浏览器

BOM

JavaScript在浏览器环境下一般由三部分组成：

ECMAscript，DOM和BOM；

- ECMAscript核心：描述了JS的语法和基本对象，我们常说的ES6，就是属于这个部分的内容；
- DOM是文档对象模型，处理网页的API，来自于W3C的标准；
- BOM是Browser Object Model，浏览器对象模型，BOM顾名思义其实就是为了控制浏览器的行为而出现的接口；

W3C、ECMAscript以及MDN关系是什么?

W3C

w3c是一个非标准化的组织，最重要的工作是发展web规范

这些规范描述了web的通信协议（比如HTML和XHTML）和其他的构建模块。平时我们接触到的标准，比如：超文本标记语言、HTML5规范、事件规范，我们平时接触到的DOM一级、二级规范都是此组织制定的。

标准制定后，就是需要各方支持实现，得到了浏览器厂商的支持，所以会有一致的表现。

ECMAscript

ECMAscript也是一种标准，这个标准主要用于标准化JavaScript这种语言的，比如我们平时使用的es6、es7等都是标准化的产物。该标准由ECMA intermational进行，TC39委员会进行监督。

在浏览器中，以V8、JSCore、SpiderMonkey等引擎进行解析。

所以，node为什么是V8引擎。

- 语法（解析规则，关键词，流程控制，对象初始化，等等）
- 错误处理机制（throw，try...catch，以及创建用户定义错误类型的能力）
- 类型（布尔值，数字，字符串，函数，对象等等）
- 全局对象，在浏览器环境中，这个全局对象就是window对象，但是ECMAscript只定义那些不特定于浏览器的API（例如：parseInt，parseFloat，decodeURL，encodeURL等等）
- 基于原型的继承机制

- 内置对象和函数（JSON，Math，Array.prototype方法，对象内省（自检、自我检查，introspection）方法等等）
- strict mode

MDN

MDN全称是Mozilla Developer Network，它和前面的w3c和ECMAscript不太一样，这个组织不是为了标准化而诞生的。在MDN的官网上的左上角写着MDN web docs，很明显这是一个转为开发者服务的开发文档。

当然，w3c和ECMAscript也有对应的文档，但是平时开发的过程中，大家都比较习惯用MDN去查询资料，主要是MDN做的太好了，有各种比较容易理解的使用说明和兼容性说明。

常见 BOM 对象

Navigator

属性：

- appCodeName：返回浏览器的代码名。
- appName：返回浏览器的名称--Netscape
- appVersion：返回浏览器的平台和版本信息
- cookieEnabled：返回指明浏览器是否启用cookie的布尔值
- platform：返回运行浏览器的操作系统平台
- userAgent：返回由客户机发送服务器的user-agent头部的值
- onLine：判断浏览器是否在线
- connection：自动检测网络状况切换清晰度

location

属性：

- hash：描述锚点，简单来说就是url#的后边
- host：url的端口+端口
- hostname：主机路径
- href：完成url
- pathname：返回当前url的路径部分
- port：端口

- protocol：协议
- search：协议
- origin：协议、主机名、端口

location的方法：

- assign：加载新的文档
- reload：重新刷新页面，相当于刷新按钮
- replace：用新的文档替换当前文档，移动设备检测时的立刻跳转

screen

属性：

- avaiHleight：返回屏幕的高度（不包括windows任务栏）
- availWidth：返回屏幕的宽度（不包括windows任务栏）
- colorDepth：返回目标设备或缓冲器上的调色板的比特深度
- height：返回屏幕的总高度
- pixelDepth：返回屏幕的颜色分辨率（每像素的位数）
- width：返回屏幕的总宽度

History

方法：

- back()：返回上一页
- forward()：返回下一页
- go()：加载history列表中的某个具体页面
- pushState()：页面不刷新，不触发onPopState事件
- replaceState()：页面不刷新，不触发onPopState事件

事件模型

```
    <div id="app">
        <p id="dom">Click</p>
    </div>

    // 冒泡：p => div => body => HTML => document
    // 捕获：document => HTML => body => div => p

    el.addEventListener(event, function, useCapture) // useCapture - false
```

```
    // 追问:
    // 1. 如何阻止事件传播
    event.stopPropagation()
    // 注意：无论向上还是向下都可以阻止 => 无法阻止默认事件的发生，如a标签的跳转

    // 2. 如何阻止默认事件的传播
    event.preventDefault()

    // 3. 相同节点绑定多个同类事件，如何阻止
    event.stopImmediatePropagation()

