## 简介

浏览器与服务器之间，采用 HTTP 协议通信。用户在浏览器地址栏键入一个网址，或者通过网页表单向服务器提交内容，这时浏览器就会向服务器发出 HTTP 请求。

1999年，微软公司发布 IE 浏览器5.0版，第一次引入新功能：允许 JavaScript 脚本向服务器发起 HTTP 请求。这个功能当时并没有引起注意，直到2004年 Gmail 发布和2005年 Google Map 发布，才引起广泛重视。2005年2月，AJAX 这个词第一次正式提出，它是 Asynchronous JavaScript and XML 的缩写，指的是通过 JavaScript 的异步通信，从服务器获取 XML 文档从中提取数据，再更新当前网页的对应部分，而不用刷新整个网页。后来，AJAX 这个词就成为 JavaScript 脚本发起 HTTP 通信的代名词，也就是说，只要用脚本发起通信，就可以叫做 AJAX 通信。W3C 也在2006年发布了它的国际标准。

### 同步和异步

Javascript语言的执行环境是"单线程"（single thread）。

所谓"单线程"，就是指一次只能完成一件任务。如果有多个任务，就必须排队，前面一个任务完成，再执行后面一个任务，以此类推。



这种模式的好处是实现起来比较简单，执行环境相对单纯；坏处是只要有一个任务耗时很长，后面的任务都必须排队等着，会拖延整个程序的执行。常见的浏览器无响应（假死），往往就是因为某一段Javascript代码长时间运行（比如死循环），导致整个页面卡在这个地方，其他任务无法执行。

为了解决这个问题，Javascript语言将任务的执行模式分成两种：同步（Synchronous）和异步（Asynchronous）。

"同步模式"就是上一段的模式，后一个任务等待前一个任务结束，然后再执行，程序的执行顺序与任务的排列顺序是一致的、同步的；"异步模式"则完全不同，每一个任务有一个或多个回调函数（callback），前一个任务结束后，不是执行后一个任务，而是执行回调函数，后一个任务则是不等前一个任务结束就执行，所以程序的执行顺序与任务的排列顺序是不一致的、异步的。

"异步模式"非常重要。在浏览器端，耗时很长的操作都应该异步执行，避免浏览器失去响应，最好的例子就是Ajax操作。在服务器端，"异步模式"甚至是唯一的模式，因为执行环境是单线程的，如果允许同步执行所有http请求，服务器性能会急剧下降，很快就会失去响应。

具体来说，AJAX 包括以下几个步骤。

1. 创建 XMLHttpRequest 实例
2. 发出 HTTP 请求
3. 接收服务器传回的数据
4. 更新网页数据

概括起来，就是一句话，AJAX 通过原生的XMLHttpRequest对象发出 HTTP 请求，得到服务器返回的数据后，再进行处理。现在，服务器返回的都是 JSON 格式的数据，XML 格式已经过时了，但是 AJAX 这个名字已经成了一个通用名词，字面含义已经消失了。

XMLHttpRequest对象是 AJAX 的主要接口，用于浏览器与服务器之间的通信。尽管名字里面有XML和Http，它实际上可以使用多种协议（比如file或ftp），发送任何格式的数据（包括字符串和二进制）。

XMLHttpRequest本身是一个构造函数，可以使用new命令生成实例。它没有任何参数。

var xhr = new XMLHttpRequest();

一旦新建实例，就可以使用open()方法发出 HTTP 请求。

xhr.open('GET', 'http://www.example.com/page.php', true);

上面代码向指定的服务器网址，发出 GET 请求。

然后，指定回调函数，监听通信状态（readyState属性）的变化。

ajax.onreadystatechange = handleStateChange;

function handleStateChange() {

// ...

