

43 | 如何使用缓存优化系统性能？

2019-08-29 刘超



你好，我是刘超。

缓存是我们提高系统性能的一项必不可少的技术，无论是前端、还是后端，都应用到了缓存技术。前端使用缓存，可以降低多次请求服务的压力；后端使用缓存，可以降低数据库操作的压力，提升读取数据的性能。

今天我们将从前端到服务端，系统了解下各个层级的缓存实现，并分别了解下各类缓存的优缺点以及应用场景。

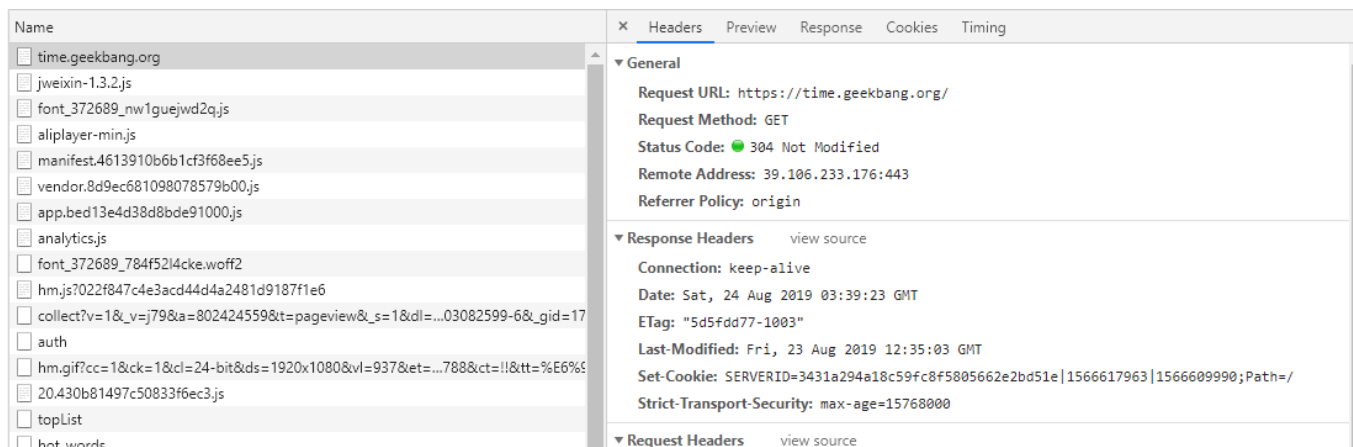
前端缓存技术

如果你是一位**Java**开发工程师，你可能会想，我们有必要去了解前端的技术吗？

不想当将军的士兵不是好士兵，作为一个技术人员，不想做架构师的开发不是好开发。作为架构工程师的话，我们就很有必要去了解前端的知识点了，这样有助于我们设计和优化系统。前端做缓存，可以缓解服务端的压力，减少带宽的占用，同时也可以提升前端的查询性能。

1. 本地缓存

平时使用拦截器（例如**Fiddler**）或浏览器**Debug**时，我们经常会发现一些接口返回**304**状态码+**Not Modified**字符串，如下图中的极客时间Web首页。



如果我们对前端缓存技术不了解，就很容易对此感到困惑。浏览器常用的一种缓存就是这种基于304响应状态实现的本地缓存了，通常这种缓存被称为协商缓存。

协商缓存，顾名思义就是与服务端协商之后，通过协商结果来判断是否使用本地缓存。

一般协商缓存可以基于请求头部中的If-Modified-Since字段与返回头部中的Last-Modified字段实现，也可以基于请求头部中的If-None-Match字段与返回头部中的ETag字段来实现。

两种方式的实现原理是一样的，前者是基于时间实现的，后者是基于一个唯一标识实现的，相对来说后者可以更加准确地判断文件内容是否被修改，避免由于时间篡改导致的不可靠问题。下面我们再来了解下整个缓存的实现流程：

