03_coderwhy前端八股文（三） - 跨域问题

01-什么是跨域？什么是同源策略？

跨域的概念解释

跨域问题通常是由浏览器的同源策略（Same-Origin Policy, SOP）引起的访问问题。

- 同源策略是浏览器的一个重要安全机制，它用于限制一个来源的文档或脚本如何能够与另一个来源的资源进行交互。

同源策略的定义：同源策略要求两个URL必须满足以下三个条件才能认为是同源：

- **协议（Protocol）**：例如，http和https是不同的协议。
- **主机（Host）**：例如，`www.example.com`和`api.example.com`是不同的主机。
- **端口（Port）**：例如，默认的8080和8081端口被认为是不同的端口。

只有当两个URL的协议、主机和端口都相同时，才被认为是同源。否则，浏览器会认为它们是跨域的。

为什么会产生跨域

跨域问题的产生和前后端分离的发展密切相关。

- 在早期，服务器端渲染的应用通常不会有跨域问题，因为前端代码和后端API都是在同一个服务器上运行的。
- 随着前后端分离的出现，前端代码和后端API经常部署在不同的服务器上，这就引发了跨域问题。
- 例如，一个网站的静态资源（HTML、CSS、JavaScript）可能部署在`www.example.com`上，而API接口则部署在`api.example.com`上。
- 浏览器在发现静态资源和API接口不在同一个源时，就会产生跨域问题。

所以，在静态资源服务器和API服务器（其他资源类同）是同一台服务器时，是没有跨域问题的。

接下来，我们就可以演示一下跨域产生和不产生的项目部署区别了。

部署同一服务器代码

我们现在使用Koa开发一个服务器：

```
const Koa = require('koa')
const KoaRouter = require('@koa/router')
const static = require('koa-static')

const app = new Koa()

app.use(static('./static'))

const router = new KoaRouter({ prefix: '/users' })
```

```
10
11
12 router.get('/list', (ctx, next) => {
13   ctx.body = ['aaa', 'bbb', 'ccc']
14 })
15
16 app.use(router.routes())
17
18 app.listen(8000, () => {
19   console.log('koa服务器启动成功~')
20 })
```

我们现在在static文件夹中开发一个index.html

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="UTF-8">
5    <meta name="viewport" content="width=device-width, initial-scale=1.0">
6    <title>Document</title>
7  </head>
8  <body>
9
10   <h1>测试跨域：页面处于同一个服务器下/不同服务器下的测试</h1>
11
12   <script>
13     fetch('http://localhost:8000/users/list').then(res => {
14       res.json().then(results => {
15         console.log(results)
16       })
17     })
18   </script>
19 </body>
20 </html>
```

测试一：通过http://localhost:8000/访问，可以正常访问，并且可以获取数据

测试二：通过http://127.0.0.1:5500/02_%E8%B7%A8%E5%9F%9F%E9%97%AE%E9%A2%98%E8%A7%A3%E6%9E%90/static/index.html访问，可以正常访问，但是出现了跨域的错误

```
Access to fetch at 'http://localhost:8000/users/list' from origin '                index.html:1
http://127.0.0.1:5500' has been blocked by CORS policy: No 'Access-Control-Allow-Origin'
header is present on the requested resource. If an opaque response serves your needs, set the
request's mode to 'no-cors' to fetch the resource with CORS disabled.

▶ GET http://localhost:8000/users/list net::ERR_FAILED 200 (OK)          index.html:13

▶ Uncaught (in promise)                                                  index.html:13
TypeError: Failed to fetch
    at index.html:13:5
```

CORS（跨域资源共享）

CORS（Cross-Origin Resource Sharing）是一种机制：

- 它使用额外的HTTP头来告诉浏览器允许从其他域（域名、协议或端口）加载资源。
- 通过CORS，服务器可以显式声明哪些源站点有权限访问它的资源。

在我们的Koa代码中我们可以进行如下设置：

