Ajax, jQuery ajax, axios和fetch介绍、区别以及优缺点

## 区别介绍

# [「Ajax」:](https://developer.mozilla.org/zh-CN/docs/Web/Guide/AJAX" \t "_blank)

# 全称Asynchronous JavaScript and XML（异步的 JavaScript 和 XML）最早出现的发送后端请求技术，隶属于原始js中，核心使用XMLHttpRequest对象，多个请求之间如果有先后关系的话，就会出现回调地狱。

# [「](https://jquery.com/" \t "_blank)**[Jquery Ajax](https://jquery.com/" \t "_blank)**[」:](https://jquery.com/" \t "_blank)

# 是 jQuery 底层 AJAX 实现。简单易用的高层实现见 $.get, $.post 等。$.ajax() 返回其创建的 XMLHttpRequest 对象。大多数情况下你无需直接操作该函数，除非你需要操作不常用的选项，以获得更多的灵活性。

# [「](http://www.axios-js.com/" \t "_blank)**[Axios](http://www.axios-js.com/" \t "_blank)**[」:](http://www.axios-js.com/" \t "_blank)

# axios不是原生JS的，需要进行安装，它不但可以在客户端使用，也可以在nodejs端使用。Axios也可以在请求和响应阶段进行拦截。同样也是基于[Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise" \t "_blank)对象的。特性：从浏览器中创建 XMLHttpRequests、从 node.js 创建 http 请求、支持 Promise API、拦截请求和响应等。

# [「](https://developer.mozilla.org/zh-CN/docs/Web/API/Fetch_API" \t "_blank)**[Fetch](https://developer.mozilla.org/zh-CN/docs/Web/API/Fetch_API" \t "_blank)**[」:](https://developer.mozilla.org/zh-CN/docs/Web/API/Fetch_API" \t "_blank) [fɛtʃ] 抵达，到达; 取来;

# Fetch 提供了对 Request 和 Response （以及其他与网络请求有关的）对象的通用定义。使之今后可以被使用到更多地应用场景中：无论是service workers、Cache API、又或者是其他处理请求和响应的方式，甚至是任何一种需要你自己在程序中生成响应的方式。Fetch号称是AJAX的替代品，是在ES6出现的，使用了ES6中的[Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise" \t "_blank)对象。Fetch是基于promise设计的。Fetch的代码结构比起ajax简单多了，参数有点像jQuery ajax。但是，一定记住fetch不是ajax的进一步封装，而是原生js。Fetch函数就是原生js，没有使用XMLHttpRequest对象。

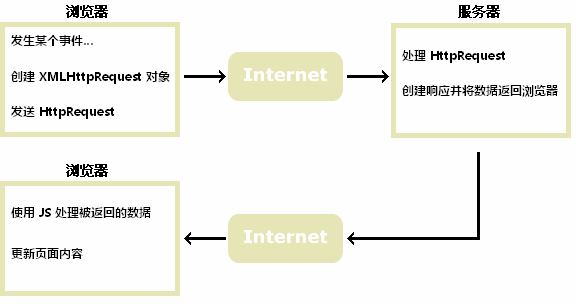
## Ajax

# AJAX = 异步 JavaScript 和 XML。

# AJAX 是一种用于创建快速动态网页的技术。

# 通过在后台与服务器进行少量数据交换，AJAX 可以使网页实现异步更新。这意味着可以在不重新加载整个网页的情况下，对网页的某部分进行更新。

**AJAX 工作原理**

****

# **XMLHttpRequest**

# 在 [AJAX](https://developer.mozilla.org/zh-CN/docs/Glossary/AJAX) 编程中，XMLHttpRequest 被大量使用。

# 使用 XMLHttpRequest（XHR）对象可以与服务器交互。您可以从URL获取数据，而无需让整个的页面刷新。这允许网页在不影响用户的操作的情况下更新页面的局部内容。

# 现代浏览器，最开始与服务器交换数据，都是通过XMLHttpRequest对象。它可以使用JSON、XML、HTML和text文本等格式发送和接收数据。

# 如何使用：①创建一个请求对象实例，②打开一个URL，③然后发送这个请求。④当传输完毕后，结果的[HTTP状态](https://developer.mozilla.org/zh-CN/docs/Web/HTTP/HTTP_response_codes" \t "https://juejin.im/post/_blank)以及返回的响应内容也可以从请求对象中获取。

