onprogress 只适用于下载响应数据（xhr.response），不适用于上传数据。

完整示例

```
let xhr = new XMLHttpRequest();
xhr.open("GET", "data.xml", true);

// 进度监控
xhr.onprogress = function (event) {
  if (event.lengthComputable) {
    let percent = (event.loaded / event.total) * 100;
    console.log(`下载进度：${percent.toFixed(2)}%`);
  }
};

// 请求成功
xhr.onload = function () {
  if (xhr.status === 200) {
    let xmlData = xhr.responseXML;
    console.log("成功加载 XML：", xmlData);
  } else {
    console.log("请求完成但失败，状态码：" + xhr.status);
  }
};

// 请求失败
xhr.onerror = function () {
  console.log("请求失败！检查网络或 URL 是否正确。");
};

// 发送请求
xhr.send();
```

Fetch

什么是 Fetch API?

Fetch API 是一个现代的、**基于 Promise 的浏览器原生 API**，它为浏览器提供了发送 HTTP 请求的新方式。

你可以把它看作是 **XMLHttpRequest** 的升级版和替代品。

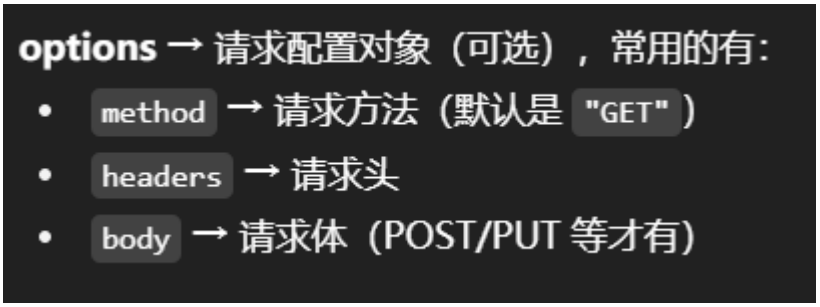
Fetch API 的核心思想是：**用更现代的 JavaScript 语法（Promise）来解决异步请求的痛点。**它使得代码更简洁、更易于阅读和维护，完美契合了现代前端开发的需求。

基本流程

语法：`fetch(url, [options])`

1.`url`: 你要请求的 URL 地址。

2.`options`: 一个可选的对象，用于配置请求，比如请求方法、请求头、请求体等。



`fetch()` **函数会立即返回一个 Promise 对象。**这个 Promise 在请求完成后会解析成一个 `Response` 对象。

```
fetch(url, options)
  .then(response => {
    // response 是一个 Response 对象
  })
  .catch(error => {
    // 网络错误（断网、跨域失败等）才会进入这里
  })
```

GET请求示例

```
fetch('https://jsonplaceholder.typicode.com/posts/1')
  .then(response => {
    if (!response.ok) { // 检查状态码
      throw new Error('HTTP 错误，状态码： ' + response.status)
    }
    return response.json() // 把响应体解析为 JSON
  })
  .then(data => {
    console.log('数据:', data)
  })
  .catch(error => {
    console.error('请求失败:', error)
  })
```

`fetch` 不会因为状态码 404 / 500 进入 `catch`，只有 **网络错误** 才会。

所以你需要手动检查 `response.ok`。

POST 请求示例

```
fetch('https://jsonplaceholder.typicode.com/posts', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json' // 声明发送的是 JSON
  },
  body: JSON.stringify({
    title: 'Hello',
    body: 'World',
    userId: 1
  })
})
  .then(res => res.json())
  .then(data => console.log('创建成功:', data))
  .catch(err => console.error('错误:', err))
```

async/await写法

```
async function getPost() {
  try {
    const res = await fetch('https://jsonplaceholder.typicode.com/posts/1')
    if (!res.ok) throw new Error('请求失败，状态码：' + res.status)

    const data = await res.json()
    console.log('数据:', data)
  } catch (err) {
    console.error('请求出错:', err)
  }
}
getPost()
```

常见配置

```
fetch('https://example.com/data', {
  method: 'GET',
  headers: {
    'Authorization': 'Bearer token123'
  },
  credentials: 'include', // 携带 cookie，默认不带
  mode: 'cors',           // 允许跨域请求（默认就是 cors）
  cache: 'no-cache',      // 禁用缓存
  redirect: 'follow',     // 自动跟随重定向
})
```

