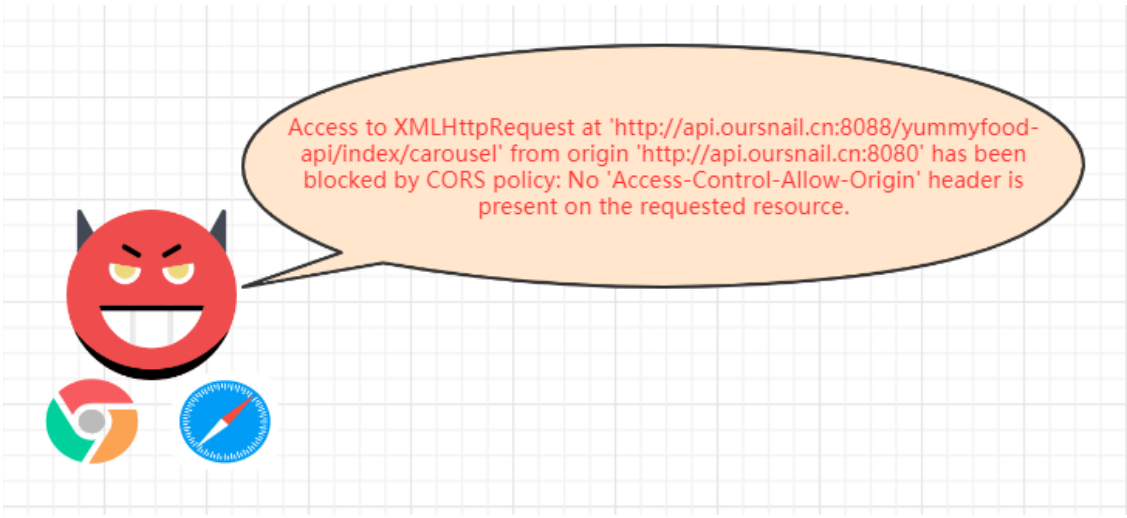
跨域问题是软件开发中经常遇到的一个问题，我们必须理解其由来，以及解决方案。

一、跨域问题的由来

在我私人的一个项目中，有在后端做过如下配置：

```
# 防止老要修改，我把能加的都加了一遍
config.addAllowedOrigin("http://localhost:8080");
config.addAllowedOrigin("http://111.231.119.253:8080");
config.addAllowedOrigin("http://111.231.119.253:80");
config.addAllowedOrigin("http://www.oursnail.cn:8080");
config.addAllowedOrigin("http://www.oursnail.cn:80");
config.addAllowedOrigin("http://oursnail.cn:8080");
config.addAllowedOrigin("http://oursnail.cn:80");
```

为什么要做这个配置呢？原因是当时的前端工程是部署在8080端口的tomcat下，而后端的接口是部署在8088端口的tomcat下，从8080访问8088，实际上属于跨域，下面会再说，如果不做以上配置，前端工程就无法正常请求后端接口数据。提示报错类似于：