```

事件委托

```
<!DOCTYPE html>

<html>

<head>
    <meta charset="UTF-8">
</head>

<body>
    <ul id="ul">
        <li>1</li>
        <li>2</li>
        <li>3</li>
        <li>4</li>
        <li>5</li>
        <li>6</li>
        <li>7</li>
        <li>8</li>
    </ul>
</body>
<script type="text/javascript">
    const ul = document.querySelector("ul");
    ul.addEventListener('click', function (e) {
        const target = e.target;
        if (target.tagName.toLowerCase() === "li") {
            const liList = this.querySelectorAll("li");
            index = Array.prototype.indexOf.call(liList, target);
            alert(`内容为${target.innerHTML}, 索引为${index}`);
        }
    })
```

```
</script>

</html>
```

实现懒加载

使用 scroll event 和 getBoundingClientRect 实现

将img的src属性值写进data-src中，监听页面滚动事件，当图片可以看见，也就是说img元素顶部距离视窗顶部距离小于视窗高度，则加载data-src中的图片路径，当img元素不能看见的时候，即img元素顶部距离视窗顶部距离小于视窗大小的时候，则不加载图片路径

```
<img data-src="./logo.png" alt="">
<img data-src="./logo.png" alt="">
<img data-src="./logo.png" alt="">

<script>
  const imgs = document.querySelectorAll('img');
  window.addEventListener('scroll', (e) => {
      imgs.forEach( img => {
          const imgTop = img.getBoundingClientRect().top;
          if(imgTop < window.innerHeight) {
              const data_src = img.getAttribute('data-src');
              img.setAttribute('src', data_src)
          }
      })
  })
</script>
```

IntersectionObserver 实现

IntersectionObserver是浏览器提供的函数，交叉观察，当目标元素和可是窗口出现交叉区域时，触发事件事件会触发两次，目标元素看见和看不见都会触发

```
<script>
const imgs = document.querySelectorAll('img');
const callback = (entries) => {
        // 回调函数会接受一个entries参数
        entries.forEach( entry => {
                if(entry.isIntersecting) {  // 此属性进入交叉区就会为true，否则为false
                        const img= entry.target;  // 获取当前进入交叉区的元素
                        const data_src = img.getAttribute('data-src');
```

```
                        img.setAttribute('src', data_src);
                        observer.unobserve(img); // 加载完图片后对当前img元素停止观察
                }
        })
}
const observer = new IntersectionObserver(callback);

imgs.forEach(img => {
        observer.observe(img)
})
</script>

```

浏览器请求

XHR

```
    // 实例化
    const xhr = new XMLHttpRequest()

    // 初始化连接
    xhr.open(method, url, async)

    // 发送请求
    xhr.send(data)

    // 接收统计
    xhr.readyStatus // 0 - 尚未调用open  1 - 已经open 2 - 已经send 3 - 已接收到请求 4 - 请求已完成

    // 接收回调
    xhr.onreadystatechange = () => {
        if (xhr.readyStatus === 4) {
            if (
                xhr.status >= 200
                    && xhr.status < 300
                    || xhr.status == 304
            ) {
                // 请求成功
            }
        }
    }

    // 设置超时
```

```
    xhr.timeout = 1000
    xhr.ontimeout = () => {
        // ^^^
    }
```

fetch

1. 默认不带 cookie(可以自己设置 credentials)

```
fetch('http://domain/service', {
    method: 'POST',
    credentials: 'same-origin' // include 可以同域发送，也可以跨域发送，*same-origin 只能同域发送，不能跨域发送，omit 忽略cookie的发送
  })
```

2. 错误不会 reject。*http 错误，比如：404、5** 不会导致 fetch 返回的 promise 标记为 reject，.catch 不会被触发想要精准判断 fetch 是否成功，需要包含 promise resolved 的情况，判断 response.ok 是不是 true*

```
fetch('http://domain/service', {
    method: 'GET'
  })
    .then((response) => {
      // 404、5** 会进入.then
      if (response.ok) {
        // 请求成功，200 201
        return response.json()
      } else {
        throw new Error('fetch error')
      }
    })
    .then((json) => {
      // todo
    })
    .catch((error) => console.log(error))
```

3. 不支持超时设置

```
function fetchTimeout(url, init, timeout = 9999) {
    return new Promise((resolve, reject) => {
      // fetch setTimeout 同时开始，谁先结束先使用哪个
```

```
        // 未超时，  fetch resolve/reject
        // 超时，  setTimeout reject
        fetch(url, init).then(resolve).catch(reject)
        setTimeout(reject, timeout)
      })
    }
```

