

AJAX-HTTP协议请求报文与响应文本结构

HTTP协议「超文本传输协议」，协议详细规定了浏览器和万维网服务器之间互相通信的规则。

请求报文

这里重点讲请求报文的格式与参数

一个请求报文包括四部分：**行、头、空行、体**

行：

包括三部分：

①**请求类型**：GET、POST、HEAD、PUT、DELETE、OPTIONS、TRACE、CONNECT，get和post
见得居多

②**url**

③**HTTP协议的版本**

1.7. 不同类型的请求及其作用

1. GET: 从服务器端读取数据
2. POST: 向服务器端添加新数据
3. PUT: 更新服务器端已经数据
4. DELETE: 删除服务器端数据

头：

头部有一些属性配置，，例如

Host:主机

Cookie:cookie数据

Content-type:内容类型，，用来规定下面**请求体内容的格式**

```
头      Host: atguigu.com
          Cookie: name=guigu
          Content-type: application/x-www-form-urlencoded
          User-Agent: chrome 83]
```

空行：必须要有一个空行

体：

可以有内容也可以没有

如果是get请求，请求体可以没有内容，如果是post请求，请求体可以不为空

请求体的格式可以是**查询体 (urlencoded， 使用query查询参数时使用，一般属性名为params，在get没有请求体的时候使用，参数最后呈现在url后面)**，也可以是**json字符串 (post传参时使用的，一般属性名为data)**

username=tom&pwd=123

{"username": "tom", "pwd": 123}

简单的示例：

```
重点是格式与参数
```
行头 POST /s?ie=utf-8 HTTP/1.1
 Host: atguigu.com
 Cookie: name=guigu
 Content-type: application/x-www-form-urlencoded
 User-Agent: chrome 83
空行
体 username=admin&password=admin
```

```

响应报文

一个响应报文同样包括四部分：**行、头、空行、体**

行：

包括三部分：

①协议版本

②响应状态码

③响应状态字符串

1.6. 常见的响应状态码

200 OK

请求成功。一般用于 GET 与 POST 请求

201 Created

已创建。成功请求并创建了新的资源

401 Unauthorized

未授权/请求要求用户的身份认证

404 Not Found

服务器无法根据客户端的请求找到资源

500 Internal Server Error

服务器内部错误，无法完成请求

头：

跟请求报文一样有一些属性，对响应报文做一些描述

Content-type:**内容类型**，规定响应体内容的格式

Set-Cookie:**服务器携带cookie到客户端**，通过响应头中的Set-Cookie来传参

```
头 Content-Type: text/html;charset=utf-8  
Content-length: 2048  
Content-encoding: gzip
```

