一个跨域问题给我整懵了

技术漫谈 2022-01-05 08:36

以下文章来源于业余码农 ，作者Amazing10

大家好，我是Levi。分享一篇来自字节的朋友的文章，希望对你有帮助。

最近在做全栈项目的时候，遇到一个问题。由于是前后端分离的项目，所以前后端实际上会运行在不同的域名上。如果是在本地开发，前后端也会分别部署在不同的端口。

这个时候，前端直接请求后端接口，就会遇到所谓的跨域问题。

```
Access to XMLHttpRequest at 'http://127.0.0.1:3001/user/info' from origin 'http://127.0.0.1:3000' has been blocked by user:1
CORS policy: Response to preflight request doesn't pass access control check: No 'Access-Control-Allow-Origin' header is
present on the requested resource.
```

跨域错误

同源策略

说到跨域，首先需要解释下为什么会出现这样的跨域问题。这其实都源于浏览器的同源策略。

同源策略是浏览器中的一个重要的安全策略，是 `Netscape` 公司在1995年引入。同源策略的作用就是为了限制不同源之间的交互，从而能够有效避免 `XSS` 、 `CSFR` 等浏览器层面的攻击。

同源指的是两个请求接口 `URL` 的协议（ `protocol` ）、域名（ `host` ）和端口（ `port` ）一致。

同源策略

比如以下例子：

URL	结果	原因
http://store.company.com/dir2/other.html	同源	只有路径不同
http://store.company.com/dir/inner/another.html	同源	只有路径不同

https://store.company.com/secure.html	失败	协议不同
http://store.company.com:81/dir/etc.html	失败	端口不同 (http:// 默认端口是80)
http://news.company.com/dir/other.html	失败	主机不同

同源与非同源接口

说到浏览器的攻击手段， `XSS` 指的是恶意攻击者往 `Web` 页面里插入恶意 `HTML` 代码，利用的是用户对指定网站的信任。

而 `CSFR` 指的是跨站请求伪造，是攻击者通过一些技术手段欺骗用户的浏览器去访问一个自己曾经认证过的网站并执行一些操作（如发邮件，发消息，甚至财产操作如转账和购买商品）。

由于浏览器曾经认证过，所以被访问的网站会认为是真正的用户操作而去执行。这利用了Web中用户身份验证的一个漏洞：**简单的身份验证只能保证请求是发自某个用户的浏览器，却不能保证请求本身是用户自愿发出的**。这实际上利用的是网站对用户网页浏览器的信任。

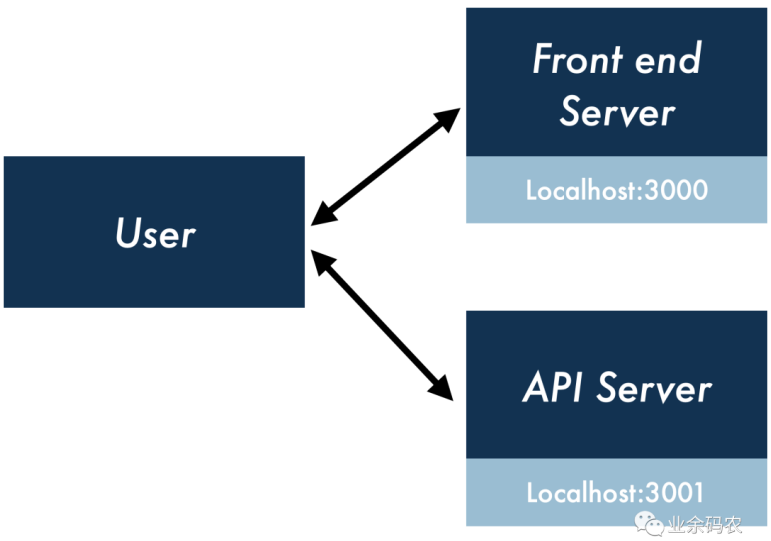
所以根据浏览器的是否同源判定，可以有选择的限制网站的一些行为。比如非同源的站点会被限制访问 `cookie` 、 `localStorage` 以及 `IndexDB` ，同时也无法获取网页 `DOM` 以及 `JavaScript` 对象，甚至 `AJAX` 的请求也会被拦截。

