前端 HTTP 学习指南

视频

HTTP1.1-PART1-Overview

HTTP1.1-PART2-Basic Concept

HTTP - PART3 - Transfer and Encoding

HTTP - PART4 - Cache

HTTP - PART5 - CORS

写在前面

文章会从前端的视角分享一些HTTP相关的知识, 希望能帮助读者对HTTP协议建立更健全的认知, 为了实现更好的学习效果, 推荐阅读本文的同时希望读者也能做一些简单的尝试或者写一些代码来辅助理解, 对于模糊的点可以在提供的学习资料中寻找, 也可以抛出来大家一起交流.

本文的实例代码都可以在 nzhl/http-learning-guide 找到. 为了方便快速实践demo, 至少以下工具是必要的:

> 1. Node 环境
> 2. 代理, 推荐 http://bifrost.bytedance.net/

1. 概览

1.1 学习资料

https://developer.mozilla.org/en-US/docs/Web/HTTP （文章大部分内容的参考, 一定程度上, 可以认为本文是精简+意译版的 MDN）

RFC 7230 - Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing (HTTP1.1 规范)

https://book.systemsapproach.org/congestion.html

1.2 特点

HTTP 协议是一个应用非常广泛的协议, 它本身非常简单, 但是一定程度上仍然能够保证:

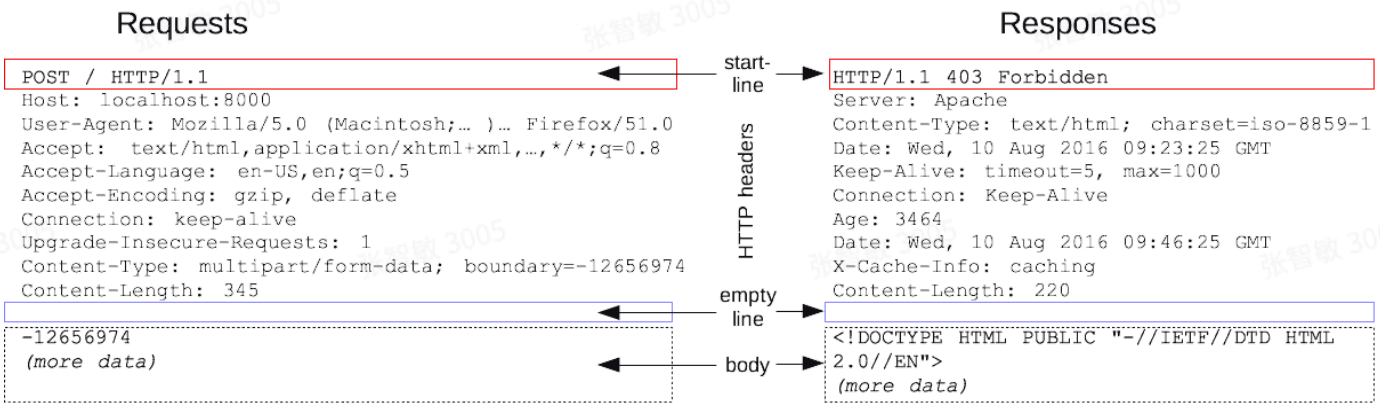
1. 可靠性: 通常基于TCP协议实现, TCP保证了其传输的可靠性
2. 安全性: 本身明文传输, 并不保证安全, 主要依赖 HTTPS
3. 有状态: 设计上是无状态的, 不过现实场景中通常通过 Cookie 来保证有记忆

1.3 结构

HTTP是典型的CS模型, 协议规定了 Client 和 Server 之间数据交换的方式, 形式上非常简单, 从 Client 发起请求, Server 根据 Client 的请求作出响应. 由于HTTP是明文传输, 报文内容都是可读的, 所以可以说是非常容易理解的.

请求和响应无一例外都由三部分组成:

1. Start Line (Request) / Status Line (Response)
2. headers
3. Body



1.4 实现一个简易的 Client & Server

1.4.1 分析需求

Ok, 现在已经拥有了足够的知识, 来尝试实现一个简易的 Client & Server, 先来分析下工作:

1. Client 需要发送 HTTP 请求, 并收到来自 Server 的响应, 常见的例子有 浏览器, Postman, curl 之类.

2. Server 需要监听某个端口, 并随时对到来的请求作出回应, 常见的例子有 你访问的任何一个网站, 后端起的API服务, 甚至包括你开发用的devServer.

1.4.2 指导文档

https://nodejs.org/api/net.html

1.4.3 代码实现

相关的代码可以在文章首部给出的代码仓库中的 src/example1 中找到

1. 首先根据上面报文结构的说明, 简单生成下报文, 这里直接写死

TypeScript

```typescript
// http.ts

// https://stackoverflow.com/questions/5757290/http-header-line-break-style
const LINE_SEPERATOR = '\r\n';

const createHttpMessage = ({
  startLine,
  headers,
  body,
}: {
  startLine: string;
  headers: string;
  body: string;
}) => {
  const headersWithLength =
    headers + `Content-length: ${body.length}` + LINE_SEPERATOR;
  return startLine + headersWithLength + LINE_SEPERATOR + body;
};

export const REQUEST = createHttpMessage({
  startLine: 'GET / HTTP/1.1' + LINE_SEPERATOR,
  headers: 'Host: github.com' + LINE_SEPERATOR,
  body: '',
});

export const RESPONSE = createHttpMessage({
  startLine: 'HTTP/1.1 200 OK' + LINE_SEPERATOR,
  headers:
    'Server: FredHomeMadeServer Node/14' +
    LINE_SEPERATOR +
    'Content-type: text/html' +
    LINE_SEPERATOR,
  body: `<html> <body> <h1> hello wrold </h1> </body> </html>`,
});
```

2. 实现一个客户端, 大致工作:
   a. 初始化一个 Socket 并与某个服务器建立连接
   b. 发送请求报文
   c. 接受响应报文

TypeScript

```typescript
// https://stackoverflow.com/questions/5757290/http-header-line-break-style
const LINE_SEPERATOR = '\r\n';

const createHttpMessage = ({
  startLine,
  headers,
  body,
}: {
  startLine: string;
  headers: string;
  body: string;
}) => {
  const headersWithLength =
    headers + `Content-length: ${body.length}` + LINE_SEPERATOR;
  return startLine + headersWithLength + LINE_SEPERATOR + body;
};

export const REQUEST = createHttpMessage({
  startLine: 'GET / HTTP/1.1' + LINE_SEPERATOR,
  headers: 'Host: myfakehost' + LINE_SEPERATOR,
  body: '',
});

export const RESPONSE = createHttpMessage({
  startLine: 'HTTP/1.1 200 OK' + LINE_SEPERATOR,
  headers:
    'Server: FredHomeMadeServer Node/14' +
    LINE_SEPERATOR +
    'Content-type: text/html' +
    LINE_SEPERATOR,
  body: `<html> <body> <h1> hello world </h1> </body> </html>`,
});
```

尝试运行代码，应该能看到 302 重定向, 这是因为 github 希望我们使用 https 访问其站点

CSS

```
  dev:client ~~ request sent ~~ +0ms
  dev:client request write finish +1ms
  dev:client HTTP/1.1 301 Moved Permanently
  dev:client Content-Length: 0
  dev:client Location: https://github.com/
  dev:client
  dev:client  +302ms
  dev:client response read end +0ms
  dev:client socket close +0ms
```

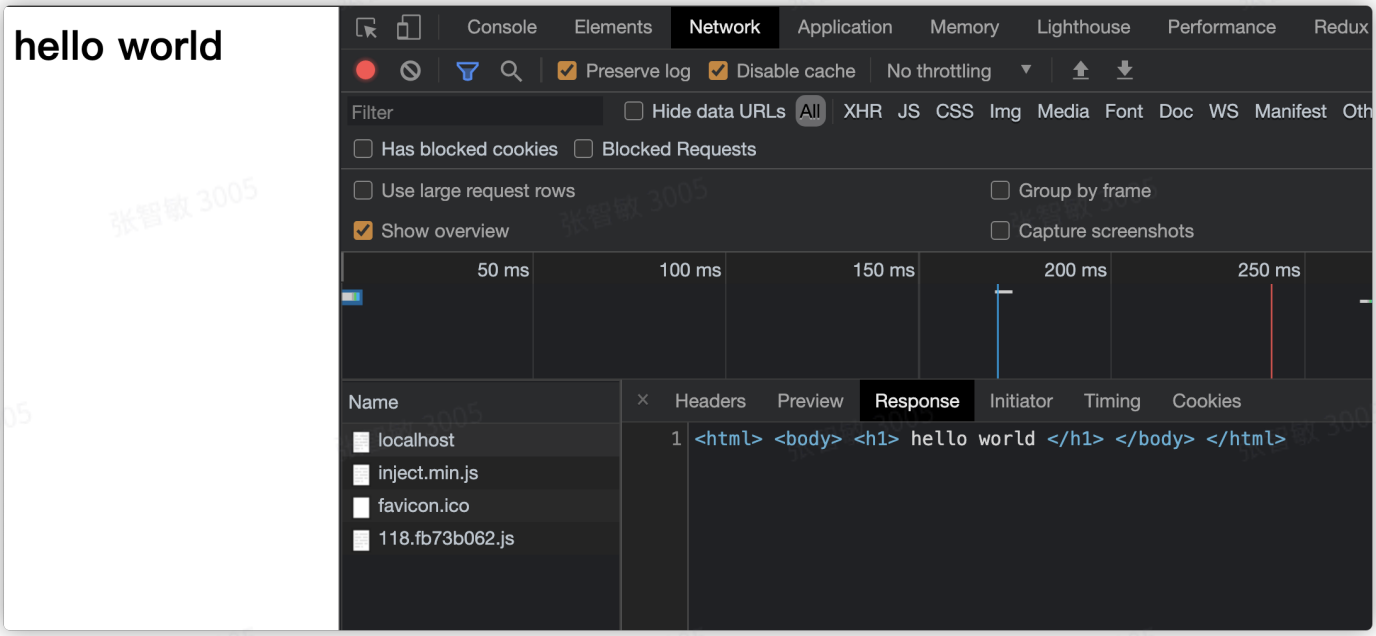
3. 实现一个服务器, 大致工作:
   a. 初始化一个 Server 并监听某个端口
   b. 解析发来的请求
   c. 作出恰当的响应

