

# 浏览器缓存机制

注意：该知识点属于性能优化领域，并且整一章节都是一个面试题。

缓存可以说是性能优化中**简单高效**的一种优化方式了，它可以**显著减少网络传输**所带来的损耗。

对于一个数据请求来说，可以分为发起网络请求、后端处理、浏览器响应三个步骤。浏览器缓存可以帮助我们**在第一和第三步骤中优化性能**。比如说直接使用缓存而不发起请求，或者发起了请求但后端存储的数据和前端一致，那么就没有必要再将数据回传回来，这样就减少了响应数据。

接下来的内容中我们将通过以下几个部分来探讨浏览器缓存机制：

- 缓存位置
- 缓存策略
- 实际场景应用缓存策略

## 缓存位置

从缓存位置上来说分为四种，并且各自有**优先级**，当依次查找缓存且都没有命中的时候，才会去请求网络

1. Service Worker
2. Memory Cache
3. Disk Cache
4. Push Cache
5. 网络请求

## Service Worker

在上一章节中我们已经介绍了 Service Worker 的内容，这里就不演示相关的代码了。

Service Worker 的缓存与浏览器其他内建的缓存机制不同，它可以让我们**自由控制**缓存哪些文件、如何匹配缓存、如何读取缓存，并且**缓存是持续性的**。












当 Service Worker 没有命中缓存的时候，我们需要去调用 `fetch` 函数获取数据。也就是说，如果我们没有在 Service Worker 命中缓存的话，会根据缓存查找优先级去查找数据。**但是不管我们是从**

Memory Cache 中还是从网络请求中获取的数据，浏览器都会显示我们是从 Service Worker 中获取的内容。

## Memory Cache

Memory Cache 也就是内存中的缓存，读取内存中的数据肯定比磁盘快。但是内存缓存虽然读取高效，可是缓存持续性很短，会随着进程的释放而释放。一旦我们关闭 Tab 页面，内存中的缓存也就被释放了。

当我们访问过页面以后，再次刷新页面，可以发现很多数据都来自于内存缓存

| Name   | Status        | Type     | Initiator                | Size                | ... | Waterfall   |
|--|---------------|----------|--------------------------|---------------------|-----|---|
|  timeline                         | 200           | document | Other                    | 12.3 KB             | ... |  |
|  manifest.af6996a28865b94880dd.js | 200           | script   | <a href="#">timeline</a> | (from memory cache) | ... |  |
|  vendor.1020ec95ad777daa7497.js   | 200           | script   | <a href="#">timeline</a> | (from memory cache) | ... |  |
|  app.e4986bf68ccecea5526a.js      | 200           | script   | <a href="#">timeline</a> | (from memory cache) | ... |  |
|  adsbygoogle.js                   | (blocked:o... | script   | <a href="#">timeline</a> | 0 B                 | ... |   |
|  icon.a87e5ae.svg                 | 200           | svg+xml  | <a href="#">timeline</a> | (from memory cache) | ... |  |

从内存中读取缓存

那么既然内存缓存这么高效，我们是不是能让数据都存放在内存中呢？

先说结论，这是不可能的。首先计算机中的内存一定比硬盘容量小得多，操作系统需要精打细算内存的使用，所以能让我们使用的内存必然不多。内存中其实可以存储大部分的文件，比如说 JSS、HTML、CSS、图片等等。但是浏览器会把哪些文件丢进内存这个过程就很玄学了，我查阅了很多资料都没有一个定论。

当然，我通过一些实践和猜测也得出了一些结论：

- 对于大文件来说，大概率是不存储在内存中的，反之优先
- 当前系统内存使用率高的话，文件优先存储进硬盘

## Disk Cache

Disk Cache 也就是存储在硬盘中的缓存，读取速度慢点，但是什么都能存储到磁盘中，比之 Memory Cache 胜在容量和存储时效性上。

在所有浏览器缓存中，Disk Cache 覆盖面基本是最大的。它会根据 HTTP Herder 中的字段判断哪些资源需要缓存，哪些资源可以不请求直接使用，哪些资源已经过期需要重新请求。并且即使在跨站点的情况下，相同地址的资源一旦被硬盘缓存下来，就不会再次去请求数据。

## Push Cache

Push Cache 是 HTTP/2 中的内容，当以上三种缓存都没有命中时，它才会被使用。并且缓存时间也很短暂，只在会话（Session）中存在，一旦会话结束就被释放。

Push Cache 在国内能够查到的资料很少，也是因为 HTTP/2 在国内不够普及，但是 HTTP/2 将会是日后的一个趋势。这里推荐阅读 [HTTP/2 push is tougher than I thought](#) 这篇文章，但是内容是英文的，我翻译一下文章中的几个结论，有能力的同学还是推荐自己阅读

- 所有的资源都能被推送，但是 Edge 和 Safari 浏览器兼容性不怎么好
- 可以推送 `no-cache` 和 `no-store` 的资源
- 一旦连接被关闭，Push Cache 就被释放
- 多个页面可以使用相同的 HTTP/2 连接，也就是说能使用同样的缓存
- Push Cache 中的缓存只能被使用一次
- 浏览器可以拒绝接受已经存在的资源推送
- 你可以给其他域名推送资源