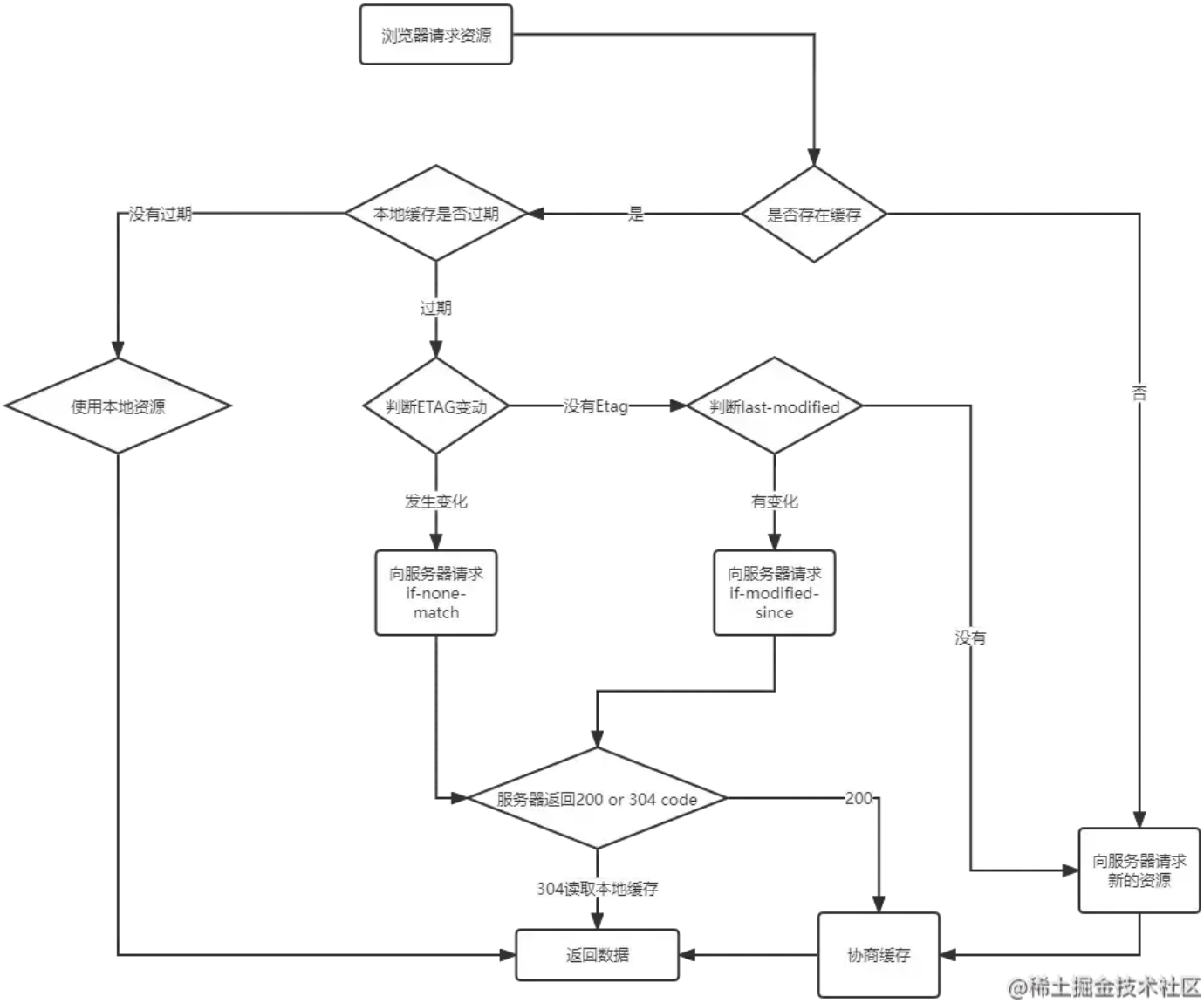
4. 中止 fetch（借用 AbortController）

```
const controller = new AbortController()

 fetch('http://domain/service', {
   method: 'GET',
   signal: controller.signal
 })
   .then((response) => response.json())
   .then((json) => console.log(json))
   .catch((error) => console.error('Error:', error))

 // 在其他使用的地方 终止请求
 controller.abort()
```

http缓存

缓存的原理是在**首次请求**后保存一份请求资源的**响应副本**，当用户**再次**发起相同请求后，如果判断缓存命中则拦截请求，将之前存储的**响应副本返回**给用户，从而避免了重新向服务器发起资源请求。

