

42 跨域解决方案

更新时间：2020-10-14 10:57:35



“最聪明的人是最不愿浪费时间的人。——但丁”

跨域及跨域解决方案也是一个稳定性比较高的考点。关于跨域如何解决这个问题，不同的人有着不同的思路。笔者之前曾经尝试在团队内做过一次头脑风暴，想看看大家能想出多少种解决跨域的办法，最后竟然数出了近10种之多（可见程序员的脑洞之大，哈哈）。

不过，面试的时候，在跨域这块想要“出奇制胜”其实是一个不太合适的事情。一来强记太多的方法，脑壳容易懵，不利于你现场发挥；二来面试官多数情况下想听的也就只有那么几个，讲得太多，对方也会不耐烦（我个人就有被打断的经历）。本节我们针对面试官比较喜闻乐见的几个思路来讲，也鼓励大家私下里结合自己的开发经验补充自己的思路，但不鼓励死磕。

什么是跨域

跨域的故事，要从“同源策略”说起了。

这里的源(origin)指的是协议、域名、端口号，同源指的是在url中协议、域名、端口号均相同。那么同源策略是浏览器的一个安全功能，不同源的脚本在没有明确授权的情况下，不能读写对方资源。

注意这个“不能读写资源”的含义，它主要限制了以下三个方面：

- Cookie、LocalStorage 和 IndexedDB 无法读取
- DOM 和 JS 对象无法获取
- Ajax请求发送不出去

只要协议、域名、端口有任何一个不同，都被当作是不同的域，这就是所谓“跨域”。

虽然同源策略带来了安全上的保证，但是实际业务中，跨域的场景实在是太多了。如果仅仅因为跨域就导致资源无法互相读写，那么我们现在看到的许多互联网功能都将原地歇菜。之所以没歇菜，是因为网络策略有其灵活性，我们可以通过一些方式来绕过同源策略、达到通信目的。

跨域解决方案

JSONP

由于js调用跨域文件是被允许的。只要我们在远程服务器上设法把数据装进js格式的文件里，就可以供客户端调用和进一步处理。

在这个思路的指导下，人们挖掘出了 JSONP。JSONP 的一个要点就是允许用户传递一个callback参数给服务端，然后服务端返回数据时会将这个callback参数作为函数名来包裹住JSON数据，这样客户端就可以随意定制自己的函数来自动处理返回数据了。

举个?，我在 HTML 里面去调用一个服务端的 callback 函数：

```
<!DOCTYPE html>
<html>
  <body>
    <script type="text/javascript">
      // callback是存在服务端的
      function callback(data) {
        console.log(data)
      }
    </script>
    // 这里引入服务端代码
    <script src="http://www.imoooc.com/jsonp.shtml" type="text/javascript" charset="utf-8"></script>
  </body>
</html>
```

然后在 <http://www.imoooc.com/jsonp.shtml> （注意它是个虚拟链接，作示例用，大家不要贴到浏览器访问哈，估计是404），这个链接请求的服务端代码可以是这样写的：

```
/**
 * jsonp 测试
 * @return
 */
@RequestMapping(value="/jsonp",method=RequestMethod.GET)
@ResponseBody
public String Jsonp(){
    return "callback('我是目标字符串');";
}
```

这段代码是用 JAVA 写的，各位不必看懂 JAVA，只需要关注到它 return 的是个什么东西：

```
return "callback('我是目标字符串');";
```

没错，return 的就是一个对 callback 的调用！因此 JSONP 方案，大家只需要记住两个角色、两件事情：

1. 浏览器脚本——定义：定义 callback，callback内是读取数据的逻辑
2. 服务端——调用：输出对 callback 的调用，把目标数据作为入参传给 callback

如此一来，就可以成功地把 `{message:"success"}` 这个跨域的信息写入目标文件里啦。

CORS

CORS是一个W3C标准，全称是"跨域资源共享"（Cross-origin resource sharing）。

它允许浏览器向不同源的服务器，发出XMLHttpRequest请求。虽然需要浏览器和服务器同时支持，但目前来看，除了低版本 IE 外，基本所有浏览器都支持该功能。

CORS的通信过程，实际上不需要什么代码层面的配合与改动，由浏览器自动实现。

对于开发者来说，CORS通信与同源的通信没有差别，至少代码上是一样的。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、追加必要的请求，但用户不会有感觉。

浏览器的行为是通用的、自动化的。因此能否实现 CORS 的关键，其实在于服务器是否对此提供支持。我们下面从过程来理解一下浏览器和服务器在 CORS 上的合作机制：

简单请求对应的 CORS 行为

浏览器会把请求分为简单请求和非简单请求，对于这两种请求，CORS 的处理过程是不同的，我们先来看简单请求：

- 请求方式为HEAD、POST 或者 GET
- http头信息不超出以下字段：Accept、Accept-Language、Content-Language、Last-Event-ID、Content-Type(限于三个值：application/x-www-form-urlencoded、multipart/form-data、text/plain)