# if (window.XMLHttpRequest) { // model browser

# xhr = new XMLHttpRequest()

# } else if (window.ActiveXObject) { // IE 6 and older

# xhr = new ActiveXObject('Microsoft.XMLHTTP')

# }

# // 第一个参数是请求方法名（get，post等）

# // 第二个参数是需要请求的接口地址

# //第三个参数是设置请求是否是异步，一般都是都是发送异步请求，同步请求可能会阻塞页面

# xhr.open('POST', url, true)

# // open方法只是设置参数，并不会发送请求 // 而请求是由send()方法发送的，并且接收一个参数，就是需要发送到服务端的数据 // 如果没有需要发送到服务端的数据，必须传入null，因为有些浏览器不许要这个参数

# xhr.send(data)

# //收到服务端响应的时候，就已经拿到了数据，响应的数据会自动填充到XHR对象中

# xhr.onreadystatechange = function () {

# try {

# // 处理响应

# if (xhr.readyState === 4) {

# // 请求正常

# if (xhr.status === 200) {

# // 处理响应

# } else {

# // 请求遇到一些问题，处理异常

# }

# } else {

# // 还处于未准备好的状态

# }

# } catch (e) {

# // 通信错误的事件中（例如服务器宕机）

# alert('Caught Exception: ' + e.description)

# }

# }

# **优点：**

# 不重新加载页面的情况下更新网页

# 在页面已加载后从服务器请求/接收数据

# 在后台向服务器发送数据。

# **缺点：**

# 使用起来也比较繁琐，需要设置很多值。

# 早期的IE浏览器有自己的实现，这样需要写兼容代码。

常用方法：

# **[XMLHttpRequest.open()](https://developer.mozilla.org/zh-CN/docs/Web/API/XMLHttpRequest/open)**初始化一个请求。

**[XMLHttpRequest.send()](https://developer.mozilla.org/zh-CN/docs/Web/API/XMLHttpRequest/send)**发送请求。如果请求是异步的（默认），那么该方法将在请求发送后立即返回。

# **XMLHttpRequest.timeout：** 这个就是可以设置请求超时，设置一个时间，单位为毫秒，如果请求时间超过设置时间，直接抛出异常

# **XMLHttpRequest.withCredentials**： 这个属性是看我们需不需要发送cookie的，因为如果是跨域请求，一般请求是不会带上cookie，需要我们手动设置这个属性

# **XMLHttpRequest.abort()：** 这个方法我们用来终止请求，当然是在请求发送并没有返回的时候

# **XMLHttpRequest.setRequestHeader()：** 这个我们刚才有用到过，就是设置请求头，更改一些我们需要的设置

# XMLHttpRequest 的实例有两种方式提交表单：

# ①使用 AJAX

# ②使用 FormData API

**Jquery Ajax**

$.ajax({

url: "/api/getWeather",

data: {

zipcode: 97201

},

success: function( result ) {

$( "#weather-temp" ).html( "<strong>" + result + "</strong> degrees" );

}

});

传统 Ajax 指的是 XMLHttpRequest（XHR）， 最早出现的发送后端请求技术，隶属于原始js中，核心使用XMLHttpRequest对象，多个请求之间如果有先后关系的话，就会出现回调地狱。

Jquery Ajax的出现是对原生XHR的封装，除此以外还增添了对JSONP的支持，Jquery Ajax经过多年的更新维护，真的已经是非常的方便了，但是随着react,vue,angular新一代框架的兴起，以及ES规范的完善，更多API的更新，它逐渐暴露了自己的不足:

1.本身是针对MVC的编程,不符合现在前端MVVM的浪潮

2.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

3.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）

4.不符合关注分离（Separation of Concerns）的原则

5.配置和调用方式非常混乱，而且基于事件的异步模型不友好

优点：

对原生XHR的封装，做了兼容处理，简化了使用。

增加了对JSONP的支持，可以简单处理部分跨域。

默认情况下，Ajax 请求使用 GET 方法。如果要使用 POST 方法，可以设定 type 参数值。这个选项也会影响 data 选项中的内容如何发送到服务器。

data 选项既可以包含一个查询字符串，比如 key1=value1&key2=value2 ，也可以是一个映射，比如 {key1: 'value1', key2: 'value2'} 。如果使用了后者的形式，则数据再发送器会被转换成查询字符串。这个处理过程也可以通过设置 processData 选项为 false 来回避。如果我们希望发送一个 XML 对象给服务器时，这种处理可能并不合适。并且在这种情况下，我们也应当改变 contentType 选项的值，用其他合适的 MIME 类型来取代默认的 application/x-www-form-urlencoded 。

var list = {};

