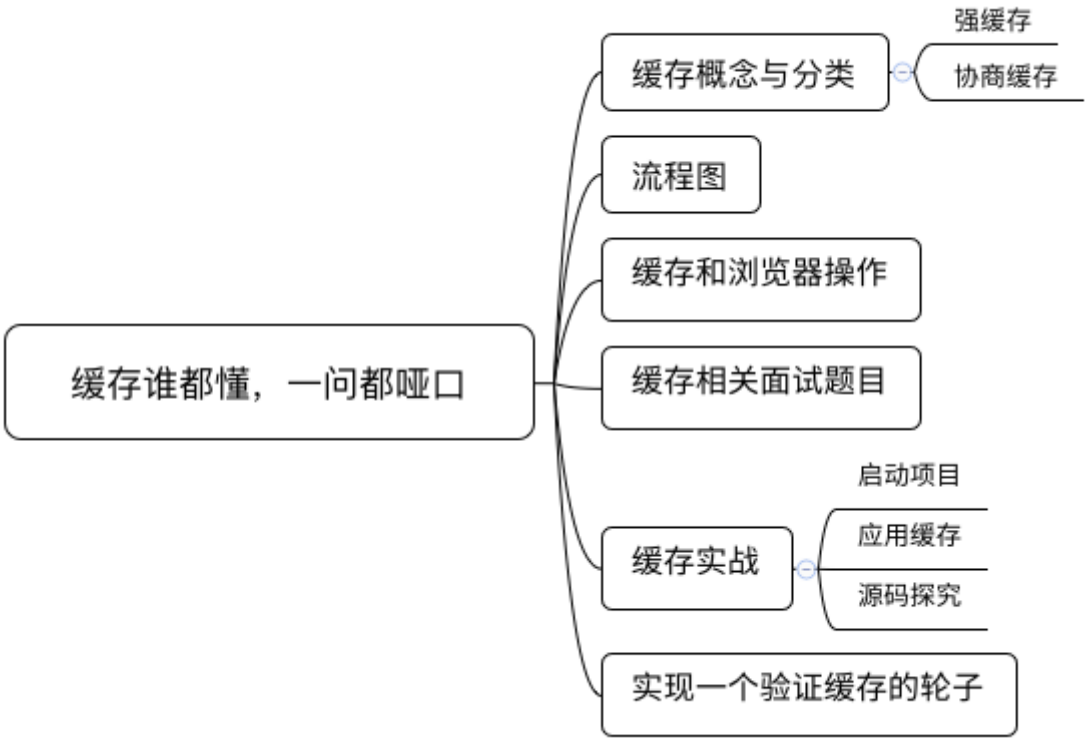
前端开发核心知识进阶：50 讲从夯实基础到突破瓶颈
来自 Lucas ... · 盐选专栏

查看详情 >

缓存谁都懂，一问都哑口

上一讲，我们了解了缓存的几种方式和基本概念；这一讲，让我们从应用和面试的角度出发，巩固理论基础，加深操作印象。



缓存和浏览器操作

缓存的重要一环是浏览器，常见浏览器行为对应的缓存行为有哪些呢？我们来做一个总结（注意，不同浏览器引擎、不同版本可能会有差别，读者可以根据不同情况酌情参考）：

当用户 Ctrl + F5 强制刷新网页时，浏览器直接从服务器加载，跳过强缓存和协商缓存

当用户仅仅敲击 F5 刷新网页时，跳过强缓存，但是仍然会进行协商缓存过程

这里我借用 Alloy Team 的图进行一个总结：

浏览器相关操作	Expires/Cache-Control	Last-Modified / Etag
地址栏回车	有效	有效
页面链接跳转	有效	有效
新开窗口	有效	有效
前进、后退	有效	有效
刷新	无效	有效
强制刷新	无效	无效

缓存相关面试题目

知识点我们已经梳理完毕，是时候刷一下经典题目来巩固了。以下题目都可以在上述知识中找到答案，我们也当做一个总结和考察。

题目一：如何禁止浏览器不缓存静态资源

在实际工作中，很多场景都需要禁用浏览器缓存。比如可以使用 Chrome 隐私模式，在代码层面可以设置相关请求头：

```
Cache-Control: no-cache, no-store, must-revalidate
```

此外，也可以给请求的资源增加一个版本号：

我们也可以使用 Meta 标签来声明缓存规则：

> 题目二：设置以下 request/response header 会有什么效果?

```
cache-control: max-age=0
```

上述响应头属于强缓存，因为 max-age 设置为 0，所以浏览器必须发请求重新验证资源。这时候会走协商缓存机制，可能返回 200 或者 304。

> 题目三：设置以下 request/response header 会有什么效果?

```
cache-control: no-cache
```

上述响应头属于强缓存，因为设置 no-cache，所以浏览器必须发请求重新验证资源。这时候会走协商缓存机制。

> 题目四：除了上述方式，还有哪种方式可以设置浏览器必须发请求重新验证资源，走协商缓存机制?