追根到底，这个跨域问题是从哪里出来的呢？那就是 浏览器的同源策略 。

什么叫做同源呢？

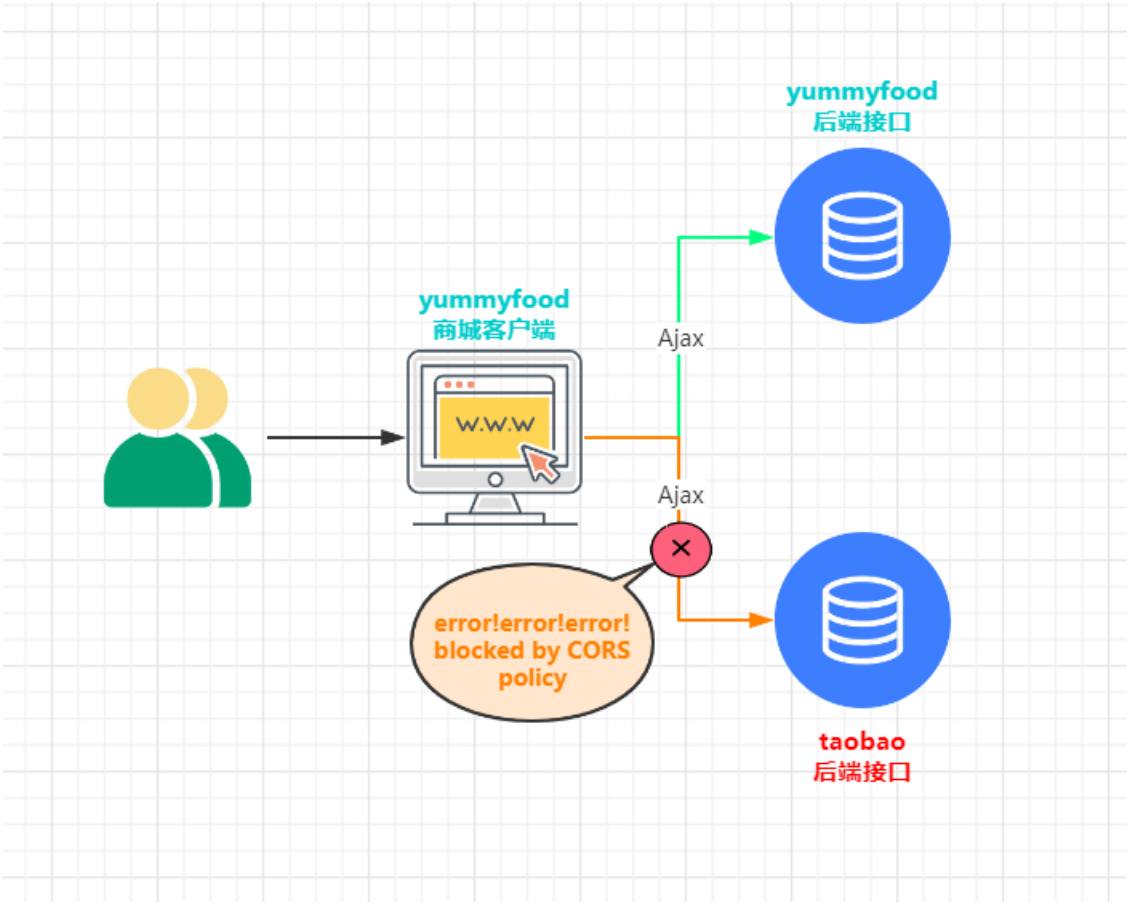
> 如果两个 URL 的 protocol、port (如果有指定的话)和 host 都相同的话，则这两个 URL 是同源。这个方案也被称为“协议/主机/端口元组”，或者直接是 “元组”。

也就是说：协议、域名、端口号都相同，只要有一个不相同，那么都是非同源。

下面几种情况就是同源或不同源的典型案例：

URL	说明	允许通信
http://www.a.com/a.js http://www.a.com/b.js	同一域名下	允许
http://www.a.com/lab/a.js http://www.a.com/script/b.js	同一域名下不同文件夹	允许
http://www.a.com:8000/a.js http://www.a.com/b.js	同一域名，不同端口	不允许
http://www.a.com/a.js https://www.a.com/b.js	同一域名，不同协议	不允许
http://www.a.com/a.js http://127.0.0.100/b.js	域名和域名对应ip	不允许
http://www.a.com/a.js http://script.a.com/b.js	主域相同，子域不同	不允许
http://www.a.com/a.js http://a.com/b.js	同一域名，不同二级域名（同上）	不允许
http://www.a.com/a.js http://www.b.com/b.js	不同域名	不允许

那么纠结这个同源不同源的问题目的是什么呢？答案是为了安全。就拿我们自己的项目来说，你受得了有人从他的网站不断地从你的网站调用接口获取数据吗？



在同源策略的限制下，非同源的网站之间不能发送 AJAX 请求，因此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。现在的问题是如何实现呢？主要通过CORS来解决，不过我们先简单说一说老的JSONP方案，再着重说下CORS如何解决。

二、JSONP跨域

浏览器只对 `XHR(XMLHttpRequest)` 请求有同源请求限制，而对 `script` 标签 `src` 属性、`link` 标签 `ref` 属性和 `img` 标签 `src` 属性没有这这种限制，利用这个“漏洞”就可以很好的解决跨域请求。`JSONP` 就是利用了 `script` 标签无同源限制的特点来实现的，当向第三方站点请求时，我们可以将此请求放在 `<script>` 标签的 `src` 属性里，这就如同我们请求一个普通的 `JS` 脚本，举个例子理解下。

假如需要从服务器（`http://www.a.com/user?id=123`）获取的数据如下：

```
{"id": 123, "name" : 张三, "age": 17}
```