$.ajax({

//请求方式 POST || GET

type : "POST",

//请求的媒体类型

contentType: "application/json;charset=UTF-8",

//请求地址

url : "http://127.0.0.1/xxxx/",

//数据，json字符串

data : JSON.stringify(list),

//请求成功

success : function(result) {

console.log(result);

},

//请求失败，包含具体的错误信息

error : function(e){

console.log(e.status);

console.log(e.responseText);

}

# $.ajax({name:value, name:value, ... })

|  |  |
| --- | --- |
| 名称 | 值/描述 |
| async | 布尔值，表示请求是否异步处理。默认是 true。 |
| beforeSend(xhr) | 发送请求前运行的函数。 |
| cache | 布尔值，表示浏览器是否缓存被请求页面。默认是 true。 |
| complete(xhr,status) | 请求完成时运行的函数（在请求成功或失败之后均调用，即在 success 和 error 函数之后）。 |
| contentType | 发送数据到服务器时所使用的内容类型。默认是："application/x-www-form-urlencoded"。 |
| context | 为所有 AJAX 相关的回调函数规定 "this" 值。 |
| data | 规定要发送到服务器的数据。 |
| dataFilter(data,type) | 用于处理 XMLHttpRequest 原始响应数据的函数。 |
| dataType | 预期的服务器响应的数据类型。 |
| error(xhr,status,error) | 如果请求失败要运行的函数。 |
| global | 布尔值，规定是否为请求触发全局 AJAX 事件处理程序。默认是 true。 |
| ifModified | 布尔值，规定是否仅在最后一次请求以来响应发生改变时才请求成功。默认是 false。 |
| jsonp | 在一个 jsonp 中重写回调函数的字符串。 |
| jsonpCallback | 在一个 jsonp 中规定回调函数的名称。 |
| password | 规定在 HTTP 访问认证请求中使用的密码。 |
| processData | 布尔值，规定通过请求发送的数据是否转换为查询字符串。默认是 true。 |
| scriptCharset | 规定请求的字符集。 |
| success(result,status,xhr) | 当请求成功时运行的函数。 |
| timeout | 设置本地的请求超时时间（以毫秒计）。 |
| traditional | 布尔值，规定是否使用参数序列化的传统样式。 |
| type | 规定请求的类型（GET 或 POST）。 |
| url | 规定发送请求的 URL。默认是当前页面。 |
| username | 规定在 HTTP 访问认证请求中使用的用户名。 |
| xhr | 用于创建 XMLHttpRequest 对象的函数。 |

# jQuery AJAX 方法：

| 方法 | 描述 |
| --- | --- |
| $.ajax() | 执行异步 AJAX 请求 |
| $.ajaxPrefilter() | 在每个请求发送之前且被 $.ajax() 处理之前，处理自定义 Ajax 选项或修改已存在选项 |
| $.ajaxSetup() | 为将来的 AJAX 请求设置默认值 |
| $.ajaxTransport() | 创建处理 Ajax 数据实际传送的对象 |
| $.get() | 使用 AJAX 的 HTTP GET 请求从服务器加载数据 |
| $.getJSON() | 使用 HTTP GET 请求从服务器加载 JSON 编码的数据 |
| $.getScript() | 使用 AJAX 的 HTTP GET 请求从服务器加载并执行 JavaScript |
| $.param() | 创建数组或对象的序列化表示形式（可用于 AJAX 请求的 URL 查询字符串） |
| $.post() | 使用 AJAX 的 HTTP POST 请求从服务器加载数据 |
| ajaxComplete() | 规定 AJAX 请求完成时运行的函数 |
| ajaxError() | 规定 AJAX 请求失败时运行的函数 |
| ajaxSend() | 规定 AJAX 请求发送之前运行的函数 |
| ajaxStart() | 规定第一个 AJAX 请求开始时运行的函数 |
| ajaxStop() | 规定所有的 AJAX 请求完成时运行的函数 |
| ajaxSuccess() | 规定 AJAX 请求成功完成时运行的函数 |
| load() | 从服务器加载数据，并把返回的数据放置到指定的元素中 dom.load(url) |
| serialize() | 编码表单元素集为字符串以便提交 |
| serializeArray() | 编码表单元素集为 names 和 values 的数组 |

