

课程目标

01

工作日常

02

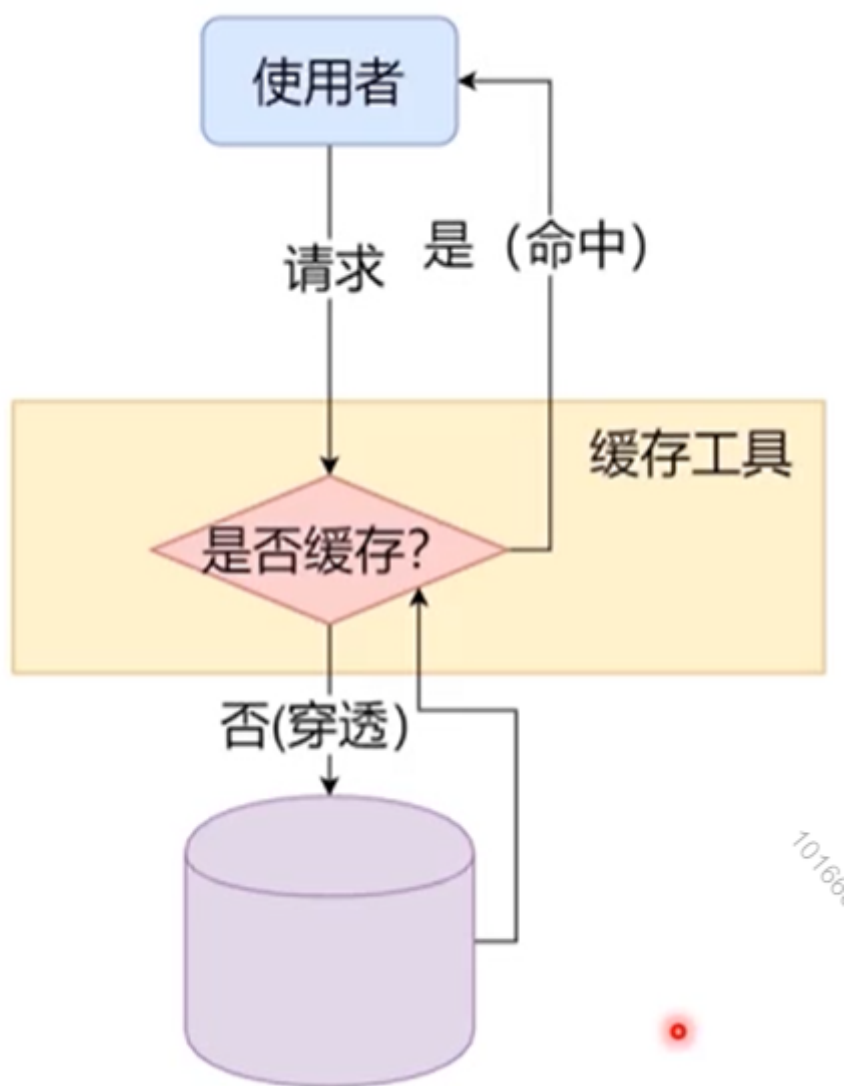
面试重点

03

架构思想

缓存介绍

缓存：存储将被用到的 数据,让数据访问更快。缓存示意图：



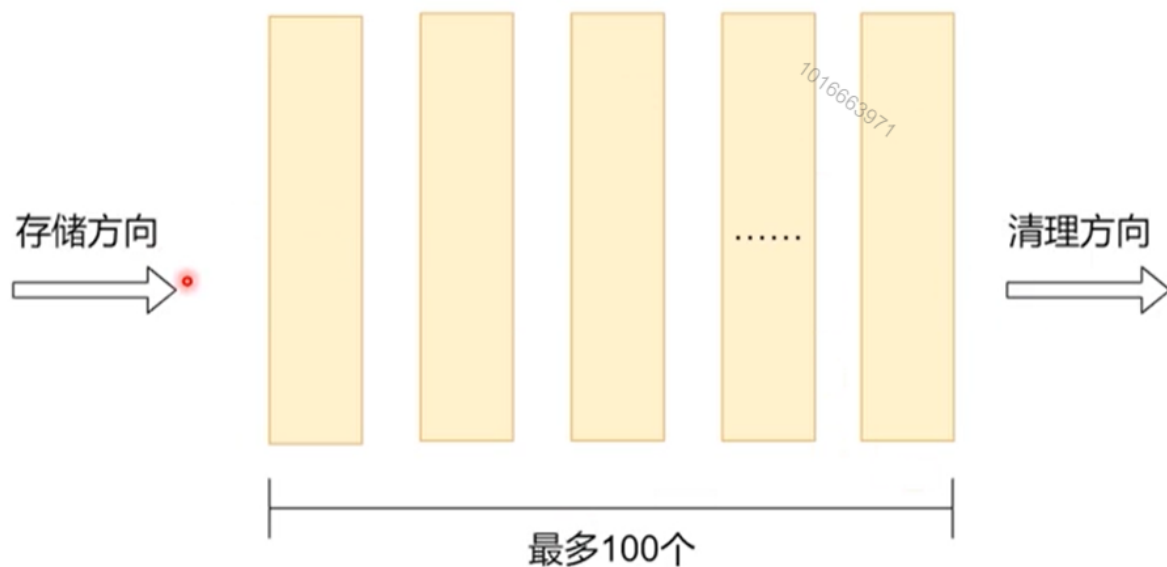
缓存这个词最早不是web端,而是在cpu和内存的设计上。最早的时候,是在很底层的时候就用到过。缓存涉及到几个概念。