上传/下载文件

```
const formData = new FormData()
formData.append('file', fileInput.files[0])

fetch('/upload', {
  method: 'POST',
  body: formData
})
```

```
const res = await fetch('/download')
const blob = await res.blob() // 转成二进制
const url = URL.createObjectURL(blob)
const a = document.createElement('a')
a.href = url
a.download = 'file.txt'
a.click()
```

fetch 的优缺点

✅ 优点

- 语法简洁，基于 Promise，支持 `async/await`。
- 直接内置在浏览器，无需引入库。
- API 设计更现代化，语义清晰。

⚠️ 缺点

- 不会自动 reject 404/500，需要自己判断。
- 不如 XHR 好用的地方：比如**上传文件进度监听**，fetch 需要配合 `ReadableStream` 实现，比较复杂。
- IE 浏览器不支持（要用 polyfill）。

- **默认不处理 HTTP 错误：** 你必须手动检查 `response.ok` 或 `response.status` 来判断请求是否真的成功。
- **不支持请求中断：** 原生的 Fetch API 不支持中止正在进行的请求。不过，现在可以通过 `AbortController` 来弥补这个不足。
- **无法监听进度：** 不像 XHR 有 `onprogress` 事件，Fetch 无法直接监听上传或下载的进度。

Axios

Axios 是一个基于 **Promise** 的 HTTP 客户端，可以同时在浏览器和 Node.js 环境下使用。它的核心目标就是：**提供一个统一、简洁、功能强大的 API 来处理 HTTP 请求。**

为什么选择 Axios?

Axios 之所以如此受欢迎，是因为它解决了实际开发中的许多痛点，提供了以下核心优势：

- **基于 Promise：** 语法简洁，支持 `async/await`，完全摆脱了回调地狱。
- **统一的 API：** 在浏览器端使用 XHR，在 Node.js 端使用原生 http 模块，但对外提供的是一套完全相同的 API。
- **请求和响应拦截器（Interceptor）：** 这是 Axios 最强大的功能之一。你可以在请求发送前或响应返回后，统一处理数据。
- **自动转换 JSON 数据：** 默认将请求体数据自动转换为 JSON，将响应数据自动解析为 JSON。
- **客户端支持防止跨站请求伪造（CSRF/XSRF）：** 提供内置机制来保护你的应用。
- **支持取消请求：** 可以在请求发送后，通过 API 随时取消它。
- **统一的错误处理：** 任何非 2xx 的状态码（如 404、500）都会被视为错误，Promise 会进入 `catch` 块，这比 Fetch 的处理方式更符合直觉。

基本使用

安装

首先，你需要通过 npm 或 yarn 安装 Axios：

Bash

```
npm install axios
# 或者
yarn add axios
```

导入

```
// src/api/request.js
import axios from 'axios'
```

创建实例

```
import axios from 'axios'

// 创建一个实例
const api = axios.create({
  baseURL: 'https://api.example.com', // 基础路径
  timeout: 5000,                      // 超时时间
  headers: {                          // 默认请求头
    'Content-Type': 'application/json'
  }
})
```

请求拦截器

```
// 2. 请求拦截器
service.interceptors.request.use(
  config => {
    // 你可以在这里统一加 token
    const token = localStorage.getItem('token')
    if (token) {
      config.headers.Authorization = `Bearer ${toke
    }
    console.log('请求拦截:', config)
    return config
  },
  error => {
    return Promise.reject(error)
  }
)
```

响应拦截器

```
// 3. 响应拦截器
service.interceptors.response.use(
  response => {
    // 直接返回数据部分，调用时不用写 res.data
    return response.data
  },
  error => {
    console.error('响应出错:', error)
    return Promise.reject(error)
  }
)
```

导出实例

```
// 4. 导出 axios 实例
export default service
```

并发请求

```
import request from './api/request'
import axios from 'axios'

axios.all([
  request.get('/posts/1'),
  request.get('/posts/2')
]).then(
  axios.spread((res1, res2) => {
    console.log('结果1:', res1)
    console.log('结果2:', res2)
  })
)
```