设置 request/response header：

```
cache-control: must-revalidate
```

> 题目五：设置以下 request/response header 会有什么效果?

```
Cache-Control: max-age=60, must-revalidate
```

如果资源在 60s 内再次访问，走强缓存，可以直接返回缓存资源内容；如果超过 60s，则必须发送网络请求到服务端，去验证资源的有效性。

题目五：据你的经验，为什么大厂都不怎么用 Etag?

大厂多使用负载分担的方式来调度 HTTP 请求。因此，同一个客户端对同一个页面的多次请求，很可能被分配到不同的服务器来相应，而根据 ETag 的计算原理，不同的服务器，有可能在资源内容没有变化的情况下，计算出不一样的 Etag，而使得缓存失效。

题目六：Yahoo 的 YSlow 页面分析工具为什么推荐关闭 ETag?

因为 Etag 计算较为复杂，所以可能会使得服务端响应变慢。

缓存实战

我们来通过几个简单的真实项目案例实操一下缓存。

启动项目

首先创建项目：

```
mkdir cache

npm init
```

之后，得到 package.json，同时声明我们的相关依赖：

```
{
  "name": "cache",
  "version": "1.0.0",
  "description": "Cache demo",
  "main": "index.js",
  "scripts": {
    "start": "nodemon ./index.js"
  },
```

```
  "keywords": [
    "cache",
    "node"
  ],
  "devDependencies": {
    "@babel/core": "latest",
    "@babel/preset-env": "latest",
    "@babel/register": "latest",
    "koa": "latest",
    "koa-conditional-get": "^2.0.0",
    "koa-etag": "^3.0.0",
    "koa-static": "latest"
  },
  "dependencies": {
    "nodemon": "latest"
  },
  "license": "ISC"
}
```

使用 nodemon 来启动并 watch Node 脚本，同时配置 .babelrc 如下：

```
{
 "presets": [
   [
     "@babel/preset-env",
     {
       "targets": {
         "node": "current"
       }
     }
   ]
 ]
}
```

在 cache/static 目录下，创建 index.html 和一张测试图片 web.png：

前端开发核心知识进阶

```
.cache img {
    display: block;
    width: 100%;
}
```

看一下我们的核心脚本 index.js，其实就是一个简单的 NodeJS 服务：

index.js：

```
require('@babel/register');
require('./cache.js');
```

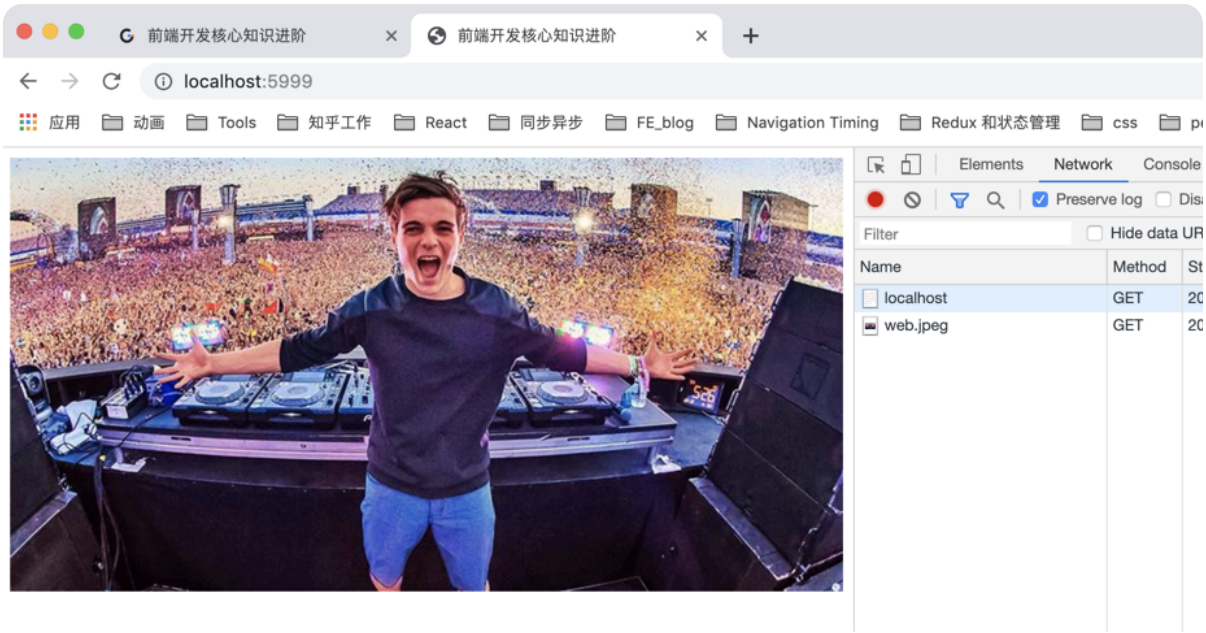
cache.js：

```
import Koa from 'koa'
import path from 'path'
import resource from 'koa-static'

const app = new Koa()
const host = 'localhost'
const port = 6666

app.use(resource(path.join(__dirname, './static')))

app.listen(port, () => {
  console.log(`server is listen in ${host}:${port}`)
})
```