- 当浏览器第一次请求访问服务器资源时，服务器会在返回这个资源的同时，在Response头部加上ETag唯一标识，这个唯一标识的值是根据当前请求的资源生成的；
- 当浏览器再次请求访问服务器中的该资源时，会在Request头部加上If-None-Match字段，该字段的值就是Response头部加上ETag唯一标识；
- 服务器再次收到请求后，会根据请求中的If-None-Match值与当前请求的资源生成的唯一标识进行比较，如果值相等，则返回304 Not Modified，如果不相等，则在Response头部加上新的ETag唯一标识，并返回资源；
- 如果浏览器收到304的请求响应状态码，则会从本地缓存中加载资源，否则更新资源。

本地缓存中除了这种协商缓存，还有一种就是强缓存的实现。

强缓存指的是只要判断缓存没有过期，则直接使用浏览器的本地缓存。如下图中，返回的是200状态码，但在size项中标识的是memory cache。

Name	Status	Type	Initiator	Size	Ti...	Waterfall
app.bed13e4d38d8bde91000.js	304	script	(index)	222 B	49...	
analytics.js	200	script	(index):22	(memory cache)	9...	
font_372689_784f5214cke.woff2	200	font	(index)	(memory cache)	0...	
hm.js?022f847c4e3acd44d4a2481d9187f1e6	304	script	(index):30	208 B	61...	
collect?v=1&_v=j79&a=802424559&t=pageview&s=1&dl=...03082...	200	gif	analytics.js:16	63 B	16...	
auth	200	xhr	vendor.8d9ec68...js:6	690 B	14...	
hm.gif?cc=1&ck=1&cl=24-bit&ds=1920x1080&vl=937&et=...788&ct...	200	gif	hm.js?022f847...:21	299 B	51...	
20.430b81497c50833f6ec3.js	304	script	manifest.4613910...js:1	350 B	14...	
topList	200	xhr	vendor.8d9ec68...js:6	8.9 KB	10...	
hot_words	200	xhr	vendor.8d9ec68...js:6	716 B	53...	
logo_pc@2x.90583da.png	304	png	(index)	345 B	18...	
favicon-32x32.jpg	200	jpeg	Other	3.2 KB	23...	
dd8cbc79f017d1b01f643c7ea929d79e.png	304	png	vendor.8d9ec68...js:1	341 B	27...	
4aeb8fb752fa21a0fd989a45d9847c3.png	304	png	vendor.8d9ec68...js:1	325 B	29...	
9c223cae33c5245a3009857582f1df1.png	304	png	vendor.8d9ec68...js:1	332 B	34...	
b683240befccbdcaa86da8f382d3a11ajpg?x-oss-process=image/resiz...	304	jpeg	vendor.8d9ec68...js:1	308 B	37...	
2d5e1c53fd7aa2a8f7663db0dae0ee12.jpg?x-oss-process=image/resi...	304	jpeg	vendor.8d9ec68...js:1	284 B	37...	
2aa01da86c8fb550b32c79f8ca4f67b0.jpg?x-oss-process=image/resiz...	304	jpeg	vendor.8d9ec68...js:1	292 B	36...	
ca39e9384b7793c4713dcc762da5f5c0.jpg?x-oss-process=image/resiz...	304	jpeg	vendor.8d9ec68...js:1	289 B	36...	

强缓存是利用**Expires**或者**Cache-Control**这两个HTTP Response Header实现的，它们都用来表示资源在客户端缓存的有效期。

Expires是一个绝对时间，而**Cache-Control**是一个相对时间，即一个过期时间大小，与协商缓存一样，基于**Expires**实现的强缓存也会因为时间问题导致缓存管理出现问题。我建议使用**Cache-Control**来实现强缓存。具体的实现流程如下：

