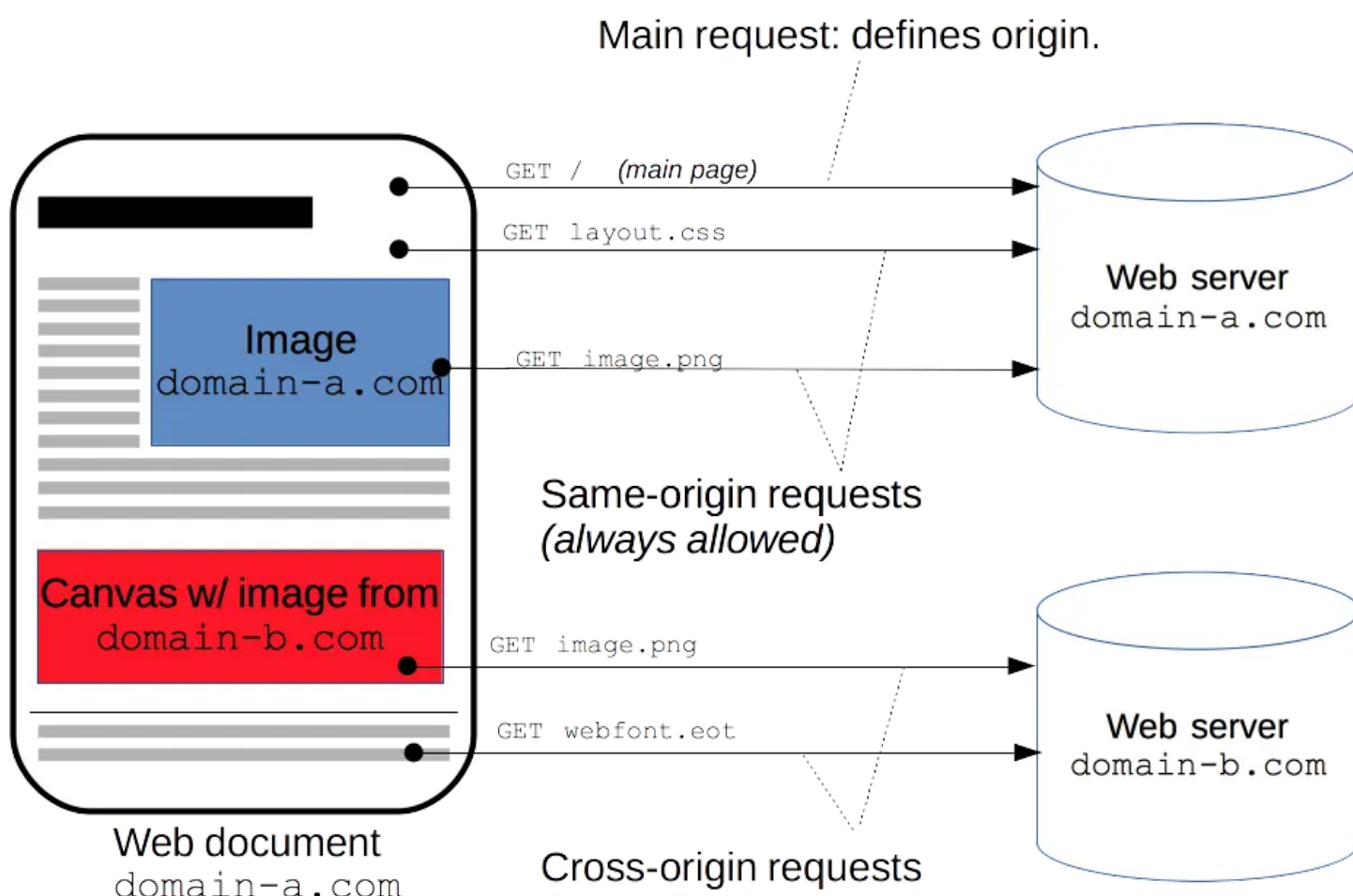


面试被跨域问麻了

跨源资源共享（Cross-Origin Resource Sharing, **CORS**）

是一种基于 **HTTP** 头的机制（划重点），该机制通过允许服务器标示除了它自己以外的其他源（域、协议或端口），使得浏览器允许这些源访问加载自己的资源。

跨源 HTTP 请求的一个例子：运行在 [domain-a.com](#) 的 JavaScript 代码使用 [XMLHttpRequest](#) 来发起一个到 [domain-b.com/data.json](#) 的请求。