- 命中(在缓存中找到请求的数据)

- 不命中(缓存中没有需要的数据)
- 命中率(命中次数/总次数)
- 缓存大小(缓存中一共可以存多少数据)
- 清空策略(如果缓存空间不够数据如何替换)

清空策略

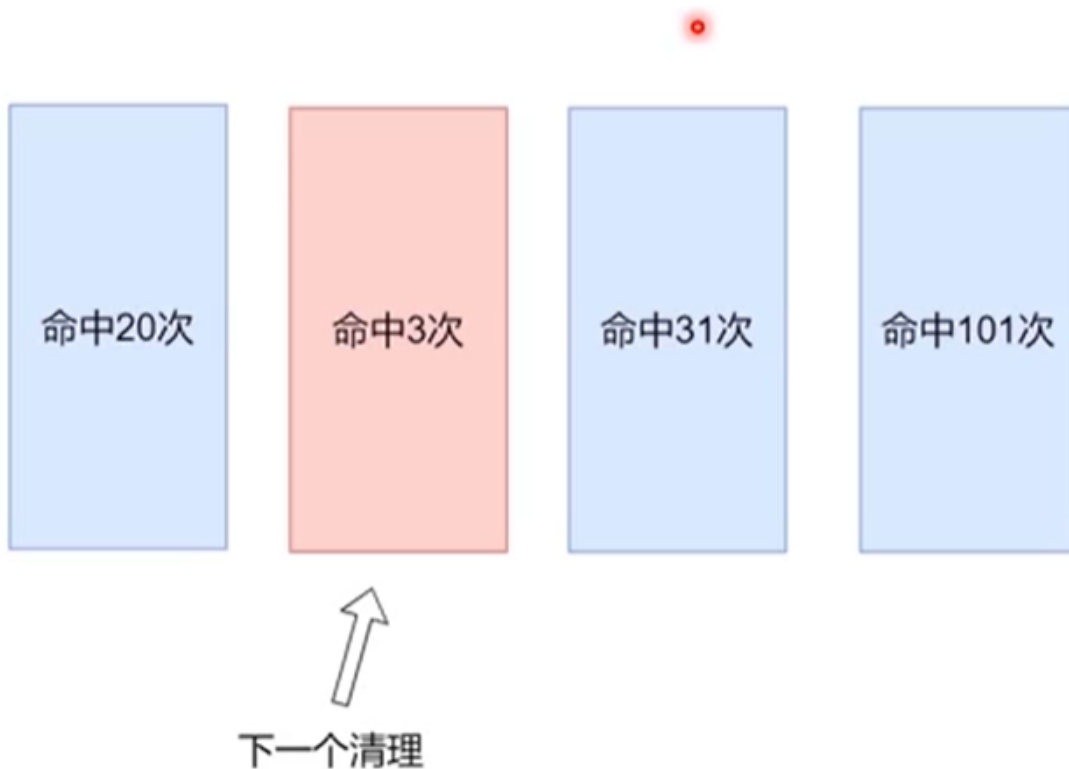
1. FIFO(先进先出)