空行: 也是必须得有

体: 响应体就是我们请求后的主要的返回结果

html 文本/json 文本/js/css/图片..各类内容

简单的示例

```
## 响应报文  
```\n行      HTTP/1.1 200 OK\n头      Content-Type: text/html;charset=utf-8\n        Content-length: 2048\n        Content-encoding: gzip\n空行\n体      <html>\n          <head>\n          </head>\n          <body>\n            <h1>尚硅谷</h1>\n          </body>\n        </html>
```

## 前后端API分类

API 的分类

REST API: restful

(1) 发送请求进行 CRUD (增删改查) 哪个操作由请求方式来决定

(2) 同一个请求路径可以进行多个操作

(3) 请求方式会用到 GET/POST/PUT/DELETE

非 REST API: restless

(1) 请求方式不决定请求的 CRUD 操作

(2) 一个请求路径只对应一个操作

(3) 一般只有 GET/POST

# Chrome浏览器网络控制台查看通信报文

在当前的页面打开控制台，打开network，刷新页面后，会看到下面的一些请求信息

注意这个例子是一个get请求

The screenshot shows the Chrome browser's Network tab in the developer tools. A search query for "谷粒学院" has been entered into the search bar. The results page from Baidu is displayed, showing various educational institutions like Tsinghua University and Peking University. The Network tab lists 35 requests. The first request, which is a GET request to the search URL, is highlighted with a yellow box. This request is labeled 'headers' in the question. Other requests listed include image files (bd\_logo1.png, result@2.png) and CSS files (iconfont-8db5f471f4.woff2).

点击左边第一个请求内容，**headers**代表的是头部信息

This screenshot shows the same setup as the previous one, but the Headers tab is selected in the Network tab's filter bar. A red arrow points to the 'Headers' tab, and another red arrow points to the 'General' section of the expanded Headers panel. This indicates that the user is examining the response headers for the first request.

打开**请求头**

点击可以查看请求行的内容

Name  
s?ie=utf-8&f=8&rsv\_bp=1&rsv\_idx=1&tn=baidu  
bd\_logo1.png  
result.png  
result@2.png  
iconfont-8db5f471f4.woff2  
icons\_441e82f.png  
image?imglist=2714642870\_204433563\_58,3087  
u=899652145,172585467&fm=85&app=79&f=j  
u=3918494008,3866563057&fm=85&app=92&f=  
img?pacompress=&imgtype=0&sec=14396196  
u=3910140465,82781925&fm=85&app=92&f=j  
ea7e0c7af4673ed4cd13dc1c2b27c1eb\_15629139  
img?pacompress=&imgtype=0&sec=14396196  
itqt=spg&c=300&coord=E|13267313.00,29891.  
019461ad292b36094ecae0789e85469\_1529549  
refresh.htm  
Host: www.baidu.com  
35 / 47 requests | 88.2 kB / 88.2 kB transferred | 1.

Headers  
Request Headers  
view source  
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,\*/\*;q=0.8,application/signed-exchange;v=b3;q=0.9  
Accept-Encoding: gzip, deflate, br  
Accept-Language: zh-CN,zh;q=0.9,en;q=0.8,la;q=0.7  
Cache-Control: max-age=0  
Connection: keep-alive  
Cookie: BD\_UPSID=B34B92B6AB4C4FA34BC8D6DC08704D; PSTM=159155979; H\_WISE\_SID=148078\_149390\_150077\_147089\_14174  
8\_150086\_148193\_148867\_150798\_150744\_147279\_150039\_150166\_149587\_149540\_149812\_148754\_147888\_148238\_148523\_150715\_127960\_149571\_1  
49907\_146551\_148616\_149718\_146732\_138425\_143667\_131423\_128699\_149007\_147528\_145601\_107311\_148186\_148933\_149251\_146395\_144966\_149278\_1503  
41\_147546\_148869\_150312\_110085; sug=3; sugstore=0; ORIGIN=0; BDUID=DC647F0BEABAFF5D52BCF8621E5B023:5L=0:NR=10:FG=1; P0000=84  
9085EBF6F3CD402E515022BCD0A1598; delPer=0; BD\_CK\_SAM=1; PSINO=2; BD\_HOME=1; COOKIE\_SESSION=5\_0\_9\_7\_14\_14\_1\_0\_9\_5\_0\_0\_554\_0\_2\_0\_1594953559  
\_0\_1594953557%7C9%23124355\_191\_1594425042%7C9; H\_PS\_PSSID=32100\_1435\_31672\_31253\_32046\_32230\_32117\_31708\_26350\_31639; H\_PS\_645EC=cfc02%2  
BKY10mRo0mQYHQN21qsEVgMsjY0JXZGdjo5uYfh9gwq7XYxuCA  
Host: www.baidu.com

点击**view source**, 里面有**请求行**和**请求头**, 这个例子只get请求, 所以没有请求体

请求行  
请求头

Name  
s?ie=utf-8&f=8&rsv\_bp=1&rsv\_idx=1&tn=baidu  
bd\_logo1.png  
result.png  
result@2.png  
iconfont-8db5f471f4.woff2  
icons\_441e82f.png  
image?imglist=2714642870\_204433563\_58,3087  
u=899652145,172585467&fm=85&app=79&f=j  
u=3918494008,3866563057&fm=85&app=92&f=  
img?pacompress=&imgtype=0&sec=14396196  
u=3910140465,82781925&fm=85&app=92&f=j  