- 当浏览器第一次请求访问服务器资源时，服务器会在返回这个资源的同时，在**Response**头部加上**Cache-Control**，**Cache-Control**中设置了过期时间大小；
- 浏览器再次请求访问服务器中的该资源时，会先通过请求资源的时间与**Cache-Control**中设置的过期时间大小，来计算出该资源是否过期，如果没有，则使用该缓存，否则请求服务器；
- 服务器再次收到请求后，会再次更新**Response**头部的**Cache-Control**。

2. 网关缓存

除了以上本地缓存，我们还可以在网关中设置缓存，也就是我们熟悉的**CDN**。

CDN缓存是通过不同地点的缓存节点缓存资源副本，当用户访问相应的资源时，会调用最近的**CDN**节点返回请求资源，这种方式常用于视频资源的缓存。

服务层缓存技术

前端缓存一般用于缓存一些不常修改的常量数据或一些资源文件，大部分接口请求的数据都缓存在了服务端，方便统一管理缓存数据。

服务端缓存的初衷是为了提升系统性能。例如，数据库由于并发查询压力过大，可以使用缓存减轻数据库压力；在后台管理中的一些报表计算类数据，每次请求都需要大量计算，消耗系统**CPU**资源，我们可以使用缓存来保存计算结果。

服务端的缓存也分为进程缓存和分布式缓存，在**Java**中进程缓存就是**JVM**实现的缓存，常见的有我们经常使用的容器类，**ArrayList**、**ConcurrentHashMap**等，分布式缓存则是基于**Redis**实现的缓存。

1. 进程缓存

对于进程缓存，虽然数据的存取会更加高效，但JVM的堆内存数量是有限的，且在分布式环境下很难同步各个服务间的缓存更新，所以我们一般缓存一些数据量不大、更新频率较低的数据。常见的实现方式如下：

```
//静态常量
public final static String url = "https://time.geekbang.org";

//list容器
public static List<String> cacheList = new Vector<String>();

//map容器
private static final Map<String, Object> cacheMap= new ConcurrentHashMap<String, Object>();
```

除了Java自带的容器可以实现进程缓存，我们还可以基于Google实现的一套内存缓存组件Guava Cache来实现。

Guava Cache适用于高并发的多线程缓存，它和ConcurrentHashMap一样，都是基于分段锁实现的并发缓存。

Guava Cache同时也实现了数据淘汰机制，当我们设置了缓存的最大值后，当存储的数据超过了最大值时，它就会使用LRU算法淘汰数据。我们可以通过以下代码了解下Guava Cache的实现：

```
public class GuavaCacheDemo {
    public static void main(String[] args) {
        Cache<String,String> cache = CacheBuilder.newBuilder()
            .maximumSize(2)
            .build();
        cache.put("key1","value1");
        cache.put("key2","value2");
        cache.put("key3","value3");
        System.out.println("第一个值: " + cache.getIfPresent("key1"));
        System.out.println("第二个值: " + cache.getIfPresent("key2"));
        System.out.println("第三个值: " + cache.getIfPresent("key3"));
    }
}
```

运行结果：

第一个值: `null`

第二个值: `value2`

第三个值: `value3`

那如果我们的数据量比较大，且数据更新频繁，又是在分布式部署的情况下，想要使用JVM堆内存作为缓存，这时我们又该如何去实现呢？

Ehcache是一个不错的选择，**Ehcache**经常在**Hibernate**中出现，主要用来缓存查询数据结果。

Ehcache是**Apache**开源的一套缓存管理类库，是基于JVM堆内存实现的缓存，同时具备多种缓存失效策略，支持磁盘持久化以及分布式缓存机制。

2. 分布式缓存

由于高并发对数据一致性的要求比较严格，我一般不建议使用**Ehcache**缓存有一致性要求的数据。对于分布式缓存，我们建议使用**Redis**来实现，**Redis**相当于一个内存数据库，由于是纯内存操作，又是基于单线程串行实现，查询性能极高，读速度超过了10W次/秒。

