缓存与数据库的双写一致性

一、前言

只要用到缓存，就有可能涉及到缓存与数据库双存储双写，你只要是双写，就一定会有数据一致性的问题。

So，如何解决一致性问题呢？

一般来说，如果允许缓存可以稍微的跟数据库偶尔有不一致的情况，也就是说如果你的系统不是严格要求，缓存和数据数据一致性的话，最好不要做这个方案。即：读请求和写请求串行化，串到一个内存队列里去。

串行化可以保证一定不会出现一致性的问题，但是它也会导致系统的吞吐量大幅度降低，用比正常情况多几倍的机器去支撑线上请求。

先介绍几个专属名词：

QPS：每秒查询率，对一个特定的查询服务器在规定时间内所处理流量多少的衡量标准。每天80%的访问集中在20%的时间中，这个20%时间叫峰值。

TPS：系统单位时间内处理请求的数量。对于无并发的应用系统来说，吞吐量与响应时间成严格反比关系，实际上此时吞吐量就是响应时间的倒数。

二、Cache Aside Pattern

这是最经典的缓存+数据库读写模式。该模式的原理很简单：

1. 在读的时候，先读缓存，缓存没有的情况下，就去读数据库，然后取出数据后放入缓存，同时返回响应。
2. 更新的时候，先更新数据库，然后再删除缓存。

为什么这种模式是删除缓存，而不是更新缓存呢？原因如下：

1. 在复杂的缓存场景中，缓存中的数据不单单是数据库中直接取出来的值，比如可能更新了一张表的一条数据中的某个字段，然后其对应的缓存数据，是需要查询另外两个表的数据进行计算出来的。
2. 更新缓存的代价有时候会更高。
3. 删除缓存不是不更新缓存，其实是一种lazy计算的思想，不要每次都重新做复杂的计算，不管它会不会被用到，而是让它到需要被用到的时候再重新做计算。

三、初级的缓存不一致问题及解决方案

Q: 先修改数据库，在删除缓存。如果删除缓存失败了，那么数据库中的数据是最新的，缓存中的数据是旧数据，就导致了不一致性问题。

A: 那么我们就先删除缓存，再修改数据库。如果数据库修改失败了，那么数据库中的是旧数据，缓存中是空的，那么数据也不会出现不一致的情况。因为在读取的时候，缓存中没有，则读取数据中旧数据，然后更新到缓存中。

四、复杂的缓存不一致问题及解决方案

先看个场景：数据发生了变更，先删除了缓存，然后要去修改数据库，还没来得及修改，一个新的请求过来，去读取缓存，发现缓存是空的，去查询数据库，查到了修改前的旧数据，放到了缓存中。随后数据变更的任务完成了数据库的数据修改。OMG，数据库和缓存中的数据就不一样了。

有人会说有这么巧的事儿发生吗，什么时候才会发生上面这么巧的事情呢？

只有在一个数据并发的进行读写操作的时候，才可能会出现这种问题。其实如果说你的并发量很低的话，特别是读并发很少的情况下，每天访问量也就1w，那么很少的情况下，会出现上面描述的不一致场景。But，如果每天的是上亿的流量，每秒并发读是几万次的话，每秒只要有数据更新的请求，就可能会出现上述的数据库+缓存不一致的情况发生。

问题场景描述清楚了，那么我们来看下怎么来解决吧~

1. 更新数据的时候，根据数据的唯一标识进行路由，将操作路由到一个jvm内存队列中。
2. 读取数据的时候，如果发现数据不在缓存中，那么将重新读取数据库+更新缓存的操作，根据唯一标识进行路由，将该操作路由到一个jvm内存队列中。
3. 一个jvm内存队列对应一个工作线程，每个工作线程串行拿到对应的操作，然后一条一条的执行。

此时，一个数据变更的操作开始进行，先删除缓存，然后再去更新数据库，但是还没完成更新。此时如果一个读请求过来，读到了空的缓存，那么可以先将更新缓存的请求发送到jvm内部队列中，此时会在队列中积压，然后同步等待缓存更新完成。

这样就保证了这么复杂的情况下，数据库+缓存的一致性问题。

当然，上面的解决方案中有一个小优化点，一个队列中，如果存在多个更新缓存的操作串在一起是没有意义的，因此可以做一些过滤。如果发现jvm内存队列中已经有了更新缓存的请求，那么久不要再放一个更新请求操作进去了。

如果读请求还在等待时间范围内，不断轮询，发现可以取到值了，那么就直接返回；如果请求等待时间超过了一定时长，那么这一次直接从数据中读取当前的就值。

五、高并发场景下，该解决方案中要注意的问题：

1. 读请求长时间阻塞

   由于读请求进行了轻度的异步化，所以一定要注意读超时的问题，每个读请求必须在超时时间范围内返回值，无论新值还是旧值。

   该解决方案中一个最大的风险点，在于可能数据更新很频繁，导致队列中积压了大量更新操作在里面，然后读请求会发生大量的超时，最后导致大量的请求直接走数据库。务必上线前，要通过一些模拟真实场景的测试，看看更新数据的频率是怎样的。另外一点，因为一个队列中，可能会积压针对多个数据项的更新操作，因此需要根据自己的业务情况进行测试，可能需要部署多个服务，每个服务分摊一些数据的更新操作。

   我们举个栗子，算一算：

   如果一个内存队列里积压100个商品的库存修改操作，每个库存修改操作要耗费10ms去完成，那么最后一个商品的读请求，可能需要等待10*100 = 1000ms = 1s后才能得到数据，这个时候就导致读请求的长时间阻塞。因此，一定要根据实际的业务系统运行状况去做一些压力测试和模拟线上环境，去看看最繁忙的时候，内存队列可能会积压多少更新操作，可能对最后的一个读请求会hang多少时间。

   如果真如上面所说的，内存队列中积压过多的更新操作，那么你就要加应用服务的机器了。

2. 读请求并发量过高

   确保恰巧碰上上述情况，还有一个风险，就是突然间大量读请求会在几十毫秒的延时hang在服务上，看服务能不能抗住，需要多少机器才能抗住最大的极限峰值。由于并不是所有的数据都在同一时间更新，缓存也不会同一时间失效，所以每次可能也就少数数据的缓存失效了，然后那些数据对应的读请求过来，并发量应该也不会特别大。

3. 多服务实例部署的请求路由

可能这个服务部署了多个实例，那么必须保证，执行数据更新操作，已经执行缓存更新操作的请求，都通过NGINX服务器路由到相同的服务实例上。比如，对同一个商品的读写请求，全部路由到同一台机器上。可以自己去做服务间，按照某个请求参数的hash路由，也可以用NGINX的hash路由功能等。

4. 热点商品的路由问题，导致请求倾斜

   万一某个商品的读写请求特别高的时候，全部打到相同的应用服务器上的相同的队列上，可能会造成某台机器上的压力过大。

   因为只有在商品数据更新的时候才会清空缓存，然后才会导致读写并发，所以要根据业务系统去看，如果更新频率不是太高的话，这个问题的影响并不是特别大，但是可能会造成某些机器的负载会高一些。

   其实后面可以通过hash一致性来解决这类哈希散列不均与的情况。