取消请求

```
import request from './api/request'

// 创建控制器
const controller = new AbortController()

// 发起请求
request.get('/posts/1', { signal: controller.signal })
  .then(res => console.log('请求结果:', res))
  .catch(err => console.error('请求被取消:', err))

// 2 秒后取消请求
setTimeout(() => {
  controller.abort()
  console.log('请求已取消')
}, 2000)
```

API介绍

1.请求方法

1. 请求方法（最常用）

axios 请求方法和 HTTP 协议的方法是一一对应的:

```
axios.get(url[, config])
axios.post(url[, data[, config]])
axios.put(url[, data[, config]])
axios.delete(url[, config])
axios.patch(url[, data[, config]])
axios.head(url[, config])
axios.options(url[, config])
```

1.get

axios.get(url[, config])

作用：从服务器获取数据。

参数：

- `url`: 请求地址
- `config`: 配置对象（可选），里面可以放 headers、params 等

```
axios.get('/api/users', {
  params: { id: 123 }
})
```

2.post

axios.post(url[, data[, config]])

作用：向服务器提交数据（常用于表单提交、创建资源）。

参数：

- `url`: 请求地址
- `data`: 请求体数据（比如 JSON、FormData）
- `config`: 配置对象（可选）

```
axios.post('/api/users', {
  name: 'Tom',
  age: 18
})
```

3.put

axios.put(url[, data[, config]])

作用：更新服务器上的资源（**整体更新**，通常替换掉原有资源）。

参数：

- `url`: 请求地址
- `data`: 请求体数据
- `config`: 配置对象（可选）

```
axios.put('/api/users/123', {
  name: 'Tom',
  age: 20
})
```

4.delete

axios.delete(url[, config])

作用：删除服务器上的资源。

参数：

- `url`: 请求地址
- `config`: 配置对象（可选，可以带 `params`）

```
js

axios.delete('/api/users/123')
```

Aixos默认不允许deleted有请求体body 需要标明

⚠ 注意：有的后端支持 `DELETE` 请求带 `body`（data），但 Axios 默认不允许，需要写成：

```
js

axios.delete('/api/users', { data: { id: 123 } })
```

5.patch

axios.patch(url[, data[, config]])

作用：更新资源的 **部分字段**（区别于 `put` 的整体更新）。

参数：

- `url`: 请求地址
- `data`: 只传需要修改的字段
- `config`: 配置对象（可选）

```
axios.patch('/api/users/123', {
  age: 21
})
```

6.head

axios.head(url[, config])

作用：只请求响应头（不返回响应体）。常用于检测资源是否存在、获取资源大小等。

参数：

- `url`: 请求地址
- `config`: 配置对象（可选）

```
axios.head('/api/users')
```

常用于向后端确认请求的数据是否存在，常用于文件下载之前，请求文件的具体数据

7.options

axios.options(url[, config])

