**浏览器缓存**

缓存是性能优化中非常重要的一环，浏览器的缓存机制对开发也是非常重要的知识点。接下来以三个部分来把浏览器的缓存机制说清楚：

* 强缓存
* 协商缓存
* 缓存位置

## [#](http://47.98.159.95/my_blog/perform/001.html" \l "%E5%BC%BA%E7%BC%93%E5%AD%98)**强缓存**

浏览器中的缓存作用分为两种情况，一种是需要发送HTTP请求，一种是不需要发送。

首先是检查强缓存，这个阶段不需要发送HTTP请求。

如何来检查呢？通过相应的字段来进行，但是说起这个字段就有点门道了。

在HTTP/1.0和HTTP/1.1当中，这个字段是不一样的。在早期，也就是HTTP/1.0时期，使用的是****Expires****，而HTTP/1.1使用的是****Cache-Control****。让我们首先来看看Expires。

### [#](http://47.98.159.95/my_blog/perform/001.html" \l "expires)**Expires**

Expires即过期时间，存在于服务端返回的响应头中，告诉浏览器在这个过期时间之前可以直接从缓存里面获取数据，无需再次请求。比如下面这样:

Expires: Wed, 22 Nov 2019 08:41:00 GMT

表示资源在2019年11月22号8点41分过期，过期了就得向服务端发请求。

这个方式看上去没什么问题，合情合理，但其实潜藏了一个坑，那就是****服务器的时间和浏览器的时间可能并不一致****，那服务器返回的这个过期时间可能就是不准确的。因此这种方式很快在后来的HTTP1.1版本中被抛弃了。

### [#](http://47.98.159.95/my_blog/perform/001.html" \l "cache-control)**Cache-Control**

在HTTP1.1中，采用了一个非常关键的字段：Cache-Control。这个字段也是存在于

它和Expires本质的不同在于它并没有采用具体的过期时间点这个方式，而是采用过期时长来控制缓存，对应的字段是****max-age****。比如这个例子:

Cache-Control:max-age=3600

代表这个响应返回后在 3600 秒，也就是一个小时之内可以直接使用缓存。

如果你觉得它只有max-age一个属性的话，那就大错特错了。

它其实可以组合非常多的指令，完成更多场景的缓存判断, 将一些关键的属性列举如下: ****public****: 客户端和代理服务器都可以缓存。因为一个请求可能要经过不同的代理服务器最后才到达目标服务器，那么结果就是不仅仅浏览器可以缓存数据，中间的任何代理节点都可以进行缓存。

****private****： 这种情况就是只有浏览器能缓存了，中间的代理服务器不能缓存。

****no-cache****: 跳过当前的强缓存，发送HTTP请求，即直接进入协商缓存阶段。

****no-store****：非常粗暴，不进行任何形式的缓存。

****s-maxage****：这和max-age长得比较像，但是区别在于s-maxage是针对代理服务器的缓存时间。

****must-revalidate****: 是缓存就会有过期的时候，加上这个字段一旦缓存过期，就必须回到源服务器验证。

值得注意的是，当****Expires****和****Cache-Control****同时存在的时候，****Cache-Control****会优先考虑。

当然，还存在一种情况，当资源缓存时间超时了，也就是强缓存失效了，接下来怎么办？没错，这样就进入到第二级屏障——****协商缓存****了。

## [#](http://47.98.159.95/my_blog/perform/001.html" \l "%E5%8D%8F%E5%95%86%E7%BC%93%E5%AD%98)**协商缓存**

强缓存失效之后，浏览器在请求头中携带相应的缓存tag来向服务器发请求，由服务器根据这个tag，来决定是否使用缓存，这就是****协商缓存****。

具体来说，这样的缓存tag分为两种: ****Last-Modified**** 和 ****ETag****。这两者各有优劣，并不存在谁对谁有绝对的优势，跟上面强缓存的两个 tag 不一样。

### [#](http://47.98.159.95/my_blog/perform/001.html" \l "last-modified)**Last-Modified**

即最后修改时间。在浏览器第一次给服务器发送请求后，服务器会在响应头中加上这个字段。

浏览器接收到后，如果再次请求，会在请求头中携带If-Modified-Since字段，这个字段的值也就是服务器传来的最后修改时间。

服务器拿到请求头中的If-Modified-Since的字段后，其实会和这个服务器中该资源的最后修改时间对比:

* 如果请求头中的这个值小于最后修改时间，说明是时候更新了。返回新的资源，跟常规的HTTP请求响应的流程一样。
* 否则返回304，告诉浏览器直接用缓存。

### [#](http://47.98.159.95/my_blog/perform/001.html" \l "etag)**ETag**

ETag 是服务器根据当前文件的内容，给文件生成的唯一标识，只要里面的内容有改动，这个值就会变。服务器通过响应头把这个值给浏览器。

浏览器接收到ETag的值，会在下次请求时，将这个值作为****If-None-Match****这个字段的内容，并放到请求头中，然后发给服务器。

服务器接收到****If-None-Match****后，会跟服务器上该资源的****ETag****进行比对:

* 如果两者不一样，说明要更新了。返回新的资源，跟常规的HTTP请求响应的流程一样。
* 否则返回304，告诉浏览器直接用缓存。

### [#](http://47.98.159.95/my_blog/perform/001.html" \l "%E4%B8%A4%E8%80%85%E5%AF%B9%E6%AF%94)**两者对比**

1. 在精准度上，ETag优于Last-Modified。优于 ETag 是按照内容给资源上标识，因此能准确感知资源的变化。而 Last-Modified 就不一样了，它在一些特殊的情况并不能准确感知资源变化，主要有两种情况:

* 编辑了资源文件，但是文件内容并没有更改，这样也会造成缓存失效。
* Last-Modified 能够感知的单位时间是秒，如果文件在 1 秒内改变了多次，那么这时候的 Last-Modified 并没有体现出修改了。

1. 在性能上，Last-Modified优于ETag，也很简单理解，Last-Modified仅仅只是记录一个时间点，而 Etag需要根据文件的具体内容生成哈希值。

另外，如果两种方式都支持的话，服务器会优先考虑ETag。

## [#](http://47.98.159.95/my_blog/perform/001.html" \l "%E7%BC%93%E5%AD%98%E4%BD%8D%E7%BD%AE)**缓存位置**

前面我们已经提到，当强缓存命中或者协商缓存中服务器返回304的时候，我们直接从缓存中获取资源。那这些资源究竟缓存在什么位置呢？

浏览器中的缓存位置一共有四种，按优先级从高到低排列分别是：

* Service Worker
* Memory Cache
* Disk Cache
* Push Cache

### [#](http://47.98.159.95/my_blog/perform/001.html" \l "service-worker)**Service Worker**

Service Worker 借鉴了 Web Worker的 思路，即让 JS 运行在主线程之外，由于它脱离了浏览器的窗体，因此无法直接访问DOM。虽然如此，但它仍然能帮助我们完成很多有用的功能，比如离线缓存、消息推送和网络代理等功能。其中的离线缓存就是 ****Service Worker Cache****。

Service Worker 同时也是 PWA 的重要实现机制，关于它的细节和特性，我们将会在后面的 PWA 的分享中详细介绍。

### [#](http://47.98.159.95/my_blog/perform/001.html" \l "memory-cache-%E5%92%8C-disk-cache)**Memory Cache 和 Disk Cache**

****Memory Cache****指的是内存缓存，从效率上讲它是最快的。但是从存活时间来讲又是最短的，当渲染进程结束后，内存缓存也就不存在了。

****Disk Cache****就是存储在磁盘中的缓存，从存取效率上讲是比内存缓存慢的，但是他的优势在于存储容量和存储时长。稍微有些计算机基础的应该很好理解，就不展开了。

好，现在问题来了，既然两者各有优劣，那浏览器如何决定将资源放进内存还是硬盘呢？主要策略如下：

* 比较大的JS、CSS文件会直接被丢进磁盘，反之丢进内存
* 内存使用率比较高的时候，文件优先进入磁盘

### [#](http://47.98.159.95/my_blog/perform/001.html" \l "push-cache)**Push Cache**

即推送缓存，这是浏览器缓存的最后一道防线。它是 HTTP/2 中的内容，虽然现在应用的并不广泛，但随着 HTTP/2 的推广，它的应用越来越广泛。关于 Push Cache，有非常多的内容可以挖掘，不过这已经不是本文的重点，大家可以参考这篇[扩展文章](https://jakearchibald.com/2017/h2-push-tougher-than-i-thought/" \t "http://47.98.159.95/my_blog/perform/_blank)。

## [#](http://47.98.159.95/my_blog/perform/001.html" \l "%E6%80%BB%E7%BB%93)**总结**

对浏览器的缓存机制来做个简要的总结:

首先通过 Cache-Control 验证强缓存是否可用

* 如果强缓存可用，直接使用
* 否则进入协商缓存，即发送 HTTP 请求，服务器通过请求头中的If-Modified-Since或者If-None-Match这些条件请求字段检查资源是否更新
  + 若资源更新，返回资源和200状态码
  + 否则，返回304，告诉浏览器直接从缓存获取资源