TypeScript

```ts
// server.ts

import { Server } from 'net';
import debug from 'debug';
import { RESPONSE } from './http';

const d = debug('dev:server')


export const startServer = (port: number, host: string, cb: () => void) => {
  const server = new Server();

  server.on('connection', reqSocket => {
    d('~~ request coming ~~');
    let tmp = Buffer.alloc(0);

    reqSocket.on('data', chunk => {
      tmp = Buffer.concat([tmp, chunk]);
      const tmpStr = tmp.toString();

      // check end of http
      // https://httpwg.org/specs/rfc7230.html#rfc.section.3.3.3
```

```
23        if (tmp.includes('Content-Length')) {
24          // todo
25        } else if (tmp.includes('Transfer-Encoding')) {
26          // todo
27        } else if (tmpStr.endsWith('\r\n\r\n')) {
28          // https://github.com/jinhailang/blog/issues/34
29          // body empty
30          d(tmp.toString())
31          reqSocket.end(RESPONSE);
32        }
33      });
34
35      // no more content to read
36      reqSocket.on('end', () => {
37        d('request read end')
38      });
39
40      // no more content to write
41      reqSocket.on('finish', () => {
42        d('response write finish')
43      });
44
45      reqSocket.on('close', () => {
46        d('socket close')
47      })
48    });
49
50    server.listen(port, host, () => {
51      d(`server listening ${host}:${port}`);
52      cb();
53    });
54  };
55
56  startServer(10086, '0.0.0.0')
```

打开浏览器, 访问 http://localhost:10086/, 应该该能看到



1.4.4 小练习

1. 沿着上面的思路, 改造 Client, 实现 Postman 儿童版 Postchild, 能自定义发送HTTP请求.
2. 沿着上面的思路, 改造 Server, 实现 http-server 简陋版, HOST 当前目录.

2. 基础的概念

下面介绍HTTP协议中一些基础的概念, 当然为了不让文章变得过于枯燥, 只会有个简要的描述, 其中一些重要的概念会在下一节中结合实际场景被强调.

2.1 学习资料

HTTP request methods - HTTP | MDN

HTTP headers - HTTP | MDN

HTTP response status codes - HTTP | MDN

理解RESTful架构 - 阮一峰的网络日志

RESTful API 设计指南 - 阮一峰的网络日志

Architectural Styles and the Design of Network-based Software Architectures

2.2 请求方法

2.2.1 GET, POST, DELETE, PUT, PATCH

这几个很常见就不说了, 直接用上面给的 HTTP request methods - HTTP | MDN 看下就很清晰了

2.2.2 OPTIONS

1. 查询目标服务器支持的请求方法.
2. 浏览器发送跨域之前, 提前咨询是否即将发送的请求是被允许的.

2.2.3 CONNECT

主要用于建立代理隧道, 在代理工具中会比较常见

2.2.4 HEAD

对于服务器而言, 接收到 HEAD 请求的处理方式应等同于 GET 请求, 唯一的区别是响应只需要包含响应头, 不需要返回 body, 效果类似 `curl -I` .

2.2.5 TRACE

服务器需要把自己收到的请求体原封不动地回传给客户端, 初衷是用来做调试, 不过由于历史上存在安全风险(见 Exploitation of Trace method), 当然部分风险在现代浏览器上已经可以规避了, 不过基于安全考虑在大部分服务器上TRACE方法仍然是禁用的.

2.3 请求/响应头

头信息为http交互过程提供了额外的信息, MDN列出了 Headers 的多种分类方式, 感觉刻意去记忆这些分类意义不大, 更多是理解每个字段的意义, 这个后面结合场景来讲. 这里只讲两点:

1. Headers的名字是大小写不敏感的, 名字和值靠冒号隔开, 值签名的空格会被忽略.
2. 我们在业务里面通常会使用 X- 来命名自定义的请求头, 不过这个习惯已经被标记为废弃了, 因为使得后续这个非标准请求头变为标准时候, 命名的变更会非常麻烦.

2.4 响应状态码

1xx

1. 101: 更换协议, 比如 websocket 建立连接的时候需要发送一个HTTP的握手包, 此时如果服务器能理解并同意升级, 就会返回101

2xx

成功相关

1. 200: 请求成功
2. 206: 部分内容, 一般在分片传输的过程中用到, 后面具体讲

3xx

重定向相关的, 比较常见的比如:

1. 301, 308: 永久重定向, 后者规定重定向后必须使用相同的请求方法
2. 302, 307: 暂时重定向, 后者规定重定向后必须使用相同的请求方法
3. 304: 内容没变: 直接使用缓存作为响应, 细节后面结合HTTP Headers 具体讲.

4xx

客户端错误, 比较常见的比如:

1. 400: 错误的请求, 一般是请求格式不对或者多了少了参数
2. 401: 未授权, 一般是身份校验没过
3. 403: 被禁止, 和401有点像, 但这里更多用作服务器了解客户端的身份但是仍然禁止, 譬如写爬虫未经允许强行爬简历
4. 404: 找不到, 一般是Url对应的页面, 资源没了, 通常是访问了已经下线或者不存在的 Url
5. 405: 方法不允许: 使用了服务器不允许的请求方法

5xx

服务端错误, 比较常见的比如:

1. 500: 服务器内部错误, 通常是服务器逻辑有问题, 内部出现了没法处理的异常
2. 502: 网关错误, 通常是服务端针对你的请求给出了非法的响应, 此时作为中间层的代理或网关将直接返回502, 譬如有时候用 whistle 代理访问后端开发机, 后端直接关机了, 就会返回502.
3. 503: 服务不可用, 通常是服务器过载了, 比如你的博客突然上了微博热榜, qps直接上百万, 这时候大概率你的服务器会返回503.
4. 504: 网关超时, 指服务器无法在给定的时间内给出响应, 此时作为中间层的代理或网关将直接返回504, 譬如有时候服务器访问数据库的耗时超过了 nginx 的默认超时时间, 就会返回504.

2.5 RESTful API

2.5.1 概念

一套比较流行的 HTTP API的设计规范, 大致规则如下:

1. 用 URI 表示资源

2. 用 HTTP METHOD 表示对资源执行的动作
    - 创建：POST
    - 删除：DELETE
    - 完整修改: PUT
    - 部分修改: PATCH
    - 查询：GET

2.5.2 示例

GITHUB的API给了一个很好的示范: Projects - GitHub Docs

```CoffeeScript
GET /projects/{project_id}  获取一个项目
PATCH /projects/{project_id} 更新一个项目
DELETE /projects/{project_id} 删除一个项目
```

2.5.3 优点

简单清晰, 可读性高, 使用广泛.

2.5.4 局限性

有些场景下基于资源的抽象不够直观, 比如我们常见的 "登录, 登出" 接口, 按照RESTful规范得:

1. `登陆: POST /user/session`
2. `登出: DELETE /user/session`

但真这么设计反而很费解, 现实中往往会直接用 `/api/login` / `/api/logout` 会清晰.

3. 传输&编码相关

3.1 Content-Type & Accept

首先在 HTTP 1.x 中, 其头部肯定是 ASCII 编码的, 但是由于HTTP传输的内容多种多样, 可以是各种类型的文件 (html, css, js, jpg, mp4等等, 更具体的可以看看 https://en.wikipedia.org/wiki/MIME), 为了让接收方清楚地了解HTTP BODY中携带的数据格式, 这就有了 Content-Type 字段. 一些常见的例子:

HTTP

```
# BODY里面是HTTP文本, 编码方式是 UTF-8
Content-Type: text/html; charset=UTF-8

# BODY里面是图片
Content-Type: image/jpeg

# BODY 里面是POST FORM-DATA
Content-Type: multipart/form-data; boundary=------------------------77f344aea6123ed5
```

提一下最后一个, multipart/form-data, 即带了多块内容的请求体, 这时候侯后面的boundary定义了如何对多块内容进行区分.