Redis除了高性能的特点之外，还支持不同类型的数据结构，常见的有string、list、set、hash等，还支持数据淘汰策略、数据持久化以及事务等。

两种缓存讲完了，接下来我们看看其中可能出现的问题。

数据库与缓存数据一致性问题

在查询缓存数据时，我们会先读取缓存，如果缓存中没有该数据，则会去数据库中查询，之后再放入到缓存中。

当我们的数据被缓存之后，一旦数据被修改（修改时也是删除缓存中的数据）或删除，我们就需要同时操作缓存和数据库。这时，就会存在一个数据不一致的问题。

例如，在并发情况下，当A操作使得数据发生删除变更，那么该操作会先删除缓存中的数据，之后再删除数据库中的数据，此时若是还没有删除成功，另外一个请求查询操作B进来了，发现缓存中已经没有了数据，则会去数据库中查询，此时发现有数据，B操作获取之后又将数据存放在了缓存中，随后数据库的数据又被删除了。此时就出现了数据不一致的情况。

那如果先删除数据库，再删除缓存呢？

我们可以试一试。在并发情况下，当A操作使得数据发生删除变更，那么该操作会先删除了数据库的操作，接下来删除缓存，失败了，那么缓存中的数据没有被删除，而数据库的数据已经被删除了，同样会存在数据不一致的问题。

所以，我们还是需要先做缓存删除操作，再去完成数据库操作。那我们又该如何避免高并发下，

数据更新删除操作所带来的数据不一致的问题呢？

通常的解决方案是，如果我们需要使用一个线程安全队列来缓存更新或删除的数据，当A操作变更数据时，会先删除一个缓存数据，此时通过线程安全的方式将缓存数据放入到队列中，并通过一个线程进行数据库的数据删除操作。

当有另一个查询请求B进来时，如果发现缓存中没有该值，则会先去队列中查看该数据是否正在被更新或删除，如果队列中有该数据，则阻塞等待，直到A操作数据库成功之后，唤醒该阻塞线程，再去数据库中查询该数据。

但其实这种实现也存在很多缺陷，例如，可能存在读请求被长时间阻塞，高并发时低吞吐量等问题。所以我们在考虑缓存时，如果数据更新比较频繁且对数据有一定的一致性要求，我通常不建议使用缓存。

缓存穿透、缓存击穿、缓存雪崩

对于分布式缓存实现大数据的存储，除了数据不一致的问题以外，还有缓存穿透、缓存击穿、缓存雪崩等问题，我们平时实现缓存代码时，应该充分、全面地考虑这些问题。

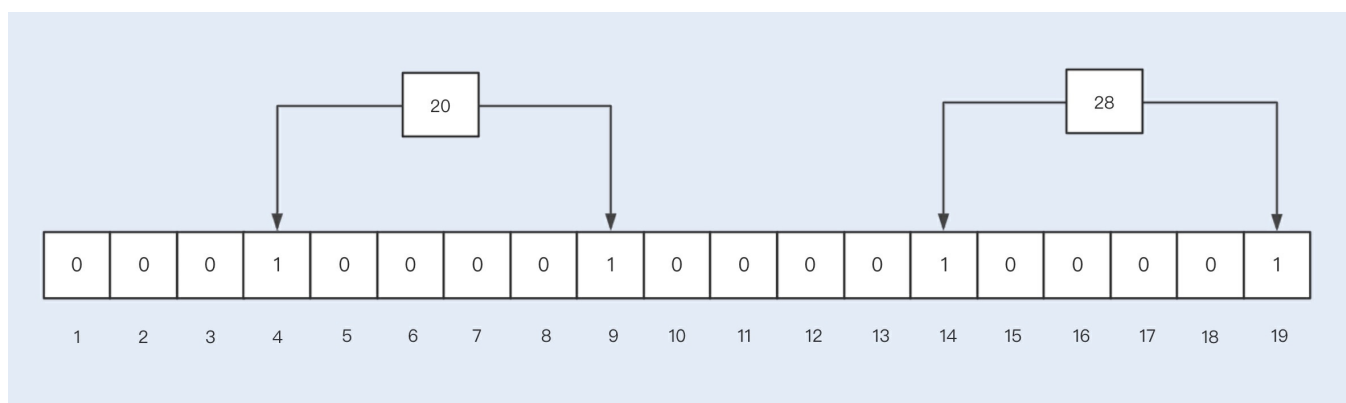
缓存穿透是指大量查询没有命中缓存，直接去到数据库中查询，如果查询量比较大，会导致数据库的查询流量大，对数据库造成压力。