```
// 手动设置CORS中间件
app.use(async (ctx, next) => {
  ctx.set('Access-Control-Allow-Origin', '*');
  ctx.set('Access-Control-Allow-Methods', 'GET, POST, PUT, DELETE, OPTIONS');
  ctx.set('Access-Control-Allow-Headers', 'Content-Type, Authorization, Accept');
  await next();
});
```

设置的解释：

- `Access-Control-Allow-Origin`：允许所有域名访问（你也可以指定特定的域名，例如`'http://example.com'`）。
- `Access-Control-Allow-Methods`：允许的HTTP请求方法。
- `Access-Control-Allow-Headers`：允许的HTTP请求头。

另外我们可以了解一下浏览器机制，关于预请求和实际请求：

预检请求（Preflight Request）：

- 对于复杂请求（如使用非简单方法：PUT, DELETE 或自定义头），浏览器会先发送一个OPTIONS请求，询问服务器是否允许跨域请求。
- 服务器如果同意跨域请求，则返回包含CORS头信息的响应。

实际请求（Actual Request）：

- 如果预检请求被允许，浏览器会发送实际请求，并且会在请求头中包含一些CORS相关的头信息。
- 服务器在响应中包含CORS头信息，这些信息会被浏览器验证。

```
// 手动设置CORS中间件
app.use(async (ctx, next) => {
  ctx.set("Access-Control-Allow-Origin", "http://127.0.0.1:5500")
  ctx.set("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE, OPTIONS")
  ctx.set("Access-Control-Allow-Headers", "Content-Type, Authorization, Accept")

  // 如果是预检请求，则直接返回204 No Content
  if (ctx.method === "OPTIONS") {
    ctx.status = 204
    return
  }

```

```
  await next()
})
```

Vite/Webpack底层方案（Node服务器）

那么在我们平时开发中，我们并不会，也不能直接去修改服务器（当然自己开发的服务器除外），那么开发过程中我们遇到跨域问题应该如何解决呢?

我相信很多同学都是做过如何类似的配置的：

webpack跨域配置：

```
module.exports = {
  //...
  devServer: {
    proxy: {
      target: 'http://localhost:8000',
      pathRewrite: {
        '^/api': ''
      },
      changeOrigin: true
    },
  },
};
```

vite跨域配置：

```
export default defineConfig({
  server: {
    proxy: {
      '/api': {
        target: 'http://localhost:3000',
        changeOrigin: true,
        rewrite: (path) => path.replace(/^\/api/, ''),
      }
    },
  },
})
```

其实不管是Webpack，还是Vite，它们底层都是利用开启一个新的Node服务器代理来解决跨域的。

它们的过程如下：

创建开发服务器：

- 使用 Node.js 的 `http` 模块创建一个本地开发服务器，监听特定端口（如 3000）。
- 这个开发服务器负责处理所有的前端请求，包括静态文件、热模块替换（HMR）、API 代理等。

使用 http-proxy 实现代理：

- Vite 或者 Webpack 使用 `http-proxy` 或 `http-proxy-middleware` 来创建代理中间件。
- 代理中间件会拦截特定路径的请求，并将这些请求转发到目标服务器。

我们这里使用express和http-proxy-middleware来实现一下代码：

```
const express = require('express')
const { createProxyMiddleware } = require('http-proxy-middleware')

const app = express()

app.use(express.static('./static'))

app.use('/api', createProxyMiddleware({
  target: 'http://localhost:8000',
  pathRewrite: {
    '^/api': ''
  },
  changeOrigin: true
}))

app.listen(8001, () => {
  console.log('proxy服务器启动成功~')
})
```

Nginx反向代理配置

windows、Mac、Linux的NGINX安装这里不再演示，大家可以去学习一下我们之前系统课的安装和使用。

- 注意：Mac电脑的配置文件在`/usr/local/etc/nginx`

我们可以可以分成两种情况：

情况一：Nginx仅仅代理API服务器

在这种情况下，前端静态资源由前端服务器直接提供，Nginx 仅用于代理 API 请求。

```
server {
  listen       8080;
  server_name  localhost;

  location / {
    root   /Users/coderwhy/Desktop/前端八股文/备课/code/02_跨域问题解析/static;
    index  index.html index.htm;
  }

  location /api/ {
    proxy_set_header Host $host; # 将客户端请求的 Host 头信息设置为原始请求的主机名
```