$.get() 方法使用 HTTP GET 请求从服务器加载数据。

$.get(URL,data,function(data,status,xhr),dataType)

ajaxSuccess() 方法规定 AJAX 请求成功完成时运行的函数。

$(document).ajaxSuccess(function(event,xhr,options))

load() 方法从服务器加载数据，并把返回的数据放置到指定的元素中。  
$(selector).load(url,data,function(response,status,xhr))

**Axios**

Axios本质上也是对原生XHR的封装，只不过它是Promise的实现版本，可以用在浏览器和 node.js 中,符合最新的ES规范

执行 GET 请求

// 为给定 ID 的 user 创建请求

axios.get('/user?ID=12345')

.then(function (response) {

console.log(response);

})

.catch(function (error) {

console.log(error);

});

// 上面的请求也可以这样做

axios.get('/user', {

params: {

ID: 12345

}

})

.then(function (response) {

console.log(response);

})

.catch(function (error) {

console.log(error);

});

执行 POST 请求

axios.post('/user', {

firstName: 'Fred',

lastName: 'Flintstone'

})

.then(function (response) {

console.log(response);

})

.catch(function (error) {

console.log(error);

});

执行多个并发请求

function getUserAccount() {

return axios.get('/user/12345');

}

function getUserPermissions() {

return axios.get('/user/12345/permissions');

}

axios.all([getUserAccount(), getUserPermissions()])

.then(axios.spread(function (acct, perms) {

// 两个请求现在都执行完成

}));

axios特性：

-从浏览器中创建 [XMLHttpRequests](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest" \t "https://juejin.im/post/_blank)

-从 node.js 创建 http 请求

-支持 [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise" \t "https://juejin.im/post/_blank) API

-拦截请求和响应

-转换请求数据和响应数据

-取消请求

-自动转换 JSON 数据

-客户端支持防御 XSRF

XSRF（Cross Site Request Forgery, 跨站域请求伪造）也称 XSRF， 是一种网络的攻击方式，它在 2007 年曾被列为互联网 20 大安全隐患之一。其他安全隐患，比如 SQL 脚本注入，跨站域脚本攻击等在近年来已经逐渐为众人熟知，很多网站也都针对他们进行了防御。然而，对于大多数人来说，CSRF 却依然是一个陌生的概念。即便是大名鼎鼎的 Gmail, 在 2007 年底也存在着 CSRF 漏洞，从而被黑客攻击而使 Gmail 的用户造成巨大的损失。客户端支持防御 XSRF，是怎么做到的呢，就是让你的每个请求都带一个从cookie中拿到的key, 根据浏览器同源策略，假冒的网站是拿不到你cookie中得key的，这样，后台就可以轻松辨别出这个请求是否是用户在假冒网站上的误导输入，从而采取正确的策略。

**Fetch**

fetch提供了对 Request 和 Response （以及其他与网络请求有关的）对象的通用定义。 fetch是一个现代的概念, 等同于 XMLHttpRequest。它提供了许多与XMLHttpRequest相同的功能，但被设计成更具可扩展性和高效性。

fetch号称是ajax的替代品，它的API是基于Promise设计的，旧版本的浏览器不支持Promise，需要使用polyfill es6-promise

fetch API 提供了一个 JavaScript接口，用于访问和操纵HTTP管道的部分，例如请求和响应。它还提供了一个全局 fetch()方法，该方法提供了一种简单，合理的方式来跨网络异步获取资源。

fetch规范与jQuery.ajax()主要有两种方式的不同，牢记：

①当接收到一个代表错误的 HTTP 状态码时，从 fetch()返回的 Promise 不会被标记为 reject， 即使该 HTTP 响应的状态码是 404 或 500。相反，它会将 Promise 状态标记为 resolve （但是会将 resolve 的返回值的 ok 属性设置为 false ），仅当网络故障时或请求被阻止时，才会标记为 reject。

②默认情况下，fetch 不会从服务端发送或接收任何 cookies, 如果站点依赖于用户 session，则会导致未经认证的请求（要发送 cookies，必须设置 credentials 选项）。自从2017年8月25日后，默认的credentials政策变更为same-originFirefox也在61.0b13中改变默认值

简单实例:

fetch(url)