通常有两种解决方案，一种是将第一次查询的空值缓存起来，同时设置一个比较短的过期时间。但这种解决方案存在一个安全漏洞，就是当黑客利用大量没有缓存的key攻击系统时，缓存的内存会被占满溢出。

另一种则是使用布隆过滤算法（BloomFilter），该算法可以用于检查一个元素是否存在，返回结果有两种：可能存在或一定不存在。这种情况很适合用来解决故意攻击系统的缓存穿透问题，在最初缓存数据时也将key值缓存在布隆过滤器的BitArray中，当有key值查询时，对于一定不存在的key值，我们可以直接返回空值，对于可能存在的key值，我们会去缓存中查询，如果没有值，再去数据库中查询。

BloomFilter的实现原理与Redis中的BitMap类似，首先初始化一个m长度的数组，并且每个bit初始化值都是0，当插入一个元素时，会使用n个hash函数来计算出n个不同的值，分别代表所在数组的位置，然后再将这些位置的值设置为1。

假设我们插入两个key值分别为20,28的元素，通过两次哈希函数取模后的值分别为4,9以及14,19，因此4,9以及14,19都被设置为1。



那为什么说**BloomFilter**返回的结果是可能存在和一定不存在呢？

假设我们查找一个元素**25**，通过n次哈希函数取模后的值为**1,9,14**。此时在**BitArray**中肯定是不存在的。而当我们查找一个元素**21**的时候，n次哈希函数取模后的值为**9,14**，此时会返回可能存在的结果，但实际上是不存在的。

BloomFilter不允许删除任何元素的，为什么？假设以上**20,25,28**三个元素都存在于**BitArray**中，取模的位置值分别为**4,9、1,9,14**以及**14,19**，如果我们要删除元素**25**，此时需要将**1,9,14**的位置都置回**0**，这样就影响**20,28**元素了。

因此，**BloomFilter**是不允许删除任何元素的，这样会导致已经删除的元素依然返回可能存在的结果，也会影响**BloomFilter**判断的准确率，解决的方法则是重建一个**BitArray**。

那什么缓存击穿呢？在高并发情况下，同时查询一个key时，key值由于某种原因突然失效（设置过期时间或缓存服务宕机），就会导致同一时间，这些请求都去查询数据库了。这种情况经常出现在查询热点数据的场景中。通常我们会在查询数据库时，使用排斥锁来实现有序地请求数据库，减少数据库的并发压力。

缓存雪崩则与缓存击穿差不多，区别就是失效缓存的规模。雪崩一般是指发生大规模的缓存失效情况，例如，缓存的过期时间同一时间过期了，缓存服务宕机了。对于大量缓存的过期时间同一时间过期的问题，我们可以采用分散过期时间来解决；而针对缓存服务宕机的情况，我们可以采用分布式集群来实现缓存服务。

总结

从前端到后端，对于一些不常变化的数据，我们都可以将其缓存起来，这样既可以提高查询效率，又可以降低请求后端的压力。对于前端来说，一些静态资源文件都是会被缓存在浏览器端，除了静态资源文件，我们还可以缓存一些常量数据，例如商品信息。

服务端的缓存，包括了**JVM**的堆内存作为缓存以及**Redis**实现的分布式缓存。如果是一些不常修改的数据，数据量小，且对缓存数据没有严格的一致性要求，我们就可以使用堆内存缓存数据，这样既实现简单，查询也非常高效。如果数据量比较大，且是经常被修改的数据，或对缓存数据有严格的一致性要求，我们就可以使用分布式缓存来存储。