```
    proxy_set_header X-Real-IP $remote_addr; # 设置 X-Real-IP 头信息，包含客户端的真实IP 地址
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for; # 设置 X-Forwarded-For 头信息，包含所有中间代理的 IP 地址
    proxy_set_header X-Forwarded-Proto $scheme; # 设置 X-Forwarded-Proto 头信息，包含客户端请求使用的协议（HTTP 或 HTTPS）

    # 添加跨域头信息
    add_header Access-Control-Allow-Origin *;
    add_header Access-Control-Allow-Methods "GET, POST, OPTIONS, PUT, DELETE";
    add_header Access-Control-Allow-Headers "Content-Type, Authorization, Accept";

    # 处理预检请求
    if ($request_method = OPTIONS) {
      return 204;
    }

    rewrite ^/api/(.*)$ /$1 break; # 因为服务器没有/api,所以做一次重写
    proxy_pass http://localhost:8000; # 代理到api服务器
  }
}
```

情况二：Nginx代理了静态资源和API服务器

在这种情况下，Nginx 既代理前端静态资源，又代理 API 请求。

- 因为静态资源和API请求都是在同一个服务器中，所以其实可以不需要设置跨域访问

```
location / {
  root   /Users/coderwhy/Desktop/前端八股文/备课/code/02_跨域问题解析/static;
  index  index.html index.htm;
}

