**前端跨域问题的解决方法（域的存在的必要性）**

**1.域问题存在的必要性：**

域问题的存在是由于同源策略引起的安全性问题的一种深入解析和资源、性能优化的解决。

**2.什么是跨域：**

异步请求，同一域下的文档，JS脚本文件试图去请求另一个域下的资源，这是广义的域的概念。

（实例）例：a.资源跳转: a链接；重定向；表单提交。

b.嵌入：<link/>;<script></script>;<img>;

<iframe>。

c.样式引入：background：URL（）；

@font-face（）。

d:JS发起的Ajax请求。

**3.同源策略 ：**

同源是指：“协议+域名+端口”都相同。

详解：同源策略/SOP（Same origin policy）是一种约定，由Netscape公司1995年引入浏览器，它是浏览器最核心也最基本的安全功能，如果缺少了同源策略，浏览器很容易受到XSS、CSFR等攻击。所谓同源是指"协议+域名+端口"三者相同，即便两个不同的域名指向同一个ip地址，也非同源。  
同源策略限制以下几种行为：  
1.) Cookie、LocalStorage 和 IndexDB 无法读取；

2.) DOM 和 Js对象无法获得；

3.) AJAX 请求不能发送。

**4.跨域的解决方法：**

1.） 通过jsonp跨域（jQuery分装的Ajax）；

2.）document.domian+iframe跨域；

3.）location.hash+iframe;

4.) window.name+iframe跨域；

5.）postMessage跨域；

6.）跨域资源共享 （CORS）;

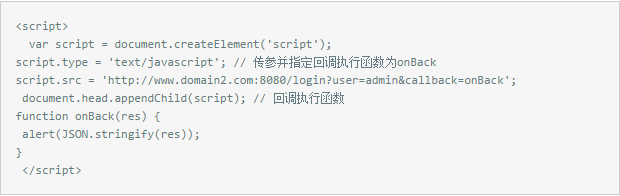
7.) nginX代理跨域；

8.）node.js中间件代理跨域；

9.）webSocket协议跨域（HTML5新增）；

10.）jsonp跨域缺点：只能实现get一种请求；

**一、 通过jsonp跨域**  
通常为了减轻web服务器的负载，我们把js、css，img等静态资源分离到另一台独立域名的服务器上，在html页面中再通过相应的标签从不同域名下加载静态资源，而被浏览器允许，基于此原理，我们可以通过动态创建script，再请求一个带参网址实现跨域通信。  
1.）原生实现：



服务端返回如下（返回时即执行全局函数）：

微信图片2.png

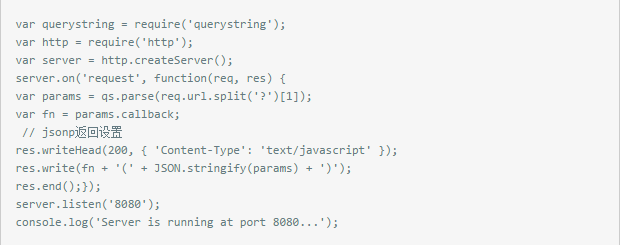
2.）jQuery/Ajax：



3.）Vue.js;



后端node.js代码示例：



jsonp缺点：只能实现get一种请求。  
**二、 document.domain + iframe跨域**  
此方案仅限主域相同，子域不同的跨域应用场景。  
实现原理：两个页面都通过js强制设置document.domain为基础主域，就实现了同域。  
1.）父窗口：

<iframe id="iframe" src="http://child.domain.com/b.html"></iframe>

<script>

document.domain = 'domain.com';

var user = 'admin';

</script>

2.）子窗口：

<script>

document.domain = 'domain.com'; // 获取父窗口中变量

alert('get js data from parent ---> ' + window.parent.user);

</script>

**三、 location.hash + iframe跨域**  
实现原理： a欲与b跨域相互通信，通过中间页c来实现。 三个页面，不同域之间利用iframe的location.hash传值，相同域之间直接js访问来通信。  
具体实现：A域：a.html -> B域：b.html -> A域：c.html，a与b不同域只能通过hash值单向通信，b与c也不同域也只能单向通信，但c与a同域，所以c可通过parent.parent访问a页面所有对象。  
1.）a.html：

<iframe id="iframe" src="http://www.domain2.com/b.html" style="display:none;"></iframe>