HTTP缓存应该算是前端开发中最常接触的缓存之一，它又可以细分为**强制缓存**和**协商缓存**，二者最大的区别在于判断缓存命中时，浏览器是否需要向服务器端进行询问以协商缓存的相关信息，进而判断是否需要就响应内容进行重新请求，下面让我们来看看HTTP缓存的具体机制及缓存的决策策略。



当浏览器发起一个资源请求时，浏览器会先判断本地是否有缓存记录，如果没有会向浏览器请求新的资源，并记录服务器返回的last-modified。

如果有缓存记录，先判断强缓存是否存在（cache-control优先于expires，后面会说），如果强缓存的时间没有过期则返回本地缓存资源（状态码为200）

如果强缓存失效了，客户端会发起请求进行协商缓存策略，首先服务器判断Etag标识符，如果客户端传来标识符和当前服务器上的标识符是一致的，则返回状态码 304 not modified（不会返回资源内容）

如果没有Etag字段，服务器会对比客户端传过来的if-modified-match，如果这两个值是一致的，此时响应头不会带有last-modified字段（因为资源没有变化，last-modified的值也不会有变化）。客户端304状态码之后读取本地缓存。如果last-modified。

如果Etag和服务器端上的不一致，重新获取新的资源，并进行协商缓存返回数据。

强缓存

对于强制缓存而言，如果浏览器判断所请求的目标资源有效命中，则直接从强制缓存中返回请求响应，**无须与服务器进行任何通信**。返回200的状态码。

> 在浏览器控制台 `NetWork` 中的体现为：
> `200 OK (from disk cache)` 或者 `200 OK (from memory cache)`

释义

- `200 OK (from disk cache)` HTTP状态码200，缓存的文件从硬盘中读取
- `200 OK (from memory cache)` HTTP状态码200，缓存的文件从内存中读取

强缓存可以通过设置两种 HTTP Header 实现：Expires 和 Cache-Control。

基于expires实现

缓存过期时间，用来指定资源到期的时间，是服务器端的具体的时间点。也就是说，Expires=max-age + 请求时间，需要和Last-modified结合使用。 Expires是Web服务器响应消息头字段，在响应http请求时告诉浏览器在过期时间前浏览器可以直接从浏览器缓存取数据，而无需再次请求。

- 在以前，我们通常会使用响应头的 `Expires` 字段去实现强缓存,现在的项目中基本上**不推荐**使用expires
- HTTP1.0协议中声明的用来 `控制缓存失效日期时间戳` 的字段
- 由服务器端指定后通过响应头告知浏览器，浏览器在接收到带有该字段的响应体后进行缓存。
- 如果浏览器再次发起相同的资源请求，便会对比 `expires` 与本地当前的时间戳。
  - 当前请求的本地时间戳小于expires的值，则说明浏览器缓存的响应还未过期，无须向服务器端再次发起请求
  - 当本地时间戳大于expires值，缓存过期，重新向服务器发起请求。（仅过期才能允许再次发送请求）
- expires判断的局限性：
  - 对本地时间戳过分依赖
  - 如果客户端本地的时间与服务器端的时间不同步，或者对客户端的时间进行主动修改，那么对于缓存过期的判断可能就无法和预期相符。
- **解决expires判断的局限性**：HTTP1.1协议开始新增了 `cache-control` 字段
  - 比如：cache-control设置了maxage=31536000的属性值来控制响应资源的有效期，它是一个以秒为单位的时间长度，表示该资源在被请求到后的31536000秒内有效，如此便可避免服务器端和客户端时间戳不同步而造成的问题。

基于cache-control实现

在HTTP/1.1中，Cache-Control是最重要的规则，主要用于控制网页缓存。比如当Cache-Control:max-age=300时，则代表在这个请求正确返回时间（浏览器也会记录下来）的5分钟内再次加载资源，就会命中强缓存。

Cache-Control:max-age=N，N就是需要缓存的秒数。从第一次请求资源的时候开始，往后N秒内，资源若再次请求，则直接从磁盘（或内存中读取），不与服务器做任何交互。

Cache-control中因为max-age后面的值是一个滑动时间，从服务器第一次返回该资源时开始倒计时。所以也就不需要比对客户端和服务端的时间，解决了Expires所存在的巨大漏洞。