ea7e0c7af4673ed4cd13dc1c2b27c1eb\_15629139  
img?pacompress=&imgtype=0&sec=14396196  
itqt=spg&c=300&coord=E|13267313.00,29891.  
019461ad292b36094ecae0789e85469\_1529549  
refresh.htm  
Host: www.baidu.com  
Connection: keep-alive  
Cache-Control: max-age=0  
Upgrade-Insecure-Requests: 1  
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/83.0.4103.116 Safari/537.36  
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,\*/\*;q=0.8,application/signed-exchange;v=b3;q=0.9  
Sec-Fetch-Site: same-origin  
Sec-Fetch-Mode: navigate  
Sec-Fetch-User: ?1  
Sec-Fetch-Dest: document  
Host: www.baidu.com  
35 / 47 requests | 88.2 kB / 88.2 kB transferred | 1.

Headers  
Request Headers  
view parsed  
GET /s?ie=utf-8&f=8&rsv\_bp=1&rsv\_idx=1&tn=baidu&wd=%E8%BD%87%E7%BD%2E%5E5%AD%64%E9%99%A2&fenlei=256&rsv\_pq=cee206f9001b4524&rsv\_t=c4f3V8R  
P66OeB1TywN3D8XFhE477NY%32F1iKNCmfXZhtuwQx2nZsh1k90e54&rqlang=cn&rsv\_enter=1&rsv\_dl=t&rsv\_sug3=30&rsv\_sug1=10&rsv\_sug7=100&rsv\_sug2=8&r  
sv\_btype=1&inputt=5536&rsv\_sug4=6239 HTTP/1.1  
Host: www.baidu.com

点击**Query String Parameters** (查询字符串参数), 这里面是对url内的参数做了一个解析, 做了一个格式化

Query String Parameters  
view source  
view URL encoded

Name  
s?ie=utf-8&f=8&rsv\_bp=1&rsv\_idx=1&tn=baidu  
bd\_logo1.png  
result.png  
result@2.png  
iconfont-8db5f471f4.woff2  
icons\_441e82f.png  
image?imglist=2714642870\_204433563\_58,3087  
u=899652145,172585467&fm=85&app=79&f=j  
u=3918494008,3866563057&fm=85&app=92&f=  
img?pacompress=&imgtype=0&sec=14396196  
u=3910140465,82781925&fm=85&app=92&f=j  
ea7e0c7af4673ed4cd13dc1c2b27c1eb\_15629139  
img?pacompress=&imgtype=0&sec=14396196  
itqt=spg&c=300&coord=E|13267313.00,29891.  
019461ad292b36094ecae0789e85469\_1529549  
refresh.htm  
Host: www.baidu.com  
Connection: keep-alive  
Cache-Control: max-age=0  
Upgrade-Insecure-Requests: 1  
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/83.0.4103.116 Safari/537.36  
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,\*/\*;q=0.8,application/signed-exchange;v=b3;q=0.9  
Sec-Fetch-Site: same-origin  
Sec-Fetch-Mode: navigate  
Sec-Fetch-User: ?1  
Sec-Fetch-Dest: document  
Host: www.baidu.com  
35 / 47 requests | 88.2 kB / 88.2 kB transferred | 1.

Headers  
Request Headers  
Query String Parameters  
view source  
view URL encoded  
ie: utf-8  
f: 8  
rsv\_bp: 1  
rsv\_idx: 1  
tn: baidu  
wd: 谷粒学院  
fenlei: 256  
rsv\_pq: cee206f9001b4524  
rsv\_t: c4f3V8RP6Go8T9ymN3D8XFhE477NYj/IikNCmfXZhtuwQx2nZsh1k90e54  
rqlang: cn  
rsv\_enter: 1

打开**响应头**, 里面都是响应头信息

The screenshot shows the Network tab in Chrome DevTools. A list of requests is on the left, and the response headers for a selected request are displayed on the right. The 'Cache-Control' header is highlighted with a blue background. Other visible headers include 'Bdpagetype: 3', 'Bdqid: 0xcdcd2f3c00190754', 'Content-Type: text/html; charset=utf-8', and 'Set-Cookie' entries.

点击**view source**查看原始的响应报文，里面有**响应行和响应头**，响应体在控制台上面导航条的第一个：response

The screenshot shows the Network tab in Chrome DevTools. A red box highlights the 'Response Headers' section. Inside this box, the status line 'HTTP/1.1 200 OK' is also highlighted. Other visible headers in the list include 'Bdpagetype: 3', 'Bdqid: 0xcdcd2f3c00190754', 'Content-Type: text/html; charset=utf-8', and 'Set-Cookie' entries.

打开**response**

The screenshot shows the Network tab in Chrome DevTools. A red box highlights the 'Response' tab. Below it, the response body content is displayed, showing HTML code for a search result page from Baidu. The content includes meta tags, a title, and a style block for the search results.

打开**preview**，这是对响应体内容解析之后的一个预览

下面举一个post请求，利用一个网站的登录过程

打开控制台的network，打开左边第一个请求的内容

在header下的**from data**中，就是**请求体**内容

再点击**view source**，就能看到请求体的原始内容，里面有登录的信息（邮箱，密码等等）

## express框架的基本使用

express简单地使用演示

使用**npm init --yes**初始化

**npm i express**来安装express框架

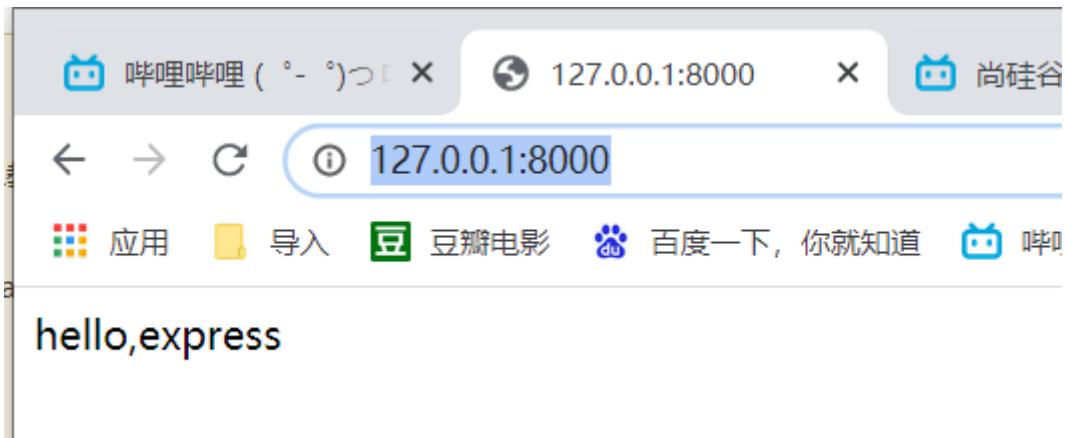
然后编写如下的js文件

