课程目标

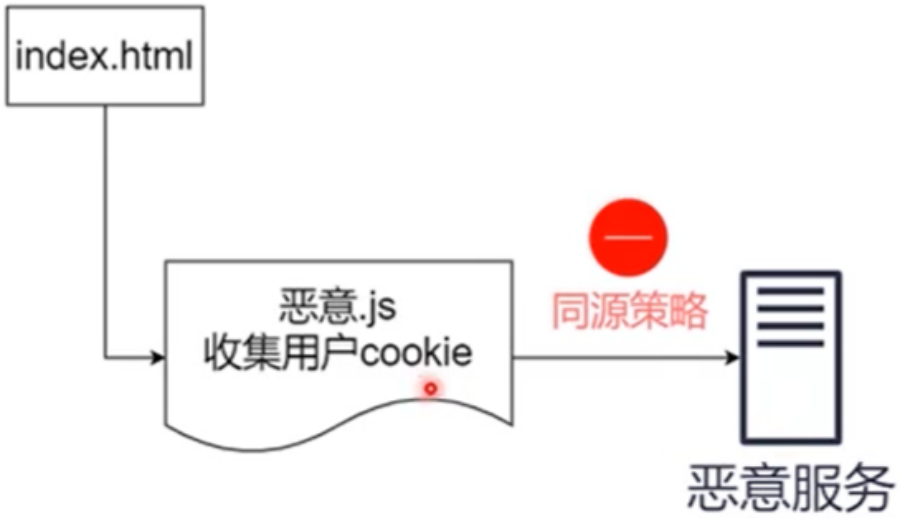
同源策略

定义:禁止一个源的脚本&文档和另外一个源的脚本&文档交互

两个URL的protocal，port和host相同，那么同源。不同源可能两家不一样的公司,即使有的公司的又两个不同源的地址，但是对于机器来说是不知道的。所以为了安全，都需要在同源环境下交互。如下

图

思考:如果两个源产生过多的交互会有什么问题?



如果在index里面有用到一个第三方的插件，第三方插件里里面有恶意的.js收集用户的cookie，发送到index.html的自己的服务器还这个可以，如果发送到其他的服务器的话，就是恶意服务。同源策略就限制了。

思考:

- **为什么不禁用不同源的js?(需要用到第三方库，不能直接禁用)**
- **应不应该允许不同源的js修改dom？(也是允许的，第三库应该允许修改dom)**
- **应不应该允许不同源的js获取远程图片内容？(不允许)**

- **应不应该允许网站提交数据到不同源的服务器？（不允许）**
- **应不应该允许网站提交cookie到不同源的服务器？（不允许）**

跨域的N种办法

定义:绕过同源策略，下面列举几种常用的

- **Jsonp**：本质是利用不限制跨域脚本执行的特点,利用跨域脚本传输数据,例如:script标签

下面是具体的json实现过程.

1. 首先在client端index.html页面写一个的json回调方法
2. 服务端动态 加载跨域数据脚本

```
var script = document.createElement('script')
script.setAttribute('src', 'data.js')
document.getElementsByTagName('head')[0].appendChild(script)
```

3. 通过加载跨域脚本的执行回到index页面的回调方法,传输数据就可以进行跨域数据传输.

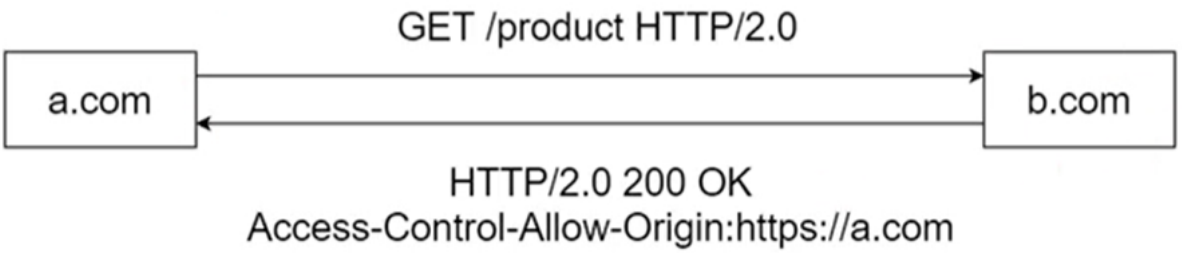
思考:

Jsonp可以用来提交数据吗?(从本质上来说,jsonp是get请求,是从标签里面的发送过去的,也是可以用来提交数据的,但是只能用get请求,因为你是直接加载一个文件,理论是很难去提取数据的,但是服务端可以有很多的实现,可以读到一个带字符的标签.根据不同的情况返回不同的script给你,这就相当于可以用来提交数据了,但是只能到get方式,请求参数放在url上,可以用get方式来提交数据,但是基本上不合适的.)