那么，使用JSONP方式请求（`http://www.a.com/user?id=123?callback=foo`）的数据将会是如下：

```
foo({"id": 123, "name" : 张三, "age": 17});
```

这时候我们前端只要定义一个foo()回调函数接收返回的数据，并动态地创建一个script标签，使其的src属性为 http://www.a.com/user?id=123?callback=foo：

```
//这里的src传入http://www.a.com/user?id=123?callback=foo
//形如的效果是：<script src="http://www.a.com/user?id=123?
callback=foo"></script>
function addScriptTag(src) {
  var script = document.createElement("script")
  script.setAttribute('type','text/javascript')
  script.src = src
  document.appendChild(script)
}
// 回调函数
function foo(res) {
  console.log(res.message);
}
```

具体的写法不过多说明，否则容易产生更多的疑问，总之我们能理解其大概的用法即可。

因为 `jsonp` 跨域的原理就是用的动态加载 `script` 的 `src` ，所以我们只能把参数通过 `url` 的方式传递,所以 `jsonp` 的 `type` 类型只能是 `GET` ！

此外，这个方案对应的接口必须要跟后端接口配合好，让后端按照特定的格式返回信息。

三、CORS

既然 `JSONP` 有缺陷，那有没有其他好办法了呢?

为了解决浏览器同源问题， `W3C` 提出了跨源资源共享，即 `CORS` (`Cross-Origin Resource Sharing`)。它允许浏览器向跨源服务器，发出 `XMLHttpRequest` 请求，从而克服了 `AJAX` 只能同源使用的限制。

跨域资源共享（CORS）：通过修改Http协议header的方式，实现跨域。说的简单点就是，通过设置HTTP的响应头信息，告知浏览器哪些情况在不符合同源策略的条件下也可以跨域访问，浏览器通过解析Http协议中的Header执行具体判断。具体的Header如下：

- `Access-Control-Allow-Origin` ： 允许哪些ip或域名可以跨域访问
- `Access-Control-Max-Age` ： 表示在多少秒之内不需要重复校验该请求的跨域访问权限

- `Access-Control-Allow-Methods`：表示允许跨域请求的HTTP方法，如：`GET`，`POST`，`PUT`，`DELETE`
- `Access-Control-Allow-Headers`：表示访问请求中允许携带哪些`Header`信息，如：`Accept`、`Accept-Language`、`Content-Language`、`Content-Type`

前端工程是部署在8080端口的tomcat下，而后端的接口是部署在8088端口的tomcat下，从8080访问8088，实际上属于跨域。因此我们的接口工程需要实现`CORS`实现跨域。`SpringBoot`中有多种方式来实现`CORS`，我这里就贴下JAVA项目里用的写法：

```
@Configuration
public class CorsConfig {
    public CorsConfig(){

    }

    @Bean
    public CorsFilter corsFilter(){
//        1、添加cors配置信息
        CorsConfiguration config = new CorsConfiguration();
//      1.1 开放哪些ip、端口、域名的访问权限，星号表示开放所有域
        config.addAllowedOrigin("*");
//      1.2 设置允许客户端携带一些cookie等信息过来
        config.setAllowCredentials(true);
//      1.3 开放哪些Http方法，允许跨域访问，比如GET、POST
        config.addAllowedMethod("*");
//      1.4 设置允许的header
        config.addAllowedHeader("*");

//        2、为url添加映射路径
        UrlBasedCorsConfigurationSource corsSource = new
UrlBasedCorsConfigurationSource();
//        configo配置信息适用所有的路由
        corsSource.registerCorsConfiguration("/**",config);

        return new CorsFilter(corsSource);
    }
}
```

其中的原理，这篇文章：【SpringBoot配置Cors解决跨域请求问题(https://www.cnblogs.com/yuansc/p/9076604.html)】说的很好。这篇文章应该是参照了阮一峰大神写的【跨域资源共享 CORS 详解(http://www.ruanyifeng.com/blog/2016/04/cors.html)】大部分内容，着重说明了简单请求和非简单请求。其中如果是简单请求的话，服务端根据请求头的`Origin`字段判断是否允许访问。如果是非简单请求，则会复杂一点，在发送真实请求前首先发送一次`OPTION`请求询问