当然, 发送之前也要考虑接收端的兼容性, 所以这就有了另一个字段 `Accept`, 它用于客服端告诉服务端, 哪些内容对于他来说是符合预期的. 常见的例子譬如

Apache

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
```

可以看到有个很神奇的 `;q=` , 这玩意后面可以接一个数值, 表示客户端对于内容的偏好权重. 不写默认是 1.0 (最高), 所以对于上面这个例子来说, 他们的优先级权重分别是:

HTTP

```
text/html,application/xhtml+xml 1.0
application/xml 0.9
image/avif,image/webp,image/apng 1.0
*/* 0.8
application/signed-exchange;v=b3 0.9
```

之前俊斌提到的, 云服务 CDN 的智能选择最佳图片格式能力, 实际上就是根据这个字段来实现的, 可以看看 格式转换 - CDN - 阿里云

3.2 Content-Encoding & Accept-Encoding

这个字段第一次看以为是描述字符编码的, 然而并不, 其实是用来描述数据传输过程中所使用的压缩算法的, 常见的算法包括:

- gzip
- compress
- deflate
- bz

HTTP

```
# 直接gzip压缩
Content-Encoding: gzip
# 先deflate再gzip二次压缩
Content-Encoding: gzip, deflate
```

当然 和 Content-Type 一致, Content-Encoding 也有个协商字段叫 `Accept-Encoding` , 它的作用是向对方表明自己接受的压缩方式, 例如

> Accept-Encoding: gzip, deflate

3.3 Connection & Content-Length & Transfer-Encoding

讲 `Content-Length` 之前先要提到一个字段叫 `**Connection**`, 它只有两种值 `close` 和 `keep-alive`, 它用于描述完成当前当前的传输任务之后, 该链接的行为. 前者表示关闭链接, 后者表示保持链接. HTTP 1.1 默认就是 keep-alive, 好处很明显, 那就是可以方便地复用 TCP 进行其他后续请求的网络传输.

这样就带来了一个新的问题, 因为连续的多个HTTP请求都是由同一个 TCP 连接承载的, 必须有个手段判断上一个HTTP相关的内容已经结束, 这就是所谓的 Content-Length, 事实上, 就如同你在第一节看到的那样, 如果你的Content-Length乱写, 比如内容比Content-Length中声明的更长, Chrome 会错误地判断你的内容已经结束, 你会发现多余的内容都无法正常现实.

当然还有另外一种情况, 有时候我们并不能提前知道我们的内容有多长, 譬如我们用 Node 服务器做 ssr 的时候, 为了更好的性能, 这个过程很有可能是流式的, 这时候就需要用到另一个字段 `Transfer-Encoding: chunked`, 它的作用是声明消息体的长度是不确定的. 一旦开启了分块传输, 接下来每个数据块会由一个16进制的数字开头跟随一个CRLF(即\r\n), 然后是内容, 然后是 (/r/n) 表示结束, 最后一个数据块必须是 `0\r\n\r\n` 即空白块, 表示消息体整个已经结束.

3.4 Accept-Ranges & Range & Content-Range & If-Range

有些时候文件过大, 为了带来更好的传输体验, 就会用到分片传输, 这里要注意区分上面所提到的 `Transfer-Encoding: chunked`, 上面指的是将一个 HTTP 请求的 Body 分成很多块, 分别传输, 本质上还是属于同一个请求. 这里指的是用多次HTTP来传输同一份文件(内容). 它的步骤大体是这样:

1. 首先需要确认服务器确实支持分片上传, 如果服务器支持分片上传的话, 响应头中会包含 Accept-Ranges: bytes, 没有这个头部或者值为 none 的话表示不支持.
2. 接下来客户端可以发送自己需要的内容部分, 这里需要用到 `Range` 字段, 大概长这样

Apache

```
# 我需要第100到200字节的内容
Range: bytes=100-200

# 我需要第100到结尾的内容
Range: bytes=100-

# 我需要100到200 以及 300到400的内容
Range: bytes=100-200, 300-400
```

3. 对于服务器而言, 也分以下几种情况进行处理:
   a. 自己不支持分片, 不过文件有, 可以返回200, 然后返回整个文件
   b. 发现文件压根没有这个范围的内容, 比如请求中声明所需要的范围超出了文件的实际长度, 此时返回416表示范围不合法
   c. 如果发现自己确实有这一块的内容, 也支持分片传输, 此时可以返回状态码 206 (第二节中提到的那个), 然后返回内容, 对于指定了多块内容的情况, Content-Type 中会返回 multipart, 对着这种 case, 响应头中会带上与 `Range` 对应的 `Content-Range` 字段

HTML

```
# 总文件大小是1000，返回了100~200的部分
Content-Range: bytes 100-200/1000

# 不知道总文件大小，但返回了100~200的部分
Content-Range: bytes 100-200/*
```

4. 当然, 为了完整性这里还要提一嘴 If-Range 字段, 它的作用是使得 Range 字段条件性生效. 换句话说, 它用于检测当前客户端保存的那部分内容是否还是新鲜的, 比如对于1个1000bytes的文件, 客户端目前拿到的部分是 0~400, 它在向客户端发送请求的时候可以这样:

Apache

```
# 文件在劳动节11点半之后发生过变化吗
# 当然 If-Range 的值还可以是 ETag
If-Range: Sat, 1 May 2021 11:30:00 GMT

# 如果没有，返回从401开始的那部分内容给我
Range: bytes=401-
```

3.5 Content-Language & Accept-Language

首先是 `Content-Language`, 它表示服务器根据你的 `Accpet-Language` 最终选择的语言, 我一开始其实是有点误解, 以为 Content-Language 指的单纯就是返回内容的语言, 然而, 按照MDN的说法, Content-Language 返回的**并不一定**是当前页面所使用的的语言, 而只是标明页面的用户的母语. 更具体来说, 一群学习英语的德国人在英语学习网站上看到的内容可能是英语, 但是此时 Content-Language 极有可能是 `de-DE`, 当然如果没有指明的话, 表示下发的文档适合所有语种的用户. 标明当前文档内容本身语言的是另一个叫 lang 的玩意, 需要声明在 html:

HTML

```
1  <html lang="zh-CN">
```

至于 `Accept-Lanuage` , 它用于表明用户对于语种的偏好, 这个值通常取决于你的浏览器设置, Chrome 的话, 可以打开设置搜索语言:

更具体的, 这个字段其实长这样:

Apache