这样一来，就能够有效限制利用浏览器以及历史访问站点来进行攻击的目的。

跨域问题

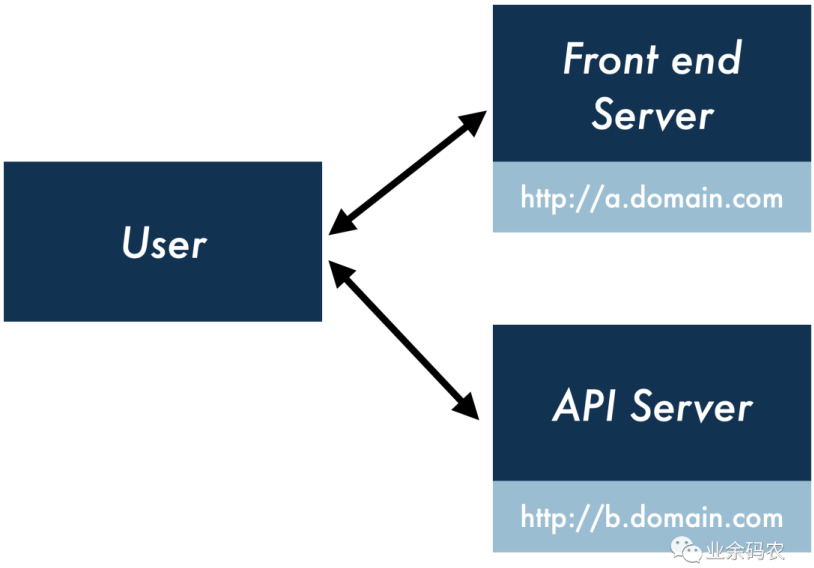
本质上浏览器不允许跨域请求好像是件好事，因为这样对于前端来说更安全。人家辛辛苦苦设计的同源策略，为啥会成为咱们的一个问题呢。