出于安全性，浏览器限制脚本内发起的跨源 HTTP 请求。例如，XMLHttpRequest 和 [Fetch API](#) 遵循 [同源策略](#)。这意味着使用这些 API 的 Web 应用程序只能从加载应用程序的同一个域请求 HTTP 资源，除非响应报文包含了正确 **CORS 响应头**。

CORS 响应头

前面提到 CORS 是一种基于 HTTP 头的机制，这些 HTTP 头决定了浏览器是否阻止前端 JavaScript 代码获取跨域资源请求的响应

因此想要了解跨域必须先了解有哪些相关的 header、

Access-Control-Allow-Origin

指示响应的资源是否可以被给定的来源共享。

有效值：`*` | `<origin>` | `null`

对于**不包含凭据**的请求，服务器会以 “`*`” 作为通配符，从而允许任意来源的请求代码都具有访问资源的权限。

尝试使用通配符来响应包含凭据的请求会导致报错。

凭据 (Credentials) 通常是指 Cookie、HTTP 认证、TLS 客户端证书等敏感信息

指定一个来源（只能指定一个）。如果服务器支持多个来源的客户端，其必须以与指定客户端匹配的来源来响应请求。

Access-Control-Allow-Credentials

指示当前请求的凭证标记为 `true` 时，是否可以公开对该请求响应。

用于在请求要求包含凭据 (credentials) 时，告知浏览器是否可以将请求的响应暴露给前端 JavaScript 代码。

当作为对**预检请求**的响应的一部分时，这能表示是否真正的请求可以使用 credentials。

`Access-Control-Allow-Credentials` 标头需要与 `XMLHttpRequest.withCredentials` 或 Fetch API 的 `Request()` 构造函数中的 `credentials` 选项结合使用。Credentials 必须在前后端都被配置才能使带 credentials 的 CORS 请求成功。

有效值：`true` 唯一的有效值。

Access-Control-Allow-Headers

用于对预检请求的响应中，指示实际的请求中可以使用哪些 HTTP 标头。

如果请求中含有 `Access-Control-Request-Headers` 字段，那么这个首部是必要的。

注意以下特定首部是一直允许的：`Accept`，`Accept-Language`，`Content-Language`，`Content-Type`（只在值属于 **MIME 类型** `application/x-www-form-urlencoded`，`multipart/form-data` 或 `text/plain` 中的一种时）。这些被称作 **simple headers**，无需特意声明它们。

有效值：

- (wildcard 通配符)

对于没有凭据的请求（没有 HTTP cookie 或 HTTP 认证信息的请求），值 "`*`" 仅作为特殊通配符值。在具有凭据局的请求中，它被视为没有特殊语义的文字标头名称 "`*`"。

`<header-name>` header 字段名

`Authorization` 标头不能使用通配符，并且始终需要明确列出。

Access-Control-Allow-Methods

响应部首 `Access-Control-Allow-Method` 在对 `preflight request`（预检请求）的应答中明确了客户端所要访问的资源允许使用的方法或方法列表。

有效值：`<method>` 用逗号隔开的允许使用的 `HTTP request methods` 列表。

Access-Control-Expose-Headers

允许服务器指示哪些响应标头可以暴露给浏览器中运行的脚本，以响应跨源请求。

默认情况下，仅暴露 `CORS 安全列表` 的响应标头，如果想要让客户端可以访问到其他的标头，服务器必须将它们放在 `Access-Control-Expose-Headers` 里列出来。

有效值：

`<header-name>` 允许客户端从响应中访问的 0 个或多个使用逗号分隔的标头名称

*（wildcard 通配符）若没有携带凭据才会被当做一个特殊的通配符。对于带有凭据的请求，会被简单地当作标头名称" * "，没有特殊的语义。不会匹配 `Authorization`，如果要暴露它需要显式指定。

