

**什么是CORS**

CORS（Cross-Origin Resource Sharing 跨源资源共享），当一个请求url的协议、域名、端口三者之间任意一与当前页面地址不同即为跨域。

例如最常见的，在一个域名下的网页中，调用另一个域名中的资源。相对于上面这种静态的调用方式，还可以通过Ajax技术来动态发起跨域请求。例如如下的方式，利用XMLHttpRequest对象发送一个GET请求，获取另一个域名下的图片内容。

<!DOCTYPE html>

<html>

<head>CORS Test</head>

<body>

<div id="img\_Div"></div>

<script type="text/javascript">

//XmlHttpRequest对象

function createXmlHttpRequest(){

if(window.ActiveXObject){ //如果是IE浏览器

return new ActiveXObject("Microsoft.XMLHTTP");

}else if(window.XMLHttpRequest){ //非IE浏览器

return new XMLHttpRequest();

}

}

function getFile() {

var img\_Container = document.getElementById("img\_Div");

var xhr = createXmlHttpRequest();

xhr.open('GET', 'http://oss.youkouyang.com/1.jpg', true);

xhr.setRequestHeader('Content-Type', 'image/jpeg');

xhr.responseType = "blob";

xhr.onload = function() {

if (this.status == 200) {

var blob = this.response;

var img = document.createElement("img");

img.onload = function(e) {

window.URL.revokeObjectURL(img.src);

};

img.src = window.URL.createObjectURL(blob);

img\_Container.appendChild(img);

}

}

xhr.send(null);

}

</script>

<div class="row">

<input type="button" onclick="getFile()" value="Get" />

</div>

</body>

</html>

**CORS的作用**

为了改善网络应用程序，开发人员要求浏览器供应商允许跨域请求。跨域请求主要用于：

调用XMLHttpRequest或fetchAPI通过跨站点方式访问资源网络字体，例如Bootstrap（通过CSS使用@font-face 跨域调用字体）通过canvas标签，绘制图表和视频。

**CORS的安全隐患**

跨域请求和Ajax技术都会极大地提高页面的体验，但同时也会带来安全的隐患，其中最主要的隐患来自于CSRF（Cross-site request forgery）跨站请求伪造。

**￼CSRF攻击的大致原理是：**

用户通过浏览器，访问正常网站A（例如某银行），通过用户的身份认证（比如用户名/密码）成功A网站。网站A产生Cookie信息并返回给用户的浏览器； 用户保持A网站页面登录状态，在同一浏览器中，打开一个新的TAB页访问恶意网站B；网站B接收到用户请求后，返回一些攻击性代码，请求A网站的资源（例如转账请求）；浏览器执行恶意代码，在用户不知情的情况下携带Cookie信息，向网站A发出请求。网站A根据用户的Cookie信息核实用户身份（此时用户在A网站是已登录状态），A网站会处理该请求，导致来自网站B的恶意请求被执行。

**CORS验证机制**

出于安全原因，浏览器限制从脚本中发起的跨域HTTP请求。默认的安全限制为同源策略， 即JavaScript或Cookie只能访问同域下的内容。

W3C推荐了一种跨域的访问验证的机制，即CORS（Cross-Origin Resource Sharing 跨源资源共享）。

这种机制让Web应用服务器能支持跨站访问控制，使跨站数据传输更加安全，减轻跨域HTTP请求的风险。

CORS验证机制需要客户端和服务端协同处理。

**CORS浏览器支持情况**

目前主流浏览器都已基本提供对跨域资源共享的支持，移动端浏览器也几乎全部支持。

￼客户端处理机制

基于上述的CSRF的风险，各主流的浏览器都会对动态的跨域请求进行特殊的验证处理。验证处理分为简单请求验证处理和预先请求验证处理。

简单请求

当请求同时满足下面两个条件时，浏览器会直接发送GET请求，在同一个请求中做跨域权限的验证。

请求方法是下列之一：

GETHEADPOST

请求头中的Content-Type请求头的值是下列之一：

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

简单请求时，浏览器会直接发送跨域请求，并在请求头中携带Origin 的header，表明这是一个跨域的请求。

服务器端接到请求后，会根据自己的跨域规则，通过Access-Control-Allow-Origin和Access-Control-Allow-Methods响应头，来返回验证结果。