## 网络请求

如果所有缓存都没有命中的话，那么只能发起请求来获取资源了。

那么为了性能上的考虑，大部分的接口都应该选择好缓存策略，接下来我们就来学习缓存策略这部分的内容。

## 缓存策略

---

通常浏览器缓存策略分为两种：**强缓存**和**协商缓存**，并且缓存策略都是通过设置 HTTP Header 来实现的。

### 强缓存

强缓存可以通过设置两种 HTTP Header 实现：`Expires` 和 `Cache-Control` 。强缓存表示在缓存期间不需要请求，`state code` 为 200。

#### Expires

```
Expires: Wed, 22 Oct 2018 08:41:00 GMT
```

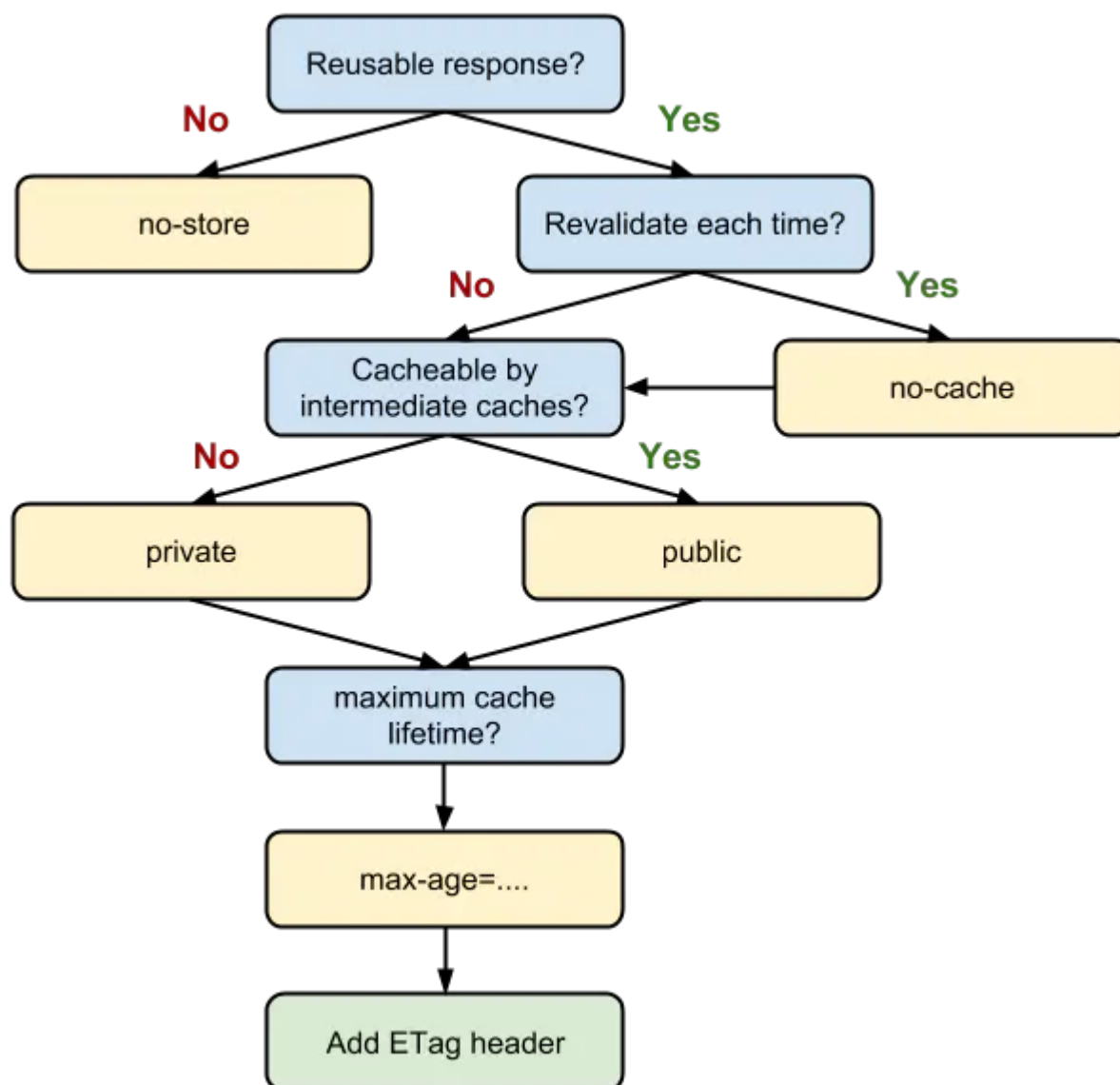
`Expires` 是 HTTP/1 的产物，表示资源会在 `Wed, 22 Oct 2018 08:41:00 GMT` 后过期，需要再次请求。并且 `Expires` 受限于本地时间，如果修改了本地时间，可能会造成缓存失效。