}

上面代码中，一旦XMLHttpRequest实例的状态发生变化，就会调用监听函数handleStateChange

一旦拿到服务器返回的数据，AJAX 不会刷新整个网页，而是只更新网页里面的相关部分，从而不打断用户正在做的事情。

注意，AJAX 只能向同源网址（协议、域名、端口都相同）发出 HTTP 请求，如果发出跨域请求，就会报错

下面是XMLHttpRequest对象简单用法的完整例子。

var xhr = new XMLHttpRequest();

xhr.onreadystatechange = function(){

// 通信成功时，状态值为4

if (xhr.readyState === 4){

if (xhr.status === 200){

console.log(xhr.responseText);

} else {

console.error(xhr.statusText);

}

}

};

xhr.onerror = function (e) {

console.error(xhr.statusText);

};

xhr.open('GET', '/endpoint', true);

xhr.send(null);

## XMLHttpRequest 的实例属性

### XMLHttpRequest.readyState

XMLHttpRequest.readyState返回一个整数，表示实例对象的当前状态。该属性只读。它可能返回以下值。

* 0，表示 XMLHttpRequest 实例已经生成，但是实例的open()方法还没有被调用。
* 1，表示open()方法已经调用，但是实例的send()方法还没有调用，仍然可以使用实例的setRequestHeader()方法，设定 HTTP 请求的头信息。
* 2，表示实例的send()方法已经调用，并且服务器返回的头信息和状态码已经收到。
* 3，表示正在接收服务器传来的数据体（body 部分）。这时，如果实例的responseType属性等于text或者空字符串，responseText属性就会包含已经收到的部分信息。
* 4，表示服务器返回的数据已经完全接收，或者本次接收已经失败。

通信过程中，每当实例对象发生状态变化，它的readyState属性的值就会改变。这个值每一次变化，都会触发readyStateChange事件。

var xhr = new XMLHttpRequest();

if (xhr.readyState === 4) {

  // 请求结束，处理服务器返回的数据

} else {

  // 显示提示“加载中……”

}

上面代码中，xhr.readyState等于4时，表明脚本发出的 HTTP 请求已经成功。其他情况，都表示 HTTP 请求还在进行中。

通常，我们只对 readyState 值为 4 的阶段感兴趣，因为这时所有数据都已经就绪。不过，必须在调用 open()之前指定 onreadystatechange事件处理程序才能确保跨浏览器兼容性。

### XMLHttpRequest.onreadystatechange

XMLHttpRequest.onreadystatechange属性指向一个监听函数。readystatechange事件发生时（实例的readyState属性变化），就会执行这个属性。

### XMLHttpRequest.response

XMLHttpRequest.response属性表示服务器返回的数据体（即 HTTP 回应的 body 部分）。它可能是任何数据类型，比如字符串、对象、二进制对象等等，具体的类型由XMLHttpRequest.responseType属性决定。该属性只读。

如果本次请求没有成功或者数据不完整，该属性等于null。但是，如果responseType属性等于text或空字符串，在请求没有结束之前（readyState等于3的阶段），response属性包含服务器已经返回的部分数据。

### XMLHttpRequest.responseType

XMLHttpRequest.responseType属性是一个字符串，表示服务器返回数据的类型。这个属性是可写的，可以在调用open()方法之后、调用send()方法之前，设置这个属性的值，告诉服务器返回指定类型的数据。如果responseType设为空字符串，就等同于默认值text。

XMLHttpRequest.responseType属性可以等于以下值。

* ""（空字符串）：等同于text，表示服务器返回文本数据。
* "arraybuffer"：ArrayBuffer 对象，表示服务器返回二进制数组。
* "blob"：Blob 对象，表示服务器返回二进制对象。
* "document"：Document 对象，表示服务器返回一个文档对象。
* "json"：JSON 对象。
* "text"：字符串。

上面几种类型之中，text类型适合大多数情况，而且直接处理文本也比较方便。

### XMLHttpRequest.responseText

XMLHttpRequest.responseText属性返回从服务器接收到的字符串，该属性为只读。只有 HTTP 请求完成接收以后，该属性才会包含完整的数据。

### XMLHttpRequest.responseXML

XMLHttpRequest.responseXML属性返回从服务器接收到的 HTML 或 XML 文档对象，该属性为只读。如果本次请求没有成功，或者收到的数据不能被解析为 XML 或 HTML，该属性等于null。