我们启动：

```
npm run start
```

得到页面：

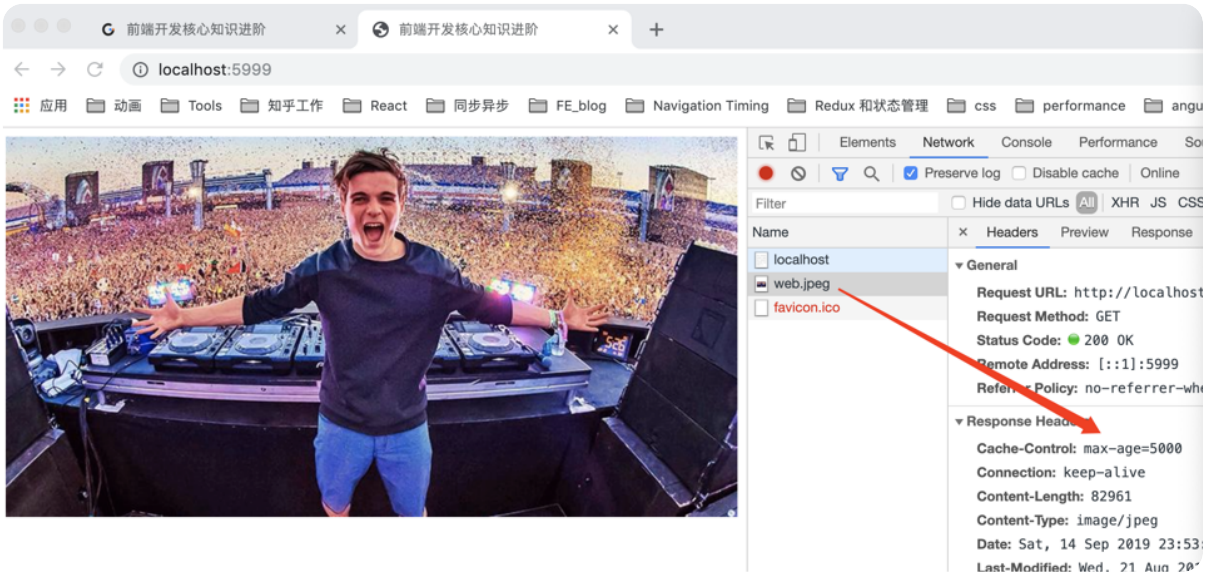


应用缓存

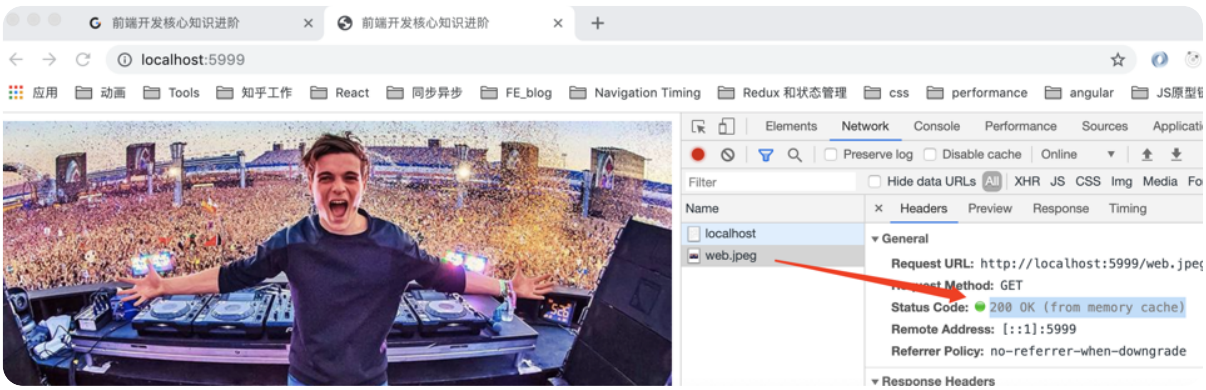
我们来尝试加入一些缓存，首先应用强缓存，只需要在响应头上加入相关字段即可：

```
import Koa from 'koa'
import path from 'path'
import resource from 'koa-static'

const app = new Koa()
const host = 'localhost'
const port = 5999

app.use(async (ctx, next) => {
  ctx.set({
    'Cache-Control': 'max-age=5000'
  })
  await next()