location /api/ {
  rewrite ^/api/(.*)$ /$1 break; # 因为服务器没有/api,所以做一次重写
  proxy_pass http://localhost:8000; # 代理到
}
```

02-描述在开发过程中遇到的跨域问题，并解释导致跨域问题产生的原因

在开发过程中，我们几乎都会遇到跨域请求问题，这主要是因为浏览器的同源策略。

- 同源策略要求执行脚本的网页的源必须与请求的资源的源相同，否则浏览器会阻止这种请求。
- 具体来说，同源策略要求协议、端口（如果有指定），和域名完全匹配。

简单例子：

- 举个例子，如果我们开发过程中，前端项目是部署在`http://localhost:3000`（本地开启的服务）。
- 而后端API部署在`http://api.example.com`，由于域名不同，直接从前端向后端发起请求，就会因为不符合同源策略而被浏览器拦截。

在开发过程中我们的解决方案主要有如下几种：

- **CORS 设置**
  - 让后端开发在测试服务器上配置 CORS（跨域资源共享）策略。
  - 这通常涉及在后端响应头中添加 `Access-Control-Allow-Origin`。
  - 例如，设置为 `*` 可以允许所有域的访问，或者指定特定的域名来限制访问只允许来自这些域的请求。
- **在Vite或Webpack中配置**
  - 通过前端开发工具如 Vite 或 Webpack 配置代理。
  - 这些工具的底层实现通常使用了如 `http-proxy` 等库，可以在本地开发环境中代理API请求到指定的后端服务，从而绕过浏览器的同源策略。

当然，还有生成环境，我们可以到时候回答Nginx。

03-什么是浏览器的同源策略？为什么浏览器会有同源策略?

同源策略是浏览器的一种基本安全特性，它用来限制一个域的文档或脚本如何与另一个域的资源进行交互。

- 具体来说，如果两个页面的协议、端口（如果指定了的话）和域名都相同，我们就认为它们是"同源"的。
- 反之，就是“不同源”，不同源时，浏览器会对其访问进行限制。

为什么浏览器会有同源策略?

- 同源策略的主要目的是为了保护用户信息的安全，防止恶意网站窃取数据。
- 没有同源策略的限制，恶意网站可以非常容易地通过脚本访问另一个网站上的敏感数据。

虽然同源策略提供了重要的安全保障，但它也限制了合法的跨域请求，这在现代的网页应用中是非常常见的需求。

04-解释正向代理和反向代理的概念，以及其在网络通信中的作用。

网络上有很多关于正向代理和反向代理的解释，要么关于概念化，要么很多时候我认为并没有把握正向代理和反向代理的核心。

正向代理

正向代理代表客户端向服务器发起请求，这意味着：

- **客户端知道代理的存在**：客户端配置了代理服务器的地址和端口，因此清楚所有的请求都通过这个代理来完成的（客户端需要这个代理）。
- **服务器可能不知道真实的发起者**：从服务器的角度看，它只看到有一个这样的请求。它无法直接知道请求的真实来源，到底是客户端自己还是代理服务器发起的。
- 也就是正向代理是客户端的配置的，这种配置并不局限于在客户端直接配置，或者放到某一台服务器来完成。

正向代理的实际用途：

1. **控制和过滤访问**：学校或企业或xx可能使用正向代理来限制访问特定类型的网站，因为这些网站受限了。
2. **提供匿名性**：代理服务器可以隐藏用户的真实IP地址，使得目标网站看到的是代理服务器的IP。
3. **缓存常访问的资源**：通过缓存常用的网页和资源，正向代理可以加快访问速度并减少外部流量。
4. 等等...

我们在Vite、Webpack中开启的本地服务器其实就是一种正向代理。

反向代理

反向代理代表服务器接受来自客户端的请求，这意味着：

- **服务器知道代理的存在**：服务器配置了反向代理，所有的客户请求都首先被代理服务器接收和处理。
- **客户端可能不知道代理的存在**：对于客户端来说，它们直接与所认为的服务器通信，实际上他们是与代理服务器通信。客户端通常不知道其请求实际上是被代理处理的。

反向代理的实际用途：

1. **负载均衡**：反向代理可以将接收到的请求分散到多个后端服务器上，从而优化资源的使用和提高响应速度。
2. **增强安全性**：通过在服务器和互联网之间设置一个中间层，反向代理能够提供防火墙的功能和阻止不安全的网络请求。
3. **压缩和优化内容**：反向代理还可以在将内容发送给用户前进行压缩，减少数据传输量，加快加载速度。
4. 等等...

我们在Nginx中配置的静态资源和API代理就是一种反向代理。

05-描述如何使用NGINX作为解决跨域的一种方法，并概述其工作原理。

NGINX来解决跨域是我们在生产环境中常用的一种手段，在不同的需求下我们使用Nginx来解决跨域访问有两种方案：

方案一：Nginx仅仅代理API服务器。

在这种配置中，NGINX 承担了处理跨域问题的责任，通过在响应头中添加跨域资源共享（CORS）相关的HTTP头，使得前端应用能够从不同的源安全地请求API。

- **优势**：将API请求集中处理，可以简化后端服务的CORS配置，特别是在后端服务分布在多个服务器或服务上时。
- **工作原理**：NGINX 作为网关接收来自前端的API请求，并将其转发到实际的后端服务。通过重写请求的URL并

添加必要的HTTP头信息，NGINX 管理跨源请求，从而允许前端代码与后端API进行交互。

方案二：Nginx代理了静态资源和API服务器

当NGINX同时代理静态资源和API请求时，它可以为前端和后端提供一个统一的访问地址。

- **优势**：这种配置简化了前端的配置，因为所有请求都发送到同一个服务器。此外，它还可以通过缓存机制提高静态资源的加载速度。
- **工作原理**：静态资源和API请求都经过NGINX，然后根据请求的路径被适当地重写并转发到相应的后端服务。
- 这种方式减少了跨域请求的复杂性，因为从客户端的角度看，所有请求都是同源的。

具体的配置方案也可以给苗馆描述一下（不用非常详细，因为这个不需要记住，但是思路要有）：

配置反向代理：

- 首先，我们通过配置 Nginx 作为反向代理来解决跨域问题。
- 这意味着所有前端到后端的请求首先会被发送到 Nginx，然后由 Nginx 转发到实际的后端服务器。

修改配置文件：

- 在 Nginx 的配置文件中，我们会设置一个 `location` 块，专门用来处理前端的 API 请求。
- 在这个块中，我们可以指定代理传递到后端服务的 URL。

添加必要的 HTTP 头：

- 为了解决跨域问题，我们需要在 Nginx 的响应中添加一些特定的 HTTP 头，最重要的是 `Access-Control-Allow-Origin`。
- 这个头部信息指明了哪些域名（origin）被允许访问资源。