从左到右,最多100个,当存入的小于100个的时候,可以继续存入,当大于100的时候,清理先缓存进来。

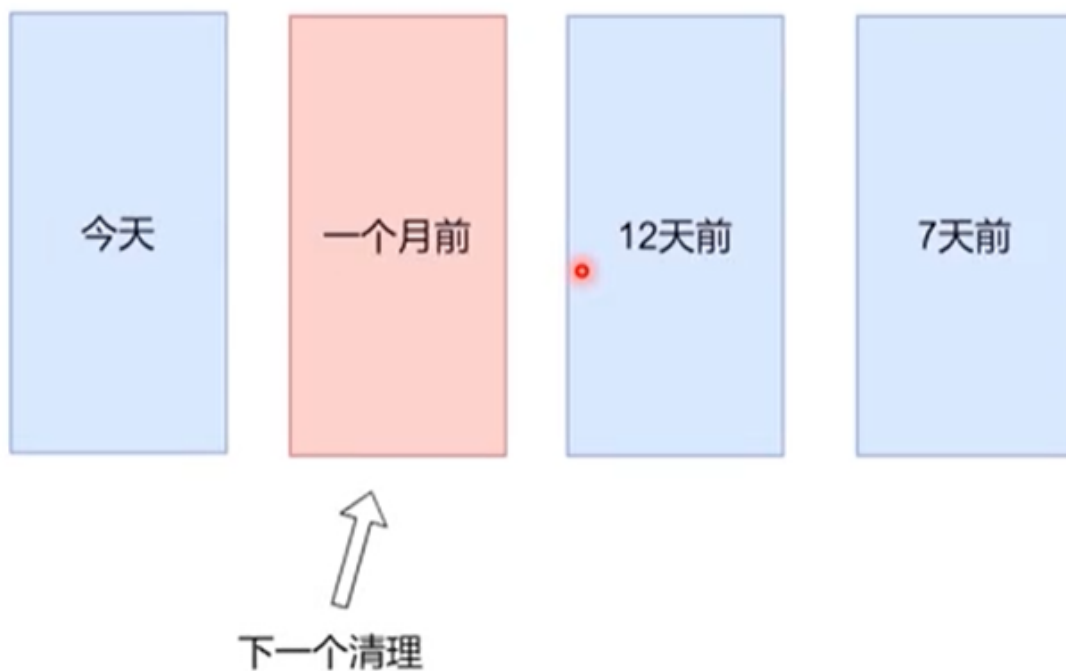
思考: 如果Javascript缓存,用Map还是Array?

2.LFU-least Frequently used



优先清空命中次数最少的。

3.LRU-least recently used



1016~
优先清空缓存时间最久的。

思考：内部实现用数组还是优先队列？

实战fifo的memory函数

斐波拉契数列使用FIFO先进先出相关代码

```
1 //斐波拉契数列
2 function fifo(n) {
3   if(n == 1 || n == 2){
4     return 1
5   }
6   return fib(n-1)+fib(n-2) //这里相加的运算也得用缓存后的 函数调用否则达不到缓存的作用
7 }
8
9 //1 1 2 3 5 这种是依赖前面的计算,规模越大 后面计算的时间就越长
10 // 1
11 // 1
12 //1+1 =2
13 //1+2=3
14 //2+3 = 5
15
16 const fib = memory(fifo,10)
17
18 //使用缓存的函数
19 function memory(fn,maxSize) { // 这里使用高阶函数
20   //[{hash:hash,value:value}]
21   const cache = []
22   return (...args)=>{
23     //这里需要看是否有缓存,有缓存直接从缓存里面直接拿,没有缓存继续调用函数
24     const hash = args.join(',')
25     // if(cache[hash])
26     //find返回的直接是数据中命中条件的值
27     const item = cache.find(item=>item.hash === hash)
28     if(item){
29       return item.value
30     }
31     const result = fn(...args)
32     cache.push({hash,value:result})
33
34     //缓存数量是否超出,超出则把缓存的最先进去的移除掉 就是移除头部
35     if(cache.length>maxSize){
36       cache.shift()
37     }
```

```
38 //最终返回result
39 return result
40 }
41 }
42
43
```