<script>

var iframe = document.getElementById('iframe');

// 向b.html传hash值 setTimeout(function() {

iframe.src = iframe.src + '#user=admin';

}, 1000);

// 开放给同域c.html的回调方法

function onCallback(res) { alert('data from c.html ---> ' + res); }</script>

2.）b.html：

<iframe id="iframe" src="http:// " style="display:none;"></iframe><script>

var iframe = document.getElementById('iframe');

// 监听a.html传来的hash值，再传给c.html

window.onhashchange = function () {

iframe.src = iframe.src + location.hash;

};

</script>

3.）c.html:

<script>

// 监听b.html传来的hash值

window.onhashchange = function () {

// 再通过操作同域a.html的js回调，将结果传回 window.parent.parent.onCallback('hello: ' + location.hash.replace('#user=', ''));

};

</script>

**四、 window.name + iframe跨域**  
window.name属性的独特之处：name值在不同的页面（甚至不同域名）加载后依旧存在，并且可以支持非常长的 name 值（2MB）。  
1.）a.html：

var proxy = function(url, callback) {

var state = 0;

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

// 加载跨域页面

iframe.src = url;

// onload事件会触发2次，第1次加载跨域页，并留存数据于window.name

iframe.onload = function() {

if (state === 1) {

// 第2次onload(同域proxy页)成功后，读取同域window.name中数据

callback(iframe.contentWindow.name);

destoryFrame();

}

else if (state === 0) {

// 第1次onload(跨域页)成功后，切换到同域代理页面 iframe.contentWindow.location = 'http://www.domain1.com/proxy.html';

state = 1;

}

};

document.body.appendChild(iframe);

// 获取数据以后销毁这个iframe，释放内存；这也保证了安全（不被其他域frame js访问）

function destoryFrame() {

iframe.contentWindow.document.write('');

iframe.contentWindow.close();

document.body.removeChild(iframe);

}

};

// 请求跨域b页面数据

proxy('http://www.domain2.com/b.html', function(data){

alert(data);

});

2.）proxy.html：

中间代理页，与a.html同域，内容为空即可。  
3.）b.html：

<script>

window.name = 'This is domain2 data!';

</script>

总结：通过iframe的src属性由外域转向本地域，跨域数据即由iframe的window.name从外域传递到本地域。这个就巧妙地绕过了浏览器的跨域访问限制，但同时它又是安全操作。  
**五、 postMessage跨域**  
postMessage是HTML5 XMLHttpRequest Level 2中的API，且是为数不多可以跨域操作的window属性之一，它可用于解决以下方面的问题：a.） 页面和其打开的新窗口的数据传递b.） 多窗口之间消息传递c.） 页面与嵌套的iframe消息传递d.） 上面三个场景的跨域数据传递  
用法：postMessage(data,origin)方法接受两个参数data： html5规范支持任意基本类型或可复制的对象，但部分浏览器只支持字符串，所以传参时最好用JSON.stringify()序列化。origin： 协议+主机+端口号，也可以设置为"\*"，表示可以传递给任意窗口，如果要指定和当前窗口同源的话设置为"/"。  
1.）a.html：

<iframe id="iframe" src="http://www.domain2.com/b.html" style="display:none;"></iframe>

<script>

var iframe = document.getElementById('iframe');

iframe.onload = function() {

var data = { name: 'aym' };

// 向domain2传送跨域数据 iframe.contentWindow.postMessage(JSON.stringify(data), 'http://www.domain2.com'); };

// 接受domain2返回数据

window.addEventListener('message', function(e) {

alert('data from domain2 ---> ' + e.data);

}, false);

</script>

2.）b.html：

<script>

// 接收domain1的数据

window.addEventListener('message', function(e) {

alert('data from domain1 ---> ' + e.data);

var data = JSON.parse(e.data);

if (data) { data.number = 16;

// 处理后再发回domain1

window.parent.postMessage(JSON.stringify(data), 'http://www.domain1.com'); }

},

false);