服务器是否允许这样的操作。预检请求通过后，浏览器会发送真实请求到服务器。这就实现了跨源请求。

二、CORS 简介

为了解决浏览器同源问题，`W3C` 提出了跨源资源共享，即 `CORS` (Cross-Origin Resource Sharing)。

`CORS` 做到了如下两点：

- 不破坏即有规则
- 服务器实现了 `CORS` 接口，就可以跨源通信

基于这两点，`CORS` 将请求分为两类：简单请求和非简单请求。

1、简单请求

在 `CORS` 出现前，发送 `HTTP` 请求时在头信息中不能包含任何自定义字段，且 `HTTP` 头信息不超过以下几个字段：

- `Accept`
- `Accept-Language`
- `Content-Language`
- `Last-Event-ID`
- `Content-Type` 只限于 [`application/x-www-form-urlencoded` 、 `multipart/form-data` 、 `text/plain`] 类型

一个简单的请求例子：

```
GET /test HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate, sdch, br
Origin: http://www.examples.com
Host: www.examples.com
```

对于简单请求，`CORS` 的策略是请求时在请求头中增加一个 `Origin` 字段，服务器收到请求后，根据该字段判断是否允许该请求访问。

1. 如果允许，则在 HTTP 头信息中添加 `Access-Control-Allow-Origin` 字段，并返回正确的结果；
2. 如果不 允许，则不在 HTTP 头信息中添加 `Access-Control-Allow-Origin` 字段 。

除了上面提到的 `Access-Control-Allow-Origin` ，还有几个字段用于描述 `CORS` 返回结果：

1. `Access-Control-Allow-Credentials` ： 可选，用户是否可以发送、处理 `cookie` ；
2. `Access-Control-Expose-Headers` ：可选，可以让用户拿到的字段。有几个字段无论设置与否都可以拿到的，包括 `Cache-Control` 、 `Content-Language` 、 `Content-Type` 、 `Expires` 、 `Last-Modified` 、 `Pragma` 。

2、非简单请求

对于非简单请求的跨源请求，**浏览器会在真实请求发出前**，增加一次 `OPTION` 请求，称为预检请求(`preflight request`)。预检请求将真实请求的信息，包括请求方法、自定义头字段、源信息添加到 HTTP 头信息字段中，询问服务器是否允许这样的操作。

例如一个 `DELETE` 请求：

```
OPTIONS /test HTTP/1.1
Origin: http://www.examples.com
Access-Control-Request-Method: DELETE
Access-Control-Request-Headers: X-Custom-Header
Host: www.examples.com
```

与 `CORS` 相关的字段有：

1. 请求使用的 `HTTP` 方法 `Access-Control-Request-Method` ；
2. 请求中包含的自定义头字段 `Access-Control-Request-Headers` 。

服务器收到请求时，需要分别对 `Origin` 、 `Access-Control-Request-Method` 、 `Access-Control-Request-Headers` 进行验证，验证通过后，会在返回 `HTTP` 头信息中添加：

```
Access-Control-Allow-Origin: http://www.examples.com
Access-Control-Allow-Methods: GET, POST, PUT, DELETE
Access-Control-Allow-Headers: X-Custom-Header
Access-Control-Allow-Credentials: true
Access-Control-Max-Age: 1728000
```

他们的含义分别是：

1. Access-Control-Allow-Methods: 真实请求允许的方法
2. Access-Control-Allow-Headers: 服务器允许使用的字段
3. Access-Control-Allow-Credentials: 是否允许用户发送、处理 cookie
4. Access-Control-Max-Age: 预检请求的有效期，单位为秒。有效期内，不会重复发送预检请求

当预检请求通过后，浏览器会发送真实请求到服务器。这就实现了跨源请求。

在文章的后半部分说明了 `SpringBoot` 设置 `CORS` 的方式和原理，它说到：

无论是通过哪种方式配置 `CORS`，其实都是在构造 `CorsConfiguration`。一个 `CORS` 配置用一个 `CorsConfiguration` 类来表示，它的定义如下：

```
public class CorsConfiguration {
    private List<String> allowedOrigins;
    private List<String> allowedMethods;
    private List<String> allowedHeaders;
    private List<String> exposedHeaders;
    private Boolean allowCredentials;
    private Long maxAge;
}
```