尝试为fetch函数扩展jsonp功能?

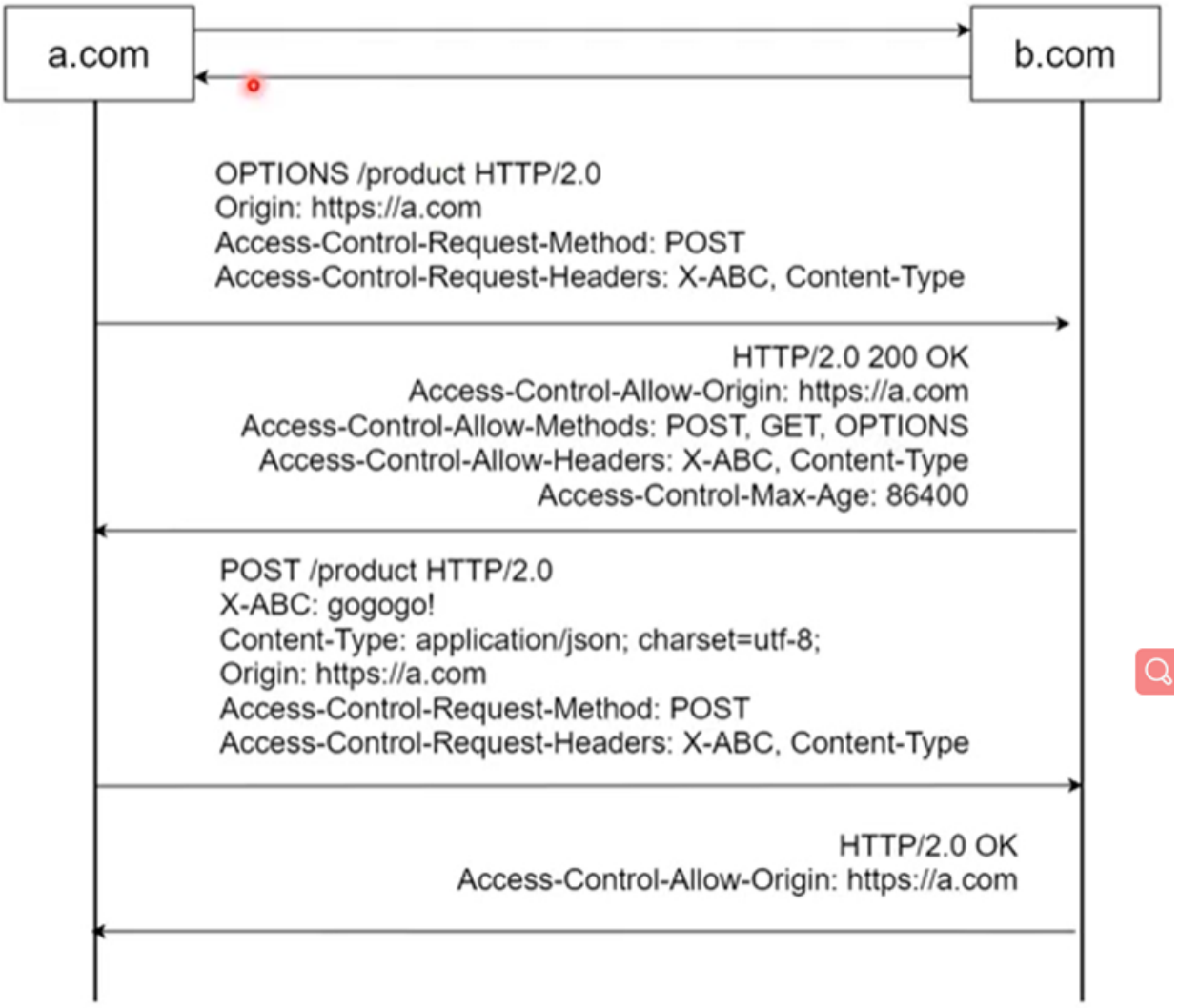
```
fetch(<jsonp-url>, {method : 'jsonp'})
    .then(data => {
      console.log(data)
  })
```

- **CORS(跨域资源公用 Cross-Origin Resource Shareing) :**使用额外HTTP头允许指定的源和另外一个源进行交互.



由此我们可以看出,同源是由浏览器控制的,浏览器看到服务端返回的Access-Control-Allow-Origin来进行相应的设置.