#### Cache-control

```
Cache-control: max-age=30
```

**Cache-Control** 出现于 HTTP/1.1，优先级高于 **Expires** 。该属性值表示资源会在 30 秒后过期，需要再次请求。

**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



多种指令配合流程图

从图中我们可以看到，我们可以将多个指令配合起来一起使用，达到多个目的。比如说我们希望资源能被缓存下来，并且是客户端和代理服务器都能缓存，还能设置缓存失效时间等等。

接下来我们就来学习一些常见指令的作用

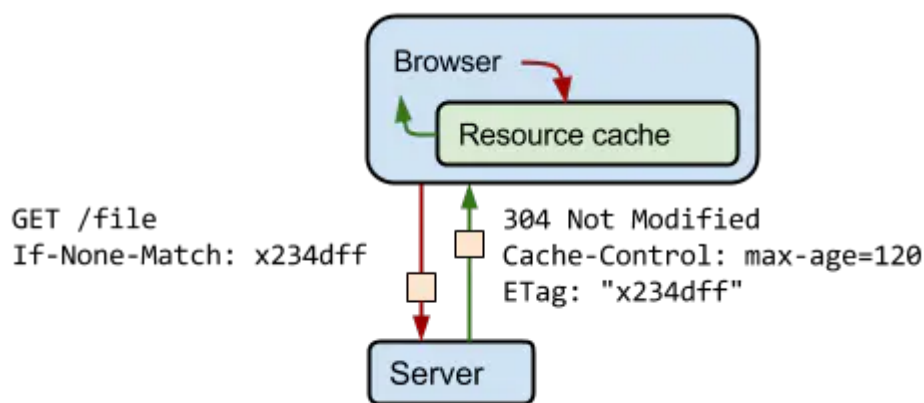
| 指令           | 作用                           |
|--------------|------------------------------|
| public       | 表示响应可以被客户端和代理服务器缓存           |
| private      | 表示响应只可以被客户端缓存                |
| max-age=30   | 缓存 30 秒后就过期，需要重新请求           |
| s-maxage=30  | 覆盖 max-age，作用一样，只在代理服务器中生效   |
| no-store     | 不缓存任何响应                      |
| no-cache     | 资源被缓存，但是立即失效，下次会发起请求验证资源是否过期 |
| max-stale=30 | 30 秒内，即使缓存过期，也使用该缓存          |
| min-fresh=30 | 希望在 30 秒内获取最新的响应             |

常见指令作用

## 协商缓存

如果缓存过期了，就需要发起请求验证资源是否有更新。协商缓存可以通过设置两种 HTTP Header 实现：`Last-Modified` 和 `ETag`。

当浏览器发起请求验证资源时，如果资源没有做改变，那么服务端就会返回 304 状态码，并且更新浏览器缓存有效期。



协商缓存

## Last-Modified 和 If-Modified-Since

`Last-Modified` 表示本地文件最后修改日期，`If-Modified-Since` 会将 `Last-Modified` 的值发送给服务器，询问服务器在该日期后资源是否有更新，有更新的话就会将新的资源发送回来，否则返回 304 状态码。

但是 `Last-Modified` 存在一些弊端：

- 如果本地打开缓存文件，即使没有对文件进行修改，但还是会造成 `Last-Modified` 被修改，服务端不能命中缓存导致发送相同的资源
- 因为 `Last-Modified` 只能以秒计时，如果在不可感知的时间内修改完成文件，那么服务端会认为资源还是命中了，不会返回正确的资源

因为以上这些弊端，所以在 HTTP / 1.1 出现了 `ETag` 。

## ETag 和 If-None-Match

`ETag` 类似于文件指纹，`If-None-Match` 会将当前 `ETag` 发送给服务器，询问该资源 `ETag` 是否变动，有变动的话就将新的资源发送回来。并且 `ETag` 优先级比 `Last-Modified` 高。

以上就是缓存策略的所有内容了，看到这里，不知道你是否存在这样一个疑问。如果什么缓存策略都没设置，那么浏览器会怎么处理？

对于这种情况，浏览器会采用一个启发式的算法，通常会取响应头中的 `Date` 减去 `Last-Modified` 值的 10% 作为缓存时间。

## 实际场景应用缓存策略

---

单纯了解理论而不付诸于实践是没有意义的，接下来我们来通过几个场景学习下如何使用这些理论。

### 频繁变动的资源

对于频繁变动的资源，首先需要使用 `Cache-Control: no-cache` 使浏览器每次都请求服务器，然后配合 `ETag` 或者 `Last-Modified` 来验证资源是否有效。这样的做法虽然不能节省请求数量，但是能显著减少响应数据大小。

### 代码文件

这里特指除了 HTML 外的代码文件，因为 HTML 文件一般不缓存或者缓存时间很短。

一般来说，现在都会使用工具来打包代码，那么我们就可以对文件名进行哈希处理，只有当代码修改后才会生成新的文件名。基于此，我们就可以给代码文件设置缓存有效期一年 `Cache-Control: max-age=31536000`，这样只有当 HTML 文件中引入的文件名发生了改变才会去下载最新的代码文件，否则就一直使用缓存。