</script>

**六、 跨域资源共享（CORS）**  
普通跨域请求：只服务端设置Access-Control-Allow-Origin即可，前端无须设置，若要带cookie请求：前后端都需要设置。  
需注意的是：由于同源策略的限制，所读取的cookie为跨域请求接口所在域的cookie，而非当前页。如果想实现当前页cookie的写入，可参考下文：七、nginx反向代理中设置proxy\_cookie\_domain 和 八、NodeJs中间件代理中cookieDomainRewrite参数的设置。  
目前，所有浏览器都支持该功能(IE8+：IE8/9需要使用XDomainRequest对象来支持CORS）)，CORS也已经成为主流的跨域解决方案。  
**1、 前端设置：**  
1.）原生ajax

// 前端设置是否带

cookiexhr.withCredentials = true;

示例代码：

var xhr = new XMLHttpRequest();// IE8/9需用window.XDomainRequest兼容

// 前端设置是否带

cookiexhr.withCredentials = true;xhr.open('post', 'http://www.domain2.com:8080/login', true);

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

xhr.send('user=admin');

xhr.onreadystatechange = function() {

if (xhr.readyState == 4 && xhr.status == 200) {

alert(xhr.responseText);

}

};

2.）jQuery ajax

$.ajax({

...

xhrFields: {

withCredentials: true // 前端设置是否带cookie

},

crossDomain: true, // 会让请求头中包含跨域的额外信息，但不会含cookie

...

});

3.）vue框架在vue-resource封装的ajax组件中加入以下代码：

Vue.http.options.credentials = true

**2、 服务端设置：**  
若后端设置成功，前端浏览器控制台则不会出现跨域报错信息，反之，说明没设成功。  
1.）Java后台：

/\* \* 导入包：import javax.servlet.http.HttpServletResponse; \* 接口参数中定义：HttpServletResponse response \*/

response.setHeader("Access-Control-Allow-Origin", "http://www.domain1.com"); // 若有端口需写全（协议+域名+端口）

response.setHeader("Access-Control-Allow-Credentials", "true");

2.）Nodejs后台示例：

var http = require('http');

var server = http.createServer();

var qs = require('querystring');

server.on('request', function(req, res) {

var postData = '';

// 数据块接收中

req.addListener('data', function(chunk) {

postData += chunk; });

// 数据接收完毕

req.addListener('end', function() {

postData = qs.parse(postData);

// 跨域后台设置

res.writeHead(200, { 'Access-Control-Allow-Credentials': 'true', // 后端允许发送Cookie 'Access-Control-Allow-Origin': 'http://www.domain1.com', // 允许访问的域（协议+域名+端口）

'Set-Cookie': 'l=a123456;Path=/;

Domain=www.domain2.com;HttpOnly' // HttpOnly:脚本无法读取cookie }); res.write(JSON.stringify(postData));

res.end();

});

});

server.listen('8080');

console.log('Server is running at port 8080...');

**七、 nginx代理跨域**  
**1、 nginx配置解决iconfont跨域**  
浏览器跨域访问js、css、img等常规静态资源被同源策略许可，但iconfont字体文件(eot|otf|ttf|woff|svg)例外，此时可在nginx的静态资源服务器中加入以下配置。

location / {

add\_header Access-Control-Allow-Origin \*;

}

**2、 nginx反向代理接口跨域**  
跨域原理： 同源策略是浏览器的安全策略，不是HTTP协议的一部分。服务器端调用HTTP接口只是使用HTTP协议，不会执行JS脚本，不需要同源策略，也就不存在跨越问题。  
实现思路：通过nginx配置一个代理服务器（域名与domain1相同，端口不同）做跳板机，反向代理访问domain2接口，并且可以顺便修改cookie中domain信息，方便当前域cookie写入，实现跨域登录。  
nginx具体配置：

#proxy服务器

server {

listen 81;

server\_name www.domain1.com;

location / {

proxy\_pass http://www.domain2.com:8080; #反向代理

proxy\_cookie\_domain www.domain2.com www.domain1.com; #修改cookie里域名

index index.html index.htm;

# 当用webpack-dev-server等中间件代理接口访问nignx时，此时无浏览器参与，故没有同源限制，下面的跨域配置可不启用

add\_header Access-Control-Allow-Origin http://www.domain1.com; #当前端只跨域不带cookie时，可为\*

add\_header Access-Control-Allow-Credentials true;

}

}

1.) 前端代码示例：

var xhr = new XMLHttpRequest();

// 前端开关：浏览器是否读写

cookiexhr.withCredentials = true;

// 访问nginx中的代理服务器

xhr.open('get', 'http://www.domain1.com:81/?user=admin', true);

xhr.send();

