写Bug RP 1.3k
2018-07-11 发布

不要再问我跨域的问题了

> 写下这篇文章后我想，要不以后就把这种基础的常见知识都归到这个“不要再问我XX的问题”，形成一系列内容，希望大家看完之后再有人问你这些问题，你心里会窃喜：“嘿嘿，是时候展现真正的技术了！”
> 一、不要再问我this的指向问题了

跨域这两个字就像一块狗皮膏药一样黏在每一个前端开发者身上，无论你在工作上或者面试中无可避免会遇到这个问题。为了应付面试，我每次都随便背几个方案，也不知道为什么要这样干，反正面完就可以扔了，我想工作上也不会用到那么多乱七八糟的方案。到了真正工作，开发环境有webpack-dev-server搞定，上线了服务端的大佬们也会配好，配了什么我不管，反正不会跨域就是了。日子也就这么混过去了，终于有一天，我觉得不能再继续这样混下去了，我一定要彻底搞懂这个东西！于是就有了这篇文章。

要掌握跨域，首先要知道为什么会有跨域这个问题出现

确实，我们这种搬砖工人就是为了混口饭吃嘛，好好的调个接口告诉我跨域了，这种阻碍我们轻松搬砖的事情真恶心！为什么会跨域？是谁在搞事情？为了找到这个问题的始作俑者，请点击浏览器的同源策略。
这么官方的东西真难懂，没关系，至少你知道了，因为浏览器的同源策略导致了跨域，就是浏览器在搞事情。
所以，浏览器为什么要搞事情？就是不想给好日子我们过？对于这样的质问，浏览器甩锅道：“同源策略限制了从同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重要安全机制。”
这么官方的话术真难懂，没关系，至少你知道了，似乎这是个安全机制。
所以，究竟为什么需要这样的安全机制？这样的安全机制解决了什么问题？别急，让我们继续研究下去。

没有同源策略限制的两大危险场景

据我了解，浏览器是从两个方面去做这个同源策略的，一是针对接口的请求，二是针对Dom的查询。试想一下没有这样的限制上述两种动作有什么危险。

没有同源策略限制的接口请求

有一个小小的东西叫cookie大家应该知道，一般用来处理登录等场景，目的是让服务端知道谁发出的这次请求。如果你请求了接口进行登录，服务端验证通过后会在响应头加入Set-Cookie字段，然后下次再发请求的时候，浏览器会自动将cookie附加在HTTP请求的头字段Cookie中，服务端就能知道这个用户已经登录过了。知道这个之后，我们来看场景：
1.你准备去清空你的购物车，于是打开了买买买网站www.maimaimai.com，然后登录成功，一看，购物车东西这么少，不行，还得买多点。
2.你在看有什么东西买的过程中，你的好基友发给你一个链接www.nidongde.com，一脸yin笑地跟你说：“你懂的”，你毫不犹豫打开了。

3.你饶有兴致地浏览着www.nidongde.com，谁知这个网站暗地里做了些不可描述的事情！由于没有同源策略的限制，它向www.maimaimai.com发起了请求！聪明的你一定想到上面的话“服务端验证通过后会在响应头加入Set-Cookie字段，然后下次再发请求的时候，浏览器会自动将cookie附加在HTTP请求的头字段Cookie中”，这样一来，这个不法网站就相当于登录了你的账号，可以为所欲为了！如果这不是一个买买买账号，而是你的银行账号，那.....
这就是传说中的CSRF攻击浅谈CSRF攻击方式。
看了这波CSRF攻击我在想，即使有了同源策略限制，但cookie是明文的，还不是一样能拿下来。于是我看了一些cookie相关的文章聊一聊 cookie、Cookie/Session的机制与安全，知道了服务端可以设置httpOnly，使得前端无法操作cookie，如果没有这样的设置，像XSS攻击就可以去获取到cookieWeb安全测试之XSS；设置secure，则保证在https的加密通信中传输以防截获。

没有同源策略限制的Dom查询

1.有一天你刚睡醒，收到一封邮件，说是你的银行账号有风险，赶紧点进www.yinghang.com改密码。你吓尿了，赶紧点进去，还是熟悉的银行登录界面，你果断输入你的账号密码，登录进去看看钱有没有少了。
2.睡眼朦胧的你没看清楚，平时访问的银行网站是www.yinhang.com，而现在访问的是www.yinghang.com，这个钓鱼网站做了什么呢?

```
// HTML
<iframe name="yinhang" src="www.yinhang.com"></iframe>
// JS
// 由于没有同源策略的限制，钓鱼网站可以直接拿到别的网站的Dom
const iframe = window.frames['yinhang']
const node = iframe.document.getElementById('你输入账号密码的Input')
console.log(`拿到了这个${node}，我还拿不到你刚刚输入的账号密码吗`)
```