Access-Control-Max-Age

表示预检请求的结果可以被缓存多久。

有效值：`<delta-seconds>`

返回结果可以被缓存的最长时间（秒）。在 Firefox 中上限是 24 小时（即 86400 秒）。在 Chromium v76 之前，上限是 10 分钟（即 600 秒），之后是 2 小时（即 7200 秒）。Chromium 同时规定了一个默认值 5 秒。如果值为 -1，则表示禁用缓存，则每次请求前都需要使用 `OPTIONS` 预检请求。

Access-Control-Request-Headers

出现于 `preflight request`（预检请求）中，用于通知服务器在真正的请求中会采用哪些请求头。

有效值：`<header-name>` 在实际请求中将要包含的一系列 HTTP 头，以逗号分隔。

Access-Control-Request-Method

出现于 `preflight request`（预检请求）中，用于通知服务器在真正的请求中会采用哪种 HTTP 方法。因为预检请求所使用的方法总是 `OPTIONS`，与实际请求所使用的方法不一样，所以这个请求头是必要的。

有效值：`<method>` 一种 HTTP 请求方法，例如 GET、POST 或 DELETE。

Origin

表示请求的**来源**（协议、主机、端口）。例如，如果一个用户代理需要请求一个页面中包含的资源，或者执行脚本中的 HTTP 请求（fetch），那么该页面的来源（origin）就可能被包含在这次请求中。

有效值：

`null` 请求来源是“隐私敏感”的，或者是 HTML 规范定义的不透明来源

`<scheme>` 请求所使用协议，通常是 HTTP 协议或者它的安全版本（HTTPS 协议）。

`<hostname>` 源站的域名或 IP 地址。

`port`（可选）服务器正在监听的端口号。缺省为服务器的默认端口（对于 HTTP 请求而言，默认端口为 80）。

代码实践

先创建一个 node 服务器

Hello world!

1. 安装 express.js

a. `yarn init`

b. `yarn add express`

2. 应用代码

新建 `app.js` 写入下面代码

```
1 const express = require('express')const app = express()const port =
  3000app.get('/', (req, res) => { res.send('Hello World!')})app.listen(port,
  () => { console.log(`Example app listening on port ${port}`)})
```

1. 启动程序

运行 `node app.js` 控制台会打印出 'Example app listening on port 3000'

打开浏览器，访问 `http://localhost:3000/` 即可看到程序响应 'Hello World!'

使用 express-generator 搭建程序框架

- `npx express-generator`

- 启动服务

- `DEBUG=myapp:* & yarn start` (在 zsh 中会报错，因为 '&' 是一个特殊字符表示将命令放入后台运行，可以使用分号分割命令，效果是相同的。)

- `DEBUG=myqpp` 是配置环境变量用于调试

- 【可选】使用 nodemon 配置热更新，不配置热更新时每次修改后需要手动重启程序

- `yarn add nodemon`

- 添加 `nodemon.json` 配置

```
1 { "watch": ["app.js", "routes/", "views/", "bin/"], "ext": "js,json",  
  "ignore": ["node_modules/"]}
```

- 使用 nodemon 启动程序 `nodemon ./bin/www`，也可配置在 `package.json` 中配置

```
1 "scripts": { "start": "nodemon ./bin/www"},
```

注意：热更新需要修改 bin 目录下的 www 文件名为 `www.js`，才能正确的监听变化

服务端代码

查看服务端代码的目录结构如下

`bin/www` 是程序入口，http 服务在这个文件中创建并启动

`public/` 静态资源

`routes/` 子路由存放位置，程序 API 都以模块的形式组织在这个文件夹中

`views/` 存放视图模板（404 等基础页面）

`app.js` 程序主体，负责创建程序，挂载路由等操作

`bin/www` 的主要内容

```
1 var app = require('../app');var http = require('http'); // 引入 node http 模块  
  var port = normalizePort(process.env.PORT || '3000'); // 合法值处理  
  app.set('port', port); // 设置端口号var server = http.createServer(app); // 创建  
  http 服务server.listen(port); // 挂载服务
```