实战:LRU算法

lru的memory函数相关代码(这里面如果计算)

```
1 //斐波拉契数列
2 function fifo(n) {
3   if(n == 1 || n == 2){
4     return 1
5   }
6   return fib(n-1)+fib(n-2) //这里相加的运算也得用缓存后的 函数调用否则达不到缓存的作用
7 }
8
9 //1 1 2 3 5 这种是依赖前面的计算,规模越大 后面计算的时间就越长
10 // 1
11 // 1
12 //1+1 =2
13 //1+2=3
14 //2+3 = 5
15
16 const fib = memory(fifo,10)
17 console.log(fib(1000));
18
19 //使用缓存的函数
20 function memory(fn,maxSize) { // 这里使用高阶函数
21   //[{hash:hash,value:value}]
22   let cache = []
23   return (...args)=>{
24     //这里需要看是否有缓存,有缓存直接从缓存里面直接拿,没有缓存继续调用函数
25     const hash = args.join(',')
26     // if(cache[hash])
```

```

27 //find返回的直接是数据中命中条件的值(如果计算条目过多的话, 这个缓存就没有意义了)
28 const item = cache.find(item=>item.hash === hash)
29 if(item){
30 //命中缓存了,记录一个当前的时间
31 item.time = new Date().getTime()
32 return item.value
33 }
34 const result = fn(...args)
35 //没有命中缓存把当前的记录当前的time
36 cache.push({hash,value:result,time:new Date().getTime()})
37
38 //缓存数量是否超出, 超出则把缓存时间最长的清除掉,
39 if(cache.length>maxSize){
40 //这里进行一个最小值的计算
41 let min = Infinity
42 let minItem = null
43 for(let item of cache){
44 if(item.time<min){
45 min = item.time
46 minItem = item
47 }
48 }
49 //求出时间最小值就是的缓存时间最长的.过滤点
50 cache = cache.filter(x=>x!==minItem)
51 }
52
53 //最终返回result
54 return result
55 }
56 }
57
58

```

HTTP缓存

Cache-Control 定义所有缓存都要遵守的行为, 是很重要的一个缓存头, 下面定义了Cache-Control的时候相关值:

可缓存性

值	含义
public	允许所有方缓存
private	值允许浏览器缓存
no-cache	每次必须先询问服务器资源是否已经更新
no-store	不使用缓存

上图定义是要不要缓存。一个是客户端的要不要缓存。一个是中间商要不要缓存(中间还有很多代理出服务器之类的)

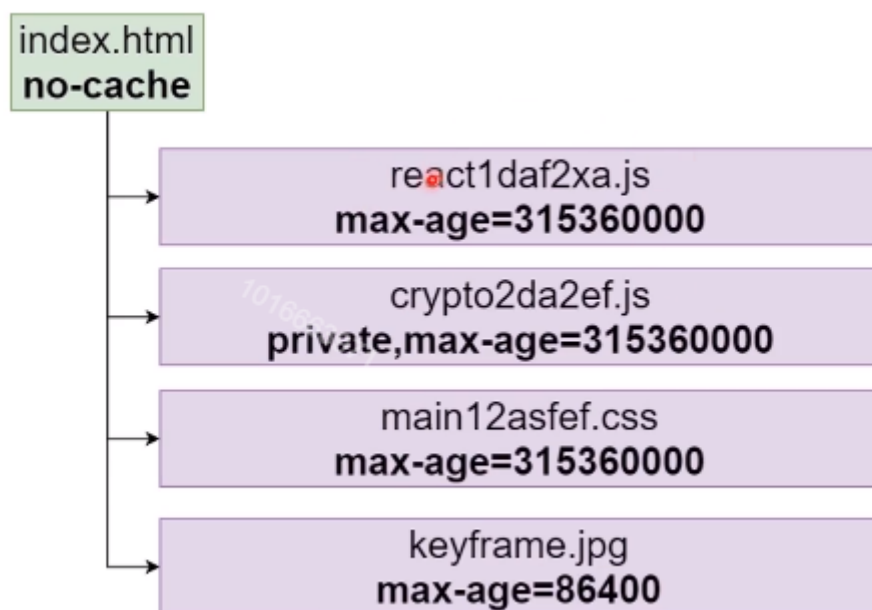
- 如果所有方都允许缓存就定义成public
- 如果所有的中间商都不允许缓存,只允许端上的缓存,就定义成private
- 每次的必须先询问服务器资源是否已经更新,每次需不需要缓存要进行协商,就定义为no-cache(no-cache不是没有缓存的意思)
- 如果希望所有都不要缓存,使用no-store

缓存期限

值	含义
max-age	秒 (存储周期)
s-maxage	秒 (共享缓存如代理等, 存储周期)