})

app.use(resource(path.join(__dirname, './static')))

app.listen(port, () => {
  console.log(`server is listen in ${host}:${port}`);
})
```

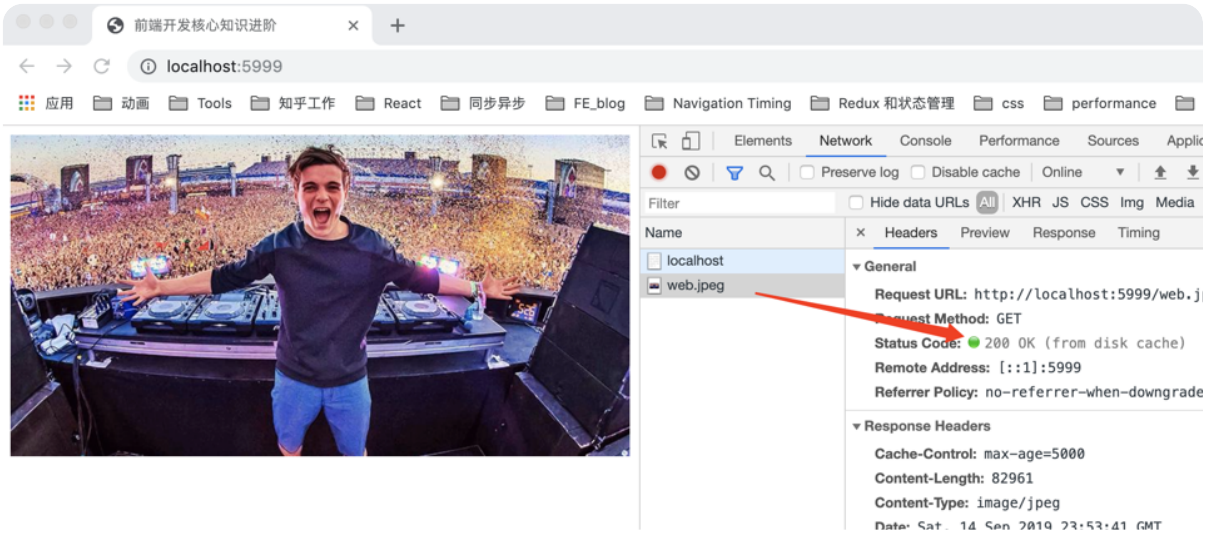
我们加入了 Cache-Control 头，设置 max-age 值为 5000。页面得到了响应：



再次刷新，得到了 200 OK（from memory cache）的标记：



当我们关掉浏览器，再次打开页面，得到了 200 OK（from disk cache）的标记。请体会与 from memory cache 的不同，memory cache 已经随着我们关闭浏览器而清除，这里是从 disk cache 取到的缓存。