2.) Nodejs后台示例：

var http = require('http');

var server = http.createServer();

var qs = require('querystring');

server.on('request', function(req, res) {

var params = qs.parse(req.url.substring(2));

// 向前台写cookie

res.writeHead(200, {

'Set-Cookie': 'l=a123456;Path=/;Domain=www.domain2.com;HttpOnly' // HttpOnly:脚本无法读取

});

res.write(JSON.stringify(params)); res.end();

});

server.listen('8080');

console.log('Server is running at port 8080...');

**八、 Nodejs中间件代理跨域**  
node中间件实现跨域代理，原理大致与nginx相同，都是通过启一个代理服务器，实现数据的转发，也可以通过设置cookieDomainRewrite参数修改响应头中cookie中域名，实现当前域的cookie写入，方便接口登录认证。  
**1、 非vue框架的跨域（2次跨域）**  
利用node + express + http-proxy-middleware搭建一个proxy服务器。  
1.）前端代码示例：

var xhr = new XMLHttpRequest();

// 前端开关：浏览器是否读写

cookiexhr.withCredentials = true;

// 访问http-proxy-middleware代理服务器

xhr.open('get', 'http://www.domain1.com:3000/login?user=admin', true);

xhr.send();

2.）中间件服务器：

var express = require('express');

var proxy = require('http-proxy-middleware');

var app = express();

app.use('/', proxy({

// 代理跨域目标接口

target: 'http://www.domain2.com:8080',

changeOrigin: true,

// 修改响应头信息，实现跨域并允许带cookie

onProxyRes: function(proxyRes, req, res) {

res.header('Access-Control-Allow-Origin', 'http://www.domain1.com'); res.header('Access-Control-Allow-Credentials', 'true'); },

// 修改响应信息中的cookie域名

cookieDomainRewrite: 'www.domain1.com' // 可以为false，表示不修改}

));

app.listen(3000);

console.log('Proxy server is listen at port 3000...');

3.）Nodejs后台同（六：nginx）  
**2、 vue框架的跨域（1次跨域）**  
利用node + webpack + webpack-dev-server代理接口跨域。在开发环境下，由于vue渲染服务和接口代理服务都是webpack-dev-server同一个，所以页面与代理接口之间不再跨域，无须设置headers跨域信息了。  
webpack.config.js部分配置：

module.exports = {

entry: {},

module: {},

...

devServer: {

historyApiFallback: true,

proxy: [{

context: '/login',

target: 'http://www.domain2.com:8080', // 代理跨域目标接口

changeOrigin: true,

cookieDomainRewrite: 'www.domain1.com' // 可以为false，表示不修改

}],

noInfo: true

}}

九、 WebSocket协议跨域  
WebSocket protocol是HTML5一种新的协议。它实现了浏览器与服务器全双工通信，同时允许跨域通讯，是server push技术的一种很好的实现。原生WebSocket API使用起来不太方便，我们使用Socket.io，它很好地封装了webSocket接口，提供了更简单、灵活的接口，也对不支持webSocket的浏览器提供了向下兼容。  
1.）前端代码：

<div>user input：<input type="text"></div>

<script src="./socket.io.js"></script>

<script>var socket = io('http://www.domain2.com:8080');

// 连接成功处理

socket.on('connect', function() {

// 监听服务端消息

socket.on('message', function(msg) {

console.log('data from server: ---> ' + msg);

});

// 监听服务端关闭

socket.on('disconnect', function() {

console.log('Server socket has closed.');

});

});

document.getElementsByTagName('input')[0].onblur = function() {

socket.send(this.value);};

</script>

2.）Nodejs socket后台：

var http = require('http');

var socket = require('socket.io');

// 启http服务

var server = http.createServer(function(req, res) {

res.writeHead(200, {

'Content-type': 'text/html'

});

res.end();

});

server.listen('8080');

console.log('Server is running at port 8080...');

// 监听socket连接

socket.listen(server).on('connection', function(client) {

// 接收信息

client.on('message', function(msg) {

client.send('hello：' + msg);

console.log('data from client: ---> ' + msg);

});

// 断开处理

client.on('disconnect', function() {

console.log('Client socket has closed.');

});

});

