**什么是跨域？**

跨域是指一个域下的文档或脚本试图去请求另一个域下的资源，这里跨域是广义的。

广义的跨域：

1.) 资源跳转： A链接、重定向、表单提交

2.) 资源嵌入： <link>、<script>、<img>、<frame>等dom标签，还有样式中background:url()、@font-face()等文件外链

3.) 脚本请求： js发起的ajax请求、dom和js对象的跨域操作等

其实我们通常所说的跨域是狭义的，是由浏览器同源策略限制的一类请求场景。

什么是同源策略？  
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

**常见跨域场景**

URL 说明 是否允许通信

http://www.domain.com/a.js

http://www.domain.com/b.js 同一域名，不同文件或路径 允许

http://www.domain.com/lab/c.js

http://www.domain.com:8000/a.js

http://www.domain.com/b.js 同一域名，不同端口 不允许

http://www.domain.com/a.js

https://www.domain.com/b.js 同一域名，不同协议 不允许

http://www.domain.com/a.js

http://192.168.4.12/b.js 域名和域名对应相同ip 不允许

http://www.domain.com/a.js

http://x.domain.com/b.js 主域相同，子域不同 不允许

http://domain.com/c.js

http://www.domain1.com/a.js

http://www.domain2.com/b.js 不同域名 不允许

**跨域解决方案**

1、 通过jsonp跨域  
2、 document.domain + 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、 nodejs中间件代理跨域  
9、 WebSocket协议跨域

**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：

<script>

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

script.type = 'text/javascript';

// 传参并指定回调执行函数为onBack

script.src = 'http://www.domain2.com:8080/login?user=admin&callback=onBack';

document.head.appendChild(script);

// 回调执行函数

function onBack(res) {

alert(JSON.stringify(res));

}

</script>

服务端返回如下（返回时即执行全局函数）：

onBack({"status": true, "user": "admin"})

2.）jquery ajax：

$.ajax({

url: 'http://www.domain2.com:8080/login',

type: 'get',

dataType: 'jsonp', // 请求方式为jsonp

jsonpCallback: "onBack", // 自定义回调函数名

data: {}

});

3.）vue.js：

this.$http.jsonp('http://www.domain2.com:8080/login', {

params: {},

jsonp: 'onBack'

}).then((res) => {

console.log(res);

})

后端node.js代码示例：

var querystring = require('querystring');

var http = require('http');

var server = http.createServer();

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

var params = qs.parse(req.url.split('?')[1]);

var fn = params.callback;

// jsonp返回设置

res.writeHead(200, { 'Content-Type': 'text/javascript' });

res.write(fn + '(' + JSON.stringify(params) + ')');

res.end();

});

server.listen('8080');

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

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：([www.domain.com/a.html)](https://link.juejin.im/?target=http%3A%2F%2Fwww.domain.com%2Fa.html))

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

<script>

document.domain = 'domain.com';

var user = 'admin';

</script>

2.）子窗口：([child.domain.com/b.html)](https://link.juejin.im/?target=http%3A%2F%2Fchild.domain.com%2Fb.html))

<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：([www.domain1.com/a.html)](https://link.juejin.im/?target=http%3A%2F%2Fwww.domain1.com%2Fa.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：([www.domain2.com/b.html)](https://link.juejin.im/?target=http%3A%2F%2Fwww.domain2.com%2Fb.html))

<iframe id="iframe" src="http://www.domain1.com/c.html" 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：([www.domain1.com/c.html)](https://link.juejin.im/?target=http%3A%2F%2Fwww.domain1.com%2Fc.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：([www.domain1.com/a.html)](https://link.juejin.im/?target=http%3A%2F%2Fwww.domain1.com%2Fa.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：([www.domain1.com/proxy....](https://link.juejin.im/?target=http%3A%2F%2Fwww.domain1.com%2Fproxy.html))  
中间代理页，与a.html同域，内容为空即可。

3.）b.html：([www.domain2.com/b.html)](https://link.juejin.im/?target=http%3A%2F%2Fwww.domain2.com%2Fb.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：([www.domain1.com/a.html)](https://link.juejin.im/?target=http%3A%2F%2Fwww.domain1.com%2Fa.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：([www.domain2.com/b.html)](https://link.juejin.im/?target=http%3A%2F%2Fwww.domain2.com%2Fb.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。  
带cookie请求：前后端都需要设置字段，另外需注意：所带cookie为跨域请求接口所在域的cookie，而非当前页。  
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cookie

xhr.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cookie

xhr.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cookie

xhr.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。

**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cookie

xhr.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.');

});

});