如果验证成功，则会直接返回访问的资源内容。

￼如果验证失败，则返回403的状态码，不会返回跨域请求的资源内容。

￼可以通过浏览器的Console查看具体的验证失败原因

￼

预先请求

当请求满足下面任意一个条件时，浏览器会先发送一个OPTION请求，用来与目标域名服务器协商决定是否可以发送实际的跨域请求。

请求方法不是下列之一：

GETHEADPOST

请求头中的Content-Type请求头的值不是下列之一：

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

浏览器在发现页面中有上述条件的动态跨域请求的时候，并不会立即执行对应的请求代码，而是会先发送Preflighted requests（预先验证请求），Preflighted requests是一个OPTION请求，用于询问要被跨域访问的服务器，是否允许当前域名下的页面发送跨域的请求。

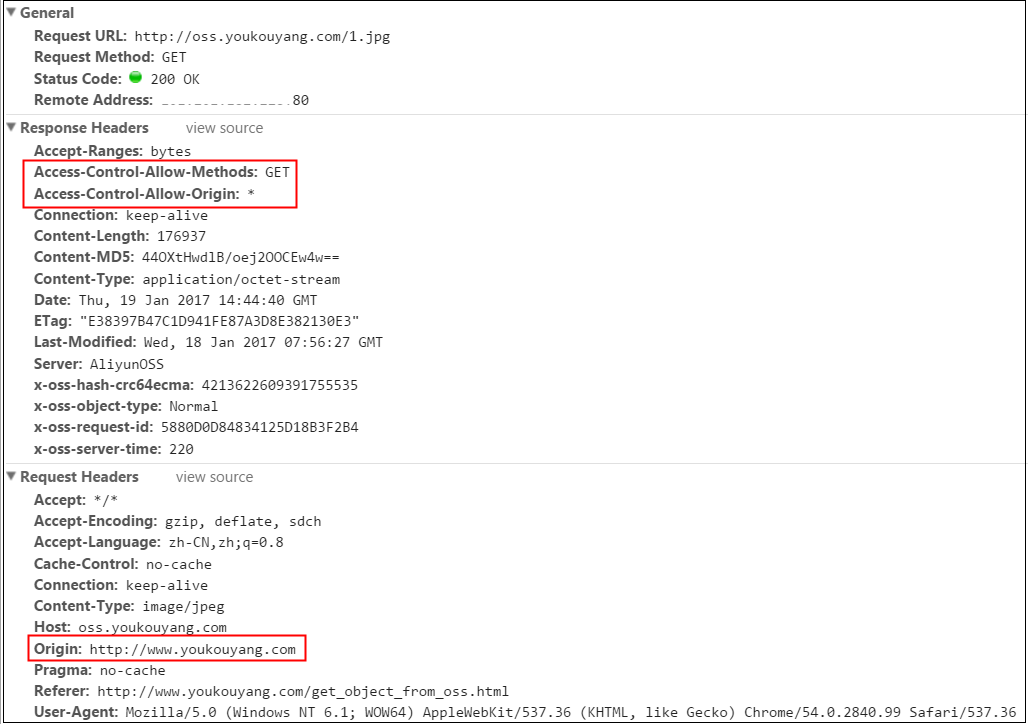
OPTIONS请求头部中会包含以下头部：Origin、Access-Control-Request-Method、Access-Control-Request-Headers。



服务器收到OPTIONS请求后，设置Access-Control-Allow-Origin、Access-Control-Allow-Method、Access-Control-Allow-Headers头部与浏览器沟通来判断是否允许这个请求。

如果Preflighted requests验证通过，浏览器才会发送真正的跨域请求。

如果Preflighted requests验证失败，则会返回403状态，浏览器不会发送真正的跨域请求。



 可以通过浏览器的Console查看具体的验证失败原因

**带认证的请求**

默认情况下，跨源请求不提供凭据(cookie、HTTP认证及客户端SSL证明等)。通过将withCredentials属性设置为true，可以指定某个请求应该发送凭据。

xhr.withCredentials = true;

如果服务器接收带凭据的请求，会用下面的HTTP头部来响应。

Access-Control-Allow-Credentials: true

服务器还可以在Preflight响应中发送这个HTTP头部，表示允许源发送带凭据的请求。

如果发送的是带凭据的请求，但服务器的响应中没有包含这个头，那么浏览器就不会把响应交给JavaScript(responseText中将是空字符串，size为0)。

注意，当withCredentials属性设置为true，需要response header中的'Access-Control-Allow-Origin'为一个确定的域名，而不能使用'\*'这样的通配符。

**服务端处理机制**

服务器端对于跨域请求的处理流程如下：

首先查看http头部有无origin字段；如果没有，或者不允许，直接当成普通请求处理，结束；如果有并且是允许的，那么再看是否是preflight(method=OPTIONS)；如果不是preflight（简单请求），就返回Allow-Origin、Allow-Credentials等，并返回正常内容。如果是preflight（预先请求），就返回Allow-Headers、Allow-Methods等，内容为空；