其实这也是没办法的事，毕竟前后端分离的项目，迫不得已就是需要进行跨域请求。比如在本地开发的环境中，很有可能端口号不一样。



本地环境中的跨域问题

在线上环境中，也同样会出现这样的情况。



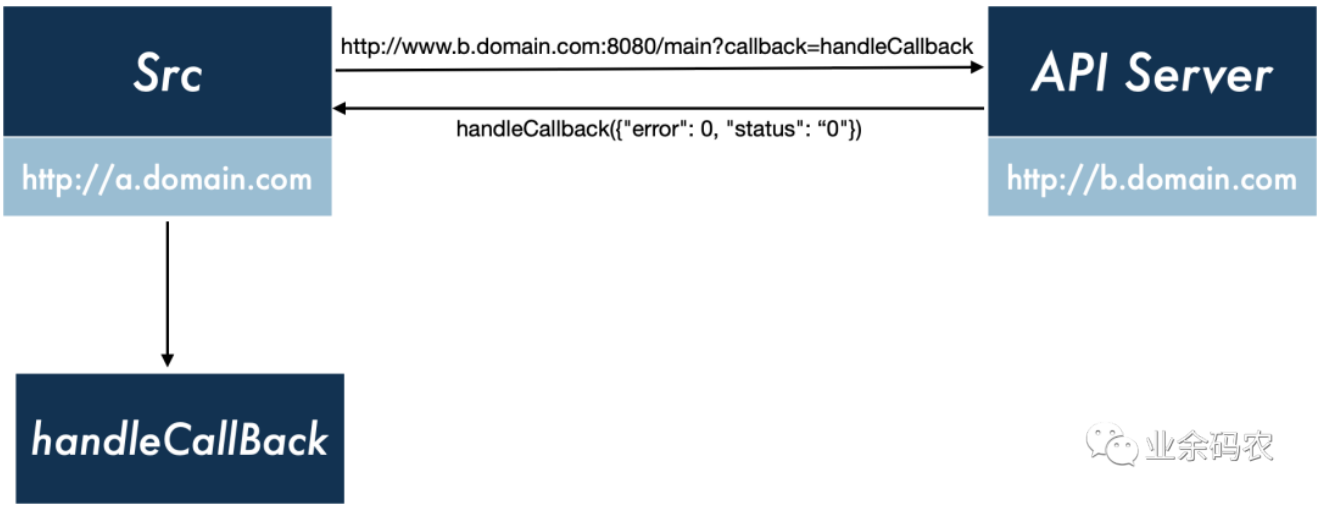
线上环境中的跨域问题

解决方案

1 JSONP跨域

上面所提到的跨域问题其实都是因为使用了 `AJAX` / `XMLHttpRequest` / `Fetch API` 的方式来发起请求，但是其实在Web页面上调用JS文件是不受跨域的影响的。不仅如此，拥有src属性的标签都拥有跨域的能力。

于是JSONP就是利用上述特点的跨域解决方案。JSONP的原理就是通过发送带有Callback参数的GET请求，服务端将接口返回数据拼凑到Callback函数中，返回给浏览器，浏览器解析执行，从而前端拿到Callback函数返回的数据。



JSONP跨域

前端代码只需要在页面中插入 `<script>` 标签，定义好回调函数，同时通过 `src` 请求后端接口即可：

```
<script>
    var script = document.createElement('script');
    script.type = 'text/javascript';
    script.src = 'http://www.b.domain.com:8080/main?callback=handleCallback';
    document.head.appendChild(script);

    // 回调函数
    function handleCallback(res) {
        alert(JSON.stringify(res));
    }
 </script>
```

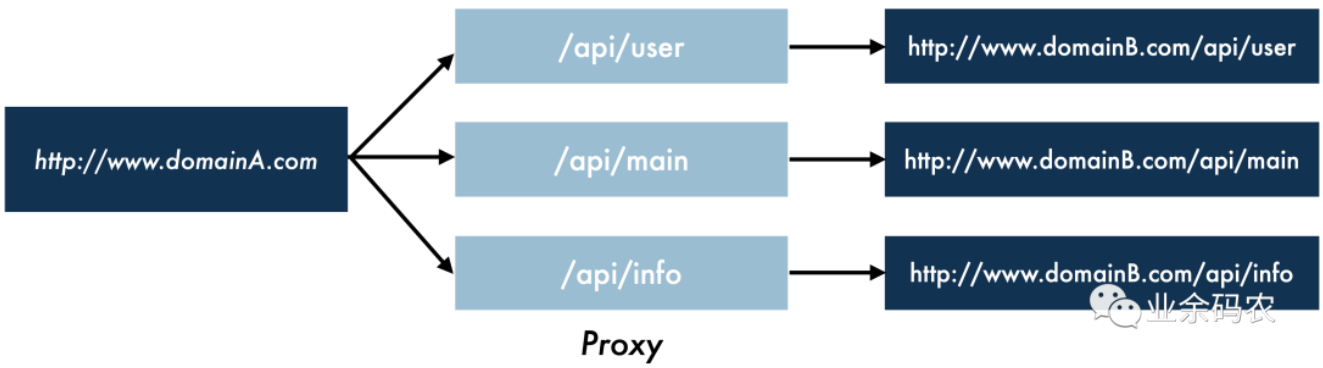
后端则需要在对应接口将客户端发送的 `callback` 参数作为函数名来包裹住 `JSON` 数据，返回数据至客户端。

```
handleCallback({"error": 0, "status": “0"})
```

`JSONP` 看起来很方便，但是实际上限制很大。由于 `script` 、 `img` 这些带 `src` 属性的标签，在引入外部资源时，使用的都是 `GET` 请求。所以 `JSONP` 也只能使用 `GET` 发送请求，这也是这种方式已经逐渐被淘汰的原因。

2 代理跨域

既然跨域问题是浏览器自己的一种保护措施，那么实际上能够通过在前后端之间加一道代理层来变相进行跨域请求。



代理跨域

Webpack Server代理

在 `webpack` 中可以通过配置 `proxy` 来快速获得接口代理的能力，同时前端请求的 `URL` 不需要带域名，代理服务器会自动自动将请求映射为同域请求。

img

可在前端 `webpack.config.js` 配置代理：

```
module.exports = {
  ...
  output: {...},
  devServer: {
    port: 3000,
    proxy: {
      "/api": {
        target: "http://localhost:3001"
      }
    }
  },
  plugins: []
};
```