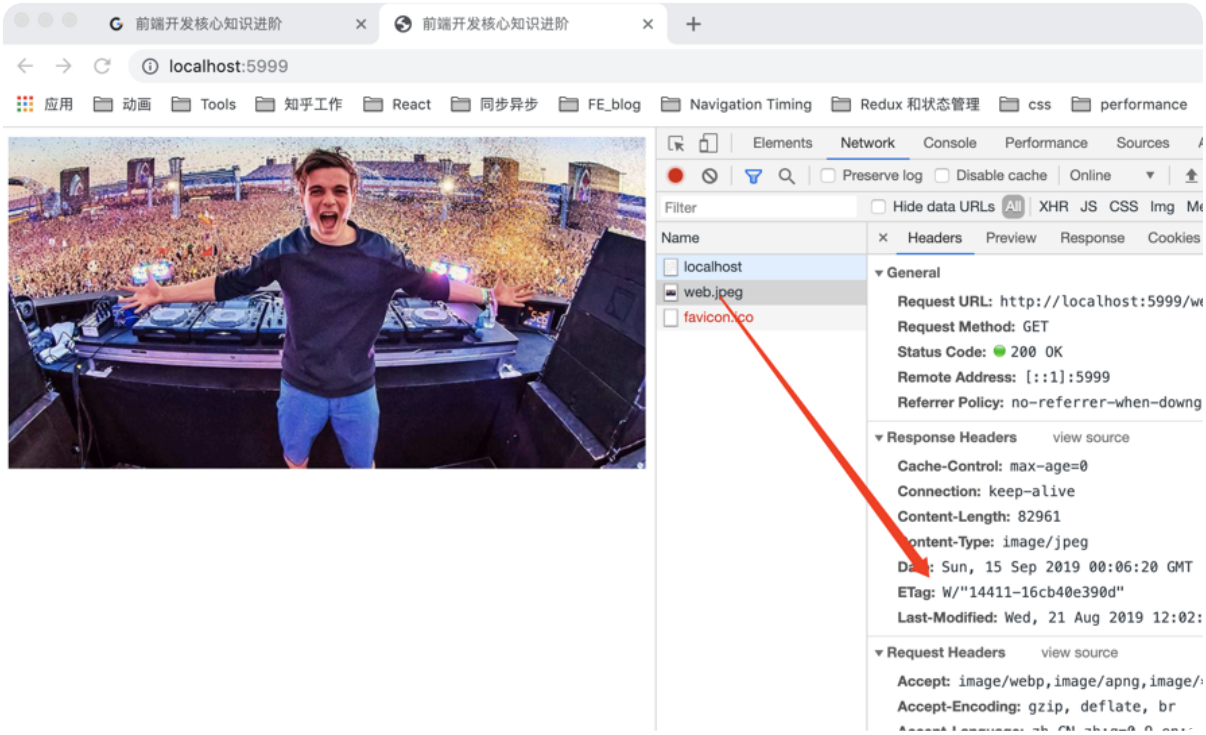
我们尝试将 max-age 改为 5 秒，5 秒后再次刷新页面，发现缓存已经失效。这里读者可以自行试验，不再截图了。

下面来试验一下协商缓存，在初始 package.json 中，已经引入了 koa-etag 和 koa-conditional-get 这两个包依赖。

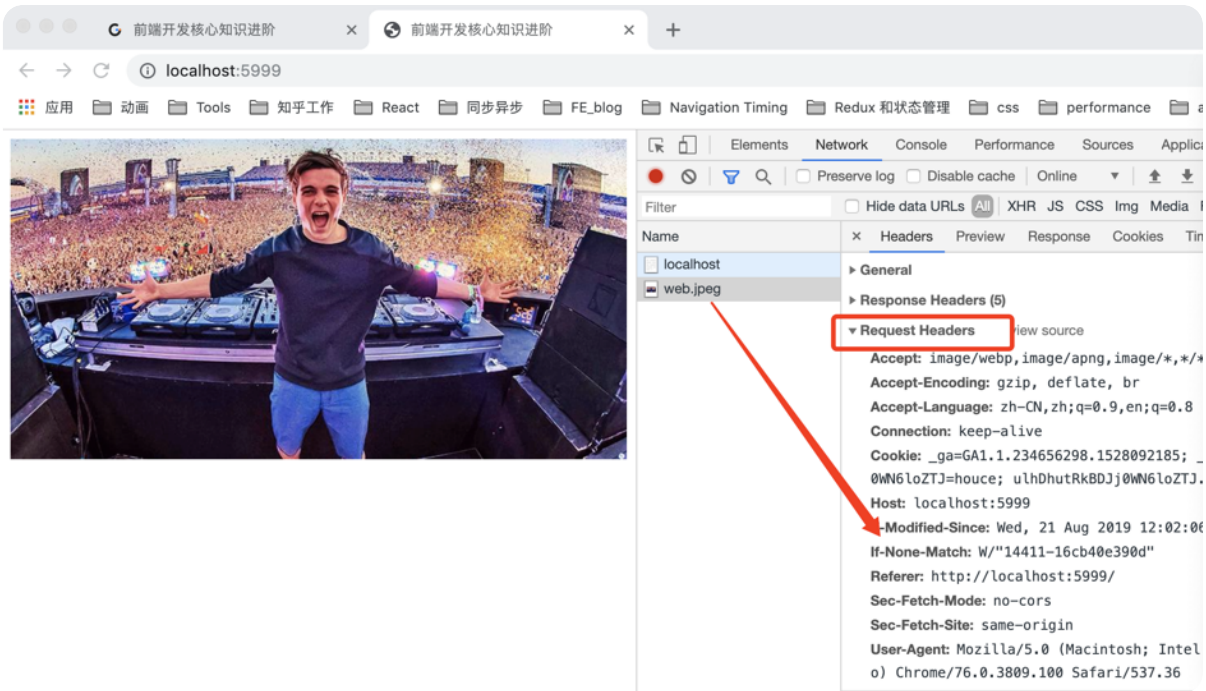
修改 cache.js 为：

```
import Koa from 'koa'
import path from 'path'
import resource from 'koa-static'
import conditional from 'koa-conditional-get'
import etag from 'koa-etag'

const app = new Koa()
const host = 'localhost'
const port = 5999

app.use(conditional())
app.use(etag())
app.use(resource(path.join(__dirname, './static')))

app.listen(port, () => {
 console.log(`server is listen in ${host}:${port}`)
})
```