由此我们知道，同源策略确实能规避一些危险，不是说有了同源策略就安全，只是说同源策略是一种浏览器最基本的安全机制，毕竟能提高一点攻击的成本。其实没有刺不穿的盾，只是攻击的成本和攻击成功后获得的利益成不成正比。

跨域正确的打开方式

经过对同源策略的了解，我们应该要消除对浏览器的误解，同源策略是浏览器做的一件好事，是用来防御来自邪门歪道的攻击，但总不能为了不让坏人进门而把全部人都拒之门外吧。没错，我们这种正人君子只要打开方式正确，就应该可以跨域。
下面将一个个演示正确打开方式，但在此之前，有些准备工作要做。为了本地演示跨域，我们需要：
1.随便跑起一份前端代码（以下前端是随便跑起来的vue），地址是http://localhost:9099。
2.随便跑起一份后端代码（以下后端是随便跑起来的node koa2），地址是http://localhost:9971。

同源策略限制下接口请求的正确打开方式

1.JSONP

在HTML标签里，一些标签比如script、img这样的获取资源的标签是没有跨域限制的，利用这一点，我们可以这样干：

后端写个小接口

```
// 处理成功失败返回格式的工具
const {successBody} = require('../utli')
class CrossDomain {
  static async jsonp (ctx) {
```

```
    // 前端传过来的参数
    const query = ctx.request.query
    // 设置一个cookies
    ctx.cookies.set('tokenId', '1')
    // query.cb是前后端约定的方法名字，其实就是后端返回一个直接执行的方法给前端，由于前端是用script标签发起的请求
    ctx.body = `${query.cb}(${JSON.stringify(successBody({msg: query.msg}, 'success'))})`
  }
}
module.exports = CrossDomain
```

简单版前端

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
  </head>
  <body>
    <script type='text/javascript'>
      // 后端返回直接执行的方法，相当于执行这个方法，由于后端把返回的数据放在方法的参数里，所以这里能拿到res。
      window.jsonpCb = function (res) {
        console.log(res)
      }
    </script>
    <script src='http://localhost:9871/api/jsonp?msg=helloJsonp&cb=jsonpCb' type='text/javascript
  </body>
</html>
```

简单封装一下前端这个套路

```
/**
 * JSONP请求工具
 * @param url 请求的地址
 * @param data 请求的参数
 * @returns {Promise<any>}
 */
const request = ({url, data}) => {
  return new Promise((resolve, reject) => {
    // 处理传参成xx=yy&aa=bb的形式
    const handleData = (data) => {
      const keys = Object.keys(data)
      const keysLen = keys.length
      return keys.reduce((pre, cur, index) => {
        const value = data[cur]
        const flag = index !== keysLen - 1 ? '&' : ''
        return `${pre}${cur}=${value}${flag}`
      }, '')
    }
    // 动态创建script标签
    const script = document.createElement('script')
    // 接口返回的数据获取
    window.jsonpCb = (res) => {
      document.body.removeChild(script)
      delete window.jsonpCb
```

2.空iframe加form

细心的朋友可能发现，JSONP只能发GET请求，因为本质上script加载资源就是GET，那么如果要发POST请求怎么办呢?

后端写个小接口

```
// 处理成功失败返回格式的工具
const {successBody} = require('../utli')
class CrossDomain {
  static async iframePost (ctx) {
    let postData = ctx.request.body
    console.log(postData)
    ctx.body = successBody({postData: postData}, 'success')
  }
}
module.exports = CrossDomain
```

前端