基于 `cache-control` 实现的强缓存是当下项目中的常规方法，而基于 `expires` 实现的强缓存不被推荐使用。

Cache-Control 可以在请求头或者响应头中设置，并且可以组合使用多种指令

cache-control的参数：

- 1. `max-age`：（单位为s)表示响应资源能被缓存多久
  - `max-age` 和 `expires` 同时存在，则以 `max-age` 为准
- 2. `s-maxage`：（单位为s)缓存在代理服务器中的过期时长（仅当设置了 `public` 属性值时才是有效的）
  - **max-age和s-maxage并不互斥。他们可以一起使用**
- 3. `no-cache`：强制进行协商缓存
  - Cache-control中设置了no-cache，那么该资源会直接跳过强缓存的校验，直接去服务器进行协商缓存
- 4. `no-store`：禁止使用任何缓存
  - 客户端的每次请求都需要服务器端给予全新的响应
  - `no-cache` 与 `no-store` 是两个互斥的属性值，不能同时设置
- 5. `public`：表示响应资源既可以被浏览器缓存，又可以被代理服务器缓存
- 6. `private`：限制了响应资源只能被浏览器缓存
  - **public和private就是决定资源是否可以在代理服务器进行缓存的属性**
  - 如果这两个属性值都没有被设置，则默认为**private**
  - **public和private** 也是一组互斥属性

Cache-control如何设置多个值呢？用逗号分割

```
1 Cache-control:max-age=10000,s-maxage=200000,public
```

如果要考虑向下兼容的话，在 `Cache-control` 不支持的时候，还是要使用 `Expires`，这也是我们当前使用的这个属性的唯一理由。

强缓存两种方式的区别

- Expires 是 HTTP/1 的产物，受限于本地时间，如果修改了本地时间，可能会造成缓存失效。
- Expires 是http1.0的产物，Cache-Control是http1.1的产物

- expires/cache-control两者同时存在的话，Cache-Control优先级高于Expires
- 在某些不支持HTTP1.1的环境下，Expires就会发挥用处
- Expires其实是过时的产物，现阶段它的存在只是一种兼容性的写法。

强缓存的缺陷

强缓存判断是否缓存的依据来自于**是否超出某个时间或者某个时间段**，而**不关心**服务器端文件**是否已经更新**，这可能会导致加载文件不是服务器端最新的内容，**那我们如何获知服务器端内容是否已经发生了更新呢**？此时我们需要用到协商缓存策略。

协商缓存

顾名思义，协商缓存就是在使用本地缓存之前，需要向服务器发起一次GET请求，与之协商当前浏览器保存的本地缓存是否已经过期。通常是采用所请求资源的最近一次的修改时间戳来判断的。

协商缓存就是强制缓存失效后，浏览器携带缓存标识向服务器发起请求，由服务器根据缓存标识决定是否使用缓存的过程，主要有以下两种情况：

- 协商缓存生效（文件未更新），返回304和Not Modified。
- 协商缓存失效（文件更新），返回200和请求结果。

协商缓存可以通过设置两种 HTTP Header 实现：Last-Modified 和 ETag 。

基于last-modified实现

实现方式/流程

基于last-modified的协商缓存实现方式是:

1. 首先需要在服务器端读出文件修改时间，
2. 将读出来的修改时间赋给响应头的 `last-modified` 字段。
3. 最后设置 `Cache-control:no-cache`

第一次请求，服务器端代码

```
1
2  const data = fs.readFileSync('./imgs/CSS.png');//读取资源
3  // 1.读取修改的时间
4  const { mtime } = fs.statSync('./imgs/CSS.png');
5  // 2.设置文件最后修改时间
6  res.setHeader('last-modified',mtime.toUTCString())
7  // 3.强制设置为协商缓存
8  res.setHeader('Cache-Control','no-cache');
9  res.end(data);
```

第二次及以后的每一次请求，服务器端代码

```
const data = fs.readFileSync('./imgs/CSS.png');
//读取资源
const { mtime } = fs.statSync('./imgs/CSS.png');
//读取修改的时间
const ifModifiedSince = req.headers['if-modified-since'];
//读取请求头携带的时间（第一次返回给客户端的文件修改时间）
if (ifModifiedSince === mtime.toUTCString()) {
    // 如果两个时间一致，则文件没被修改过，返回304
    res.statusCode = 304;
    res.end();//因为缓存生效，不需要返回数据
    return;// 避免返回新的last-modified
}
res.setHeader('last-modified', mtime.toUTCString())
// 设置文件最后修改时间
res.setHeader('Cache-Control', 'no-cache');
// 强制设置为协商缓存
res.end(data);
```