满足这两个条件的，就是简单请求。对于简单请求，浏览器直接发出CORS请求。具体来说，就是在头信息之中，增加一个Origin字段：

```
Origin: http://imooc.com
```

Origin字段用来说明，本次请求来自哪个源（协议 + 域名 + 端口）。服务器根据这个值，决定是否同意这次请求。服务器处理的结果，分为两种情况：

- 不同意：如果Origin指定的源，不在许可范围内，服务器会返回一个正常的HTTP回应；浏览器发现，这个回应的头信息没有包含Access-Control-Allow-Origin字段，就知道出错了，从而抛出一个错误，被XMLHttpRequest的onerror回调函数捕获。
- 同意：如果Origin指定的域名在许可范围内，服务器返回的响应，会多出这个关键的头信息字段：

```
Access-Control-Allow-Origin: http://imooc.com
```

这个字段用于说明服务器接纳哪些域名。它的值要么是请求时Origin字段的值，要么是一个*——表示接受任意域名的请求。

复杂请求对应的 CORS 行为

有一些请求对服务器有着特殊的要求，比如请求方法是PUT或DELETE，或者Content-Type字段的类型是application/json。

非简单请求的CORS请求，会在正式通信之前，增加一次HTTP查询请求，称为"预检"请求（preflight）。

这个 **preflight** 的作用在于，确认当前网页所在的域名是否在服务器的许可名单之中、明确可以使用的 **HTTP** 请求方法和头信息字段。只有在这个请求返回成功的情况下，浏览器才会发出正式的请求。

这样做的目的是为了“避免无用功”。要知道，一般来说，正式请求要携带一些信息，它体积可能比较大。如果我们背着这么大一个包袱到了服务端那边，却发现对方根本不接受你，那岂不是白费力气了嘛。所以说，发送正式请求前先“预检”，就跟结婚之前要先订婚一样，是一个必要的确认动作。

注：“预检”请求用的请求方法是 **OPTIONS**，这也是一个小小的考点。

CORS 和 JSONP 的对比

CORS 的优势，往往是相对于 **JSONP** 来说的：**JSONP**只支持**GET**请求，而**CORS**支持所有类型的**HTTP**请求。但相应地，**JSONP**在低版本 **IE** 上也可以畅通无阻，**CORS** 就没有这么好的兼容性了。

postMessage跨域

这个 **API** 从 **H5** 开始支持，通过注册监听信息的**Message**事件、调用发送信息的**postMessage**方法，我们可以实现跨窗口通信。

从广义上讲，一个窗口可以获得对另一个窗口的引用（比如 `targetWindow = window.opener`），然后在窗口上调用 `targetWindow.postMessage()` 方法分发一个 **MessageEvent** 消息。接收消息的窗口可以根据需要自由处理此事件。传递给 `window.postMessage()` 的参数（比如 `message`）将通过消息事件对象暴露给接收消息的窗口。

发送信息的postMessage方法

```
otherWindow.postMessage(message, targetOrigin, [transfer]);
```

这里`otherWindow`是对目标窗口的引用；`message`是要发送的消息；`targetOrigin`是限定消息接受范围，一般是字符串或者URI，星号 `*` 则意味着不限制。

接受信息的message事件

```
var onmessage = function(event) {
  var data = event.data;
  var origin = event.origin;
}
if(typeof window.addEventListener != 'undefined'){
  window.addEventListener('message', onmessage, false);
}else if(typeof window.attachEvent != 'undefined'){
  window.attachEvent('onmessage', onmessage);
}
```

流程演示

下面我们通过一个例子来理解这个过程：

a页面（<http://www.imooc.com/b.html>）中对消息接受和派发的处理：

```
// a 页面
<iframe id="iframe" src="http://www.imooc.com/b.html" style="display:none;"></iframe>
<script>
  var iframe = document.getElementById('iframe');
  iframe.onload = function() {
    var data = {
      name: 'xiuyan'
    };
    // a 页面向 b 页面派发消息
    iframe.contentWindow.postMessage(JSON.stringify(data), 'http://www.neal.cn');
  };

  // a 页面接受 b 页面的消息
  window.addEventListener("message", function( event ) {
    console.log('data from b is', event.data)
  });
</script>
```

b 页面（<http://www.imooc.com/b.html>）对消息接受和派发的处理：

```
<script>
  // 接收 a 页面的数据
  window.addEventListener('message', function(e) {
    console.log('data from a is', event.data)

    var data = JSON.parse(e.data);
    if (data) {
      data.age = 100;

      // 派发数据到 a 页面
      window.parent.postMessage(JSON.stringify(data), 'http://www.imooc.com');
    }
  }, false);
</script>
```

小结

关于跨域的解决方案，其实还有很多，比如 **iframe**（个人不推荐，如果有兴趣可以了解下）、**cookie** 跨域或者反向代理跨域等等。但是原则上，各位需要优先掌握本节提及的这几种思路。还是那句话，回答跨域问题，不要想着剑走偏锋，中规中矩是最妥帖的。

}