- max-age 缓存多少秒
- s-maxage 设置中间方代理缓存多少秒 只在最终的终端,可能是浏览器,或者其他终端应用。

Cache-control常见用法



像我们一个js文件 react库 jq公用的库 vue 这种情况下，我们把他的缓存期限设置很大，因为这种文件变动很小，更新周期是非常慢的，而且更新之后，也会更新后面的hash值，重新加载这些js文件。

像我们自己私有的加密库,我们通常会结合private 只允许浏览器端缓存。不允许中间商去缓存，这里有敏感的计算。造成不必要的麻烦。css设置max-age的也是比较长，变动比较小。类似图片的话 max-age 只设置了一天，图片也没带hash，因为存在更新的问题。这个可以根据自己不同的需要去指定缓存的策略。

缓存分为两个大的类目：

- 强制缓存
- 协商缓存

强制缓存：不去服务器对比(缓存生效不再发送请求)

Cache-Control: max-age=600
Expires: <最后期限>

Expires 在这个日期前都进行强制缓存，超过这个日期就不再缓存了（很少用）

下面就在服务端设置强缓存

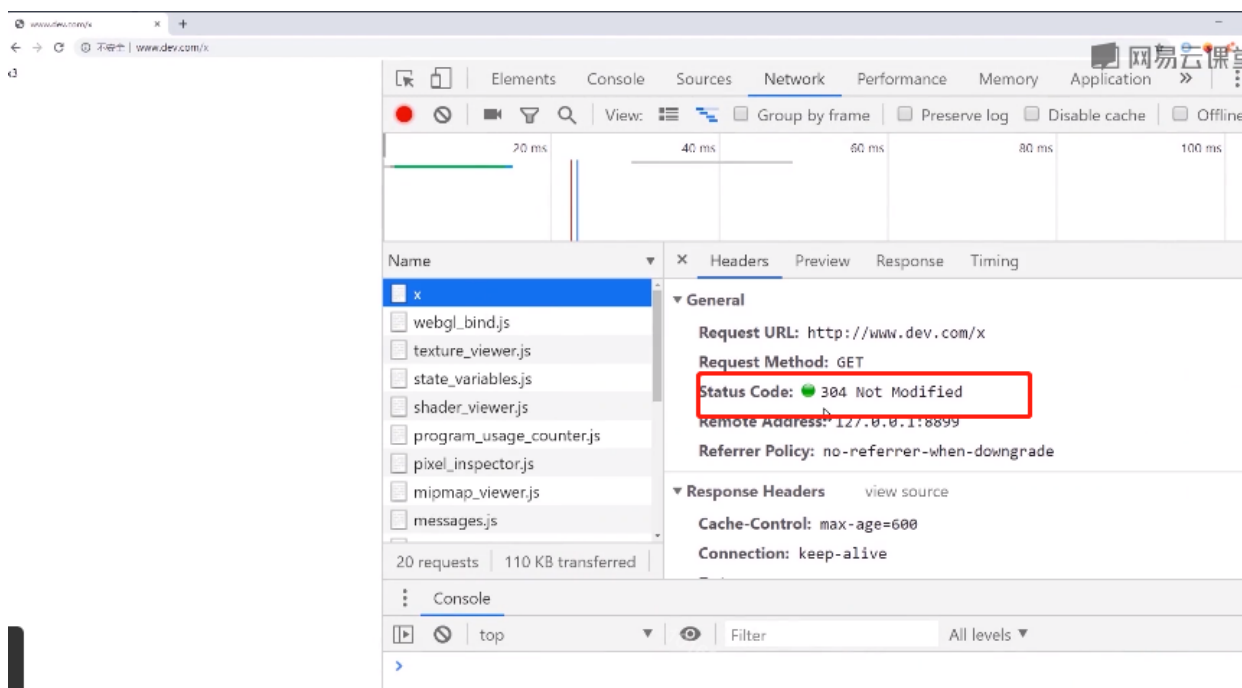

```

1  const express = require('express')
2
3  const app = express()
4
5  app.get('/x', (req, res) => {
6    res.set("Cache-Control", 'max-age=600')
7    res.send("x3")
8  })
9
10 app.listen(3000)