```
express-test.js App.vue util.js common-list.vue Profi
1 //引入express
2 const express = require('express');
3
4 // 创建应用对象
5 const app = express();
6
7 // 创建路由规则
8 // request 是对请求报文的封装
9 // response是对请求报文的封装
10 app.get('/',(request,response)=>{
11 // 设置相应
12 response.send('hello,express');
13 })
14
15 // 监听端口启动服务
16 app.listen(8000,()=>{
17 console.log("服务已经启动， 8000端口监听中...");
18 })
```

在终端执行**node express-test.js**,结果如下

```
PS D:\ajax-test> node express-test.js
服务已经启动， 8000端口监听中...
```

打开浏览器，输入IP地址<http://127.0.0.1:8000/>， 打开



## json-server

json-server可以用来快速帮我们搭建一个http服务

在之后的axios学习过程中可以充当服务器的角色

使用json-server创建的接口是**REST API**, 可以进行增删改查

使用步骤: 1.安装 2.创建文件 3.启动服务

1.安装

**npm install -g json-server**

2.创建一个db.json文件

文件内容如下:

```
{
 "posts": [
 { "id": 1, "title": "json-server", "author": "typicode" }
],
 "comments": [
 { "id": 1, "body": "some comment", "postId": 1 }
],
 "profile": { "name": "typicode" }
}
```

上面json中的三个属性分别代表文章、评论、个人

3.启动服务

**json-server --watch db.json**

```
D:\axios-test>json-server --watch db.json
\\{^_~}/ hi!
Loading db.json
Done

Resources
http://localhost:3000/posts
http://localhost:3000/comments
http://localhost:3000/profile

Home
http://localhost:3000

Type s + enter at any time to create a snapshot of the database
Watching...
```

里面有resources资源的访问，里面有三个链接,分别可以访问上面的json里面的数据

编写一个HTML文件，来测试这个提供的rest api接口的**get、post、put、delete请求**

```
<!DOCTYPE html>
<html>
 <head>
 <meta charset="utf-8">
 <title></title>
 </head>
 <body>
 <div>
 <button onclick="testGet()">GET请求</button>
```

```

 <button onclick="testPost()">POST请求</button>
 <button onclick="testPut()">PUT请求</button>
 <button onclick="testDelete()">DELETE请求</button>
 </div>
 <script src="https://cdn.bootcss.com/axios/0.19.0/axios.js"></script>
 <script>

 function testGet(){
 // axios.get('http://localhost:3000/posts')
 // axios.get('http://localhost:3000/posts/1')//param参数
 axios.get('http://localhost:3000/posts?id=1')//query参数
 .then(response =>{
 console.log('/posts', response.data);
 })
 }

 function testPost(){
 // axios.get('http://localhost:3000/posts')
 // axios.get('http://localhost:3000/posts/1')//param参数
 axios.post('http://localhost:3000/posts',
 {"title": "json-server3", "author": "typicode3"})//post传参, 不用写
 id, id自动生成
 .then(response =>{
 console.log('/posts', response.data);
 })
 }

 // 修改参数请求
 function testPut() {
 axios.put('http://localhost:3000/posts/3',
 {"title": "json-server...", "author": "typicode..."})
 .then(response => {
 console.log('/posts put', response.data)
 })
 }

 // 删除参数请求
 function testDelete() {
 axios.delete('http://localhost:3000/posts/3')
 .then(response => {
 console.log('/posts delete', response.data)
 })
 }