Nginx反向代理

实现思路其实与 `webpack` 代理一致，无非是通过 `Nginx` 作为跳板机而已。

Nginx反向代理

举个 `Nginx` 配置的例子，就是把本地端口 `3000` 代理到 `3001`，这样在本地就能够进行跨域调试了。

```
```

```
server {
    listen       3000;
    server_name  localhost;

    location /api {
        proxy_pass   http://localhost:3001;  #反向代理
    }
}
```

Node中间件代理

原理都是类似的，只不过是将代理操作设置在了后端。若是 `node` 项目的话，可以直接利用 `http-proxy-middleware` 插件进行代理。本质上 `webpack` 也是用这个包做代理服务的，只不过现在把这个放在服务端。

node中间件代理

一个 `node` + `express` + `http-proxy-middleware` 的例子：

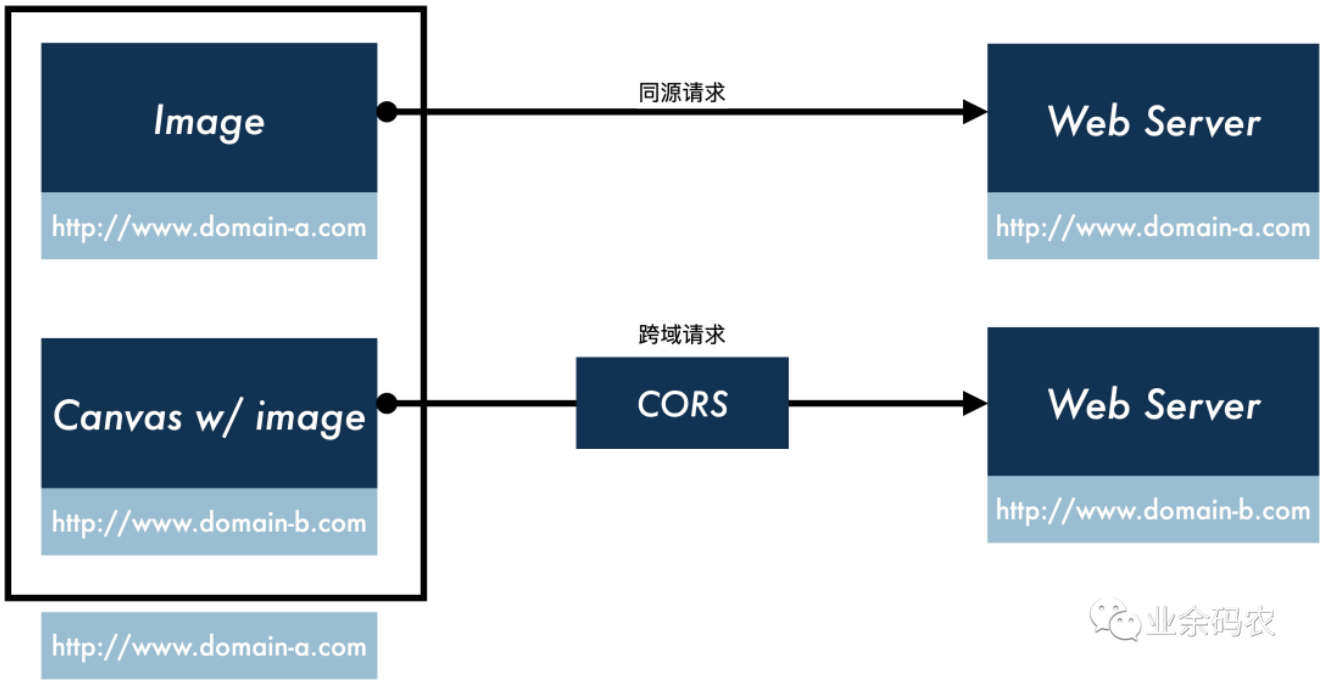
```
var express = require('express');
var proxy = require('http-proxy-middleware');
var app = express();

app.use('/', proxy({
    // 代理跨域目标接口
    target: 'http://localhost:3001',
    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', 'localhost');
        res.header('Access-Control-Allow-Credentials', 'true');
    },
}));

app.listen(3000);
```