在使用后端缓存时，我们应该注意数据库和缓存数据的修改导致的数据不一致问题，如果对缓存与数据库数据有非常严格的一致性要求，我就不建议使用缓存了。

同时，我们应该针对大量请求缓存的接口做好预防工作，防止查询缓存的接口出现缓存穿透、缓存击穿和缓存雪崩等问题。

思考题

在基于Redis实现的分布式缓存中，我们更新数据时，为什么建议直接将缓存中的数据删除，而不是更新缓存中的数据呢？

期待在留言区看到你的答案。也欢迎你点击“请朋友读”，把今天的内容分享给身边的朋友，邀请他一起讨论。



Java 性能调优实战

覆盖 80% 以上 Java 应用调优场景

刘超

金山软件西山居技术经理



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

精选留言



QQ怪

3

学到了很多，挺收益的，思考题：更新效率太低，代价很大，且不一定被访问的频率高，不高则没必要缓存，还不如直接删掉，而且还容易出现数据不一致问题

2019-08-29

作者回复

▮ 对的，两个并发写去更新还存在一致性的问题。不过，在删除缓存后，记得读取数据需要加锁或延时等待，防止读取脏数据。

2019-08-30



AiSmart4J

👍 2

还有就是更新缓存代价大。如果缓存里的数据不是把数据库里的数据直接存下来，而是需要经过某种复杂的运算，那么这种不必要的更新会带来更大的浪费。

2019-08-29

作者回复

对的，这也是一种情况

2019-08-30



AiSmart4J

👍 2

如果是更新数据库再操作缓存的话，此时更新缓存的操作不是必须的。可能缓存里的数据并没有被读到，就会被下一次更新MySQL操作带来Redis更新操作覆盖，那么本次更新操作就是无意义的。

2019-08-29



yungoo

👍 2

基于redis集中缓存更新数据采用删除而不是直接更新缓存的原因之一：避免二类更新丢失问题。

分布式系统中当存在并发数据更新时，因无法保证更新操作顺序的时间一致性，从而导致旧值覆盖新值。

如：

t1时间点，A进程发起更新key1为1的P1操作。

t1+x时间点，B进程发起更新key1为2的P2操作。

其中P1 -> P2，数据库中值为2。

而redis收到的指令，可能因网络抖动或者STW，实际为P2 -> P1，缓存的值为1，造成数据不一致。

2019-08-29

作者回复

存在并发更新时数据不一致问题

2019-08-30



godtrue

👍 1

我觉得看场景，我们是电商的核心系统，计算全部依赖缓存，我们的缓存是经过复杂计算的结构数据，每天定时任务刷新，更新是全部是先添加有效数据后删除无效数据。添加有效数据时，如果数据存在就是更新操作了啦！我觉得挺OK的

2019-09-12



Better me

👍 1

布隆过滤器为什么要经过n个hash函数散列，有什么特别的考虑吗

2019-08-29

| 作者回复

这是为了计算不同的位置，通过不同位置置1，得出一个数值。

2019-08-30



Maxwell

👍 1

老师您说的：通常我们会在查询数据库时，使用排斥锁来实现有序地请求数据库，减少数据库的并发压力。这个通常哪些方案？

2019-08-29

| 作者回复

最常用的就是使用同步锁或Lock锁实现。

2019-08-30



绿里奇迹

👍 0

“当黑客利用大量没有缓存的 key 攻击系统时，缓存的内存会被占满溢出”。但是即使使用bloom filter，这些不存在的key也会被判断为不存在，于是继续在缓存中创建该key，直到缓存被占满。请问使用bloom filter为什么能解决这类故意攻击的问题？

2019-10-20



星星

👍 0

老师，我有个问题，分布式更新或者删除缓存的时候，为啥不对这个操作加锁呢，例如a线程更新或者删除缓存，并更新数据库，然后解锁。此时b线程在争夺锁。并且持有锁。是不是性能问题，所以不这样，还是我理解错了？