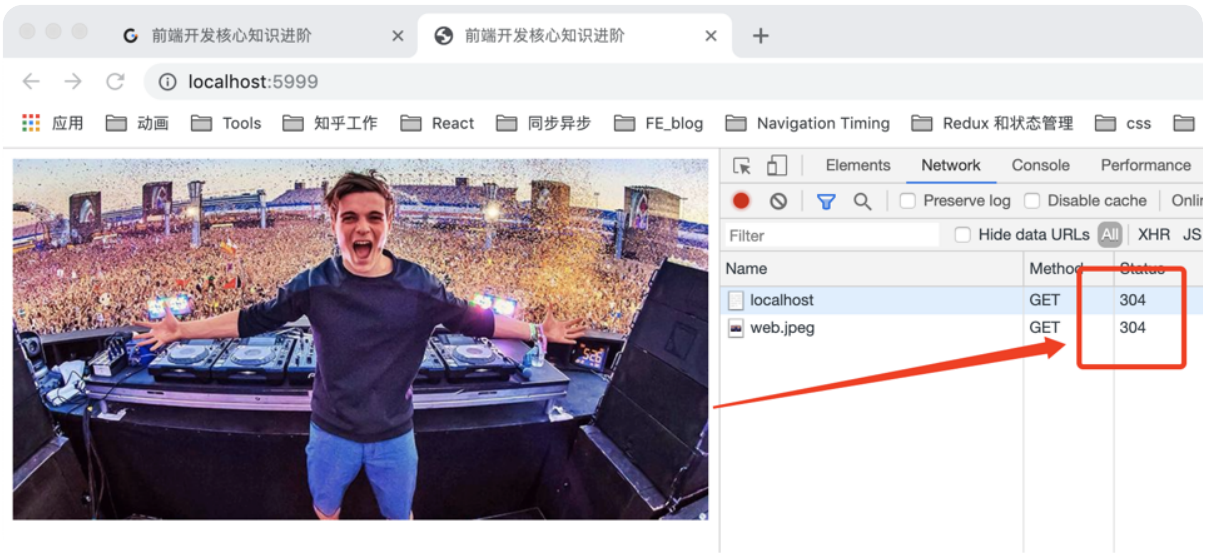
一切都很简单：



我们再次刷新浏览器，这次找到请求头，得到了 If-None-Match 字段，且内容与上一次的响应头相同。



因为我们的图片并没有发生变化，所以得到了 304 响应头。



读者可以自行尝试替换图片来验证内容。

这里我们主要使用了 Koa 库，如果对于原生 NodeJS，这里截取一个代码片段，供大家参考，该代码主要实现了 【if-modified-since/last-modified】头：

```
http.createServer((req, res) => {
    let { pathname } = url.parse(req.url, true)

    let absolutePath = path.join(__dirname, pathname)

    fs.stat(path.join(__dirname, pathname), (err, stat) => {
        // 路径不存在
        if(err) {
            res.statusCode = 404
            res.end('Not Fount')
            return
        }

        if(stat.isFile()) {
            res.setHeader('Last-Modified', stat.ctime.toGMTStr
ing())

            if(req.headers['if-modified-since'] === stat.ctime
.toGMTString()) {
```

```
                res.statusCode = 304
                res.end()
                return
            }

            fs.createReadStream(absolutePath).pipe(res)
        }
    })
})
```