 </script>
</body>
</html>

```

# XHR的理解和使用

## 概念

使用 XMLHttpRequest (XHR)对象可以与服务器交互, 也就是发送 ajax 请求

前端可以获取到数据, 而**无需让整个的页面刷新。**

这使得 Web 页面可以只更新页面的局部，而不影响用户的操作。

## 区别一般 http 请求与 ajax 请求

1. ajax 请求是一种特别的 http 请求
2. 对服务器端来说，没有任何区别，**区别在浏览器端**
3. 浏览器端发请求：**只有 XHR 或 fetch 发出的才是 ajax 请求**，其它所有的都是非 ajax 请求
4. 浏览器端接收到响应
  - (1) **一般请求**: 浏览器一般会直接显示响应体数据，也就是我们常说的刷新/跳转页面
  - (2) **ajax 请求**: 浏览器不会对界面进行任何更新操作，只是调用监视的**回调函数**并传入响应相关数据，**也就是手动的局部更新，对比一般的http请求是自动的全部更新**

## API

XMLHttpRequest(): 创建 XHR 对象的构造函数

status: 响应状态码值, 比如 200, 404

statusText: 响应状态文本, 比如404对应的文本一般为not found

readyState: 标识请求状态的只读属性

0: 初始

1: open()之后

2: send()之后

3: 请求中

4: **请求完成**，并不等于成功，还要判断状态码才行

onreadystatechange: 绑定 **readyState 改变**的监听

responseType: 指定响应数据类型，如果是'**json**'，得到响应后**自动解析响应体数据**，使用**response**去取响应体的值就是解析好的

response: 响应体数据, 类型取决于 **responseType** 的指定，如果没有使用**responseType**解析数据，还要手动解析

timeout: 指定请求超时时间, **默认为 0** 代表没有限制

ontimeout: 绑定超时的监听，是一个**回调函数**

onerror: 绑定请求网络错误的监听

open(): 初始化一个请求, 参数为: (method, url[, async])，三个参数分别为**请求方法、地址、异步/同步**（是一个布尔值，可以不写，**默认是true异步的**）

send(data): 发送请求

abort(): 中断请求

getResponseHeader(name): 获取指定名称的响应头值，（意思应该是**获得响应头某个属性的值**）

getAllResponseHeaders(): 获取所有响应头组成的字符串

setRequestHeader(name, value): 设置请求头

# XHR 的 ajax 封装(简单版 axios)

自己封装一个函数，相当于手动实现axios，因为axios也是一个函数

定义一个axios函数

函数参数是一个配置对象config

整个文件的详细代码

```
<!DOCTYPE html>
<html>
 <head>
 <meta charset="utf-8">
 <title>自定义axios函数（仿axios功能）</title>
 </head>
 <body>
 <button onclick="testGet()">发送GET请求</button>

 <button onclick="testPost()">发送POST请求</button>

 <button onclick="testPut()">发送PUT请求</button>

 <button onclick="testDelete()">发送Delete请求</button>

 </body>
 <script>

 // 点击事件返回函数
 // 1. 默认get请求，从服务器获取数据
 function testGet(){
 // 调用下面定义的axios函数
 axios({
 url:'http://localhost:3000/posts',
 // method为默认的get, params、data为空
 params: {
 id: 1,
 xxx: 'abc'
 }
 }).then(
 response =>{
 console.log(response);
 },
 error =>{
 alert(error.message)
 }
)
 }

 // 2. post请求，向服务器端添加数据
 function testPost(){
 // 调用下面定义的axios函数
 axios({
 url:'http://localhost:3000/posts',
 method: 'POST',
 data:{
 "title": "json-server...",
 "author": "typicode..."
 }
 })
 }
 </script>

```

```
 }
 }).then(
 response =>{
 console.log(response);
 },
 error =>{
 alert(error.message)
 }
)
}

// 3. put请求,向服务器端更新数据
function testPut(){
 // 调用下面定义的axios函数
 axios({
 url:'http://localhost:3000/posts/1',//准备更新服务器id为1的数据
 method:'put',//这里有可能是小写,如果是小写则需要进行一些处理,将其转化为大
写
 data:{
 "title": "json-server++",
 "author": "typicode++"
 }
 }).then(
 response =>{
 console.log(response);
 },
 error =>{
 alert(error.message)
 }
)
}

// 3. delete请求,向服务器端删除数据
function testDelete(){
 // 调用下面定义的axios函数
 axios({
 url:'http://localhost:3000/posts/2',//准备更新服务器id为2的数据
 method:'delete',//这里有可能是小写,如果是小写则需要进行一些处理,将其转化
为大写
 }).then(
 response =>{
 console.log(response);
 },
 error =>{
 alert(error.message)
 }
)
}

// 自定义axios函数
function axios({
 url,
 method='GET',//这里为什么使用等号,因为这里不是给对象中的属性赋值,而是定义函数
的时候为形参定义默认值,只不过使用了参数的结构
 params={},//get请求时,用query查询参数时使用
 data={} //post请求携带参数时使用
}){
```

```

// 返回一个Promise对象
return new Promise((resolve, reject) => {

 // 处理method(将method格式转大写)
 method = method.toUpperCase()