`app` 的主要内容

```
1 var express = require("express")var path = require("path")var cookieParser =  
  require("cookie-parser")var app = express()// 挂载视图引擎app.set("views",  
  path.join(__dirname, "views"))app.set("view engine", "jade")// 添加中间件  
  app.use(express.json()) // 识别请求体中的 jsonapp.use(express.urlencoded({  
  extended: false })) // 识别请求体中的字符串和数字app.use(cookieParser()) // 解析  
  cookie// 这些解析结果最终都会添加到 req 中// 静态文件路径映射，将 public 目录映射到  
  /static 上app.use('/static', express.static(path.join(__dirname, "public"), {  
  setHeaders: function(res, path, stat) {    res.header('Access-Control-Allow-  
  Origin', '*') // 添加跨域请求的头  } })))// 引入路由配置var indexRouter =  
  require("./routes/index")var usersRouter = require("./routes/users")const  
  testRouter = require('./routes/test')// 挂载路由app.use("/",  
  indexRouter)app.use("/users", usersRouter)app.use('/api', testRouter)// 捕获
```

```
404app.use(function (req, res, next) { next(createError(404))})// 错误处理
app.use(function (err, req, res, next) { // set locals, only providing error
in development res.locals.message = err.message res.locals.error =
req.app.get("env") === "development" ? err : {} // render the error page
res.status(err.status || 500) res.render("error")})module.exports = app
```

get 请求

创建一个支持跨域的 `get` 请求只需要在 `response` 的 `header` 上添加对应的标识即可，`routes/test.js` 文件内容

```
1 var express = require('express');var router = express.Router();/* GET home
page. */router.get('/', function(req, res, next) { // 允许跨域配置, get 的跨域请
求也需要配置 res.header('Access-Control-Allow-Origin', '*'); res.send('success
response');});module.exports = router;
```

前端 `axios` 请求代码

```
1 axios.get('/api',{ params: { name: "Ginlon" } }).then(res => {
  console.log(res.headers)})
```

post 请求

服务端

```
1 router.post("/update", function (req, res, next) { res.header("Access-Control-
Allow-Origin", "*") res.send("success update")})
```

前端

```
1 axios.post("/api/update").then((res) => { console.log(res.data)})
```

简单地 post 请求依然可以通过配置 `Access-Control-Allow-Origin: *` 来进行跨域请求，但是我们的 post 请求携带参数时，如果只配置 `Access-Control-Allow-Origin` 仍然会被跨域拦截，并且可以看到浏览器发起了一个 `preflight` 预检请求

<input type="checkbox"/> update?age=18	COR...	xhr	xhr.js:251	0 B	1...
<input type="checkbox"/> update?age=18	200	preflight	预检 ↻	0 B	3...

由于跨域机制是由 `http` 的 `header` 控制的因此通过对比两次 `post` 请求的 `header` 不难发现，携带参数的 `post` 请求的 `header` 中多了一个 `Content-Type` 项，这是 `axios` 为了使服务端可以正确的解析字符串而自动添加的标识。

▼ 请求标头

⚠ 当前显示的是预配标头。 [了解详情](#)

Accept:	application/json, text/plain, */*
Content-Type:	application/json
Referer:	http://127.0.0.1:517
Sec-Ch-Ua:	"Chromium";v="110" "Not)A;Brand";v="2" "Google Chrome";v="116"
Sec-Ch-Ua-Mobile:	?0
Sec-Ch-Ua-Platform:	"macOS"
User-Agent:	Mozilla/5.0 (Macintosh Intel Mac OS X 10_ AppleWebKit/537.3 (KHTML, like Gecko

由于整个跨域系统都是基于 `header` 的，查看前面的 `header` 说明，不难发现 `Access-Control-Allow-Headers` 中的描述

`Content-Type` 只在值属于 **MIME 类型** `application/x-www-form-urlencoded`, `multipart/form-data` 或 `text/plain` 中的一种时, 才被称作 **simple headers**, 而无需特意声明

而我们此处的 `application/json` 显然不在其中, 因此我们只要在服务端添加一个对应的 `options` 请求的配置即可

```
1 router.options("/update", function (req, res, next) { res.header("Access-Control-Allow-Origin", "*") // 是否允许跨域 res.header("Access-Control-Allow-Headers", "Content-Type") // 允许携带的 header 标识 res.send() res.end()})
```