该项目源码，读者可以在这里找到。

源码探究

在上面应用 Etag 试验当中，使用了 koa-etag 这个包，这里我们就来了解一下这个包的实现。

源码如下：

```
var calculate = require('etag');
var Stream = require('stream');
var fs = require('mz/fs');

module.exports = etag;

function etag(options) {
  return function etag(ctx, next) {
    return next()
      .then(() => getResponseEntity(ctx))
      .then(entity => setEtag(ctx, entity, options));
  };
}

function getResponseEntity(ctx, options) {
  // no body
  var body = ctx.body;
  if (!body || ctx.response.get('ETag')) return;
```

```
  // type
  var status = ctx.status / 100 | 0;

  // 2xx
  if (2 != status) return;

  if (body instanceof Stream) {
    if (!body.path) return;
    return fs.stat(body.path).catch(noop);
  } else if (('string' == typeof body) || Buffer.isBuffer(body
)) {
    return body;
  } else {
    return JSON.stringify(body);
  }
}


function setEtag(ctx, entity, options) {
  if (!entity) return;

  ctx.response.etag = calculate(entity, options);
}

function noop() {}
```

我们看整个 etag 库就是一个中间件，它首先调用 getResponseEntity 方法获取响应体，根据 body 最终调用了 setEtag 方法，根据响应内容生产 etag。最终生成 etag 的计算过程又利用了 etag 这个包，再来看一下 etag 库：

```
'use strict'

module.exports = etag

var crypto = require('crypto')
var Stats = require('fs').Stats
```

```
var toString = Object.prototype.toString

function entitytag (entity) {
  if (entity.length === 0) {
    // fast-path empty
    return '"0-2jmj7l5rSw0yVb/vlWAYkK/YBwk"'
  }

  // compute hash of entity
  var hash = crypto
    .createHash('sha1')
    .update(entity, 'utf8')
    .digest('base64')
    .substring(0, 27)

  // compute length of entity
  var len = typeof entity === 'string'
    ? Buffer.byteLength(entity, 'utf8')
    : entity.length

  return '"' + len.toString(16) + '-' + hash + '"'
}

function etag (entity, options) {
  if (entity == null) {
    throw new TypeError('argument entity is required')
  }

  // support fs.Stats object
  var isStats = isstats(entity)
  var weak = options && typeof options.weak === 'boolean'
    ? options.weak
    : isStats

  // validate argument
  if (!isStats && typeof entity !== 'string' && !Buffer.isBuff
er(entity)) {
```

```
    throw new TypeError('argument entity must be string, Buffe
r, or fs.Stats')
  }

  // generate entity tag
  var tag = isStats
    ? stattag(entity)
    : entitytag(entity)

  return weak
    ? 'W/' + tag
    : tag
}

function isstats (obj) {
  // genuine fs.Stats
  if (typeof Stats === 'function' && obj instanceof Stats) {
    return true
  }

  // quack quack
  return obj && typeof obj === 'object' &&
    'ctime' in obj && toString.call(obj.ctime) === '[object Da
te]' &&
    'mtime' in obj && toString.call(obj.mtime) === '[object Da
te]' &&
    'ino' in obj && typeof obj.ino === 'number' &&
    'size' in obj && typeof obj.size === 'number'
}

function stattag (stat) {
  var mtime = stat.mtime.getTime().toString(16)
  var size = stat.size.toString(16)

  return '"' + size + '-' + mtime + '"'
}
```

etag 方法接受一个 entity 最为入参一，entity 可以是 string、Buffer 或者 Stats 类型。如果是 Stats 类型，那么 etag 的生成方法会有不同：

```
var mtime = stat.mtime.getTime().toString(16)
var size = stat.size.toString(16)

return '"' + size + '-' + mtime + '"'
```

主要是根据 Stats 类型的 entity 的 mtime 和 size 特征，拼成一个 etag 即可。

如果是正常 String 或者 Buffer 类型，etag 的生成方法依赖了内置 crypto 包，主要是根据 entity 生成 hash，hash 生成主要依赖了 sha1 加密方法：

```
var hash = crypto
    .createHash('sha1')
    .update(entity, 'utf8')
    .digest('base64')
```