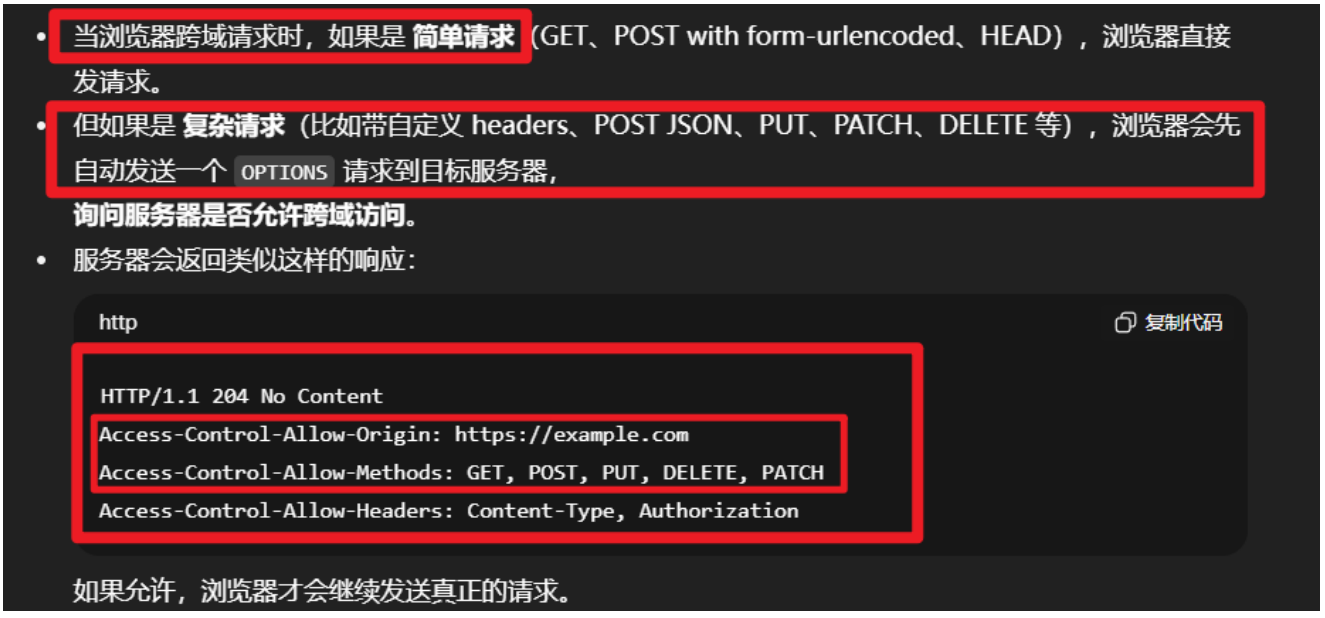
作用：**请求服务器支持的 HTTP 方法（CORS 预检请求就是 `OPTIONS`）。**

参数：

- `url`: 请求地址
- `config`: 配置对象（可选）

```
axios.options('/api/users')
```

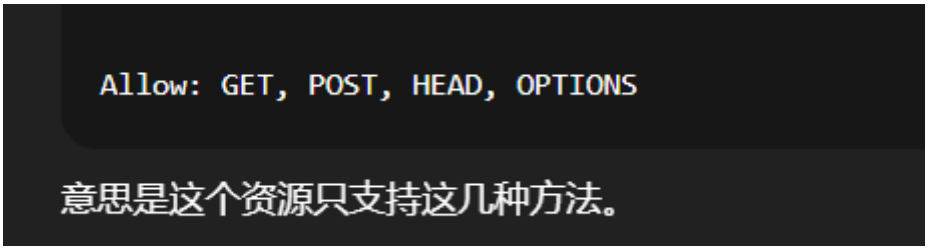
常用于CORS预检请求



✅ 所以你在**前端写 Axios 时经常会看到 先发一个 OPTIONS**，这就是浏览器在帮你做跨域检测。

以及询问服务器支持哪些HTTP方法

有时候客户端需要知道**某个资源支持什么操作**，可以手动发一个 `OPTIONS`。
服务器可能返回：

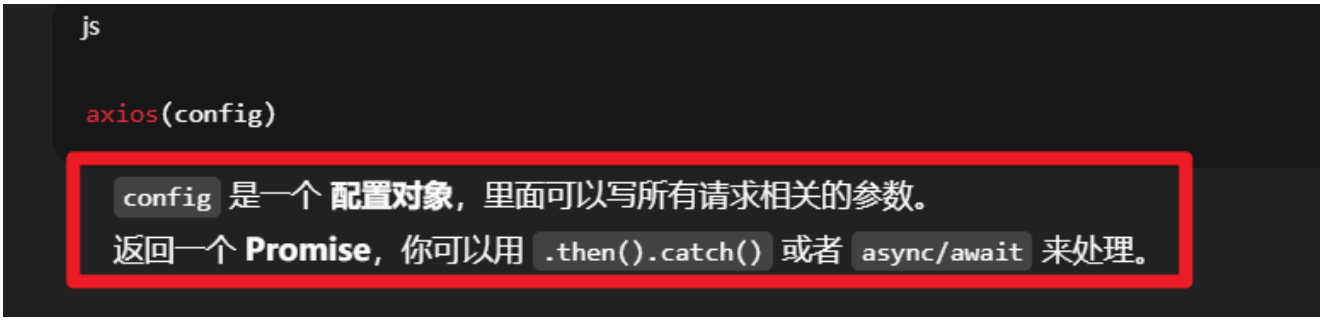


`OPTIONS` 最常见的用途是 **跨域预检请求**，由浏览器自动发起，前端开发者一般不用管。

其次是 **检查服务器支持的请求方法。**

2.axios(config)