预检 (之前一些简单的get和post请求直接设置一下Access-Control-Allow-origin就可以进行跨域了,但是一些复杂的请求如同下图)

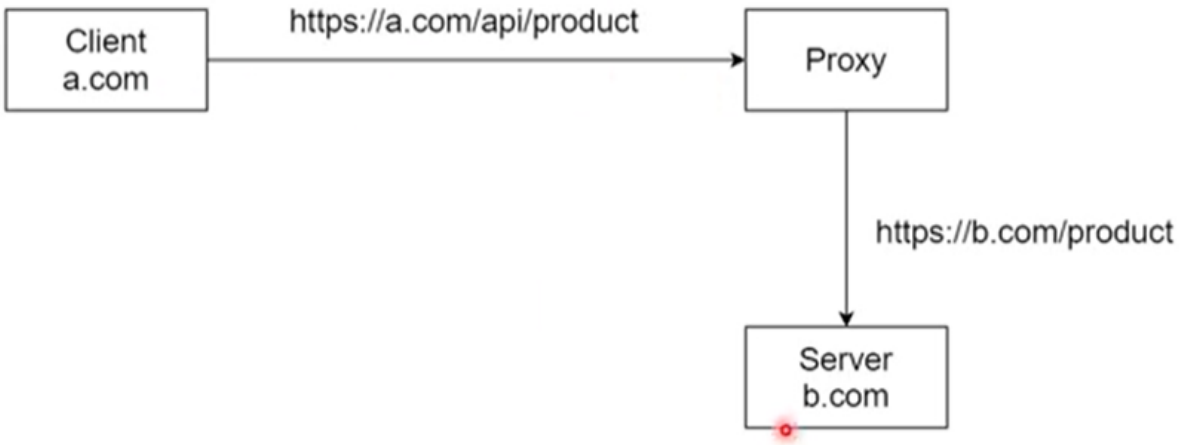


a.com向b.com发送请求,会发送一个自定义的头部,这时候是一个复杂的请求.a.com先发送一个的options请求, 先询问b.com可以发送POST请求吗?可以携带自定义的X-ABC 和Content-Type的headers.服务端先看一下是否能够支持客户端的要求,如果能够支持,返回

200 允许跨域的源 请求方式 请求自定义的headers 还有这个要求的有效时间 Access-Control-Allow-Origin.这时候client才会真正的发送数据.server端才返回真正的数据,
总体是CORS分为两种一种简单请求.一种复杂请求(需要发options请求),这个浏览器说了算,如果是简单请求,浏览器直接发过去了.

- **代理**

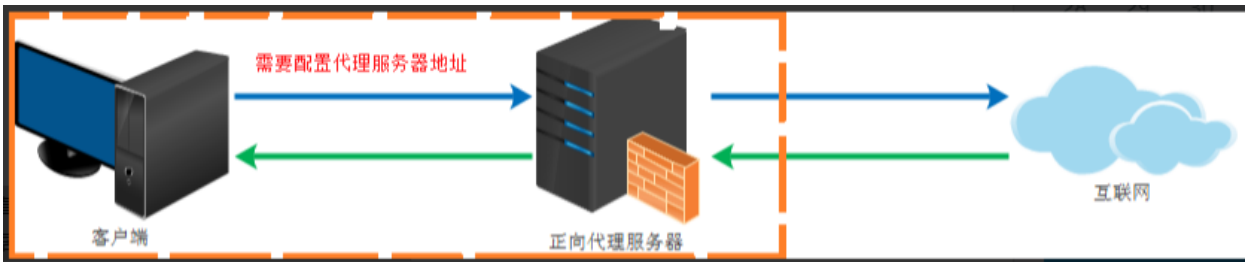
定义:代理将不同源的资源代理到同源的资源.node非常适合做这个代理,现在前端工程师建立一个node层来处理跨域.



如图,client先发请求到代理,然后再通过Proxy转发到的Server 一般可以通过nginx反向代理,代理不是浏览器，所以代理可以直接请求server端,没有管它.server数据回到代理,代理再回到client端,这样没有违反同源策略.代理没有options请求,但是代理把整个链路加长了.本来是一对一,现在多了一个中间的传话者.