```
Accept-Language: en,zh-CN;q=0.9,zh;q=0.8
```

根据 Content negotiation - HTTP | MDN 的描述, 最佳实践应该是实际上技术人员应该在页面上提供一个语言选择的菜单, 而只是把 `Accept-Language` 作为兜底的情况, 因为实际上很多用户并不知道要怎么修改这个字段. 此外实际我观察下来, 无论是 google 还是 Stack Overflow, 浏览器虽然都在请求时把 Accept-Language 给带上了, 但是实际的 Content-Language 却并没有返回.

4. 缓存相关

4.1 学习资料

HTTP caching - HTTP | MDN

4.2 概述

缓存可以认为是一个计算机领域非常常见的优化策略, 一方面它能有效缓解了服务器的负载压力, 另一方面变相提升了服务的性能, 改善了用户体验. 但是与此同时, 它也引入了一些复杂性, 比如如何保证合理地配置缓存而避免将”过期“的信息呈现给用户呢?

受到篇幅限制, 这一节侧重点会放在与 HTTP 相关的缓存知识, 另一方面即使是这样, 也会选择性地删减掉一些具体细节, 此外组里已经有了一篇不错的 HTTP 缓存科普文章, 大家可以对照着看, 互为补充.

4.3 适用范围

缓存虽然好, 但是要明确的是, 并不是所有的场景都适合做缓存, 合适缓存优化有几个条件:

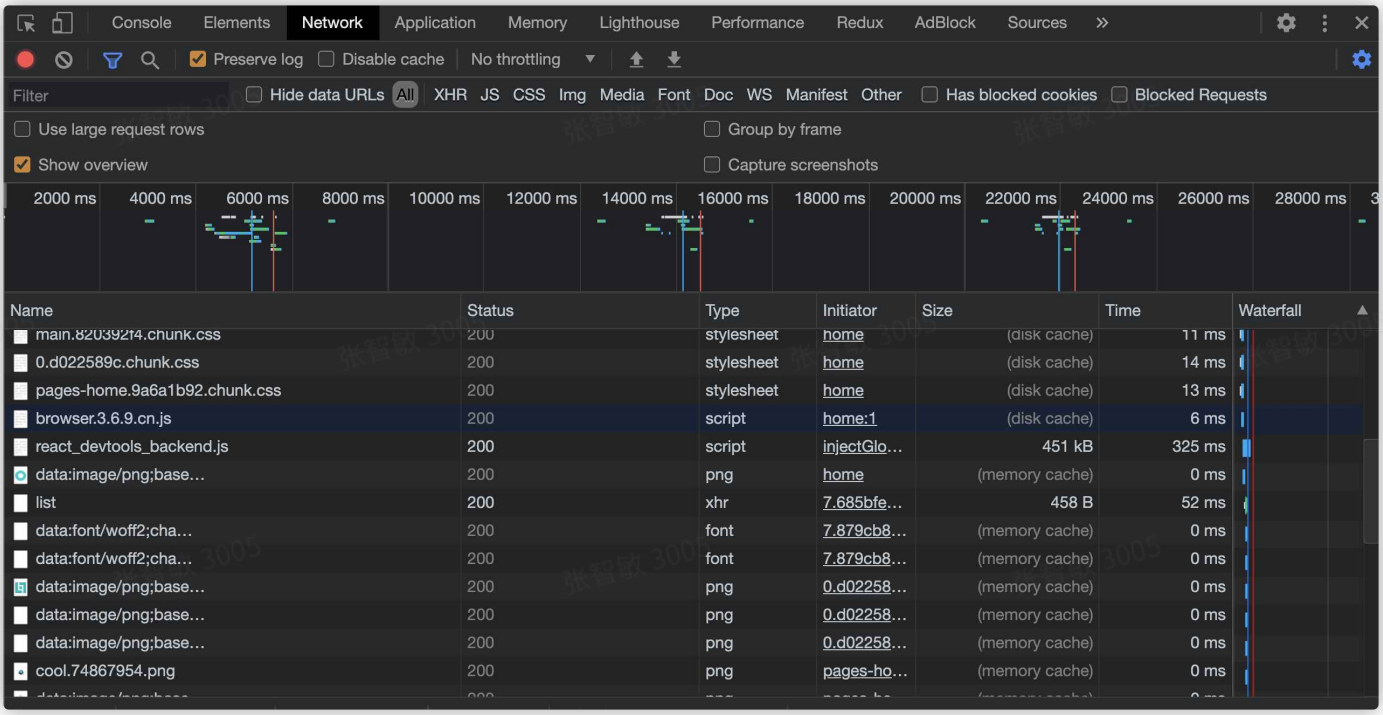
1. 首先第一点, 缓存更多是针对资源的读取, 资源的更新/修改的动作本身是很难缓存的
2. 进一步讲, 必须确保读取到的资源在一定时间内, 一定(用户)范围内是可复用的, 因为只有这样的场景缓存优化才有意义.

具体到HTTP协议, 可以看出来大部分情况下只有 GET 请求做缓存优化才有意义. 原因是 GET 请求往往表示对 HTML / JS / CSS / 图片 / 某段JSON 等等资源的获取, 其次它本身对服务器也没有副作用, 所以 GET 请求的结果相对来说会更可复用.

4.4 分类

在MDN上缓存被分成了两类: Private Cache (也叫 Local Cache) 和 Shared Cache.

1. Private Cache 指的是针对单个用户(准确来说是每个浏览器)的缓存, 例如经常能在控制台中看到的 Memory/Disk Cache, 这属于浏览器自身对于资源的缓存, 因为存储在本地, 所以又叫做 Local Cache. 浏览器会按照一定的规则来决定是否复用这些缓存, 这些规则和我们下面要讲的HTTP 请求头是息息相关的.



2. Shared Cache 指的是可以被多方复用的 cache, 具体来说, 从浏览器(客户端)到最终的服务器之间可能存在用户配置的代理, 各级 CDN 的节点, 以及服务端的反向代理, 理论上他们都可以有自己的缓存, 这一类缓存都算是 Shared Cache. HTTP Headers 中有些指令是专为 Shared Cache 准备的.

了解了这个概念, Cache-Control的这两个指令就很容易理解了:

- Cache-Control: public: 表示任何中间商(浏览器, 代理, CDN 等), 一些不常被缓存的case (譬如post请求)都可以缓存.

  这里刘华提了个有意思的问题, CDN是如何缓存HTTPS内容的, 之前也没思考过这个问题, 就大致搜索了下, 其实本质上还是MITM Can https content ever be cached at CDN datacentres?
- Cache-Control: private: 表示只有浏览器能缓存

4.5 具体的过程

弄清了缓存的分类, 我们直接来看看缓存应用的具体流程, 如果从用户发起请求来看, 整个缓存的逻辑分以下几个阶段

4.5.1 Service Worker

Service Worker是一种特殊的 Worker, 它可以拦截从当前页面发出的请求, 所以本质上可以认为它实际可以算是架设在浏览器侧的代理服务器. 虽然不能直接访问 DOM, XHR, localStorage, 但是却提供了 cache 相关的存储 API. 不过从概念上并不算是HTTP协议的一部分, 另一方面兼容性也是不算特别好, 感兴趣的同学可以看看 Service Worker API - Web APIs | MDN

4.5.2 不缓存

即 Cache-Contol: no store，表示直接跳过缓存环节, 直接向服务端请求最新的内容. 不过要注意如果你是想真的拿到服务端响应的最新资源, 直接 no-store 可能不太好使, 参见

> Note that this will not prevent a valid *pre-existing* cached response being returned. Clients can set `max-age=0` to also clear existing cache responses, as this forces the cache to revalidate with the server (no other directives have an effect when used with `no-store`).

换句话说你得

> The no-store directive will prevent a new resource being cached, but it will not prevent the cache from responding with a non-stale resource that was cached as the result of an earlier request. Setting max-age=0 as well forces the cache to revalidate (clears the cache).
>
> `Cache-Control: no-store, max-age=0`

4.5.3 强制缓存 & 协商缓存

强制缓存指的是浏览器在缓存未失效的情况下, 直接避免向后端发起请求转而直接使用本地的缓存, 那如何确认浏览器缓存是否失效呢? 相关响应头有Pragma, Expire 以及替代者 Cache-Control, 其中前两个由HTTP1.0提出, 而替代者是HTTP1.1 的新规范, 为缓存提供了更大的操作空间:

- Pragma 只有一个可选值 no-cache, 与Cache-Control中的no-cache同意, 当然这里并不是字面意思不缓存, 而是虽然会缓存内容, 但是使用缓存之前, 必须要和最终的服务器再次确认. 不过严格来讲, HTTP协议中并没有明确规范 Pragma 的行为, 所以它的行为可能会在不同实现上有所差别, 所以在支持 Cache-Control 场景下来时尽量避免使用这个字段.

- 刚才提到 no-cache 作为响应头的时候, 意思是每次获取资源都需要和最终的服务器再次确认. 但其实 `Cache-Control: no-cache` 也可以作为请求头, 其行为和作为响应头还是有一定差别的, 作为请求头发送的时候, 表示服务端必须返回一份全新的数据而不能使用缓存, 具体可以看 What's the difference between Cache-Control: max-age=0 and no-cache?
- Expires的值为服务端返回的到期时间，例如 `Expires: Wed, 21 Oct 2015 07:28:00 GMT`, 即下一次请求时，请求时间如果早于上面这个到期时间，那么直接使用缓存数据. 当然也返回0, 表示资源已经过期了, 是否利用缓存需要前往服务端进行验证.
- Expires 的问题在于, 当客户端和服务器端时间存在误差时, 容易导致缓存策略提前或延后失效. 基于这个背景, HTTP1.1提出的新的HTTP Header "Cache-Control", 首先它的优先级高于 Expire, 它里面有关过期的字段有这么几个:
  - max-age=<seconds>: 指令的值是资源的最大过期时间, 这样就避免了Expire 使用绝对过期时间导致的问题. （request & response）
  - s-maxage=<seconds> : 这个是专为 Shared Cache 准备, 在Private Cache场景下 (浏览器侧)会被忽略, 含义同上. (response only)
  - max-stale[=<seconds>]: 服务端响应的内容如果过期(不超过x秒), 那么仍然使用缓存. (request only)
  - min-fresh=<seconds>: 服务端响应的内容至少在x秒内需要是有效的. (request only)
  - stale-while-revalidate=<seconds>: 之前比较火的请求库swr命名就是出自这里, 表示客户端在响应内容过期(不超过x秒)的时间里, 客户端会可以先接受过期的缓存, 但是在后台仍然向服务端请求一个最新的. (experimental)
  - stale-if-error=<seconds>： 表示当请求内容的请求失败时, 如果最初请求(上一次成功) 的请求过期(不超过x秒), 那么会直接使用缓存内容进行兜底. (experimental)

到这可能会困惑, Cache Control 指令很多, 而且作用上有一些能相互补充, 实际中怎么选择呢？其实不需要选择, 完全可以一次性可以指定多个不矛盾的指令, 用逗号隔开, 比如说, `Cache-Control: public, max-age=604800` 是完全合法的.

4.5.4 协商缓存

协商缓存指的是判断发现浏览器缓存已经失效的情况下, 向服务端发起请求, 验证缓存的有效性. 相关的请求头也有几套 `Last-Modified / If-Modified-Since` 以及 `Etag / If-None-Match`.

- 前者的使用方法是, 客户端在请求头中带上 If-Modified-Since 字段, 其值是服务端上次请求返回的 Last-Modified, 真实例子大概长这样 `Last-Modified: Wed, 21 Oct 2015 07:28:00 GMT`

- 这种方式精度偏低, 所以协议也固定了一种优先级更高的协商方式即 `Etag / If-None-Match`, 交互方式与前者类似, 唯一的区别是 Etag 是基于响应内容生成的, 会有更高的精度.

服务端收到请求后应该根据实际的情况判断当前客户端所持有的资源是否仍然有效, 若有效则返回 304, 否则将最新的响应内容返回.

4.6 主流程之外

4.6.1 缓存的标识符

这里提一个在很多资料中被忽略的细节, 类似于数据库中的记录, 缓存需要有一个唯一标识符号, 换句话说就是, 如何判断请求a 和 请求b 本质上是同一个请求? 其实这里有多条标准:

1. 首要的当然是 URI 和 请求方法, 这个是最核心的, 如果这两个不相等, 那肯定不能缓存.
2. 然后就要提到另一个Http Header -- Vary， 意思是除了URI和method以外, 也需要参考 vary 中声明的这些字段, 只有他们都一致, 才可以认为这两个请求可共用缓存.
    - Vary: *
    - Vary: <header-name>, <header-name>, ...

前者作用和 `Cache-Control: no-store` 一致, 后者表示列举的这些字段也作为判断两个请求是否复用缓存的条件.

比较常见的用法如 `Vary: User-Agent`， 这样可以防止把PC端版本的页面错误地返回给移动端用户.

4.6.2 浏览器刷新

打开控制台后, 共有三种刷新方式, 这里简单介绍下三种方式与我们上面提到的缓存策略之间的关系, 从现象来看

- "正常重新加载"时浏览器只会对当前地址栏中地址对应的请求使用 `max-age: 0`, 对于直接依赖的文件是仍然按照正常的请求流程.
- "硬性重新加载"时浏览器不但会对地址栏中地址所对应的请求发送做特殊处理, 也会对该请求产生的直接依赖文件进行特殊处理, 且处理方式是 `no-cache` 而不是 `max-age: 0`, 关于他们的区别可以看看 https://stackoverflow.com/a/1383359/5817139, 简单来说就是, 前者将不依赖协商缓存的校验机制, 服务端必须最终返回一个份新的非缓存的数据, 而后者则取访问链路上最近的(经过协商后)合理的节点的缓存.
- 清除缓存并硬性重新加载: 这个就比较简单了, 就是在硬性重新加载之前先清除浏览器的缓存, 这前者的唯一区别是, 现在连间接依赖(譬如用户浏览的是index.html, 它引入了一个a.js, 那么这个a.js算

直接依赖, 如果a.js依赖另一个文件 b.js, 此时b.js算间接依赖)也不再走缓存流程.

4.7 代码实现

相关的代码可以在文章首部给出的代码仓库中的 src/example2 中找到

上面聊了这么多, 其实这块整体还是偏概念的. 下面尝试给出一个简单的实例进行验证来加深大家的理解.

1. 首先我们来写一个简单的html来负责给服务端发请求:

HTML