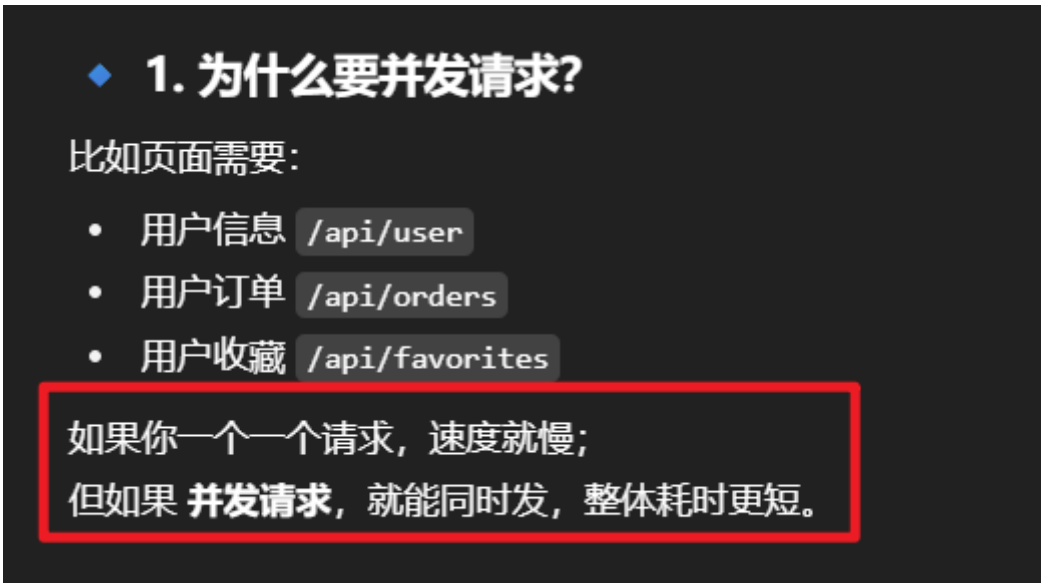
`axios(config)` 就是 **Axios 的核心调用方式**，前面那些 `axios.get` / `axios.post` 都是它的语法糖而已。



`axios(config)` 是 Axios 的核心方法，**万能写法。**

其它 **`axios.get / post / put ...`** 只是对它的封装，方便常见请求的书写。

3.axios.all并发



Axios.all实际是对Promise.all的封装

(1) axios.all()

接收一个 **数组**，并发执行里面的请求。

```js
import axios from 'axios'

axios.all([
  axios.get('/api/user'),
  axios.get('/api/orders'),
  axios.get('/api/favorites')
]).then(axios.spread((userRes, ordersRes, favoritesRes) => {
  console.log('用户信息:', userRes.data)
  console.log('订单:', ordersRes.data)
  console.log('收藏:', favoritesRes.data)
}))
```

接收一个数组，然后并发执行里面的请求

本质是对Promise的封装

(2) `Promise.all()`

其实 `axios.all` 就是对 `Promise.all` 的封装，所以直接用原生也可以：

```js
Promise.all([
  axios.get('/api/user'),
  axios.get('/api/orders'),
  axios.get('/api/favorites')
]).then(([userRes, ordersRes, favoritesRes]) => {
  console.log(userRes.data, ordersRes.data, favoritesRes.data)
})
```

同时也支持async/await写法

```
async function fetchData() {
  try {
    const [userRes, ordersRes, favoritesRes] = await Promise.all([
      axios.get('/api/user'),
      axios.get('/api/orders'),
      axios.get('/api/favorites')
    ])

    console.log('用户:', userRes.data)
    console.log('订单:', ordersRes.data)
    console.log('收藏:', favoritesRes.data)
  } catch (error) {
    console.error('请求失败:', error)
  }
}
```