Access-Control-Allow-Origin

Access-Control-Allow-Origin头中携带了服务器端验证后的允许的跨域请求域名，可以是一个具体的域名或是一个\*（表示任意域名）。简单请求时，浏览器会根据此响应头的内容决定是否给脚本返回相应内容，预先验证请求时，浏览器会根据此响应头决定是否发送实际的跨域请求。

Access-Control-Max-Age

Access-Control-Max-Age用于告知浏览器可以将预先检查请求返回结果缓存的时间，在缓存有效期内，浏览器会使用缓存的预先检查结果判断是否发送跨域请求。

**2、jsonp**

跨域的安全限制都是对浏览器端来说的，服务器端是不存在跨域安全限制的。

浏览器的同源策略限制从一个源加载的文档或脚本与来自另一个源的资源进行交互。

如果协议，端口和主机对于两个页面是相同的，则两个页面具有相同的源，否则就是不同源的。

如果要在js里发起跨域请求，则要进行一些特殊处理了。或者，你可以把请求发到自己的服务端，再通过后台代码发起请求，再将数据返回前端。

最常见的一种跨域方式，其背后原理就是利用了script标签不受同源策略的限制，在页面中动态插入了script，script标签的src属性就是后端api接口的地址，并且以get的方式将前端回调处理函数名称告诉后端，后端在响应请求时会将回调返还，并且将数据以参数的形式传递回去。

前端:

//http://127.0.0.1:8888/jsonp.html

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

script.src = 'http://127.0.0.1:2333/jsonpHandler?callback=\_callback' document.body.appendChild(script); //插入script标签

//回调处理函数 \_callback

var \_callback = function(obj){

for(key in obj){

console.log('key: ' + key +' value: ' + obj[key]);

} }

后端:

//http://127.0.0.1:2333/jsonpHandle

r app.get('/jsonpHandler', (req,res) => {

let callback = req.query.callback;

let obj = { type : 'jsonp', name : 'weapon-x' };

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

res.end(callback + '(' + JSON.stringify(obj) + ')'); })

jquery的jsonp方式跨域请求：

服务端代码不变，js代码如下：最简单的方式，只需配置一个dataType:'jsonp'，就可以发起一个跨域请求。jsonp指定服务器返回的数据类型为jsonp格式，可以看发起的请求路径，自动带了一个callback=xxx，xxx是jquery随机生成的一个回调函数名称。

这里的success就跟上面的showData一样，如果有success函数则默认success()作为回调函数。

$.ajax({ 13 url: "http://localhost:9090/student", 14 type: "GET", 15 dataType: "jsonp", //指定服务器返回的数据类型 16 success: function (data) { 17 var result = JSON.stringify(data); //json对象转成字符串 18 $("#text").val(result); 19 }

jsonp的实现方式其实就是<script>脚本请求地址的方式一样，只是ajax的jsonp对其做了封装，所以可想而知，jsonp是不支持POST方式的。

**postMessage()**

方法允许来自不同源的脚本采用异步方式进行有限的通信，可以实现跨文本档、多窗口、跨域消息传递。

postMessage(data,origin)方法接受两个参数

1.data:要传递的数据，html5规范中提到该参数可以是JavaScript的任意基本类型或可复制的对象，然而并不是所有浏览器都做到了这点儿，部分浏览器只能处理字符串参数，所以我们在传递参数的时候需要使用JSON.stringify()方法对对象参数序列化，在低版本IE中引用json2.js可以实现类似效果。

2.origin：字符串参数，指明目标窗口的源，协议+主机+端口号[+URL]，URL会被忽略，所以可以不写，这个参数是为了安全考虑，postMessage()方法只会将message传递给指定窗口，当然如果愿意也可以建参数设置为"\*"，这样可以传递给任意窗口

!DOCTYPE html>

<html>

<head>

<title>Post Message</title>

</head>

<body>

<div style="width:200px; float:left; margin-right:200px;border:solid 1px #333;">

<div id="color">Frame Color</div>

</div>

<div>

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

</div>

<script type="text/javascript">

window.onload=function(){

window.frames[0].postMessage('getcolor','http://lslib.com');

}

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

var color=e.data;

document.getElementById('color').style.backgroundColor=color;

},false);

</script>

</body>

</html>

<http://test.com/index.html>

<!doctype html>

<html>

<head>

<style type="text/css">

html,body{

height:100%;

margin:0px;