 // 处理query参数(拼接到url上) 将params中的参数转换成id=1&xxx=abc的
 // query形式
 let queryString = ''//query字符串
 // 进行拼接处理转换成id=1&xxx=abc的query形式
 Object.keys(params).forEach(key => {
 queryString += `${key}=${params[key]}&`
 })
 if (queryString) { // id=1&xxx=abc&
 // 去除最后的&
 queryString = queryString.substring(0, queryString.length-1)
 // 将最后的结果接到url, 但是这样有一个缺陷, 这样默认url后面是没有query
 // 参数的, 所以这里只是一个非常简单的实现
 url += '?' + queryString
 }

 // 执行异步ajax请求
 // 创建xhr对象
 const request = new XMLHttpRequest()
 // 打开连接(初始化请求, 并没有开始请求)
 request.open(method, url, true)
 // 发送请求, 这里规定delete只能发送query参数
 if(method==='GET' || method==='DELETE'){
 request.send()
 }
 else if(method === 'POST' || method==='PUT'){
 request.setRequestHeader('Content-
 type', 'application/json; charset=utf-8')//请求头设置, 告诉服务器请求体的格式是JSON
 request.send(JSON.stringify(data))//post 请求中发送JSON格式的请
 // 求体参数
 }

 // 绑定状态改变的监听, 这一步放在发送请求前面后面都可以, 放在前面合理一点, 放
 // 在后面也可以, 因为send是异步操作, 不会在请求结束才执行这个监听函数
 request.onreadystatechange = function (){

 // 如果请求没有完成, 直接结束
 if (request.readyState!=4) {
 return
 }
 // 如果响应状态码在[200, 300)之间代表成功, 否则失败
 const {status, statusText} = request
 // 如果请求成功了, 调用resolve()
 if (status>=200 && status<=299) {
 // 准备结果数据对象response
 const response = {
 data: JSON.parse(request.response),
 status,
 statusText
 }
 resolve(response)
 }
 else { //如果请求失败了, 调用reject()
 reject(new Error('request error status is ' + status))
 }
 }
}

```

```
 }
 }

})
}

</script>
</html>
```

## axios 的理解和使用

axios 是什么?

前端最流行的 ajax 请求库

react/vue 官方都推荐使用 axios 发 ajax 请求

## axios 特点

基本 promise 的异步 ajax 请求库

浏览器端/node端都可以使用

支持请求 / 响应拦截器

支持请求取消

请求/响应数据转换

批量发送多个请求

## axios 常用语法

注意下面的用法中前两个是把axios当做函数来使用，后面的是把它当做对象来使用

(可以这样理解，前面两个函数写法时最普通的写法，类型什么的用method来指定，可以指定任何请求，但是后面的一些写法为了简便，将axios改成对象来使用，不同类型的请求通过调用对象不同的属性或方法来实现)

axios(config): 通用/最本质的发任意类型请求的方式

axios(url[, config]): 可以只指定 url 发 get 请求

axios.request(config): 等同于 axios(config)

axios.get(url[, config]): 发 get 请求

axios.delete(url[, config]): 发 delete 请求

axios.post(url[, data, config]): 发 post 请求，**data中的数据如果是对象形式，默认是用的json格式**

axios.put(url[, data, config]): 发 put 请求

axios.defaults.xxx: 请求的默认全局配置

axios.interceptors.request.use(): 添加请求拦截器

axios.interceptors.response.use(): 添加响应拦截器

axios.create([config]): 创建一个新的 axios(它没有下面的功能), 也能当做函数使用, 也能当做对象使用, 但是有一些 axios 功能这个创建的实例没有

axios.CancelToken(): 用于创建取消请求的 token 对象

axios.CancelToken(): 用于创建取消请求的 token 对象

axios.isCancel(): 是否是一个取消请求的错误

axios.all(promises): 用于批量执行多个异步请求

axios.spread(): 用来指定接收所有成功数据的回调函数的方法

axios 默认是 get 请求

method 中类型大小写都行

Content-Type 值默认是 application/x-www-form-urlencoded

## 难点语法的理解和使用

### axios.create(config)

根据指定配置创建一个新的 axios

为什么要用这个语法?

这样做是为了解决一个常见的需求, 当我们有两类不同的 axios 请求, 可以使用 axios.defaults 来做默认的全局配置, 方便我们使用, 但是有两类啊, 这两类默认的全局配置不同, axios 全局配置就做不到了

这时候就需要 axios.create 创建不同的 axios, 在创建的时候设置默认的配置, 在调用的时候就再配置详细的内容

```
const instance = axios.create({
 baseURL: 'http://localhost:4000'
})