`Spring` 中对 `CORS` 规则的校验，都是通过委托给 `DefaultCorsProcessor` 实现的。

`DefaultCorsProcessor` 处理过程如下：

1. 判断依据是 `Header` 中是否包含 `Origin`。如果包含则说明为 `CORS` 请求，转到 2；否则，说明不是 `CORS` 请求，不作任何处理。
2. 判断 `response` 的 `Header` 是否已经包含 `Access-Control-Allow-Origin`，如果包含，证明已经被处理过了，转到 3，否则不再处理。
3. 判断是否同源，如果是则转交给负责该请求的类处理
4. 是否配置了 `CORS` 规则，如果没有配置，且是预检请求，则拒绝该请求，如果没有配置，且不是预检请求，则交给负责该请求的类处理。如果配置了，则对该请求进行校验。

校验就是根据 `CorsConfiguration` 这个类的配置进行判断：

1. 判断 `origin` 是否合法
2. 判断 `method` 是否合法
3. 判断 `header` 是否合法
4. 如果全部合法，则在 `response header` 中添加响应的字段，并交给负责该请求的类处理，如果不合法，则拒绝该请求。

那么我们从上面代码的最后一行 `CorsFilter(corsSource)` 进去看看是不是这个原理吧。

```
82      @Override
83      protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
84              FilterChain filterChain) throws ServletException, IOException {
85
86          if (CorsUtils.isCorsRequest(request)) {          1、判断是否是CORS请求，即是不是跨域请求
87              CorsConfiguration corsConfiguration = this.configSource.getCorsConfiguration(request);
88              if (corsConfiguration != null) {
89                  boolean isValid = this.processor.processRequest(corsConfiguration, request, response);   2、处理请求
90                  if (!isValid || CorsUtils.isPreFlightRequest(request)) {
91                      return;
92                  }
93              }
94          }
95
96          filterChain.doFilter(request, response);
97      }
98
99  }
```

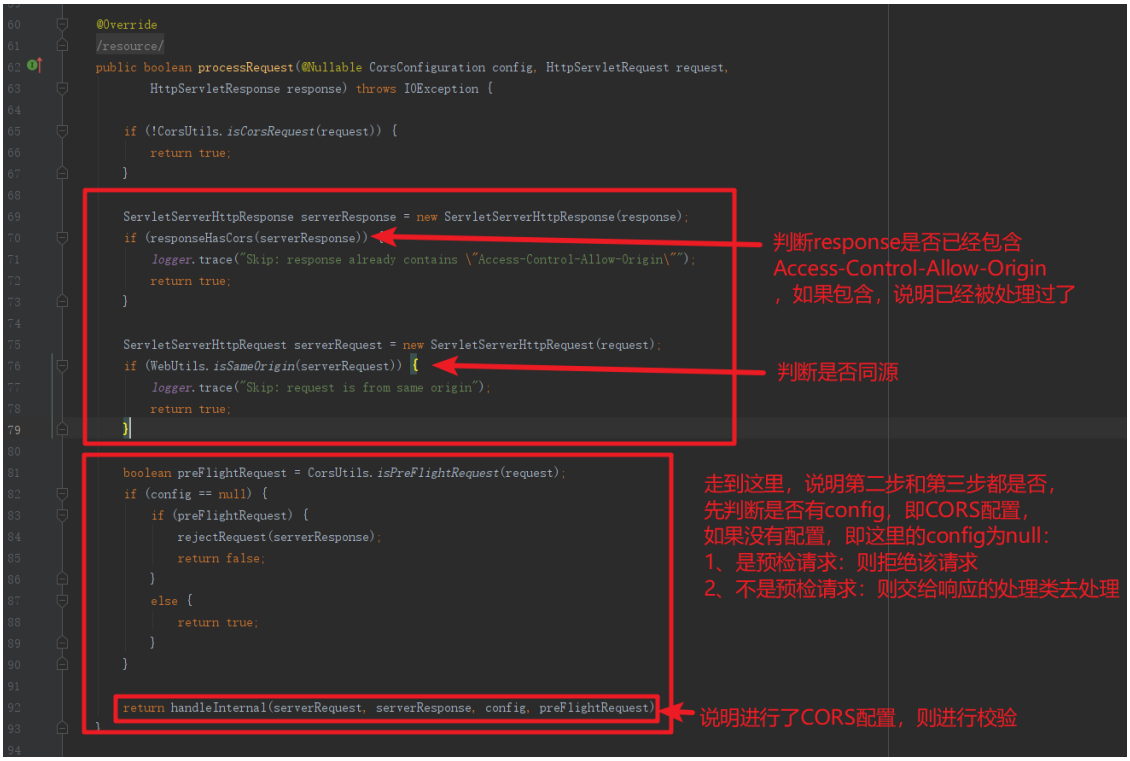
至于如何判断是否是 `CORS` 请求的呢?

```
31  public abstract class CorsUtils {
32
33      /**
34       * Returns {@code true} if the request is a valid CORS one.
35       */
36      public static boolean isCorsRequest(HttpServletRequest request) {
37          return (request.getHeader(HttpHeaders.ORIGIN) != null);          判断请求头中是否包含origin字段
38      }
39
40      /**
41       * Returns {@code true} if the request is a valid CORS pre-flight one.
42       */
43      public static boolean isPreFlightRequest(HttpServletRequest request) {
44          return (isCorsRequest(request) && HttpMethod.OPTIONS.matches(request.getMethod()) &&
45                  request.getHeader(HttpHeaders.ACCESS_CONTROL_REQUEST_METHOD) != null);
46      }
47
48  }
```