.then(response => {

if (response.ok) {

return response.json();

}

})

.then(data => console.log(data))

.catch(err => console.log(err))

优雅异步实现async/await：

async function test(){

try{

let res = await fetch(url)

if(res.status == 200){

await res.json()

}

} catch(err){

console.log(err)

}

}

fetch是比较底层的API，很多情况下都需要我们再次封装。

fetch() 接受第二个可选参数，一个可以控制不同配置的 init 对象：

postData('http://example.com/answer', {answer: 42})

.then(data => console.log(data)) // JSON from `response.json()` call

.catch(error => console.error(error))

function postData(url, data) {

// Default options are marked with \*

return fetch(url, {

body: JSON.stringify(data), // must match 'Content-Type' header

cache: 'no-cache', // \*default, no-cache, reload, force-cache, only-if-cached

credentials: 'same-origin', // include, same-origin, \*omit

headers: {

'user-agent': 'Mozilla/4.0 MDN Example',

'content-type': 'application/json'

},

method: 'POST', // \*GET, POST, PUT, DELETE, etc.

mode: 'cors', // no-cors, cors, \*same-origin

redirect: 'follow', // manual, \*follow, error

referrer: 'no-referrer', // \*client, no-referrer

})

.then(response => response.json()) // parses response to JSON

}

fetch的优点：

1.语法简洁，更加语义化

2.基于标准 Promise 实现，支持 async/await

3.同构方便，使用 [isomorphic-fetch](https://github.com/matthew-andrews/isomorphic-fetch" \t "_blank)

4.更加底层，提供的API丰富（request, response）

5.脱离了XHR，是ES规范里新的实现

缺点：

1. fetch只对网络请求报错，对400，500都当做成功的请求，需要封装去处理  
   2. fetch默认不会带cookie，需要添加配置项  
   3. fetch不支持abort，不支持超时控制，使用setTimeout及Promise.reject的实现

的超时控制并不能阻止请求过程继续在后台运行，造成了流量的浪费

4. fetch没有办法原生监测请求的进度，而XHR可以

fetch在前端的应用上有一项xhr怎么也比不上的能力：跨域的处理

我们都知道因为同源策略的问题，浏览器的请求是可能随便跨域的——一定要有跨域头或者借助JSONP，但是，fetch中可以设置mode为"no-cors"（不跨域），如下所示:

fetch('/users.json', {

method: 'post',

mode: 'no-cors',

data: {}

}).then(function() { /\* handle response \*/ });

**fetch方法**可以接收两个参数input和options。

①input 参数可以是字符串，包含要获取资源的 URL。也可以是一个 [Request](https://developer.mozilla.org/zh-CN/docs/Web/API/Request" \t "_blank) 对象。

②options 是一个可选参数。一个配置项对象，包括所有对请求的设置。可选的参数有：

这个也是**Request对象**

method: 请求使用的方法，如 GET、POST。  
headers: 请求的头信息，包含与请求关联的Headers对象或者ByteString。  
body: 请求参数可能是一个 Blob、BufferSource、FormData、URLSearchParams 或者 USVString 对象。

注意 GET 或 HEAD 方法的请求不能包含 body 信息  
mode: 请求的模式，如 cors、 no-cors 或者 same-origin。

cache: 缓存模式，如default, reload, no-cache  
credentials: 请求的 credentials，如 omit、same-origin 或者 include。

为了在当前域名内自动发送cookie ，必须提供这个选项。

**Response**

上面我们说了fetch的返回的是一个Promise对象。然后会携带Response 对象。

Response对象：

属性：

status (number) - HTTP请求结果参数，在100–599 范围， 200 为成功

statusText (String) - 服务器返回的状态报告

ok (boolean) - 如果返回200表示请求成功则为true

headers (Headers) - 返回头部信息，下面详细介绍

url(String) - 请求的地址

方法：

text() 以string的形式生成请求text

json 生成JSON.parse(responseText)的结果

blob 生成一个Blob

arrayBuffer() 生成一个ArrayBuffer

formData 生成格式化的数据，用于其他请求

其他方法：

clone()

Response.error()

Response.redirect()

response.headers

has(name) (boolean) 判断是否存在该信息头

get(name) (String) 获取信息头的数据

getAll(name) (Array) 获取所有头部数据

set(name, value) 添加headers的内容

delete(name) 删除header的信息

forEach 循环读取header的信息