```HTML
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>X</title>
</head>
<body>
    <button id="b1">getTime (no-store)</button>
    <div style="border: 1px solid red;margin: 12px 0 24px 0;">
      <div id="d1"></div>
    </div>

    <button id="b2">getTime (no-cache)</button>
    <div style="border: 1px solid red;margin: 12px 0 24px 0;">
      <div id="d2"></div>
    </div>

    <button id="b3">getTime (max-age=5; Etag=invalid)</button>
    <div style="border: 1px solid red;margin: 12px 0 24px 0;">
      <div id="d3"></div>
    </div>

    <button id="b4">getTime (max-age=5; Etag=1234)</button>
    <div style="border: 1px solid red;margin: 12px 0 24px 0;">
      <div id="d4"></div>
    </div>
```

```
    <button id="b5">getTime (max-age=5; Etag=1234; Vary)</button>
    <div style="border: 1px solid red;margin: 12px 0 24px 0;">
      <div id="d5"></div>
    </div>
</body>

<script>
  const bind = (btnId, divId, headers) => {
    const btn$ = document.getElementById(btnId)
    const div$ = document.getElementById(divId)
    btn$.onclick = async () => {
      const res = await fetch(`/api/${headers['Cache-Control']}/${btnId}`, {
        headers: {
          ...headers,
          // 思考下原样传送 no cache 会发生什么
          // 和传空有什么区别
          'Cache-Control': headers['Cache-Control'] === 'no-cache'
            ? ''
            : headers['Cache-Control'],
          'x-date': Date.now(),
        },
      })
      div$.innerHTML = await res.text()
    }
  }
  bind('b1', 'd1', {
    'Cache-Control': 'no-store',
  })

  bind('b2', 'd2', {
    'Cache-Control': 'no-cache',
  })

  bind('b3', 'd3', {
    'Cache-Control': 'max-age=5',
  })
  bind('b4', 'd4', {
    'Cache-Control': 'max-age=5'
  })
  bind('b5', 'd5', {
    'Cache-Control': 'max-age=5',
  })
</script>
```

```
72  </html>
```

这里思路很简单, 就是分别往服务端发送各种带不同缓存头的 header, 方便服务端对应返回不同的缓存头 (注意, 真实情况下服务端对于不同缓存头的处理会更加复杂, 这里我们主要是验证浏览器对于缓存头的处理).

2. 接下来写个简单的 server, 为了防止三方 server 对于缓存会有额外的处理, 这里我们选择直接用 Node 的原生模块来起一个 server

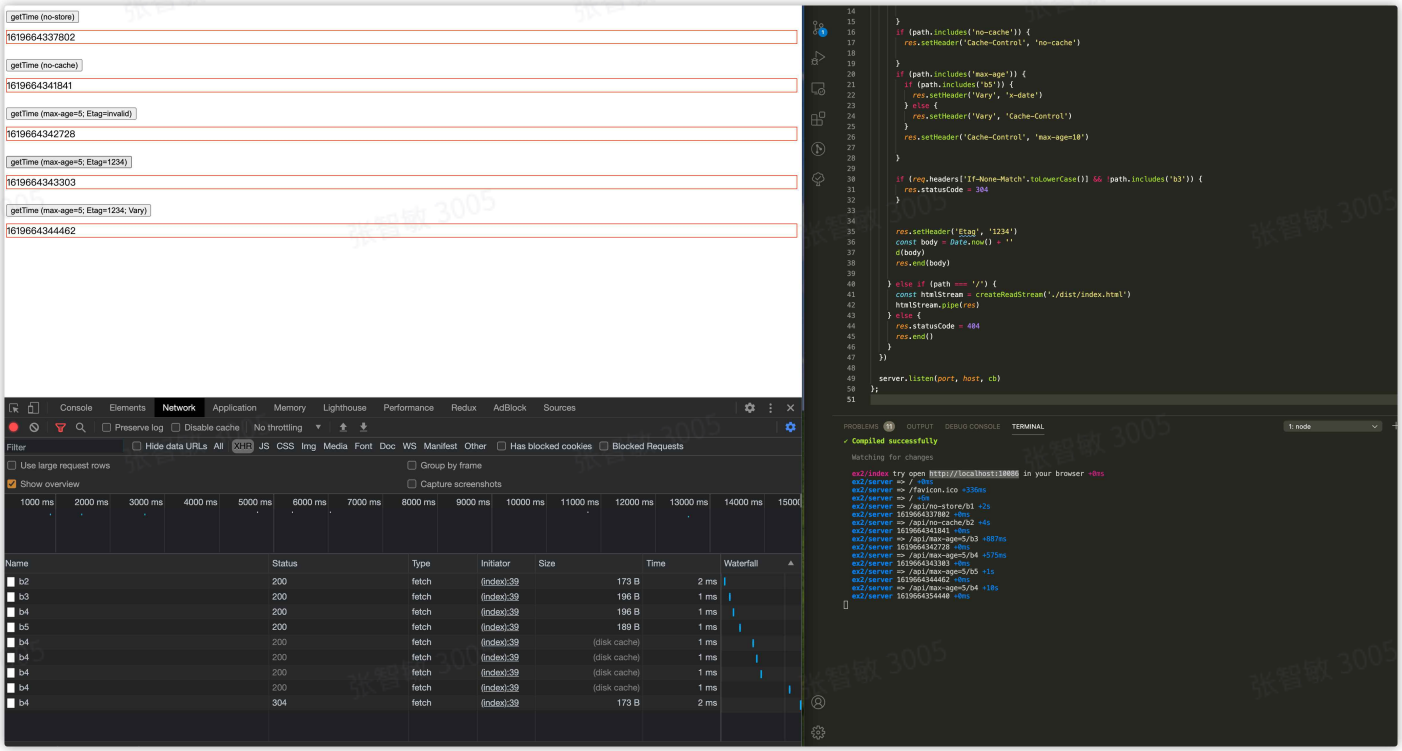
TypeScript

```typescript
import debug from 'debug';
import { createReadStream } from "fs";
import { createServer } from "http";

const d = debug('ex2/server');

export const startServer = (port: number, host: string, cb: () => void) => {
  const server = createServer((req, res) => {
    d(`=> ${req.url}`)
    const path = req.url || '';
    if (path.startsWith('/api')) {
      if (path.includes('no-store')) {
        res.setHeader('Cache-Control', 'no-store')

      }
      if (path.includes('no-cache')) {
        res.setHeader('Cache-Control', 'no-cache')

      }
      if (path.includes('max-age')) {
        if (path.includes('b5')) {
          res.setHeader('Vary', 'x-date')
        } else {
          res.setHeader('Vary', 'Cache-Control')
        }
        res.setHeader('Cache-Control', 'max-age=10')

      }

      if (req.headers['If-None-Match'.toLowerCase()] && !path.includes('b3')) {
        res.statusCode = 304
      }
```