该属性生效的前提是 HTTP 回应的Content-Type头信息等于text/xml或application/xml。这要求在发送请求前，XMLHttpRequest.responseType属性要设为document。如果 HTTP 回应的Content-Type头信息不等于text/xml和application/xml，但是想从responseXML拿到数据（即把数据按照 DOM 格式解析），那么需要手动调用XMLHttpRequest.overrideMimeType()方法，强制进行 XML 解析。

该属性得到的数据，是直接解析后的文档 DOM 树。

### XMLHttpRequest.status，XMLHttpRequest.statusText

XMLHttpRequest.status属性返回一个整数，表示服务器回应的 HTTP 状态码。一般来说，如果通信成功的话，这个状态码是200；如果服务器没有返回状态码，那么这个属性默认是200。请求发出之前，该属性为0。该属性只读。

* 200, OK，访问正常
* 301, Moved Permanently，永久移动
* 302, Move temporarily，暂时移动
* 304, Not Modified，未修改
* 307, Temporary Redirect，暂时重定向
* 401, Unauthorized，未授权
* 403, Forbidden，禁止访问
* 404, Not Found，未发现指定网址
* 500, Internal Server Error，服务器发生错误

基本上，只有2xx和304的状态码，表示服务器返回是正常状态。

XMLHttpRequest.statusText属性返回一个字符串，表示服务器发送的状态提示。不同于status属性，该属性包含整个状态信息，比如“OK”和“Not Found”。在请求发送之前（即调用open()方法之前），该属性的值是空字符串；如果服务器没有返回状态提示，该属性的值默认为"“OK”。该属性为只读属性。

### XMLHttpRequest.timeout，XMLHttpRequestEventTarget.ontimeout

XMLHttpRequest.timeout属性返回一个整数，表示多少毫秒后，如果请求仍然没有得到结果，就会自动终止。如果该属性等于0，就表示没有时间限制。

XMLHttpRequestEventTarget.ontimeout属性用于设置一个监听函数，如果发生 timeout 事件，就会执行这个监听函数。

## XMLHttpRequest 的实例方法

### XMLHttpRequest.open()

XMLHttpRequest.open()方法用于指定 HTTP 请求的参数，或者说初始化 XMLHttpRequest 实例对象。它一共可以接受五个参数。

void open(

string method,

string url,

optional boolean async,

optional string user,

optional string password

);

* method：表示 HTTP 动词方法，比如GET、POST、PUT、DELETE、HEAD等。
* url: 表示请求发送目标 URL。
* async: 布尔值，表示请求是否为异步，默认为true。如果设为false，则send()方法只有等到收到服务器返回了结果，才会进行下一步操作。该参数可选。由于同步 AJAX 请求会造成浏览器失去响应，许多浏览器已经禁止在主线程使用，只允许 Worker 里面使用。所以，这个参数轻易不应该设为false。
* user：表示用于认证的用户名，默认为空字符串。该参数可选。
* password：表示用于认证的密码，默认为空字符串。该参数可选。

注意，如果对使用过open()方法的 AJAX 请求，再次使用这个方法，等同于调用abort()，即终止请求。

### XMLHttpRequest.send()

XMLHttpRequest.send()方法用于实际发出 HTTP 请求。它的参数是可选的，如果不带参数，就表示 HTTP 请求只有一个 URL，没有数据体，典型例子就是 GET 请求；如果带有参数，就表示除了头信息，还带有包含具体数据的信息体，典型例子就是 POST 请求。

下面是 GET 请求的例子。

var xhr = new XMLHttpRequest();

xhr.open('GET',

'http://www.example.com/?id=' + encodeURIComponent(id),

true

);

xhr.send(null);

上面代码中，GET请求的参数，作为查询字符串附加在 URL 后面。

下面是发送 POST 请求的例子。

var xhr = new XMLHttpRequest();

var data = 'email='

+ encodeURIComponent(email)

+ '&password='