3 CORS跨域

CORS（`Cross-Origin Resource Sharing`）是指跨域资源共享，它是一个浏览器侧的机制，能够允许服务器标示除了它自己以外的其它域，这样浏览器就可以进行跨域访问加载资源。

一般现代浏览器都支持`CORS`跨域，只有那种古老的浏览器，比如`IE10`以下的才不支持。

这意味着，实际上浏览器虽然会采取同源策略来限制跨域访问，但是同时又给服务端提供了一个选择，即通过`CORS`来可选的提供跨域能力。



CORS跨域

比如上图这个例子，左边代表的是前端网页，右边代表的是服务器。前端部署在`domain-a`域名下，但是有两个资源需要请求来自不同域名的资源。

当资源来源与前端本身所在的域不一致时，便会发生跨域请求。此时可通过`CORS`来控制是否允许进行跨域资源的请求。

`CORS`是浏览器提供的能力，而实现`CORS`的控制和通信是在服务端进行。也就是只要服务端对相应域允许`CORS`，那么便可进行跨域通信。

简单请求

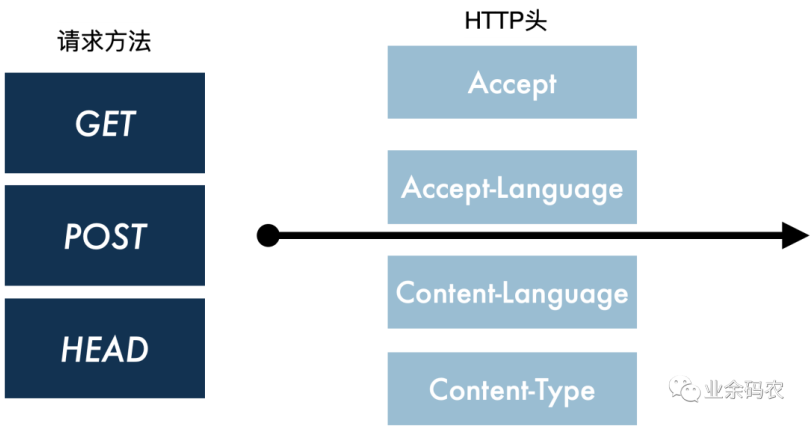
浏览器根据请求方法以及`HTTP`头部信息将`CORS`请求分成两类，简单请求和非简单请求。

若满足下列条件，则视为简单请求：

- 请求方法属于`GET`、`POST`、`HEAD`中的一种

- HTTP头部仅包含 `Accept` 、 `Accept-Language` 、 `Content-Language` 、 `Content-Type` 。

其中Content-Type的值仅限于 `text/plain` `multipart/form-data` `application/x-www-form-urlencoded`