```
33
34
35          res.setHeader('Etag', '1234')
36          const body = Date.now() + ''
37          d(body)
38          res.end(body)
39
40        } else if (path === '/') {
41          const htmlStream = createReadStream('./dist/index.html')
42          htmlStream.pipe(res)
43        } else {
44          res.statusCode = 404
45          res.end()
46        }
47      })
48
49      server.listen(port, host, cb)
50  };
```

思路也很简单, 根据客户端的请求作出对应的响应.

3. 完事之后，拉起服务器, 打开 http://localhost:10086, 一切顺利应该能看到如图, 尝试点击各个按钮观察控制台以及页面行为, 验证自己的想法

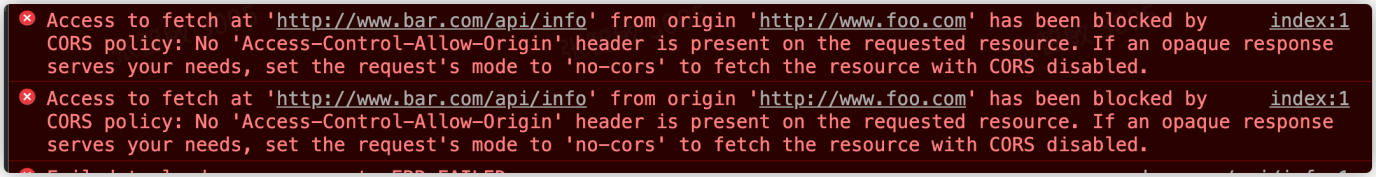


4.8 小练习

1. 修改上述代码, 探究一下 `Cache-Control: no-store` 作为请求头和响应头的区别.
2. 沿着上面的思路, 尝试验证代理服务器对于缓存的影响

5. 跨域相关

5.1 长啥样



5.2 学习资料

Same-origin policy - Web security | MDN

5.3 同源策略

为了定义两个url之间的亲疏关系 (即是否来自相同的服务器, 或者相互信任的组织), w3c 提出了同源的概念, 同源必须满足三个条件:

1. protocol 相同
2. host 相同
3. port 相同

譬如在页面 https://www.bytedance.com 上, 以下请求的情况是:

1. http://www.bytedance.com 协议不同
2. https://www.bytedance.net host不同
3. https://www.bytedance.com:10086 port不同

4. https://www.bytedance.com/api 同源

再来看看同源策略的定义

> The **same-origin policy** is a critical security mechanism that restricts how a document or script loaded from one origin can interact with a resource from another origin. It helps isolate potentially malicious documents, reducing possible attack vectors.

更通俗的, 现代网页应用同时与多个 url 发生交互是很常见的事情, 出于安全考虑, 浏览器会根据 url 之间的亲疏关系来对它们的交互进行不同程度的限制. 很自然地, 借用我们上面的定义, 同源的 url 更亲近, 倾向于互相信任, 它们之间的交互会获得更多的自由, 反之, 不同源在交互过程中会受到限制, 更具体的, MDN定义了三种类型的跨域交互:

1. 首先是跨域写, 比如类似重定向, 表单提交等, MDN表示这通常是被允许的. 这里听起来可能会有些费解, 我的理解是, 这里写是发生在跨域域名的服务器上, 因为抛开服务器相应的鉴权逻辑或者防御机制不谈, 这个请求确实被发送到了服务器, 认为服务器被 “写” 了是说得通的.
2. 然后是跨域的嵌入资源, 通常也是被允许的, 比如所谓的 script, link, img, video, iframe 等等, 这个很好理解, 一些早期的跨域方案实际上就是依赖了嵌入资源不受跨域限制这一点.
3. 最后一种是跨域读, MDN表示这通常是被禁止的, 这里读的主体是浏览器, 如果没有配置跨域头, 即使服务端返回了相应的信息, 浏览器也会将其截断, 这个就是我们平时看到的 `Access xxx url has been blocked by CORS policy`,

5.4 跨域手段

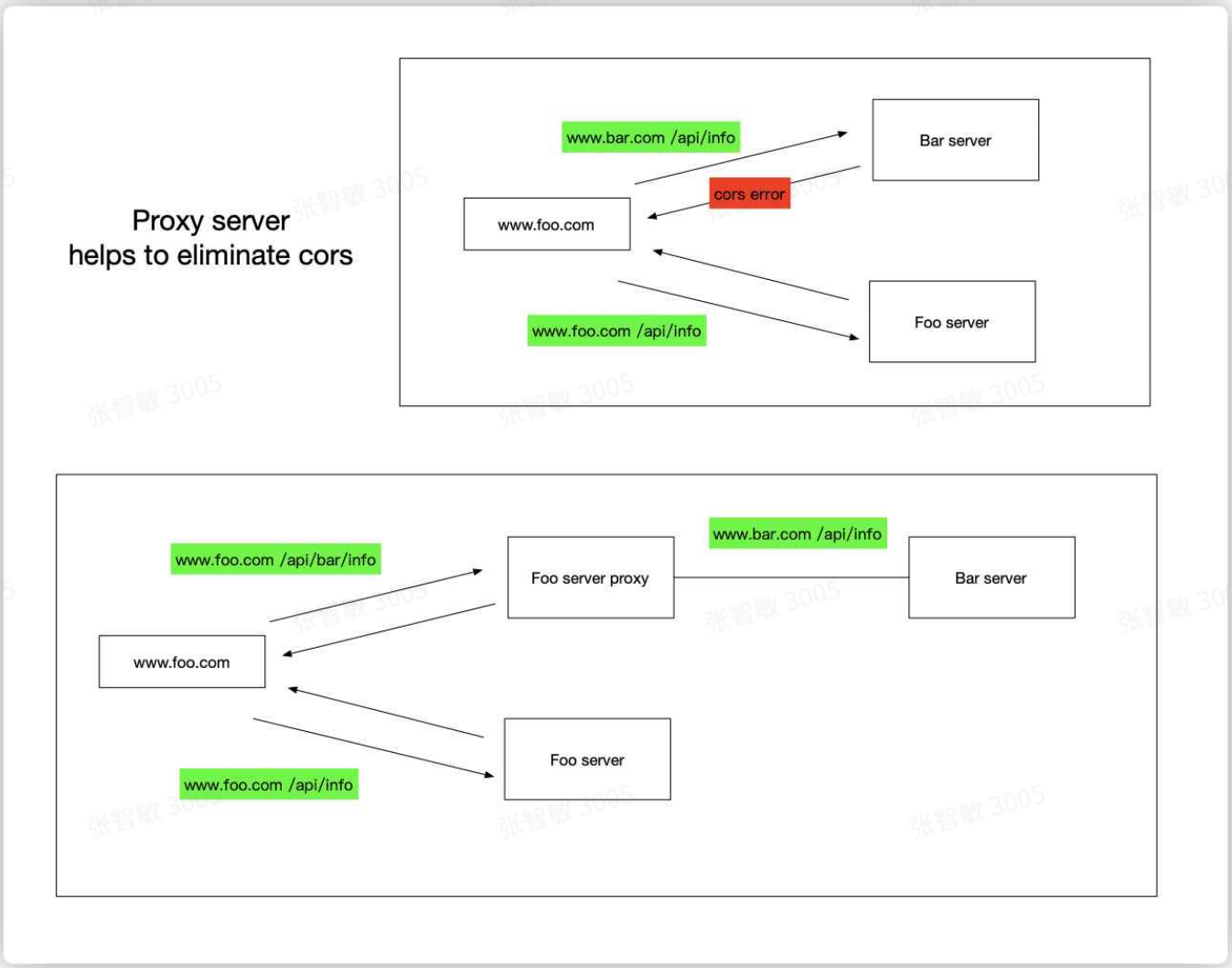
现实中其实有很多不同源的 url 实际是来自同一厂商, 它们之间也有丰富的资源交互的需求. 所以可以看到的是, 同源策略的存在在提升了web安全的同时, 也导致了诸多不便, 我个人理解跨域实际上就是在解决 **“如何安全地在不同源但是相互信任的 url 之间进行资源交互”** 的问题.

随手Google一下, 可以发现大量跨域解决方案, 比较常见的有譬如 proxy, jsonp, iframe 以及 cors. 严格来讲其实只有 CORS 和 proxy 可以算是正统的解决思路, 其他的解法都比较 tricky, 不过为了介绍的完整性, 依然会简单介绍除CORS以外的其他的一些跨域思路.

5.4.1 proxy

流程

这个准确来说不算从根本上解决了问题, 做法很简单, 既然跨域发生在两个url不同源的情况下, 那就加一层反向代理把其中一个url的链接映射到到另一个url上, 如图所示:



局限性

相当于变相地把服务放在了同一个域名下, 大部分情况是, 如果两个服务能放在同一个域名下, 最初设计的时候就放了.

5.4.2 jsonp (json with padding)

流程

jsonp是早期使用比较广泛的一种跨域手段, 利用的就是我们上面提到的嵌入资源不存在跨域限制的这一点. 假设在 www.foo.com 依赖了bar服务器的一些信息, 那么在页面上我们需要:

1. 定义一个响应的处理函数

```typescript
const handleResponse = (infoJson) => {
  // logic with info
}
```

2. 动态创建一个script标签, 其 src 为 `https://www.bar.com/api/info?callback=handleResponse`

按照约定, bar 服务器需要发返回js文件, 其内容为一个被我们声明的callback函数包裹的json, 这就是所谓的 JSON (Json with Padding)