流程：

- 客户端第一次请求目标资源的时，服务器返回的响应标头包含 `last-modified` 字段，值是 该资源的最后一次修改的时间戳 ，以及 `cache-control: no-cache`
- 当客户端再次请求该资源的时候，会携带一个 `if-modified-since` 字段，其值正是上次响应头中 `last-modified` 的字段值
- 如果客户端请求头携带的 `if-modified-since` 字段对应的时间与**目标资源的时间戳进行**对比，如果没有变化则返回一个 `304` 状态码。

> 需要注意的是：协商缓存判断缓存有效的响应状态码是304，但是如果是强制缓存判断有效的话，响应状态码是200。

last-modified的不足

- last-modified是根据请求资源的最后修改时间戳进行判断的，虽然请求的文件资源进行了编辑，但是**内容并没有发生任何变化，时间戳也会更新**，从而导致协商缓存时关于有效性的判断验证为失效，需要重新进行完整的资源请求。浪费了网络的带看资源，延长获取资源的时间
- 标识文件资源**修改的时间戳单位是秒**，如果文件修改的速度非常快，假设在几百**毫秒内**完成，那么通过时间戳的方式来验证缓存的有效性，是无法识别出该次文件资源的更新的。

综合，以上两种不足可知，基于 `last-modified` 实现的协商缓存，服务器无法根据资源修改的时间戳识别出真正的更新，进而导致重新发起了请求

基于Etag实现

为了弥补通过时间戳判断的不足，从HTTP1.1规范开始新增了一个Etag的头信息，即实体标签。 其内容主要是服务器为不同的资源进行哈希计算所生成的一个字符串，该字符串类似于文件指纹，只要文件内容编码存在差异，对应的Etag对文件资源进行更精准的变化感知。

也就是说， `ETag` 就是将原先基于 `last-modified` 协商缓存的比较**时间戳**的形式修改成了比较**文件指纹**。

文件指纹:根据文件内容计算出的唯一哈希值。文件内容一旦改变则指纹改变。

服务端代码

```
const data = fs.readFileSync('./imgs/CSS.png');
const etagContent = etag(data);

const ifNoneMatch = req.headers['if-none-match'];
if (ifNoneMatch === etagContent) {
    res.statusCode = 304;
    res.end();
    return;
}

res.setHeader('etag',etagContent)
res.setHeader('Cache-Control','no-cache');
res.end(data);
```


实现方式/流程

- 第一次请求资源时，服务端将要返回给客户端的数据通过 `ETag` 模块进行哈希计算生成一个字符串，这个字符串类似于文件指纹。
- 第二次请求资源时，客户端自动从缓存中读取出上一次服务端返回的 `ETag` 也就是文件指纹。并赋给请求头的 `if-None-Match` 字段，让上一次的文件指纹跟随请求一起回到服务端。
- 检测客户端的请求标头中的 `if-None-Match` 字段的值和第一步计算的值是否一致，一致则返回304。
- 如果不一致则返回etag标头和Cache-Control：no-cache。

不足

> 在协商缓存中，Etag并非last-modified的替代方案而是一种补充方案，因为依旧存在一些弊端。

- 服务器对于生成文件资源的Etag需要付出额外的计算开销，如果资源的尺寸比较大，数量较多且修改频繁，那么生成的Etag的过程就会影响服务器的性能。
- Etag字段值的生成分为 `强验证` 和 `弱验证` ，**强验证根据资源内容进行生成，能够保证每个字节都相同，弱验证则根据资源的部分属性来生成，生成速度快但无法确保每个字节都相同**，并且在服务器集群场景下，也会因为准确不够而降低协商缓存有效性的成功率，所以恰当的方式是根据具体的资源使用场景选择恰当的缓存校验方式。

Etag/last-modified对比

- 首先在精确度上，Etag要优于Last-Modified。
  Last-Modified的时间单位是秒，如果某个文件在1秒内改变了多次，那么他们的Last-Modified其实并没有体现出来修改，但是Etag每次都会改变确保了精度；如果是负载均衡的服务器，各个服务器生成的Last-Modified也有可能不一致。
- 第二在性能上，Etag要逊于Last-Modified
  Last-Modified只需要记录时间，而Etag需要服务器通过算法来计算出一个hash值。
- 第三在优先级上，服务器校验优先考虑Etag

课程代码

浏览器.zip
12.71KB