很好，就是判断是否包含 `origin` 字段，包含了则认为是一个CORS请求，第一步得到了验证。

> 1、Host：描述请求将被发送的目的地，包括且仅仅包括域名和端口号。 HTTP/1.1 的所有请求报文中必须包含一个Host头字段，且只能设置一个。
> 2、Origin：用来说明请求从哪里发起的，包括且仅仅包括协议和域名。这个参数一般只存在于CORS跨域请求中，可以看到response有对应的header：Access-Control-Allow-Origin。
> 3、Referer：当浏览器向web服务器发送请求的时候，一般会带上Referer，告诉服务器我是从哪个页面链接过来的，服务器籍此可以获得一些信息用于处理。比如从我主页上链接到一个朋友那里，他的服务器就能够从HTTP Referer中统计出每天有多少用户点击我主页上的链接访问他的网站。

下面进入 `processRequest` 方法：



进行了一系列的判断：

- 是否包含 `Origin` 字段，包含则是 `CORS` 请求，不是的话则直接转交给负责该请求的类处理。
- 是否同源、`response` 的 `Header` 是否包含 `Access-Control-Allow-Origin`，任意一个符合则直接转交给负责该请求的类处理。
- 是否有 `CORS` 配置，即我们的 `corsFilter`，有的话则去一一校验，没有，则判断是否是预检请求
    - 是预检请求，则拒绝该请求，真正结束了。

    - 不是预检请求，则直接转交给负责该请求的类处理。
- 有 `CORS` 配置，就不管是不是预检请求了，都走校验逻辑
    - 其中有一项校验不通过，返回false，则流程真正结束，决绝该请求。

```
protected boolean handleInternal(ServerHttpRequest request, ServerHttpResponse response,
        CorsConfiguration config, boolean preFlightRequest) throws IOException {

    String requestOrigin = request.getHeaders().getOrigin();
    String allowOrigin = checkOrigin(config, requestOrigin);
    HttpHeaders responseHeaders = response.getHeaders();

    responseHeaders.addAll(HttpHeaders.VARY, Arrays.asList(HttpHeaders.ORIGIN,
            HttpHeaders.ACCESS_CONTROL_REQUEST_METHOD, HttpHeaders.ACCESS_CONTROL_REQUEST_HEADERS));

    if (allowOrigin == null) {
        logger.debug("Reject: '" + requestOrigin + "' origin is not allowed");
        rejectRequest(response);
        return false;
    }

    HttpMethod requestMethod = getMethodToUse(request, preFlightRequest);
    List<HttpMethod> allowMethods = checkMethods(config, requestMethod);
    if (allowMethods == null) {
        logger.debug("Reject: HTTP '" + requestMethod + "' is not allowed");
        rejectRequest(response);
        return false;
    }

    List<String> requestHeaders = getHeadersToUse(request, preFlightRequest);
    List<String> allowHeaders = checkHeaders(config, requestHeaders);
    if (preFlightRequest && allowHeaders == null) {
        logger.debug("Reject: headers '" + requestHeaders + "' are not allowed");
        rejectRequest(response);
        return false;
    }

    responseHeaders.setAccessControlAllowOrigin(allowOrigin);
```