```typescript
handleResponse({
    // info content
    name: 'xxx',
    age: 24,
})

```

此时一旦js被返回, 其内容被浏览器解析, 则自动调用 handleResponse, 这就完成了一次跨域的数据交换.

局限性

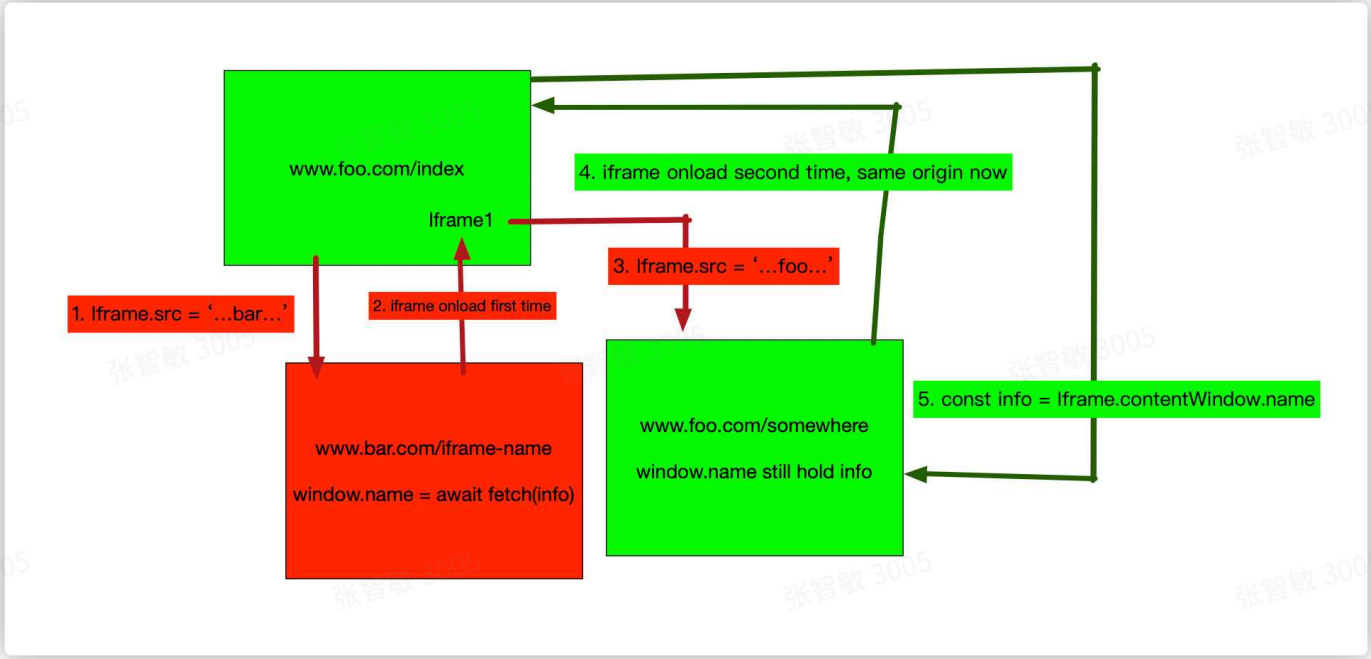
这个方法其实存在不少安全隐患,

- 比较典型的譬如 CSRF, 攻击者写一个有问题的网页a, 在其中使用 jsonp 访问了存在安全隐患的服务器b的某个隐私数据接口. 此时如果用户进入这个问题网页a, 会在不知情的情况下访问服务器b, 假设用户曾经在b登陆过, 那么他的数据自然会被带给问题网页a, 此时攻击者顺利拿到了用户的隐私数据. 可以考虑对于接口b做一些处理, 例如加上 csrf-token.

5.4.3 iframe

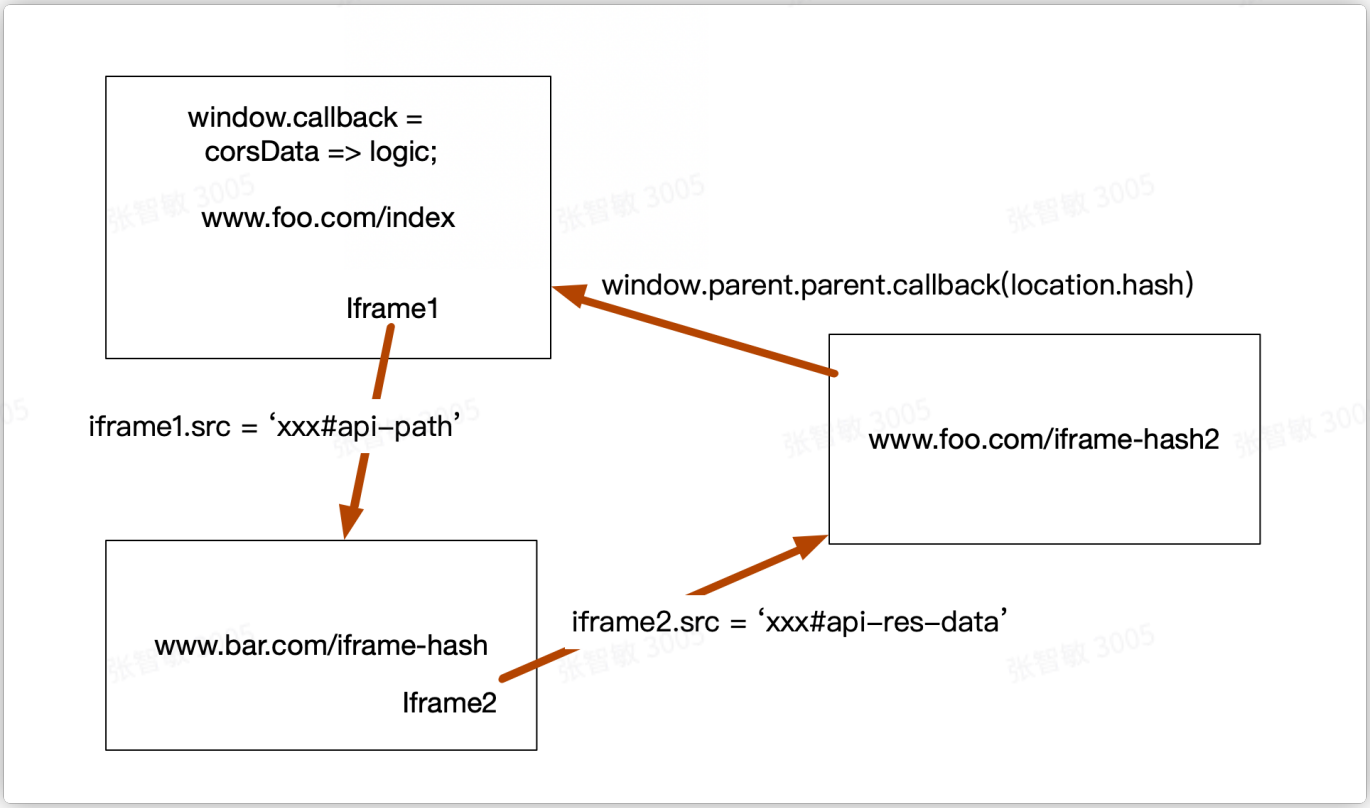
window.name 跨域

可以利用`window.name`只要页面不关闭, 即使发生跳转仍然会保留的这个性质, 我们可以在主窗口中打开一个需要跨域访问的地址 (这个过程本质完成了将父窗口对子窗口的通讯), 这个地址返回的html的script标签中把主窗口需要的信息放在`window.name`中, 主窗口一旦监听到iframe的onload事件, 就将iframe的src设置为与主窗口同源的域名, 此时`window.name`可访问, 进而完成了子窗口信息向父窗口的回传.



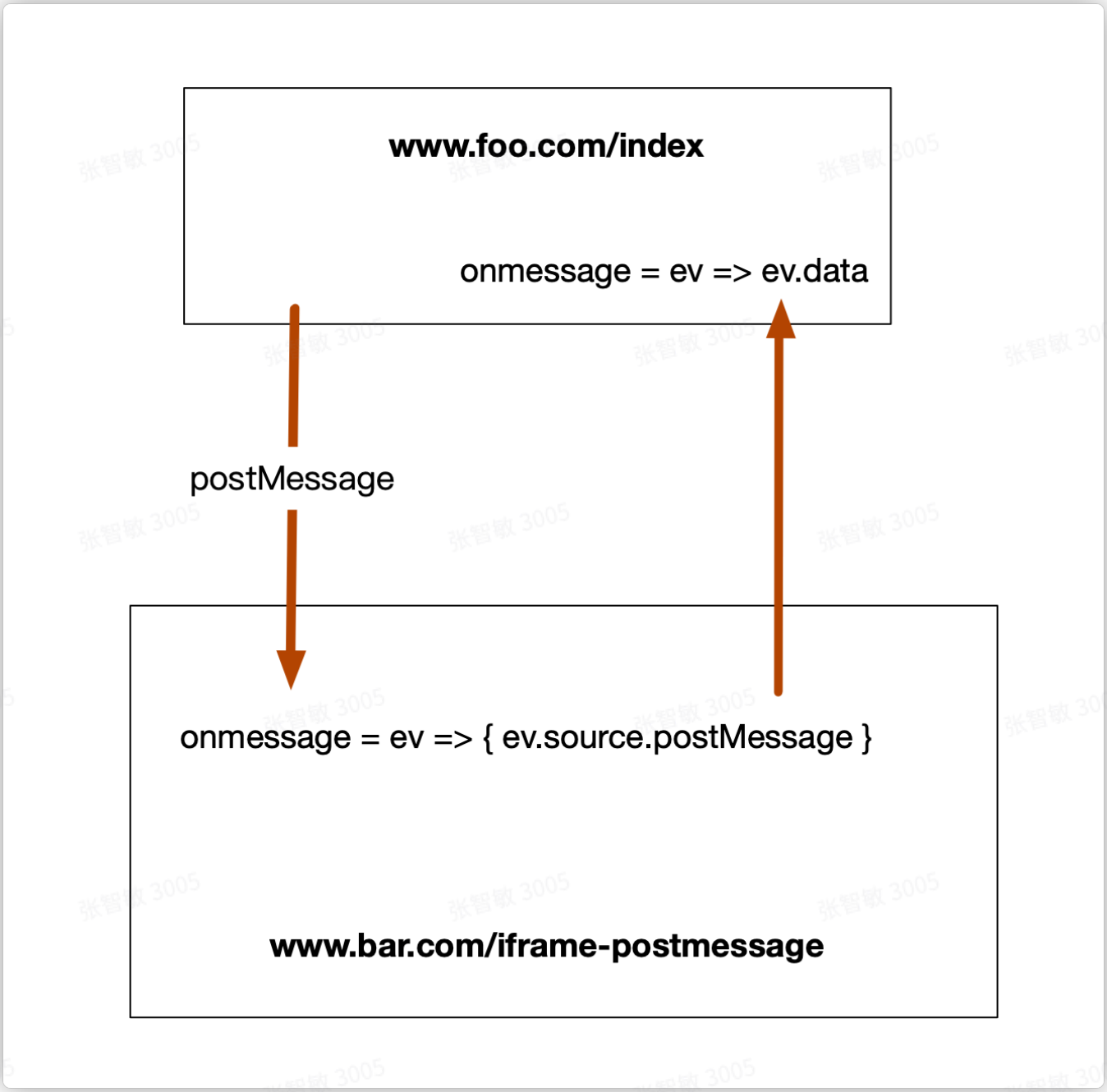
window.hash 跨域

基于哈希改变并不会触发请求, 父窗口可以通过改变跨域子iframe的hash来完成父对子的通讯, 而子窗口为了实现将信息回传给父窗口, 可以通过在子窗口中再次新建一个与父窗口同域的iframe, 同样用哈希的方式将信息汇入, 由于孙子窗口与爷爷窗口同域, 所以可以通过`window.parent.parent`获取到爷爷窗口, 基于这种方式将信息回传.



postMessage 跨域

可以看出来上面两种方式本质上都是试图在跨域场景下进行iframe通讯, 实际上是比较tricky, 现在最常见的是基于 postMessage 实现通讯, 这也是h5实现的一部分, 可以认为是比较官方的做法.



本质就是父窗口利用 `iframe.contentWindow.postMessage` 发消息, `window.onMessage` 收消息. 而子窗口利用 `window.onMessage` 收到消息后, 再利用 `event.source.postMessage` 回复消息.

5.4.4 CORS 头

Ok, 看了各种花里胡哨的跨域手段, 现在回归到目前最推崇的也是标准的方式, 配置 CORS 跨域头跨域. 前面也说了, 跨域本质上是浏览器的安全策略, 所以配置 CORS 跨域头本质上就是通知浏览器 "第三方服务器认可这个请求, 确认了它的合法性".

当请求发生跨域请求时, 譬如在 http://www.foo.com 的页面上发送了一个发往 http://www.bar.com 的请求, 接下来分以下几步:

1. 判断是否是简单请求

如果一个请求满足以下条件:

1. 请求方法: GET, HEAD, POST
2. 请求头中只包括: Accept, Accept-Language, Content-Type (字段只能是 text/plain, multipart/form-data, application/x-www-form-urlencoded), DPR, Downlink, Save-Data, Viewport-Width, Width. (有人说怎么没看到Cookie之类的, 这些被定义为 forbidden-header-name, 显然也是被允许的).
3. `XMLHttpRequest.upload` 上没有挂事件监听, 这个主要是用来跟踪上传进度的, 换句话说就是不需要追踪进度的请求
4. 请求中没有使用 `ReadableStream`， 一般只有上传文件时会用到这个.

2. 实际发送请求

如果是简单请求, 则浏览器直接发送请求, 否则需要发送 OPTIONS 方法的 Preflighted 请求, 来向服务器核验是否接下来这个请求是可接受的. 发送 Preflighted 请求的时候需要关注这几个字段(简单请求不需要), 它用来表明接下来的正式请求会包含哪些头字段&哪种方法:

> Access-Control-Request-Method: GET
>
> Access-Control-Request-Headers: x-lang,x-session-key

3. 浏览器收到请求

无论是否是简单请求, 浏览器将重点关注以下几个响应头字段:

> Access-Control-Allow-Headers: x-lang,x-session-key,content-type
>
> Access-Control-Allow-Methods: GET, HEAD, OPTIONS, POST, PUT, PATCH, DELETE
>
> Access-Control-Allow-Origin: https://gdev-sandbox-sg.bytedance.net
>
> Access-Control-Max-Age： 600
>
> Access-Control-Expose-Headers