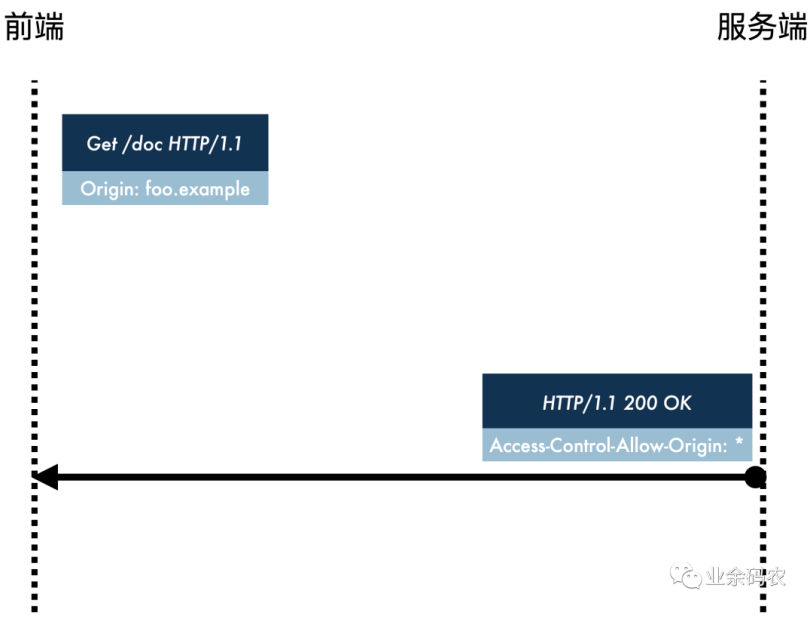


简单请求

- 请求中的任意 `XMLHttpRequestUpload` 对象均没有注册任何事件监听器； `XMLHttpRequestUpload` 对象可以使用 `XMLHttpRequest.upload` 属性访问。

- 请求中没有使用 `ReadableStream` 对象。

简单请求的流程很简单：



简单请求流程

- 浏览器发出 `CORS` 请求时，在头部添加 `Origin` 字段(最后一行)，表明请求域：比如

```
GET /resources/public-data/ HTTP/1.1
Host: bar.other
User-Agent:
```

```
Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5;
en-US; rv:1.9.1b3pre) Gecko/20081130 Minefield/3.1b3pre
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Connection: keep-alive
Referer: http://foo.example/examples/access-control/simpleXSInvocation.html
Origin: http://foo.exampl
```

服务端收到Origin，决定是否同意跨域请求:

- 如果同意请求，服务端会在返回的响应中添加 `CORS` 相关的头部，比如其中 `Access-Control-Allow-Origin` 是必须字段，表示能接受请求的域名， `*` 表示任意域名。
- 若不同意请求，服务端会返回一个正常的 `HTTP` 响应，由于不包含 `Access-Control-Allow-Origin` ，会被浏览器发现，从而抛出本文最上面的错误。

```
HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 00:23:53 GMT
Server: Apache/2.0.61
Access-Control-Allow-Origin: *
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: application/xml
```

非简单请求

不满足简单请求之外的请求，都是非简单请求。

非简单请求的特点是，在发送正式请求之前，会先发起一个预检请求。只要当服务端同意预检请求时，才会发送正式的请求。

比如这里举一个例子，假设需要发送一个请求，该请求包含自定义的头部字段 `X-PINGOTHER` ，同时 `Content-Type` 为 `application/xml` 。