// 使用 instance 发请求
instance({
 url: '/xxx' // 请求 4000
})
```

## 拦截器函数/ajax 请求/请求的回调函数的调用顺序

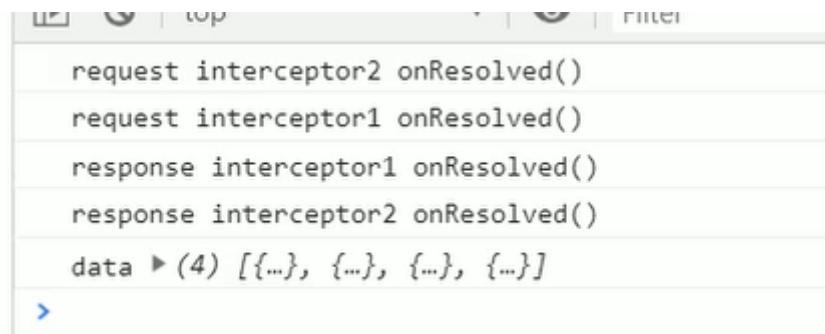
基本内容在vue中已经讲过

下面是分别定义了两个请求拦截器，两个响应拦截器，和一个发送axios完成之后的成功/失败处理

```
// 添加请求拦截器(回调函数)
axios.interceptors.request.use(
 config => {
 console.log('request interceptor1 onResolved()')
 return config
 },
 error => {
 console.log('request interceptor1 onRejected()')
 return Promise.reject(error);
 }
)
axios.interceptors.request.use(
 config => {
 console.log('request interceptor2 onResolved()')
 return config
 },
 error => {
 console.log('request interceptor2 onRejected()')
 return Promise.reject(error);
 }
)
// 添加响应拦截器
axios.interceptors.response.use(
 response => {
 console.log('response interceptor1 onResolved()')
 return response
 },
 function (error) {
 console.log('response interceptor1 onRejected()')
 return Promise.reject(error);
 }
)
axios.interceptors.response.use(
 response => {
 console.log('response interceptor2 onResolved()')
 return response
 },
 function (error) {
 console.log('response interceptor2 onRejected()')
 return Promise.reject(error);
 }
)

axios.get('http://localhost:3000/posts')
 .then(response => {
 console.log('data', response.data)
 })
 .catch(error => {
 console.log('error', error.message)
 })
```

但是最后的执行顺序如下，有一点需要注意，**请求拦截器是后定义先执行，响应拦截器是先定义先执行**，然后在执行axios的then和catch，**并且这些拦截器是一步步串起来的，每一个响应器都要返回config或者response**



The screenshot shows the Network tab of a browser's developer tools. It lists five entries:

- request interceptor2 onResolved()
- request interceptor1 onResolved()
- response interceptor1 onResolved()
- response interceptor2 onResolved()
- data ▶ (4) [ {...}, {...}, {...}, {...} ]

## 取消请求

使用**express框架**来模拟一个服务器

服务器的**server.js**文件如下，服务器里面返回参数，使用了一个延时函数，模拟请求数据的延时效果，延时时间为1~3秒

```
const express = require('express')
const cors = require('cors')

const app = express()

// 使用cors，允许跨域
app.use(cors())
// 能解析urlencoded格式的post请求体参数
app.use(express.urlencoded())
// 能解析json格式的请求体参数
app.use(express.json())

app.get('/products1', (req, res) => {
 setTimeout(() => {
 res.send([
 {id: 1, name: 'product1.1'},
 {id: 2, name: 'product1.2'},
 {id: 3, name: 'product1.3'}
])
 }, 1000 + Math.random()*2000);
})

app.get('/products2', (req, res) => {
 setTimeout(() => {
 res.send([
 {
 id: 1,
 name: 'product2.1'
 },
 {
 id: 2,
 name: 'product2.2'
 },
 {
 id: 3,
 name: 'product2.3'
 }
])
 }, 1000 + Math.random()*2000);
})
```

```

 {
 id: 3,
 name: 'product2.3'
 }
],
}, 1000 + Math.random() * 2000);

}

app.listen(4000, () => {
 console.log('server app start on port 4000')
})

```

下面是客户端页面的代码，使用axios中的**CancelToken方法**，

### 点击按钮, 取消某个正在请求中的请求

**简单流程概括：**

配置 cancelToken 对象

保存用于取消请求的 cancel 函数

在后面特定时机调用 cancel 函数取消请求

在错误回调中判断如果 error 是 cancel, 做相应处理

### 注意看里面的注释

```

<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="UTF-8">
 <meta name="viewport" content="width=device-width, initial-scale=1.0">
 <meta http-equiv="X-UA-Compatible" content="ie=edge">
 <title>取消请求</title>
</head>
<body>
 <button onclick="getProducts1()">获取商品列表1</button>