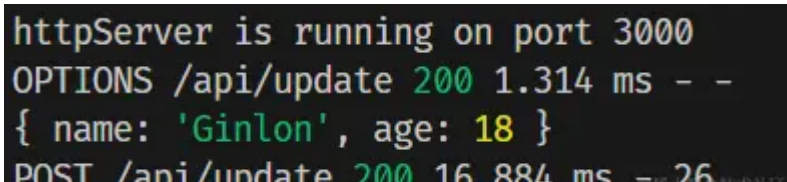
这样 `post` 请求就可以正常访问了。

为了知道 `ContentType` 的作用可以使用 `XMLHttpRequest` 自己创建一个 POST 请求, 不添加 `Content-Type`

```
1 、const xhr = new XMLHttpRequest()xhr.onreadystatechange = function () { if (xhr.readyState === XMLHttpRequest.DONE && xhr.status === 200) { console.log(xhr.responseText) }}xhr.open("POST", "http://localhost:3000/api/update", true)xhr.send( JSON.stringify({ name: "Ginlon", age: "18", }))
```

可以在服务端使用 `console.log` 打印 `request.body` 来查看两次请求什么不一样

```
1 router.post("/update", function (req, res, next) { console.log(req.body) res.header("Access-Control-Allow-Origin", "*") // 完成响应 res.send("success response api") res.end()})
```



```
httpServer is running on port 3000
OPTIONS /api/update 200 1.314 ms - -
{ name: 'Ginlon', age: 18 }
POST /api/update 200 16.884 ms - 26
```

可以看到上面的日志是 axios 发出的带有 `Content-Type` 的 post 请求的输出,

下面是我们自己创建的不带 `Content-Type` 的 post 请求, 可以发现当缺少 `Content-Type` 时服务端负责解析的中间件就不能够识别到 request 中携带的数据。

vite 的跨域配置

由于 vite 服务器默认运行在 127.0.0.1:5137 端口, 与服务器的 `http://localhost:3000` 不一致, 因此会触发浏览器的跨域机制。

有两种解决方式，一是服务端进行配置，在接口中添加跨域相关的 http header，这样前端就可以直接访问服务器地址无需额外的处理。

服务端无法提供支持时，前端可以自己搭建服务器转发请求，再将结果返回给浏览器从而避免浏览器的跨域限制。

通过添加配置，vite 可以帮我们快速的搭建一个本地服务器转发请求。

```
1 defineConfig({  server: {    proxy: {      // 带选项写法：  
      http://127.0.0.1:5173/api/bar -> http://localhost:3000/bar    "/api": {  
      target: "http://localhost:3000",      changeOrigin: true,      rewrite:  
      (path) => path.replace(/^/api/, ""),    },    },  }, })
```

这是一个典型的配置，[官网](#)有很详细的说明这里不做重点展开。

只看配置似懂非懂，我们可以通过发起一个跨域请求，简单地分析一下 vite 是如何完成转发的来加深理解

整个过程大体可以分为三步

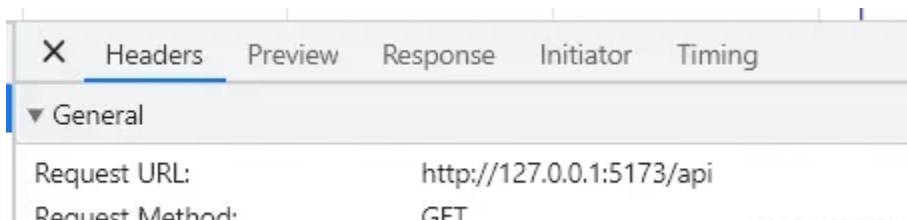
1. 浏览器向 vite 服务器发送请求
2. vite 服务器接收请求并根据配置的转发规则转发请求
3. vite 服务器收到响应，并将响应返回给浏览器

首先我们发起一个请求

```
1 axios.get('/api/someApi')
```

js 发起的 get 请求也会触发跨域，在地址栏中直接访问 get 地址不会跨域（因为不是 js 发起的请求，而是直接由浏览器内部进程发起的请求）

首先这个请求的地址并不是一个完整的 URL，因此浏览器会将它视为一个相对路径，而我们的页面根目录本就是 vite 服务器，于是就完成了第一步，浏览器向 vite 服务器发送请求，完整的请求地址可以在**网络**面板看到