◆ 4. 高阶用法：并发 + 限制数量

有时候你需要请求几十个接口，但又不能一下子全发（可能会压垮服务器）。
这时可以用 **第三方库（p-limit）** 或者自己写一个并发控制器。

4.axios.spread

`axios.spread` 是 Axios 提供的一个 **辅助工具函数**，用来配合 `axios.all()` 使用。
作用就是：
👉 **把多个请求的结果展开成单独的参数传给回调函数。**

其实就是帮你避免写一堆 `res[0]`, `res[1]`, `res[2]`, 让代码更直观。

不用 `spread` 的写法：

```js
axios.all([
  axios.get('/api/user'),
  axios.get('/api/orders')
]).then(responses => {
  console.log(responses[0].data) // 用户信息
  console.log(responses[1].data) // 订单
})
```

这里 `responses` 是一个数组，你得自己用下标取。

2. `axios.spread` 本质

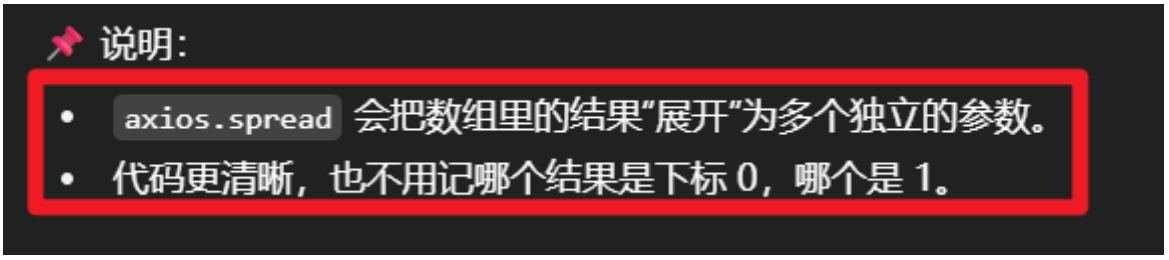
- `axios.spread` 只是个**工具函数**，用来"展开"数组，把数组里的元素依次当作参数传递给回调函数。

源码大概就是这样：

```js
axios.spread = function(callback) {
  return function wrap(arr) {
    return callback.apply(null, arr)
    // 等价于 callback(arr[0], arr[1], arr[2], ...)
  }
}
```

then 接收的本来是数组（[userRes, ordersRes]）

spread 把这个数组“展开”成了参数（(userRes, ordersRes)）。



因为 Promise.all 支持解构，所以根本用不到 spread：

5.取消请求

Axios 的 **取消请求机制** 是一个比较实用的功能，**特别是在防抖搜索、切换路由时取消无效请求、避免重复提交的场景**里很常见。

Axios 一共经历了 **两代取消机制**

1.老版（CancelToken）机制

CancelToken 的原理是 Axios 在发送请求前，保存一个取消函数。

取消时触发函数，Axios 会在内部中止请求，并让返回的 Promise 进入 catch 分支。

创建一个 CancelToken，并把取消请求的函数 c 暴露给外部变量 cancel，方便在需要的时候手动触发请求取消。

2. AbortController

AbortController 是浏览器和 Node.js 提供的一个原生 API，用来中止异步任务（最常见就是网络请求）。

它生成一个 signal，这个 signal 会传给请求。

以后如果**调用 `controller.abort()`，所有绑定了这个 `signal` 的请求都会被取消。**

使用

Axios 内部做了适配，你可以直接在 **`axios(config)` 或 `axios.get` 的配置里传入 `signal`。**

取消多个请求

AbortController 可以用 **一个 signal 取消多个请求**:

```js
const controller = new AbortController()

axios.get('/api/user', { signal: controller.signal })
axios.get('/api/orders', { signal: controller.signal })

// 一次取消多个请求
controller.abort()
```

一旦调用 `abort()`，signal 会进入 **终止状态**，不能恢复，**需要重新创建一个新的 `AbortController` 才能发新请求。**

在 Axios 里，如果取消，错误对象是 `CanceledError` 类型，可以判断:

```js
if (err.name === 'CanceledError') {
  console.log('请求被取消')
}
```

新版取消机制就是依赖 **AbortController.signal**，**Axios 内部监听它，一旦 `abort()`，请求就会被取消。**更标准，更通用，也能在 `fetch` 和 `axios` 之间复用。