2019-09-29

| 作者回复

没有理解错，锁和我文中提到的队列都是解决redis缓存数据一致性问题的方案，这种解决方案会带来一定的性能损耗

2019-09-30



Liam

👍 0

把数据库的数据全部加载到bitmap？

2019-09-05

| 作者回复

是的

2019-09-07



Liam

👍 0

bloom filter怎么初始化呢？刚开始bit array都是0吧，不可能直接拒绝掉呀？难道是把数据库的

2019-09-05

| 作者回复

初始化时将数据加载到bit array中

2019-09-07



Maxwell

👍 0

老师高并发时会不会卡？影响吞吐量，涉及到要更改公用数据如消费积分总量、每天的消费积

分额度，现在没加锁，靠数据库的事务更新，担心以后qps上来了数据库扛不住

2019-08-30

作者回复

使用队列来异步更新数据，没有压力

2019-09-07



victoriest

0

只看 模块七 值回票价

2019-08-30



疯狂咸鱼

0

浏览器缓存就是常说的http缓存么？

2019-08-29

作者回复

对的

2019-08-30



许童童

0

直接更新缓存中的数据，因为请求到达的顺序无法保证，有可能后请求的数据覆盖前请求的数据。直接将数据删除，就是一种幂等的操作，删除后，再去数据库拉数据，就不会有覆写的问题。

2019-08-29

作者回复

对的，如果两个并发写去更新还存在一致性的问题，还不如直接删除，等下次读取的时候再次写入缓存中。不过，在删除缓存后，记得读取数据需要加锁或延时等待，防止读取脏数据。

2019-08-30



Loubobooo

0

课后题：原因很简单

1. 很多时候，在复杂点的缓存场景，缓存不单单是数据库中直接取出来的值。比如可能更新了某个表的一个字段，然后其对应的缓存，是需要查询另外两个表的数据并进行运算，才能计算出缓存最新的值的。

另外更新缓存的代价有时候是很高的。每次修改数据库的时候，都一定要将其对应的缓存更新一份，这样做的代价较高。如果遇到复杂的缓存数据计算的场景，缓存频繁更新，但这个缓存到底会不会被频繁访问到？如果没有，这个缓存的效率就很低了

2019-08-29

作者回复

回答很全面

2019-08-30



撒旦的堕落

0

老师说 缓存数据库一致性问题时 当一个线程缓存删除 而数据库中没有来得及删除时 另一个线程来请求数据 发现缓存中数据不存在去队列中判断 如果数据在更新中 则等待 然后唤醒 不过如果是不同进程中的线程呢 怎么唤醒？感觉这种方式要维护的数据更多了 要把删除的缓存取出来

放到队列中 然后更新完成后 还要删除队列中的数据 为了应对分布式的情况 还要使用的是分布式队列 效率变低了 有木有更好的办法呢

2019-08-29

作者回复

暂时没有想到更好的

2019-08-30



-W.LI-

0

老师真棒，全能。

CDN的缓存策略是copy服务端的，协商缓存和强缓存？如果有些静态资源，服务端开发没做缓存策略，CDN还会缓存么？实际开发中用过一次CDN。是在资源路径前，拼接一段CDN路径。具体不知

课后习题，如果并发操作时，虽然redis是单线程的但是没法保证网络延时下，先更新数据库。也先更新缓存。个人感觉失效一个key比写一个key开销小。网络传输上看，update还得传一个value的值，redis更新还得写缓存感觉也是失效慢。并发情况写两次(除开正确性)有一次的写完全浪费。

2019-08-29

作者回复

通常我们是会指定一些静态资源文件上传到CDN上去，并且通过版本号来更新。例如，我们的js资源文件是xxx001.js，如果我们更新了该资源文件，则将xxx002.js推送到CDN上，同时前端的访问路径也更新访问资源路径。

2019-08-30