附：  
同源策略限制从一个源加载的文档或脚本如何与来自另一个源的资源进行交互。这是一个用于隔离潜在恶意文件的关键的安全机制。但是有时候跨域请求资源是合理的需求，本文尝试从多篇文章中汇总至今存在的所有跨域请求解决方案。  
跨域请求  
首先需要了解的是同源和跨源的概念。对于相同源，其定义为：如果协议、端口（如果指定了一个）和主机对于两个页面是相同的，则两个页面具有相同的源。只要三者之一任意一点有不同，那么就为不同源。当一个资源从与该资源本身所在的服务器的域或端口不同的域或不同的端口请求一个资源时，资源会发起一个跨域 HTTP 请求。而有关跨域请求受到限制的原因可以参考如下 MDN 文档片段：  
跨域不一定是浏览器限制了发起跨站请求，而也可能是跨站请求可以正常发起，但是返回结果被浏览器拦截了。最好的例子是 CSRF 跨站攻击原理，请求是发送到了后端服务器无论是否跨域！注意：有些浏览器不允许从 HTTPS 的域跨域访问 HTTP，比如 Chrome 和 Firefox，这些浏览器在请求还未发出的时候就会拦截请求，这是一个特例。

解决方法汇总  
以下我们由简及深介绍各种存在的跨域请求解决方案，包括 document.domain, location.hash, window.name, window.postMessage, JSONP, WebSocket, CORS  
。  
document.domain  
document.domain  
的作用是用来获取/设置当前文档的原始域部分，例如：  
// 对于文档 [www.example.xxx/good.htmldocument.domain=](http://www.example.xxx/good.htmldocument.domain=)"[www.example.xxx](http://www.example.xxx)"// 对于URI <http://developer.mozilla.org/en/docs/DOM> document.domain="[developer.mozilla.org](http://developer.mozilla.org)"

如果当前文档的域无法识别，那么 domain 属性会返回 null。  
在根域范围内，Mozilla允许你把domain属性的值设置为它的上一级域。例如，在 developer.mozilla.org 域内，可以把domain设置为 "mozilla.org" 但不能设置为 "mozilla.com" 或者"org"。

因此，若两个源所用协议、端口一致，主域相同而二级域名不同的话，可以借鉴该方法解决跨域请求。  
比如若我们在 [a.github.io](https://link.juejin.im/?target=http%3A%2F%2Fa.github.io) 页面执行以下语句：  
document.domain = "github.io"

那么之后页面对 github.io  
发起请求时页面则会成功通过对 github.io  
的同源检测。比较直接的一个操作是，当我们在 a.github.io  
页面中利用 iframe 去加载 github.io  
时，通过如上的赋值后，我们可以在 a.github.io  
页面中去操作 iframe 里的内容。  
我们同时考虑另一种情况：存在两个子域名 a.github.io  
以及 b.github.io  
， 其中前者域名下网页 a.html 通过 iframe 引入了后者域名下的 b.html，此时在 a.html 中是无法直接操作 b.html 的内容的。  
同样利用 document.domain  
，我们在两个页面中均加入  
document.domain='github.io'

这样在以上的 a.html 中就可以操作通过 iframe 引入的 b.html 了。  
**document.domain** 的优点在于解决了主语相同的跨域请求，但是其缺点也是很明显的：比如一个站点受到攻击后，另一个站点会因此引起安全漏洞；若一个页面中引入多个 iframe，想要操作所有的 iframe 则需要设置相同的 domain。  
location.hash  
location.hash  
是一个可读可写的字符串，该字符串是 URL 的锚部分（从 # 号开始的部分）。例如：  
// 对于页面 <http://example.com:1234/test.htm#part2location.hash> = "#part2"

同时，由于我们知道改变 hash 并不会导致页面刷新，所以可以利用 hash 在不同源间传递数据。  
假设 github.io  
域名下 a.html 和 shaonian.eu  
域名下 b.html 存在跨域请求，那么利用 location.hash 的一个解决方案如下：  
a.html 页面中创建一个隐藏的 iframe， src 指向 b.html，其中 src 中可以通过 hash 传入参数给 b.html  
b.html 页面在处理完传入的 hash 后通过修改 a.html 的 hash 值达到将数据传送给 a.html 的目的  
a.html 页面添加一个定时器，每隔一定时间判断自身的 location.hash 是否变化，以此响应处理