+ encodeURIComponent(password);

xhr.open('POST', 'http://www.example.com', true);

xhr.setRequestHeader('Content-Type','application/x-www-form-urlencoded');

xhr.send(data);

### XMLHttpRequest.setRequestHeader()

XMLHttpRequest.setRequestHeader()方法用于设置浏览器发送的 HTTP 请求的头信息。该方法必须在open()之后、send()之前调用。如果该方法多次调用，设定同一个字段，则每一次调用的值会被合并成一个单一的值发送。

该方法接受两个参数。第一个参数是字符串，表示头信息的字段名，第二个参数是字段值。

xhr.setRequestHeader('Content-Type', 'application/json');

xhr.setRequestHeader('Content-Length', JSON.stringify(data).length);

xhr.send(JSON.stringify(data));

上面代码首先设置头信息Content-Type，表示发送 JSON 格式的数据；然后设置Content-Length，表示数据长度；最后发送 JSON 数据。

## XMLHttpRequest 实例的事件

### readyStateChange 事件

readyState属性的值发生改变，就会触发 readyStateChange 事件。

我们可以通过onReadyStateChange属性，指定这个事件的监听函数，对不同状态进行不同处理。尤其是当状态变为4的时候，表示通信成功，这时回调函数就可以处理服务器传送回来的数据。

### progress 事件

上传文件时，XMLHTTPRequest 实例对象本身和实例的upload属性，都有一个progress事件，会不断返回上传的进度。

var xhr = new XMLHttpRequest();

function updateProgress (oEvent) {

if (oEvent.lengthComputable) {

var percentComplete = oEvent.loaded / oEvent.total;

} else {

console.log('无法计算进展');

}

}

### load 事件、error 事件、abort 事件

load 事件表示服务器传来的数据接收完毕，error 事件表示请求出错，abort 事件表示请求被中断（比如用户取消请求）。

var xhr = new XMLHttpRequest();

xhr.addEventListener('load', transferComplete);

xhr.addEventListener('error', transferFailed);

xhr.addEventListener('abort', transferCanceled);

xhr.open();

function transferComplete() {

console.log('数据接收完毕');

}

function transferFailed() {

console.log('数据接收出错');

}

function transferCanceled() {

console.log('用户取消接收');

}

## 同源限制

浏览器安全的基石是“同源政策”（[same-origin policy](https://en.wikipedia.org/wiki/Same-origin_policy)）。

### 含义

1995年，同源政策由 Netscape 公司引入浏览器。目前，所有浏览器都实行这个政策。

最初，它的含义是指，A 网页设置的 Cookie，B 网页不能打开，除非这两个网页“同源”。所谓“同源”指的是”三个相同“。

* 协议相同
* 域名相同
* 端口相同

举例来说，http://www.example.com/dir/page.html这个网址，协议是http://，域名是www.example.com，端口是80（默认端口可以省略），它的同源情况如下。

* http://www.example.com/dir2/other.html：同源
* http://example.com/dir/other.html：不同源（域名不同）
* http://v2.www.example.com/dir/other.html：不同源（域名不同）
* http://www.example.com:81/dir/other.html：不同源（端口不同）
* https://www.example.com/dir/page.html：不同源（协议不同）

### 目的

同源政策的目的，是为了保证用户信息的安全，防止恶意的网站窃取数据。

设想这样一种情况：A 网站是一家银行，用户登录以后，A 网站在用户的机器上设置了一个 Cookie，包含了一些隐私信息（比如存款总额）。用户离开 A 网站以后，又去访问 B 网站，如果没有同源限制，B 网站可以读取 A 网站的 Cookie，那么隐私信息就会泄漏。更可怕的是，Cookie 往往用来保存用户的登录状态，如果用户没有退出登录，其他网站就可以冒充用户，为所欲为。因为浏览器同时还规定，提交表单不受同源政策的限制。