 <button onclick="getProducts2()">获取商品列表2</button>

 <button onclick="cancelReq()">取消请求</button>

 <script src="https://cdn.bootcss.com/axios/0.19.0/axios.js"></script>
 <script>
 let cancel // 用于保存取消请求的函数，正在请求的过程中将取消请求的函数赋值给这个变量，一旦请求完成（成功或失败）就将这个变量赋值null，这样完成一个效果：请求过程中可以使用这个请求取消函数，请求结束这个请求取消函数就不存在了

 function getProducts1() {
 axios({
 url: 'http://localhost:4000/products1',
 cancelToken: new axios.CancelToken((c) => { // c是用于取消当前请求的函数，是这个方法默认的一个参数
 // 保存取消函数，用于之后可能需要取消当前请求
 cancel = c
 })
 }).then(
 response => {

```

```

cancel = null
console.log('请求1成功了', response.data)
},
error => {
cancel = null
console.log('请求1失败了', error.message, error)//如果是正常请求失败，error
返回的对象类型是一个Error对象。但是我们如果在请求的过程中手动取消了，请求也算是请求失败，同样会
调用这里的回调函数，但是，这个error返回的是一个cancel对象，里面的message是我们具体调用上面的
cancel函数时传入的参数（见下面cancelReq函数中使用的cancel函数）
}

}

}

function getProducts2() {
axios({
url: 'http://localhost:4000/products2'
}).then(
response => {
cancel = null
console.log('请求2成功了', response.data)
},
error => {
cancel = null
console.log('请求2失败了', error.message)
}
)
}

function cancelReq() {
// alert('取消请求')
// 执行取消请求的函数
if (typeof cancel === 'function') {
cancel('强制取消请求')//cancel函数传入的参数可以当做终止请求的错误信息，函数会执行请
求失败的then里面的出错回调函数（默认的参数从原来的error类型变成了cancel类型），这个参数就成
了error.message(error是then出错回调函数的参数，只不过这时已经变成了cancel类型）
} else {
console.log('没有可取消的请求')
}
}
</script>
</body>
</html>

```

**另一个功能实现，再点击一个按钮发送请求时，如果上一个请求还没有结束，将未完成的请求取消掉，执行新的请求**

在上面的代码上进行修改，在准备发请求前，取消未完成的请求，并且引入拦截器的使用

```

<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">

```

```

<meta name="viewport" content="width=device-width, initial-scale=1.0">
<meta http-equiv="X-UA-Compatible" content="ie=edge">
<title>取消请求</title>
</head>
<body>
 <button onclick="getProducts1()">获取商品列表1</button>

 <button onclick="getProducts2()">获取商品列表2</button>

 <button onclick="cancelReq()">取消请求</button>

 <script src="https://cdn.bootcss.com/axios/0.19.0/axios.js"></script>
 <script>

 // 添加请求拦截器
 axios.interceptors.request.use((config) => {
 // 在准备发请求前，取消未完成的请求
 if (typeof cancel === 'function') {
 cancel('取消请求')
 }
 // 添加一个cancelToken的配置，这里的config就是axios函数中的配置
 config.cancelToken = new axios.CancelToken((c) => { // c是用于取消当前请求的函数
 // 保存取消函数，用于之后可能需要取消当前请求
 cancel = c
 })
 return config
 })

 // 添加响应拦截器
 axios.interceptors.response.use(
 response => {
 cancel = null
 return response
 },
 error => {
 if (axios.isCancel(error)) { // 取消请求的错误，isCancel方法是用来判断error是不是cancel类型
 // cancel = null // 上一个例子加了这个，这里不要加，加了之后每次请求被取消，紧接着发送新的请求，但是旧的请求还未真正结束，因为这里是异步操作，旧的请求将cancel置空，就相当于把第二个请求的cancel置空，第二个请求就无法取消了
 console.log('请求取消的错误', error.message) // 做相应处理
 // 中断promise链接，这里axios里的then后的错误处理就没有用了，这样可以使axios里的then后的错误处理专注于处理非终止的错误
 return new Promise(() => {})
 } else { // 请求出错了
 cancel = null
 // 将错误向下传递
 // throw error
 return Promise.reject(error)
 }
 }
)

```

```

let cancel // 用于保存取消请求的函数
function getProducts1() {
 axios({
 url: 'http://localhost:4000/products1',
 }).then(

```

```
 response => {
 console.log('请求1成功了', response.data)
 },
 error => { // 只用处理请求失败的情况，取消请求这种错误就不用处理
 console.log('请求1失败了', error.message)
 }
)
 }
}

function getProducts2() {

 axios({
 url: 'http://localhost:4000/products2',
 }).then(
 response => {
 console.log('请求2成功了', response.data)
 },
 error => {
 console.log('请求2失败了', error.message)
 }
)
}

function cancelReq() {
 // alert('取消请求')
 // 执行取消请求的函数
 if (typeof cancel === 'function') {
 cancel('强制取消请求')
 } else {
 console.log('没有可取消的请求')
 }
}
</script>
</body>
</html>
```