逐个进行校验

整套流程走完了，经过源码翻看，确认上面推荐阅读的文章描述的流程还是很靠谱的。

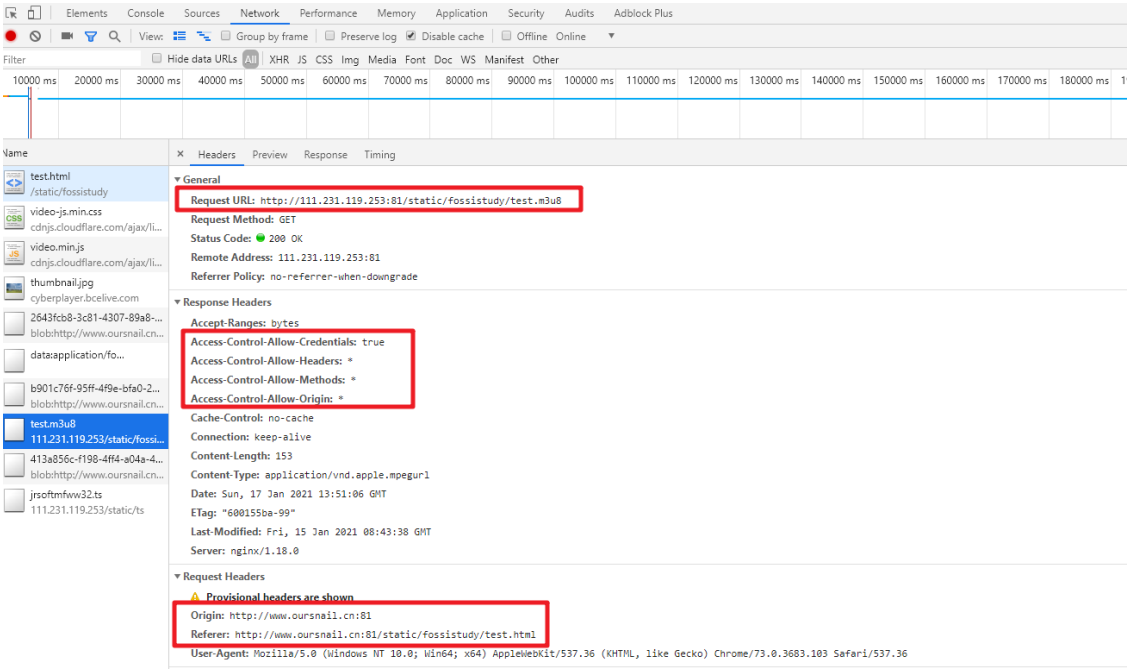
四、Nginx解决跨域

今天的主角nginx姗姗来迟，不过有了前面的铺垫，到了这里就比较简单啦。配置的项跟 `SpringBoot` 很像，只是nginx只需要在配置文件中做配置即可。

```
#允许跨域请求的域，*代表所有
add_header 'Access-Control-Allow-Origin' *;
#允许带上cookie请求
add_header 'Access-Control-Allow-Credentials' 'true';
#允许请求的方法，比如 GET/POST/PUT/DELETE
add_header 'Access-Control-Allow-Methods' *;
#允许请求的header
add_header 'Access-Control-Allow-Headers' *;
```

```
    server {
        listen       81;
        server_name  www.oursnail.cn;

        add_header Cache-Control no-cache;


#允许跨域请求的域，*代表所有
add_header 'Access-Control-Allow-Origin' *;
#允许带上cookie请求
add_header 'Access-Control-Allow-Credentials' 'true';
#允许请求的方法，比如 GET/POST/PUT/DELETE
add_header 'Access-Control-Allow-Methods' *;
#允许请求的header
add_header 'Access-Control-Allow-Headers' *;

        location / {
                root /home/mux/;
        }

        location /static {
            alias /home/sopftpuser/ftp;
        }

        error_page   500 502 503 504  /50x.html;
        location = /50x.html {
            root   html;
        }
    }
```

还是上节的观看视频，浏览器中先请求了`http://www.oursnail.cn:81/static/fossistudy/test.html`，里面它会去请求`http://111.231.119.253:81/static/fossistudy/test.m3u8`，最后请求到`http://111.231.119.253:81/static/ts/jrsoftmfww32.ts`，我找了下其中一个链接，看了下请求头和响应头：



我们可以看到，这里的配置跟`SpringBoot`中的配置可以一一对应上，实际上就是一样的意思，因此这里就不对这些配置项重复说明啦。

这里的Nginx作为一个反向代理，相当于一个网关，这样所有的后端只需要放行Nginx的访问即可，大大减少了后端工程的配置。是一个比较好的方案。