正向代理：正向代理类似一个跳板机，代理访问外部资源。

比如我们国内访问谷歌，直接访问访问不到，我们可以通过一个正向代理服务器，请求发到代理服，代理服务器能够访问谷歌，这样由代理去谷歌取到返回数据，再返回给我们，这样我们就能访问谷歌了



用途：

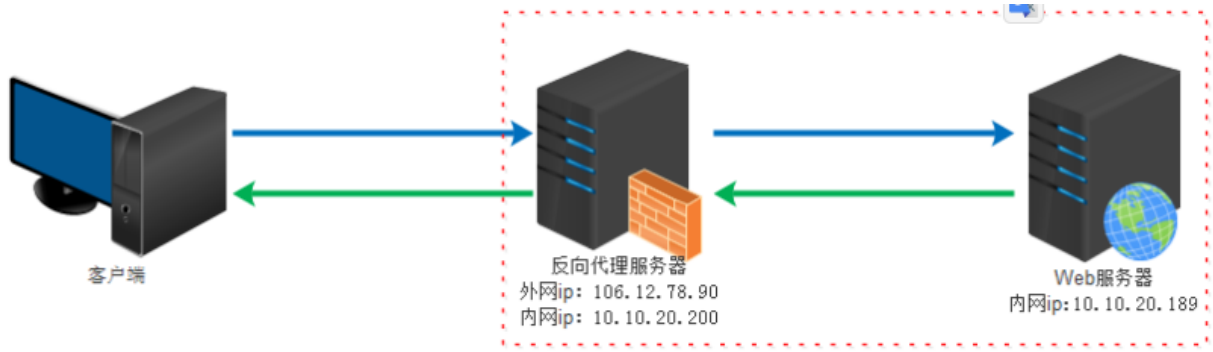
（1）访问原来无法访问的资源，如google

（2）可以做缓存，加速访问资源

（3）对客户端访问授权，上网进行认证

（4）代理可以记录用户访问记录（上网行为管理），对外隐藏用户信息

方向代理: 反向代理（Reverse Proxy）实际运行方式是指以代理服务器来接受internet上的连接请求，然后将请求转发给内部网络上的服务器，并将从服务器上得到的结果返回给internet上请求连接的客户端，此时代理服务器对外就表现为一个服务器



用途：

（1）保证内网的安全，阻止web攻击，大型网站，通常将反向代理作为公网访问地址，Web服务器是内网

（2）负载均衡，通过反向代理服务器来优化网站的负载

总结：

正向代理即是客户端代理, 代理客户端, 服务端不知道实际发起请求的客户端.proxy和client同属一个LAN，对server透明；

反向代理即是服务端代理, 代理服务端, 客户端不知道实际提供服务的服务端。proxy和server同属一个LAN，对client透明。

实战CORS(Fetch+node.js)

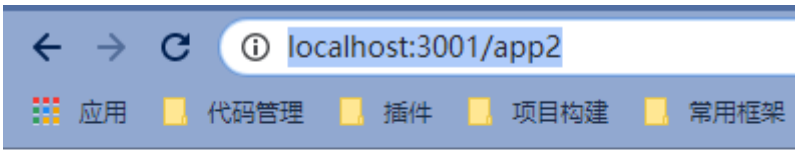
- **观察node在服务端实现CORS跨域**
- **观察浏览器中fetch的使用方法**
- **观察options预检要求**

服务端启动两个不同服务:

```
const express = require('express')
const app1 = express()
const app2 = express()
//app1 app2服务
app1.get('/app1',(req,res)=>{
 res.send('进入app1服务')
```

```
})

app1.listen(3000,()=>{
  console.log('进入3000服务')
})


app2.get('/app2',(req,res)=> {
  res.send('进入app2服务')
})

app2.listen(3001,()=>{
  console.log('进入3001服务')
})