> Access-Control-Allow-Credentials: true

含义

1. 前三个分别表示浏览器可信任的请求头字段, 请求方法以及请求域.
2. Access-Control-Max-Age 表示有关这个请求预检查请求中的 Access-Control-Allow-Headers, Access-Control-Allow-Methods 的最大缓存时间. (Chrome 默认不能超过10分钟)
3. Access-Control-Expose-Headers 了解这个得知道一个背景, 那就是对于跨域请求默认浏览器中的JavaScript代码只能拿到部分响应头字段, 这个字段是用于方便服务器声明其他可以被JavaScript代码读到的请求头
4. Access-Control-Allow-Credentials 这个字段需要配合 fetch 中的 `credentials: 'include'` 使用, 对于跨域的 get 请求而言, 假设请求时带上了 cookie, 如果响应中没有给到 Access-Control-Allow-Credentials: true (值得注意的是, 这个头字段值只能是 true 没有 false, 否则就干脆不要写这个字段), 那么此时即使资源被成功返回并且响应头带上了 allow-orgin 之类的, 响应结果依然会被浏览器舍弃. 对于 POST 请求, 如果 fetch 的时候 `credentials: 'include'` Preflight 没有返回 Access-Control-Allow-Credentials: true, 那基本宣告了CORS失败.

5.5 安全

下面介绍的两个字段头其实和我们通常理解的跨域关系不大, 但概念上也与跨域有一些关联.

5.5.1 Referrer-Policy

介绍这个字段的作用之前我们需要先了解一下 referer 字段.

众所周知, referer 字段的作用是向服务器标明当前这个请求是从哪个页面发起的, 默认情况下, referer 字段给出的信息非常详细, 例如在 https://github.com/google/zx?tes=1 发出的请求头中, referer长这样:

> referer: https://github.com/google/zx?tes=1

这个信息有用的时候确实很有用, 不过从安全的角度来讲, 有点过去完整了, 一定程度上暴露了用户的隐私. 这时候就有了我们所谓的 Referrer-Policy, 它的作用就是可以对 referer 字段的粒度进行控制, 它的值以下几种， 都很好理解就不多赘述了

> Referrer-Policy: no-referrer
>
> Referrer-Policy: no-referrer-when-downgrade

> Referrer-Policy: origin
>
> Referrer-Policy: origin-when-cross-origin
>
> Referrer-Policy: same-origin
>
> Referrer-Policy: strict-origin
>
> Referrer-Policy: strict-origin-when-cross-origin
>
> Referrer-Policy: unsafe-url

当然, 它也可以被设置在html中或是一些 a标签里, 作用都是类似的

> <meta name="referrer" content="origin">
>
> <a href="http://example.com" rel="noreferrer">

5.5.2 Content Security Policy (CSP)

之前讲了 CORS, 本质是基于浏览器的同源策略执行的, CSP则更近一步, 允许开发者使用响应头来细粒度地对页面需要访问的资源进行控制, 即白名单, 当然和 Refrrer-Policy 一样, 同样也支持在 Html 中定义 meta 头来声明 CSP. CSP的出现很大意义上是为了解决网络安全问题, 最常见的例如 XSS. 大致格式:

```HTML
Content-Security-Policy: <policy-directive>; <policy-directive>

<policy-directive>: <directive> <value>
```

具体的头结构我举个例子:

> 譬如 Content-Security-Policy: default-src 'self'; connect-src http://www.foo.com; style-src 'unsafe-inline'; script-src 'unsafe-inline';

1. `default-src` 表示兜底, `self` 表示只允许当前域.
2. `connect-src` 表示脚本可访问的链接, 后面接的是具体的域名, 意思是该页面的脚本只被允许访问 http://www.foo.com .

3. `style-src` 表示被允许的样式表来源, `unsafe-inline` 表示允许使用link标签作为来定义页面的样式.

这一块概念较多, 这里仅仅做简单介绍, 更具体地可以对着 Content-Security-Policy - HTTP | MDN 配合实例代码中的 example5 进行探索和学习.

The End

感觉前端HTTP相关的内容就是这些, 如果有错误或者不完整的地方, 欢迎各位补充~