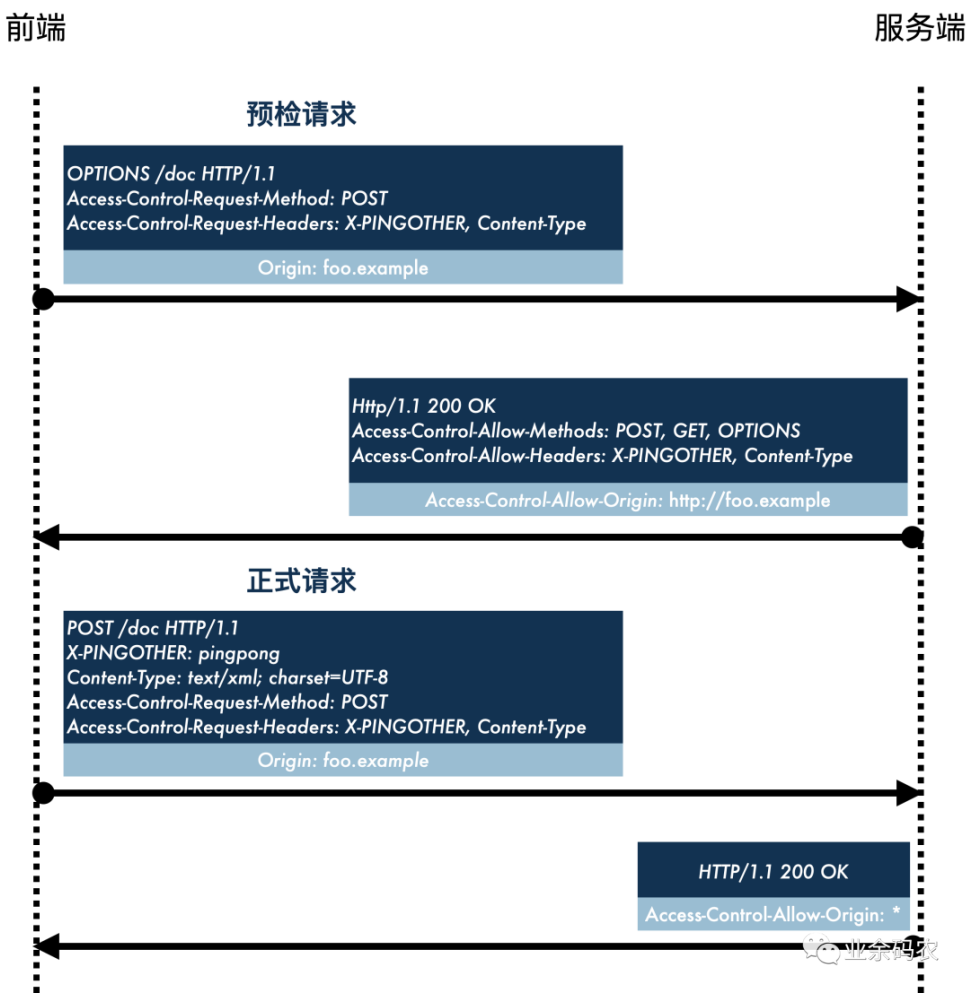
可以看出这个请求是妥妥的非简单请求，所以需要走预检流程。

- 预检请求用的是 `OPTIONS` 请求方法，在请求头部会表明 `Origin` ，同时还需要带上两个特殊的头部字段。
    - `Access-Control-Request-Method` ：用于表明正式请求会用到哪些 `HTTP` 请求方法，比如例子中的 `POST` ;

- `Access-Control-Request-Headers`：用于表明正式请求会用哪些额外发送的头部字段，比如例子中的 `X-PINGOTHER` 和 `Content-Type` 。

- 服务端接收到预检请求后，会检查上面这几个字段，确定是否可以接受跨域请求。如果可以，就会在响应的头部中添加 `Access-Control-Request-Method` 和 `Access-Control-Request-Headers` 字段用来表示可接受的请求方法和请求头。

- 浏览器接收到了预检成功的响应后，才会开始发起正式请求，正式请求的过程就跟简单请求基本一致了。也就是在请求头中添加 `Origin` 字段，同时服务端的响应也返回相应的 `CORS` 必须字段。



非简单请求流程

虽然说 `CORS` 跨域的方案是浏览器支持的机制，但是实现确实在服务端。但是其实工作量并不大，只需要设置允许跨域的域名、 `HTTP` 头部以及请求方式等参数即可。

比如在 `node` + `express` 的项目只需要添加以下代码就可以实现任意域名跨域的目的。

```
app.all('*', function (req: express.Request, res: express.Response,
    next: express.NextFunction) {
  //设置允许跨域的域名，*代表允许任意域名跨域
  res.header("Access-Control-Allow-Origin", "*");
```

```
  //允许的header类型
  res.header("Access-Control-Allow-Headers", "*");
  //跨域允许的请求方式
  res.header("Access-Control-Allow-Methods", "DELETE,PUT,POST,GET,OPTIONS");
  if (req.method.toLowerCase() === 'options')
    res.sendStatus(200)  //让options尝试请求快速结束
  else
    next()
})
```

总结

跨域问题是 `Web` 开发中很常见的问题，解决起来其实也并不复杂。除了上面的方案，也还存在像 `Iframe` 、 `postMessag` e以及 `websocket` 等方案。

但是总的来说不如上面这三种常用，一个比较正经的前后端分离项目更多的还是使用 `CORS` 方案进行跨域。省时省力又省心。

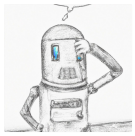
喜欢此内容的人还喜欢

ChatGPT全球爆火，它会取代人类吗？（免注册使用教程）

AIGC聊天机器人

微信把玩ChatGPT后的一些感想

相学长



ORCAD原理图检查17.4

便件小白