}

</style>

</head>

<body style="height:100%;">

<div id="container" onclick="changeColor();" style="widht:100%; height:100%; background-color:rgb(204, 102, 0);">

click to change color

</div>

<script type="text/javascript">

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

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

if(e.source!=window.parent) return;

var color=container.style.backgroundColor;

window.parent.postMessage(color,'\*');

},false);

function changeColor () {

var color=container.style.backgroundColor;

if(color=='rgb(204, 102, 0)'){

color='rgb(204, 204, 0)';

}else{

color='rgb(204,102,0)';

}

container.style.backgroundColor=color;

window.parent.postMessage(color,'\*');

}

</script>

</body>

</html>

<http://lslib.com/lslib.html>

三、规避方法

**同源策略的限制如图**

**1、cookie的读取：** 如果两个网页的一级域名一样，二级域名不同。

此时可以通过设置document.domain来共享cookie,举个栗子： A网页域名：https:\\a.example.com B网页域名：https:\\b.example.com 此时A、B网页js脚本同时设置document.domain = "example.com"; 现在A页面设置cookie： docuemnt.cookie = "title=hello"; 此时B页面就可以通过 document.cookie读取到A页面设置的cookie。 还可以在服务器设置cookie的时候指定cookie所属域名为一级域名，这样在二级、三级域名下就不用做任何设置就可以读取到这个cookie 注意事项：

1、这种方式只适用于一级域名相同

2、只适用于cookie和iframe(在页面 http://www.example.com/a.html 中设置document.domain:

在页面 http://example.com/b.html 中也设置document.domain，而且这也是必须的，虽然这个文档的domain就是example.com,但是还是必须显示的设置document.domain的值：

这样我们就可以通过js访问到iframe中的各种属性和对象了。),localstorage与IndexDB无法使用。

**2.通过H5的postMessage读写其他窗口的localstorage.**举个例子 父窗口https://a.com 子窗口https://b.com 父窗口向子窗口发送消息： var win = document.getElementsByTagName('iframe')[0].contentWindow; var obj = { name: 'Jack' }; win.postMessage(JSON.stringify({key: 'storage', data: obj}), 'http://bbb.com'); 子窗口写入父窗口的localStorage window.onmessage = function(e){ if(e.orgin !== "https://b.com"){ return; } var data = JSON.parse(e.data); localstorage.setItem(data.key,JSON.stringify(data.data)); }

**3、DOM的获取**  三种规避方法：1、片段识别符；2、window.name；3、window.postMessage 1 片段识别符值的是url的#后面的部分，这一部分的改变不会引起页面的刷新，但这一个变化可以通过window.onhaschange监听到。因此父窗口可以把信息写到子窗口的片段标识符中。如： 父窗口发送信息 var src = orginURL + "#" + data; document.getElementById("myIFrame").src = src;

子窗口接受信息 window.onhaschange = function(){ var message = window.location.hash; } 同样地子窗口也可以改变父窗口片段标识符

2、window.name

window对象有个name属性，该属性有个特征：即在一个窗口(window)的生命周期内,窗口载入的所有的页面都是共享一个window.name的，每个页面对window.name都有读写的权限，window.name是持久存在一个窗口载入过的所有页面中的，并不会因新页面的载入而进行重置。如果有需要，其中的任

**WEBSOCKET**

首先Websocket是基于HTTP协议的，或者说借用了HTTP的协议来完成一部分握手。

解决了http请求的两个问题

**问题一**：Websocket出现了。他解决了HTTP的这几个难题。首先，被动性，当服务器完成协议升级后（HTTP->Websocket），服务端就可以主动推送信息给客户端啦。

这样就可以解决客服处理速度过慢的问题了。

同时，在传统的方式上，要不断的建立，关闭HTTP协议，由于HTTP是非状态性的，每次都要重新传输 identity info （鉴别信息），来告诉服务端你是谁。

虽然接线员很快速，但是每次都要听这么一堆，效率也会有所下降的，同时还得不断把这些信息转交给客服，不但浪费客服的处理时间，而且还会在网路传输中消耗过多的流量/时间。

**问题二**：但是Websocket只需要一次HTTP握手，所以说整个通讯过程是建立在一次连接/状态中，也就避免了HTTP的非状态性，服务端会一直知道你的信息，直到你关闭请求，这样就解决了接线员要反复解析HTTP协议，还要查看identity info的信息。

更详细的讲解：

<https://www.cnblogs.com/fuqiang88/p/5956363.html>