所以每次请求都建立新的controllller

```js
function fetchData() {
  const controller = new AbortController() // 新建
  axios.get('/api/data', { signal: controller.signal })
    .then(res => console.log(res.data))
    .catch(err => {
      if (err.name === 'CanceledError') {
        console.log('请求被取消')
      } else {
        console.error(err)
      }
    })
  return controller // 返回给外部方便取消
}
```

3.对比

对比点	CancelToken（旧版）	AbortController（新版）
标准	Axios 自己实现的	浏览器/Node 原生标准
使用方式	`new CancelToken(executor)`	`new AbortController()`
多请求共享取消	可以（同一个 cancelToken）	可以（同一个 signal）
状态检测	`axios.isCancel(err)`	`err.name === "CanceledError"`
未来支持	已废弃	推荐使用

6.其他工具函数

- `axios.isCancel(error)`
  判断某个错误是不是“请求被取消”的错误。
- `axios.defaults`
  全局默认配置。

```js
axios.defaults.baseURL = 'https://api.example.com'
axios.defaults.timeout = 3000
```

- `axios.getUri(config)`
  根据配置生成最终的请求 URL（调试用）。

7.Axios.create

完整示例可以配置config选项

```
import axios from 'axios';

// 创建 axios 实例
const instance = axios.create({
  // 1️⃣ 基础配置
  baseURL: 'https://api.example.com',    // 基础请求路径
  url: '/users',                         // 默认的请求路径（通常不用）
  method: 'get',                         // 默认请求方法
  timeout: 5000,                         // 超时时间（毫秒）
  timeoutErrorMessage: '请求超时了，请重试', // 超时错误信息

  // 2️⃣ 请求参数
  params: { lang: 'zh-CN' },             // URL 查询参数，会拼接到 URL 后面
  data: { key: 'value' },                // 默认请求体（一般 POST 才用到）

  // 3️⃣ 请求头
  headers: {
    'Content-Type': 'application/json',
    'X-Custom-Header': 'foobar'
  },

  // 4️⃣ 跨域与凭证
  withCredentials: true,                 // 跨域时是否携带 cookie
  xsrfCookieName: 'XSRF-TOKEN',          // CSRF Token 的 cookie 名称
  xsrfHeaderName: 'X-XSRF-TOKEN',        // CSRF Token 的请求头名称

  // 5️⃣ 响应相关
  responseType: 'json',                  // 响应数据类型：json / blob / document / arraybu
  responseEncoding: 'utf8',              // 响应的编码（Node.js 环境下用）
  maxContentLength: 2000,                // 允许的响应内容最大字节数（超出报错）
  maxBodyLength: 2000,                   // 允许的请求体最大字节数

  // 6️⃣ 重定向（Node.js 环境）
  maxRedirects: 5,                       // 最大重定向次数
  decompress: true,                      // 是否自动解压 gzip/deflate 响应（默认 true）

  // 7️⃣ 代理设置
  proxy: {
    protocol: 'http',
    host: '127.0.0.1',
    port: 9000,
    auth: {
      username: 'user',
      password: 'pass'
    }
```

```
  },

  // 8 认证
  auth: {                              // HTTP 基本认证
    username: 'pinkQQx',
    password: '123456'
  },

  // 9 高级功能
  paramsSerializer: params => {        // 自定义参数序列化方法
    return new URLSearchParams(params).toString();
  },
  transitional: {                      // 兼容旧版本 axios 的配置
    silentJSONParsing: true,
    forcedJSONParsing: true,
    clarifyTimeoutError: false
  },
  signal: new AbortController().signal, // 通过 AbortController 取消请求
  onUploadProgress: progressEvent => {  // 上传进度回调（浏览器环境）
    console.log('上传进度：', progressEvent.loaded);
  },
  onDownloadProgress: progressEvent => { // 下载进度回调（浏览器环境）
    console.log('下载进度：', progressEvent.loaded);
  },
  validateStatus: status => {          // 自定义状态码校验
    return status >= 200 && status < 400; // 只要状态码 < 400 都算成功
  }
});

// 使用实例
instance.get('/users').then(res => {
  console.log(res.data);
});
```