```

设置 600s的强制缓存，600是内不会像服务端进行请求了。

客户端请求，出现的304协商缓存，这是为什么呢？服务端已经设置了强缓存的，为什么不生效了，而且强缓存的优先级是要大于协商缓存的

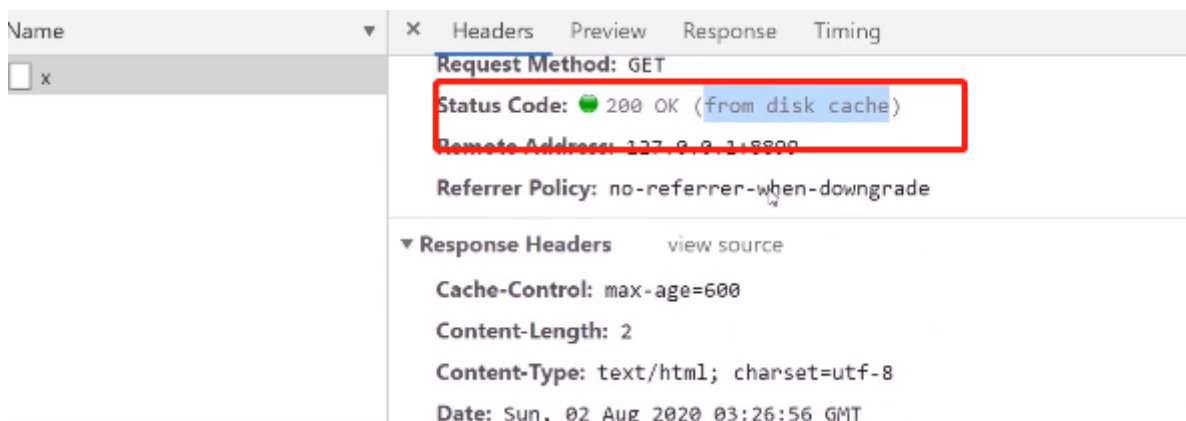


原因是 浏览器认为这个请求是http://www.dev.com/x 像是在请求某个资源，把强制缓存关了。如图所示

```
e/webp,image/apng,*/*;q=0.8,application/signed-exchan
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9,zh-CN;q=0.8,zh;q=0.7
Cache-Control: max-age=0
Host: www.dev.com
If-None-Match: W/"2-Fdo9qmiwb0ALxNEQPwVhVzzTa4o"
Proxy-Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
537.36 (KHTML, like Gecko) Chrome/75.0.3770.142 Safari
```

max-age设置成 0 就如同设置了no cache一样，每次都要去询问服务端，是否更新了资源。这样就变成了协商缓存.这个是浏览器默认的,我们可以通过fetch方式去请求，如下结果

```
> fetch('/x')
< ▶ Promise {<pending>}
> |
```



The screenshot shows the Chrome DevTools Network tab. A request named 'x' is selected. The 'Headers' tab is active, showing the following details:

- Request Method: GET
- Status Code: 200 OK (from disk cache)
- Remote Address: 127.0.0.1:5500
- Referrer Policy: no-referrer-when-downgrade

Below the request details, the 'Response Headers' are expanded, showing:

- Cache-Control: max-age=600
- Content-Length: 2
- Content-Type: text/html; charset=utf-8
- Date: Sun, 02 Aug 2020 03:26:56 GMT

这就变成了强制缓存了。可以通过服务端的日志可以看到请求是否到达服务端，强制缓存应用场景一般是一些不经常变动的静态资源。DNS就是一个强制缓存。经常变动的资源就不适合了

协商缓存：每次请求需要向服务器端请求对比,缓存生效不传回body

1. last-Modified和 if-Modified-Since

返回:

Last-Modified: <昨天>

请求:

If-Modified-Since: <昨天>

最常见的就是**last-Modified**，服务端先返回一个last-Modified，客户端最后一次修改的时间是什么时候，客户端的每次请求 **if-Modified-Since**都把服务端上次返回来的 last-Modified带上。如果没有变化则返回304 Not Modified，但是不会返回资源内容；如果有变化，就正常返回资源内容。当服务器返回304 Not Modified的响应时，response header中不会再添加Last-Modified的header，因为既然资源没有变化，那么Last-Modified也就不会改变，这是服务器返回304时的response header。

服务端设置demo:

```
cachejs > 300 app
1  const express = require('express')
2
3  const app = express()
4
5  app.set('etag', false)
6  app.get('/x', (req, res) => {
7    res.set("Last-Modified", 'Sun Aug 02 2020 11:32:38 GMT+0800')
8    res.send("x4")
9  })
10
11 app.listen(3000)
```

2.E-Tag和If-None-Match

返回:

E-Tag: 1234567

请求:

If-None-Match: 1234567

另外一个常见的就是**E-Tag**。也是用的最多的一种协商缓存，第一次请求服务器端带上一个E-Tag，客户端下次请求就会带上**If-None-Match**：服务端上次返回的E-Tag。If-None-Match能够匹配下次服务端生成的E-Tag，304命中协商缓存，否者更新新的内容。服务端设置E-tag，没有看到特殊的E-Tag设置，可以看到服务端默认设置E-tag了

```
1 const express = require('express')
2
3 const app = express()
4
5 app.get('/x', (req, res) => {
6   res.send("x5")
7 })
8
9 app.listen(3000)
```

查看客户端:





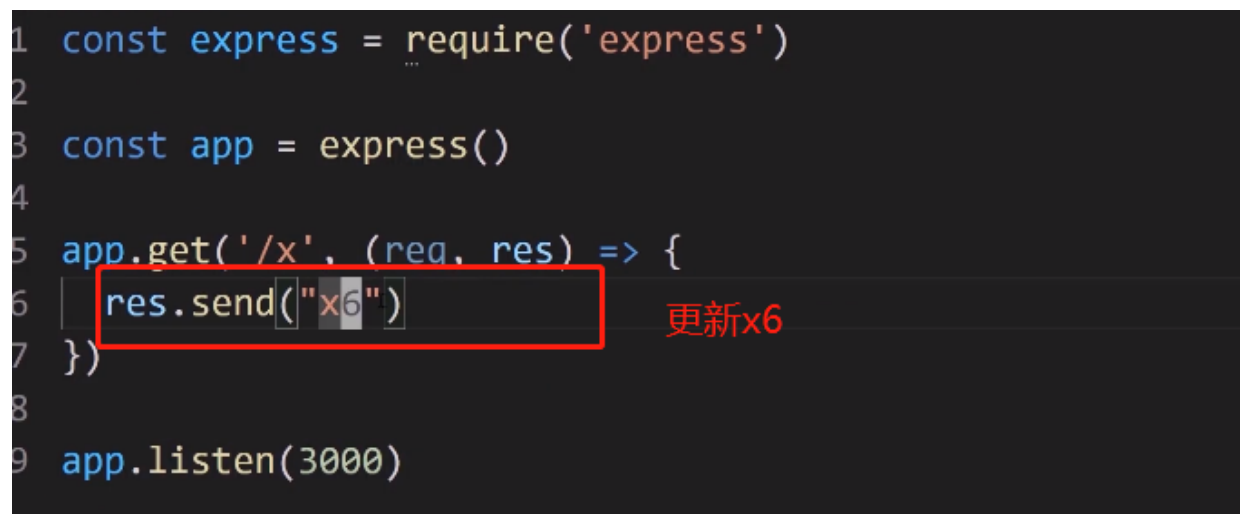
Status Code: 304 Not Modified

Remote Address: 127.0.0.1:8899

Referrer Policy: no-referrer-when-downgrade

发现两者匹配上了所有命中304协商缓存。

一旦更新后。



Request URL: http://www.dev.com/x

Request Method: GET

Status Code: 200 OK

Remote Address: 127.0.0.1:8899

Referrer Policy: no-referrer-when-downgrade

发现两者匹配不上,就重新请求了

一般浏览器会默认进行缓存行为，但是浏览器缓存设置是可由服务器端生成的
浏览器通过置身机制判定服务器返回的请求头【cache-control, expires, last-Modified, ETag等】实现缓存，关键的请求头有cache-control, expires, last-Modified, ETag等。
浏览器缓存方式有两种（强缓存和协商缓存）

课程小结

■ 发布新的静态资源的时候，如何更新缓存？

■ HTTP缓存有大小限制吗？FIFO还是LRU

1. 大厂有个处理静态资源的方案,每次发布静态资源的文件名都不同(我们的静态资源都是由HTML或者js的引用到了，最后是把那些引用都改成新的文件名地址，所以我们每次更新资源都是不删旧资源，只是添加新资源，有人说这种网络上静态资源会原来越多，这个是真的，但是运营商不对存储付费的，静态资源付的是流量费用，至于如何清理缓存，则是遵循几个fifo LRU等清理缓存策略，因为只是一个cdn，最终还是需要cdn回源的)
2. HTTP肯定是由大小限制的，服务端需要跟各端协商，我们关心的web浏览器也是由缓存大小限制的。对于我们web端来说，浏览器已经实现了相关的缓存策略的