值得一提的是，很多项目会将服务器地址封装到 axios 的 baseUrl 中，这会使路径成为绝对路径，而无法发送到 vite 服务器，自然就不会通过 vite 服务器转发，因此如果后端不进行对应的配置，直接添加 baseUrl 会导致跨域问题。

第二步，vite 接收到了来自浏览器的请求，于是 vite 会在 proxy 查找匹配的转发规则，当匹配到相应的转发规则时，vite 会根据相应的配置重新拼装路径（target、rewrite），然后使用 nodejs 的 http api 向目标地址发送请求。（websocket 和 https 同理）

第三步，vite 会收到服务器的响应，由于响应是由本地 vite 服务器接收，而不是浏览器，因此即使后端不添加跨域相关的 header，我们也可以拿到响应信息，然后 vite 会将响应返回给浏览器，至此就完成了整个请求。

整个过程对浏览器而言只是与本地的 vite 服务器进行交互，浏览器自始至终都只访问了 127.0.0.1:5173 因此也就不存在跨域问题了。

携带 cookie 的跨域请求

同源请求时，服务端可以通过在响应头中添加 Set-Cookie 字段来设置 cookie

```
1 router.get("/setCookie", function (req, res, next) { res.cookie("remember-me", "2", { expires: new Date(Date.now() + 900000), }) res.send()})
```

查看 http 的 response 头可以看到 Set-Cookie 字段，其包含了要设置的属性值和一些配置选项，具体每个配置的作用就不在此赘述。

```
Set-Cookie: remember-me=2; Path=/; Expires=Mon, 28 Aug 2023 11:00:50 GMT
```

Set-Cookie 会被浏览器从响应头中过滤掉，而不传给 javascript 脚本，但依然可以通过 document.cookie 访问到 cookie，如果不想前端访问 cookie 则可以在发送 cookie 时设置 httpOnly 属性，这样前端就无法通过 document.cookie 访问和修改对应的 cookie。

注意：我们本地启动的前端地址和服务端地址并不同源，因此我们可以通过配置 vite 的 server.proxy 实现转发请求，从而绕过跨域机制，完成同源设置 cookie 的请求。

上面的方法只能用于同源的请求，当跨域时即使服务端添加了 Access-Control-Allow-Origin 也不能通过 Set-Cookie 实现跨域的 cookie 设置，浏览器会自动过滤掉相关的 header。

为了限制 cookie 的滥用，浏览器禁止了跨域传递 cookie，当想要跨域传递 cookie 时则必须设置 SameSite 属性，只有当 SameSite 设置为 None 时才能够跨域传递 cookie，现在设置 SameSite=None 属性的 cookie 必须同时设置 Secure 属性，也就是说只能用于安全上下文（https）。

创建 https 服务，需要使用证书和密钥，自己的 demo 可以通过自签名证书来提供 https 服务。

即使使用了 https 服务也只是能够与当前跨域站点通信时设置和携带 cookie，这些 cookie 仍然不会在第三方请求时携带。

img 和 canvas 的跨域问题

通常 img 标签加载的图像数据不与 Javascript 交互，因此 img 标签是被允许加载跨域图像的，但是如果想要通过 Javascript 访问图像数据时就会被浏览器的跨域策略阻止，比如使用 canvas 的

`getImageData` api 访问图像数据时浏览器会报出错误

如果想要 canvas 可以访问 img 中的图像数据，就需要配置 img 标签的 `crossorigin` 属性，添加了 `crossorigin` 属性的图像会使用 CORS 完成图像资源的抓取，通过 CORS 获取到的图像不会被标记为“污染(tainted)”，便可以使用 Javascript 访问图像的数据。

`crossorigin` 允许的值：

`anonymous` 发送忽略凭据的跨域请求（不携带 cookie，X.509证书或 `Authorization` 标头）

`use-credentials` 发送携带凭据的跨域请求，如果服务端没有配置 `Access-Control-Allow-Credentials:true` 响应标头浏览器会将图片标记为被污染，且限制对图像数据的访问。

类似的 video、audio、svg 标签也存在同样地问题，video、audio 也有 `crossorigin` 属性，前端使用了 `crossorigin` 属性后服务端也需要添加对应的 header。