了解了这些，如果面试官再问「Etag 的生成方法」，我想读者已经能够有一定底气了。

实现一个验证缓存的轮子

分析完关于 etag 的这个库，我们来尝试自救造一个轮子，也当作留给大家的一个作业。这个轮子的需要完成验证缓存是否可用的功能，它接受请求头和响应头，并根据这两个头部，返回一个布尔值，表示缓存是否可用。

应用举例：

```
var reqHeaders = { 'if-none-match': '"foo"' }
var resHeaders = { 'etag': '"bar"' }
isFresh(reqHeaders, resHeaders)
// => false

var reqHeaders = { 'if-none-match': '"foo"' }
```

```
var resHeaders = { 'etag': '"foo"' }
isFresh(reqHeaders, resHeaders)
// => true
```

在业务端使用时，可以直接：

```
var isFresh = require('is-fresh')
var http = require('http')

var server = http.createServer(function (req, res) {

  if (isFresh(req.headers, {
    'etag': res.getHeader('ETag'),
    'last-modified': res.getHeader('Last-Modified')
  })) {
    res.statusCode = 304
    res.end()
    return
  }

  res.statusCode = 200
  res.end('hello, world!')
})

server.listen(3000)
```

实现这道题目的前提就是先要了解缓存的基本知识，知晓缓存优先级。我们应该先验证 cache-control，之后验证 If-None-Match，之后是 If-Modified-Since。了解了这些，我们按部就班不难实现：

```
var CACHE_CONTROL_NO_CACHE_REGEXP = /(?:^|,)\s*?no-cache\s*?
(?:,|$)/

function fresh (reqHeaders, resHeaders) {
  // fields
  var modifiedSince = reqHeaders['if-modified-since']
```

```
  var noneMatch = reqHeaders['if-none-match']

  if (!modifiedSince && !noneMatch) {
    return false
  }

  var cacheControl = reqHeaders['cache-control']
  if (cacheControl && CACHE_CONTROL_NO_CACHE_REGEXP.test(cache
Control)) {
    return false
  }

  // if-none-match
  if (noneMatch && noneMatch !== '*') {
    var etag = resHeaders['etag']

    if (!etag) {
      return false
    }

    var etagStale = true
    var matches = parseTokenList(noneMatch)
    for (var i = 0; i < matches.length; i++) {
      var match = matches[i]
      if (match === etag || match === 'W/' + etag || 'W/' + ma
tch === etag) {
        etagStale = false
        break
      }
    }

    if (etagStale) {
      return false
    }
  }

  // if-modified-since
  if (modifiedSince) {
```

```
    var lastModified = resHeaders['last-modified']
    var modifiedStale = !lastModified || !(parseHttpDate(lastM
odified) <= parseHttpDate(modifiedSince))

    if (modifiedStale) {
      return false
    }
  }

  return true
}


function parseHttpDate (date) {
  var timestamp = date && Date.parse(date)

  return typeof timestamp === 'number'
    ? timestamp
    : NaN
}


function parseTokenList (str) {
  var end = 0
  var list = []
  var start = 0

  for (var i = 0, len = str.length; i < len; i++) {
    switch (str.charCodeAt(i)) {
      case 0x20: /*   */
        if (start === end) {
          start = end = i + 1
        }
        break
      case 0x2c: /* , */
        list.push(str.substring(start, end))
        start = end = i + 1
        break
```

```
        default:
          end = i + 1
          break
      }
    }

    list.push(str.substring(start, end))

    return list
  }
```

这个实现比较简单，读者可以尝试解读该源码，如果这两讲的内容你已经融会贯通，上述实现并不困难。

当然，缓存的轮子却也没有「想象的那么简单」，「上述的代码强健性是否足够」？「API 设计是否优雅」？等这些话题值得思考。也希望在整个内容完结后，针对实战代码的优化和调试，应用的踩坑和解决能够大家继续交流。我们也会针对上述代码，展开更多内容。

总结

我们通过两讲的学习，介绍了缓存这一热门话题。缓存体现了理论规范和实战结合的美妙，是网络应用经验的结晶。建议读者可以多观察大型门户网站、页面应用，并结合工程化知识来看待并学习缓存。

点击查看下一节

HTTP 的深思：我从何而来，去向何处