以上步骤中需要注意第二点：如何在 iframe 页面中修改 父亲页面的 hash 值。由于在 IE 和 Chrome 下，两个不同域的页面是不允许 parent.location.hash  
这样赋值的，所以对于这种情况，我们需要在父亲页面域名下添加另一个页面来实现跨域请求，具体如下：  
假设 a.html 中 iframe 引入了 b.html, 数据需要在这两个页面之间传递，且 c.html 是一个与 a.html 同源的页面  
a.html 通过 iframe 将数据通过 hash 传给 b.html  
b.html 通过 iframe 将数据通过 hash 传给 c.html  
c.html 通过 parent.parent.location.hash  
设置 a.html 的 hash 达到传递数据的目的

**location.bash** 方法的优点在于可以解决域名完全不同的跨域请求，并且可以实现双向通讯；而缺点则包括以下几点：  
利用这种方法传递的数据量受到 url 大小的限制，传递数据类型有限  
由于数据直接暴露在 url 中则存在安全问题  
若浏览器不支持 onhashchange  
事件，则需要通过轮训来获知 url 的变化  
有些浏览器会在 hash 变化时产生历史记录，因此可能影响用户体验

window.name  
该属性用于获取/设置窗口的名称。其特征在于：一个窗口的生命周期内，窗口载入的所有页面共享该值，且都具有对该属性的读写权限。这意味着如果不修改该值，那么在不同页面加载之后该值也不会变，且其支持长达 2MB 的存储量。  
利用该特性我们可以将跨域请求用如下步骤解决：  
在 a.github.io/a.html 中创建 iframe 指向 b.github.io/b.html (页面会将自身的 window.name 附在 iframe 上)  
给 a.github.io/a.html 添加监听 iframe 的 onload 事件，在该事件中将 iframe 的 src 设置为本地域的代理文件（代理文件和a.html处于同一域下，可以相互通信），同时可以传出 iframe 的 name 值  
获取数据后销毁 iframe，释放内存，同时也保证了安全

**window.name** 的优势在于巧妙地绕过了浏览器的跨域访问限制，但同时它又是安全操作。  
window.postMessage  
HTML5 为了解决这个问题，引入了一个全新的 API：跨文档通信 API（Cross-document messaging）。这个 API 为 window 对象新增了一个 window.postMessage 方法，允许跨窗口通信，不论这两个窗口是否同源。  
API 的详细使用方法请见 [MDN](https://link.juejin.im/?target=https%3A%2F%2Fdeveloper.mozilla.org%2Fen-US%2Fdocs%2FWeb%2FAPI%2FWindow%2FpostMessage)。  
JSONP  
JSONP, 全称 JSON with Padding，是使用 AJAX 实现的请求不同源的跨域。其基本原理：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。  
以下为一个例子，由于 test.js 返回的内容直接作为代码运行，所以只要 a.html 中定义了 callback  
函数, 它就会立即被调用。  
// 当前页面 [a.com/a.html](http://a.com/a.html)<script type="text/javascript">//回调函数function callback(data) { alert(data.message);}</script><script type="text/javascript" src="http://b.com/test.js"></script>// test.js// 调用callback函数，并以json数据形式作为阐述传递，完成回调callback({message:"success"});

为了保证 script 的灵活，我们可以通过 JavaScript 动态创建 script 标签，并通过 HTTP 参数向服务器传入回调函数名，案例如下所示：  
<script type="text/javascript"> // 添加<script>标签的方法 function addScriptTag(src){ var script = document.createElement('script'); script.setAttribute("type","text/javascript"); script.src = src; document.body.appendChild(script); } window.onload = function(){ // 搜索apple，将自定义的回调函数名result传入callback参数中 addScriptTag("<http://ajax.googleapis.com/ajax/services/search/web?v=1.0&q=apple&callback=result>"); } // 自定义的回调函数result function result(data) { // 我们就简单的获取apple搜索结果的第一条记录中url数据 alert(data.responseData.results[0].unescapedUrl); }</script>

jQuery 有相应的 JSONP 的实现方法，见 [API](https://link.juejin.im/?target=http%3A%2F%2Fapi.jquery.com%2Fjquery.getjson%2F)。  
**JSONP**的优点在于简单适用，老式浏览器全部支持，服务器改造小。不需要XMLHttpRequest或ActiveX的支持；但缺点是只支持 GET 请求。  
WebSocket  
WebSocket 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。  
CORS  
CORS是一个W3C标准，全称是"跨域资源共享"（Cross-origin resource sharing）。它允许浏览器向跨源服务器，发出XMLHttpRequest请求，从而克服了AJAX只能同源使用的限制。