```

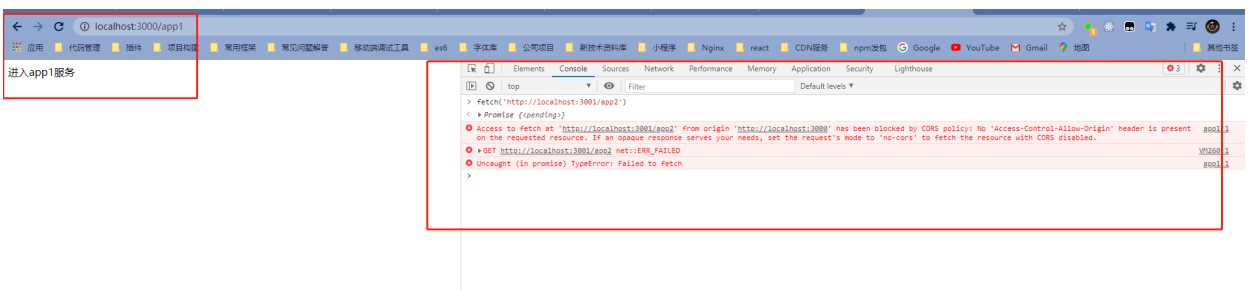
客户端请求app1和app2服务



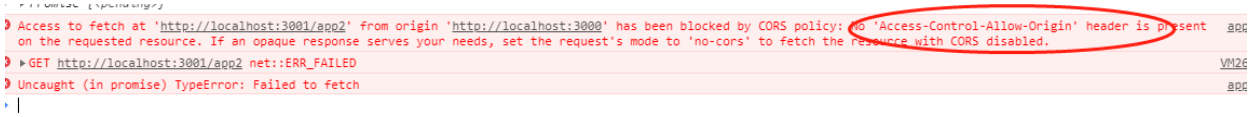
进入app2服务

进入app1服务

在3001服务中使用fetch方式请求3002服务就出现同源策略限制(跨域)



浏览器提示信息说.可通过设置respone头部来的解决跨域.



通过的设置Access-Control-Allow-Origin 只是get和post简单请求 可以解决同源限制,put方式是复杂请求,还得在options请求设置.如代码所示

```
app2.get('/app2',(req,res)=> {
    res.set("Access-Control-Allow-Origin","http://localhost:3000")
    res.send( data: '进入app2服务')
})

app2.post('/app2',(req,res)=> {
    res.set("Access-Control-Allow-Origin","http://localhost:3000")
    res.send( data: '进入app2服务')
})
```

在http://localhost:3000/源上再去请求http://localhost:3001/app2

```
> fetch('http://localhost:3001/app2')
< ▶ Promise {<pending>}
>
```

```
> fetch('http://localhost:3001/app2',{method:"POST"})
< ▶ Promise {<pending>}
>
```

就没有出现跨域的相关提示了.

如果我们需要设置headers中的content-type生效,需要先设置的 options请求.因为这是一个复杂的请求,如上图所示,得先进行一次options请求,所以得先服务端设置得 跨域 和 头部设置

```
app2.options('/app2',(req,res)=> {
    res.set("Access-Control-Allow-Origin","http://localhost:3000")
    res.set("Access-Control-Allow-Headers","content-type")
    res.send( data: '进入app2服务')
})
```

设置完后,就不会出现同源限制阻塞了

```
> fetch('http://localhost:3001/app2',{method:'POST',headers:{'Content-type':'application/json'}})
< ▶ Promise {<pending>}
>
```

前面说了get 和post简单请求可以通过设置Access-Control-Allow-Origin ,但是put请求还得设置

```
app2.options('/app2',(req,res)=> {
    res.set("Access-Control-Allow-Origin","http://localhost:3000")
    res.set("Access-Control-Allow-Headers","content-type")
    res.set("Access-Control-Allow-Methods","PUT")//不能限制get和post还有head请求，所以get post head无需设置此项
    res.send( data: '进入app2服务')
})
```

```
> fetch('http://localhost:3001/app2',{method:'PUT'})
< ▶ Promise {<pending>}
>
```

总结: 使用CORS进行跨域得时候,要区分简单请求和复杂请求,简单请求如如get和post server端只需要设置Access-Control-Allow-Origin相应即可跨域. 复杂请求 例如 设置header put delete等等 都需要在options进行相关设置.

思考:

1. 每次请求携带token,client和server该如何设置?

```
app2.options('/app2',(req,res)=> {
    res.set("Access-Control-Allow-Origin","http://localhost:3000")
    res.set("Access-Control-Allow-Headers","content-type,token")
    res.set("Access-Control-Allow-Methods","DELETE,PUT")//不能限制get和post还有head请求，所以get post
    res.sendStatus( statusCode: 200)
})
```

2.两个子域之间的交互算不算跨域? 子域和子域之间算跨域.子域和父域也会跨域.
即使是共源,子域和父域 子域和子域之间都交互,只要不同源,就会出现跨域

client设置{mode:"no-cors"} 透明请求,虽然是通过了,但是还是没有绕过同源策略.

通过设置 Access-Control-Allow-Origin:"*" 可以设置所有子域名和父域名绕过同源策略
这是不安全的.这只适合开发模式.

```
app2.get('/app2',(req,res)=> {
    res.set("Access-Control-Allow-Origin","*")
    res.send( data: '进入app2服务2')
})
```

Access-Control-Allow-Origin:会设置指定的域名还有子域名,保证安全性和可靠性.

课程小结

- 理解跨域要解决的问题而不是记住跨域这个现象和处理方法
- 子域名和父域名交互的时候因为子域名可以修改自己的域为父域名可以解决跨域问题
- 理解策略要解决的问题，而不是记住处理方法
- 决定用哪种策略核心的原则是在安全性前提下获得最优效率