```
const requestPost = ({url, data}) => {
  // 首先创建一个用来发送数据的iframe.
  const iframe = document.createElement('iframe')
  iframe.name = 'iframePost'
  iframe.style.display = 'none'
  document.body.appendChild(iframe)
  const form = document.createElement('form')
  const node = document.createElement('input')
  // 注册iframe的load事件处理程序,如果你需要在响应返回时执行一些操作的话.
  iframe.addEventListener('load', function () {
    console.log('post success')
  })

  form.action = url
  // 在指定的iframe中执行form
  form.target = iframe.name
  form.method = 'post'
  for (let name in data) {
    node.name = name
    node.value = data[name].toString()
    form.appendChild(node.cloneNode())
  }
  // 表单元素需要添加到主文档中.
  form.style.display = 'none'
```

3.CORS

CORS是一个W3C标准，全称是"跨域资源共享"（Cross-origin resource sharing）跨域资源共享 CORS 详解。看名字就知道这是处理跨域问题的标准做法。CORS有两种请求，简单请求和非简单请求。

> 这里引用上面链接阮一峰老师的文章说明一下简单请求和非简单请求。
> 浏览器将CORS请求分成两类：简单请求（simple request）和非简单请求（not-so-simple request）。

只要同时满足以下两大条件，就属于简单请求。

（1) 请求方法是以下三种方法之一：

- HEAD
- GET
- POST

（2）HTTP的头信息不超出以下几种字段：

- Accept
- Accept-Language
- Content-Language
- Last-Event-ID
- Content-Type：只限于三个值application/x-www-form-urlencoded、multipart/form-data、text/plain

1.简单请求

后端

```
// 处理成功失败返回格式的工具
const {successBody} = require('../utli')
class CrossDomain {
  static async cors (ctx) {
    const query = ctx.request.query
    // *时cookie不会在http请求中带上
    ctx.set('Access-Control-Allow-Origin', '*')
    ctx.cookies.set('tokenId', '2')
    ctx.body = successBody({msg: query.msg}, 'success')
  }
}
module.exports = CrossDomain
```

前端什么也不用干，就是正常发请求就可以，如果需要带cookie的话，前后端都要设置一下，下面那个非简单请求例子会看到。

```
fetch(`http://localhost:9871/api/cors?msg=helloCors`).then(res => {
  console.log(res)
})
```

2.非简单请求

非简单请求会发出一次预检测请求，返回码是204，预检测通过才会真正发出请求，这才返回200。这里通过前端发请求的时候增加一个额外的headers来触发非简单请求。

后端

```
// 处理成功失败返回格式的工具
const {successBody} = require('../utli')
class CrossDomain {
  static async cors (ctx) {
```

```
    const query = ctx.request.query
    // 如果需要http请求中带上cookie，需要前后端都设置credentials，且后端设置指定的origin
    ctx.set('Access-Control-Allow-Origin', 'http://localhost:9099')
    ctx.set('Access-Control-Allow-Credentials', true)
    // 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）
    // 这种情况下除了设置origin，还需要设置Access-Control-Request-Method以及Access-Control-Request-Hea
    ctx.set('Access-Control-Request-Method', 'PUT,POST,GET,DELETE,OPTIONS')
    ctx.set('Access-Control-Allow-Headers', 'Origin, X-Requested-With, Content-Type, Accept, t')
    ctx.cookies.set('tokenId', '2')

    ctx.body = successBody({msg: query.msg}, 'success')
  }
}
module.exports = CrossDomain
```

一个接口就要写这么多代码，如果想所有接口都统一处理，有什么更优雅的方式呢？见下面的koa2-cors。

```
const path = require('path')
const Koa = require('koa')
const koaStatic = require('koa-static')
const bodyParser = require('koa-bodyparser')
const router = require('./router')
const cors = require('koa2-cors')
const app = new Koa()
const port = 9871
app.use(bodyParser())
// 处理静态资源 这里是前端build好之后的目录
app.use(koaStatic(
  path.resolve(__dirname, '../dist')
))
// 处理cors
app.use(cors({
  origin: function (ctx) {
    return 'http://localhost:9099'
  },
  credentials: true,
  allowMethods: ['GET', 'POST', 'DELETE'],
  allowHeaders: ['t', 'Content-Type']
}))
// 路由
app.use(router.routes()).use(router.allowedMethods())
```

前端

```
fetch(`http://localhost:9871/api/cors?msg=helloCors`, {
  // 需要带上cookie
  credentials: 'include',
  // 这里添加额外的headers来触发非简单请求
  headers: {
    't': 'extra headers'
  }
}).then(res => {
  console.log(res)
})
```

4.代理

想一下，如果我们请求的时候还是用前端的域名，然后有个东西帮我们把这个请求转发到真正的后端域名上，不就避免跨域了吗？这时候，Nginx出场了。

Nginx配置