由此可见，同源政策是必需的，否则 Cookie 可以共享，互联网就毫无安全可言了。

### 限制范围

随着互联网的发展，同源政策越来越严格。目前，如果非同源，共有三种行为受到限制。

（1） 无法读取非同源网页的 Cookie、LocalStorage 和 IndexedDB。

（2） 无法接触非同源网页的 DOM。

（3） 无法向非同源地址发送 AJAX 请求（可以发送，但浏览器会拒绝接受响应）。

同源政策规定，AJAX 请求只能发给同源的网址，否则就报错。

除了架设服务器代理（浏览器请求同源服务器，再由后者请求外部服务），有三种方法规避这个限制。

* JSONP
* WebSocket
* CORS

### JSONP

JSONP 是服务器与客户端跨源通信的常用方法。最大特点就是简单适用，老式浏览器全部支持，服务端改造非常小。

它的基本思想是，网页通过添加一个<script>元素，向服务器请求 JSON 数据，这种做法不受同源政策限制；服务器收到请求后，将数据放在一个指定名字的回调函数里传回来。

首先，网页动态插入<script>元素，由它向跨源网址发出请求。

function addScriptTag(src) {

var script = document.createElement('script');

script.setAttribute("type","text/javascript");

script.src = src;

document.body.appendChild(script);

}

window.onload = function () {

addScriptTag('http://example.com/ip?callback=foo');

}

function foo(data) {

console.log('Your public IP address is: ' + data.ip);

};

上面代码通过动态添加<script>元素，向服务器example.com发出请求。注意，该请求的查询字符串有一个callback参数，用来指定回调函数的名字，这对于 JSONP 是必需的。

服务器收到这个请求以后，会将数据放在回调函数的参数位置返回。

foo({

"ip": "8.8.8.8"

});

由于<script>元素请求的脚本，直接作为代码运行。这时，只要浏览器定义了foo函数，该函数就会立即调用。作为参数的 JSON 数据被视为 JavaScript 对象，而不是字符串，因此避免了使用JSON.parse的步骤。

### WebSocket

WebSocket 是一种通信协议，使用ws://（非加密）和wss://（加密）作为协议前缀。该协议不实行同源政策，**只要服务器支持**，就可以通过它进行跨源通信。

下面是一个例子，浏览器发出的 WebSocket 请求的头信息（摘自[维基百科](https://en.wikipedia.org/wiki/WebSocket)）。

GET /chat HTTP/1.1Host: server.example.comUpgrade: websocketConnection: UpgradeSec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==Sec-WebSocket-Protocol: chat, superchatSec-WebSocket-Version: 13Origin: http://example.com

上面代码中，有一个字段是Origin，表示该请求的请求源（origin），即发自哪个域名。

正是因为有了Origin这个字段，所以 WebSocket 才没有实行同源政策。因为服务器可以根据这个字段，判断是否许可本次通信。如果该域名在白名单内，服务器就会做出如下回应。

HTTP/1.1 101 Switching ProtocolsUpgrade: websocketConnection: UpgradeSec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=Sec-WebSocket-Protocol: chat

### CORS

CORS 是跨源资源分享（Cross-Origin Resource Sharing）的缩写。它是 W3C 标准，属于跨源 AJAX 请求的根本解决方法。相比 JSONP 只能发GET请求，CORS 允许任何类型的请求。

CORS 是一个 W3C 标准，全称是“跨域资源共享”（Cross-origin resource sharing）。它允许浏览器向跨域的服务器，发出XMLHttpRequest请求，从而克服了 AJAX 只能同源使用的限制。

#### 简介

CORS 需要浏览器和服务器同时支持。目前，所有浏览器都支持该功能。

整个 CORS 通信过程，都是浏览器自动完成，不需要用户参与。对于开发者来说，CORS 通信与普通的 AJAX 通信没有差别，代码完全一样。浏览器一旦发现 AJAX 请求跨域，就会自动添加一些附加的头信息，有时还会多出一次附加的请求，但用户不会有感知。**因此，实现 CORS 通信的关键是服务器**。只要服务器实现了 CORS 接口，就可以跨域通信。

表单在历史上一直可以跨域发出请求。