跨域资源共享（ CORS ）机制允许 Web 应用服务器进行跨域访问控制，从而使跨域数据传输得以安全进行。其需要服务端和客户端同时支持。  
跨域资源共享标准（ cross-origin sharing standard ）允许在下列场景中使用跨域 HTTP 请求：  
由 XMLHttpRequest 或 Fetch 发起的跨域 HTTP 请求  
Web 字体 (CSS 中通过 @font-face 使用跨域字体资源), 因此，网站就可以发布 TrueType 字体资源，并只允许已授权网站进行跨站调用  
WebGL 贴图  
使用 drawImage 将 Images/video 画面绘制到 canvas  
样式表（使用 CSSOM）  
Scripts (未处理的异常)

CORS 存在以下三种主要场景，分别是**简单请求，预检请求和附带身份凭证的请求**。  
**简单请求**：若只使用 GET, HEAD 或者 POST 请求，且除 CORS 安全的首部字段集合外，无人为设置该集合之外的其他首部字段，同时 Content-Type 值属于下列之一，那么该请求则可以被视为简单请求：

application/x-www-form-urlencodedmultipart/form-datatext/plain

此情况下，若服务端返回的 Access-Control-Allow-Origin: \*  
，则表明该资源可以被任意外域访问。若要指定仅允许来自某些域的访问，需要将 \*  
设定为该域，例如：  
Access-Control-Allow-Origin: <http://foo.example>

**预检请求**：与前述简单请求不同，该要求必须首先使用 OPTIONS 方法发起一个预检请求到服务器，以获知服务器是否允许该实际请求。当请求满足以下三个条件任意之一时， 即应首先发送预检请求：

使用了 PUT, DELETE, CONNECT, OPTIONS, TRACE, PATCH 中任一的 HTTP 方法  
人为设置了对 CORS 安全的首部字段集合之外的其他首部字段  
Content-Type 的值不属于下列之一

application/x-www-form-urlencodedmultipart/form-datatext/plain

预检请求完成之后（通过 OPTIONS 方法实现），才发送实际请求。一个示范 HTTP 请求如下所示：  
var invocation = new XMLHttpRequest();var url = '<http://bar.other/resources/post-here/';var> body = '<?xml version="1.0"?><person><name>Arun</name></person>'; function callOtherDomain(){ if(invocation) { invocation.open('POST', url, true); invocation.setRequestHeader('X-PINGOTHER', 'pingpong'); invocation.setRequestHeader('Content-Type', 'application/xml'); invocation.onreadystatechange = handler; invocation.send(body); }}

**附带身份凭证的请求**：这种方式的特点在于能够在跨域请求时向服务器发送凭证请求，例如 Cookies (withCredentials 标志设置为 true)。

一般而言，对于跨域 XMLHttpRequest 或 Fetch 请求，浏览器不会发送身份凭证信息。如果要发送凭证信息，需要设置 XMLHttpRequest 的某个特殊标志位。但是需要注意的是，如果服务器端的响应中未携带 Access-Control-Allow-Credentials: true  
，浏览器将不会把响应内容返回给请求的发送者。  
附带身份凭证的请求与通配符  
对于附带身份凭证的请求，服务器不得设置 Access-Control-Allow-Origin 的值为“*”。  
这是因为请求的首部中携带了 Cookie 信息，如果 Access-Control-Allow-Origin 的值为“*”，请求将会失败。而将 Access-Control-Allow-Origin 的值设置为 [foo.example，则请求将成功执行。](https://link.juejin.im/?target=http%3A%2F%2Ffoo.example%25EF%25BC%258C%25E5%2588%2599%25E8%25AF%25B7%25E6%25B1%2582%25E5%25B0%2586%25E6%2588%2590%25E5%258A%259F%25E6%2589%25A7%25E8%25A1%258C%25E3%2580%2582)  
另外，响应首部中也携带了 Set-Cookie 字段，尝试对 Cookie 进行修改。如果操作失败，将会抛出异常。

本文转自：作者：CatherinePlans  
 链接：http://www.jianshu.com/p/ad799a7801e1