```
server{
    # 监听9099端口
    listen 9099;
    # 域名是localhost
    server_name localhost;
    #凡是localhost:9099/api这个样子的，都转发到真正的服务端地址http://localhost:9871
    location ^~ /api {
        proxy_pass http://localhost:9871;
    }
}
```

前端就不用干什么事情了，除了写接口，也没后端什么事情了

```
// 请求的时候直接用回前端这边的域名http://localhost:9099，这就不会跨域，然后Nginx监听到凡是localhost:9099/a
fetch('http://localhost:9099/api/iframePost', {
  method: 'POST',
  headers: {
    'Accept': 'application/json',
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({
    msg: 'helloIframePost'
  })
})
```

Nginx转发的方式似乎很方便！但这种使用也是看场景的，如果后端接口是一个公共的API，比如一些公共服务获取天气什么的，前端调用的时候总不能让运维去配置一下Nginx，如果兼容性没问题（IE 10或者以上），CROS才是更通用的做法吧。

同源策略限制下Dom查询的正确打开方式

1.postMessage

window.postMessage() 是HTML5的一个接口，专注实现不同窗口不同页面的跨域通讯。

为了演示方便，我们将hosts改一下：127.0.0.1 crossDomain.com，现在访问域名crossDomain.com就等于访问127.0.0.1。

这里是http://localhost:9099/#/crossDomain，发消息方

```
<template>
  <div>
    <button @click="postMessage">给http://crossDomain.com:9099发消息</button>
    <iframe name="crossDomainIframe" src="http://crossdomain.com:9099"></iframe>
  </div>
</template>
```

```
<script>
export default {
  mounted () {
    window.addEventListener('message', (e) => {
      // 这里一定要对来源做校验
      if (e.origin === 'http://crossdomain.com:9099') {
        // 来自http://crossdomain.com:9099的结果回复
        console.log(e.data)
      }
    })
  },
  methods: {
    // 向http://crossdomain.com:9099发消息
    postMessage () {
      const iframe = window.frames['crossDomainIframe']
```

这里是http://crossdomain.com:9099，接收消息方

```
<template>
  <div>
    我是http://crossdomain.com:9099
  </div>
</template>

<script>
export default {
  mounted () {
    window.addEventListener('message', (e) => {
      // 这里一定要对来源做校验
      if (e.origin === 'http://localhost:9099') {
        // http://localhost:9099发来的信息
        console.log(e.data)
        // e.source可以是回信的对象，其实就是http://localhost:9099窗口对象(window)的引用
        // e.origin可以作为targetOrigin
        e.source.postMessage(`我是[http://crossdomain.com:9099]，我知道了兄弟，这就是你想知道的结果：
      }
    })
  }
}
</script>
```

结果可以看到：

```
我是[http://localhost:9099]，麻烦你查一下你那边有没有id为app的Dom                    index.vue?d1bf:36
我是[http://crossdomain.com:9099]，我知道了兄弟，这就是你想知道的结果：有id为app的Dom   index.vue?8d06:117
```

2.document.domain

这种方式只适合主域名相同，但子域名不同的iframe跨域。

比如主域名是http://crossdomain.com:9099，子域名是http://child.crossdomain.com:9099，这种情况下给两个页面指定一下document.domain即document.domain = crossdomain.com就可以访问各自的window对象了。

3.canvas操作图片的跨域问题

这个应该是一个比较冷门的跨域问题，张大神已经写过了我就不再班门弄斧了解决canvas图片getImageData,toDataURL

跨域问题

最后

希望看完这篇文章之后，再有人问跨域的问题，你可以嘴角微微上扬，冷笑一声：“不要再问我跨域的问题了。”扬长而去。

你可能感兴趣的

你可能感兴趣的文章

- **如何真正理解用Nginx代理来解决同源策略** 小纯纯 前端
- **跨域请求** kowalski javascript html
- **跨域问题之ajax** 弩马 ajax跨域 跨域
- **再也不学AJAX了！（三）跨域获取资源 ② - JSONP & CORS** libinfs 跨域 ajax javascript
- **解决ajax跨域问题** 萧逸 javascript
- **跨域问题汇总** 陈凤娟 javascript html5
- **iframe跨域POST提交** 枫s的思念 javascript post form
- **跨域，了解一下** 如水 javascript html5

50 条评论 默认排序 时间排序

鲸落丶 · 7月11日

工作中确实不怎么遇到，但是遇到一次就很头疼了，233

赞 回复