16 | 缓存策略：如何使用缓存来减少磁盘IO？

2019-08-29 李玥

你好，我是李玥。这节课，我们一起来聊一聊缓存策略。

现代的消息队列，都使用磁盘文件来存储消息。因为磁盘是一个持久化的存储，即使服务器掉电也不会丢失数据。绝大多数用于生产系统的服务器，都会使用多块儿磁盘组成磁盘阵列，这样不仅服务器掉电不会丢失数据，即使其中的一块儿磁盘发生故障，也可以把数据从其他磁盘中恢复出来。

使用磁盘的另外一个原因是，磁盘很便宜，这样我们就可以用比较低的成本，来存储海量的消息。所以，不仅仅是消息队列，几乎所有的存储系统的数据，都需要保存到磁盘上。

但是，磁盘它有一个致命的问题，就是读写速度很慢。它有多慢呢？一般来说SSD（固态硬盘）每秒钟可以读写几千次，如果说我们的程序在处理业务请求的时候直接来读写磁盘，假设处理每次请求需要读写3～5次，即使每次请求的数据量不大，你的程序最多每秒也就能处理1000次左右的请求。

而内存的随机读写速度是磁盘的10万倍！所以，**使用内存作为缓存来加速应用程序的访问速度，是几乎所有高性能系统都会采用的方法。**

缓存的思想很简单，就是把低速存储的数据，复制一份副本放到高速的存储中，用来加速数据的访问。缓存使用起来也非常简单，很多同学在做一些业务系统的时候，在一些执行比较慢的方法上加上一个@Cacheable的注解，就可以使用缓存来提升它的访问性能了。

但是，你是否考虑过，采用@Cacheable注解的方式缓存的命中率如何？或者说怎样才能提高缓存的命中率？缓存是否总能返回最新的数据？如果缓存返回了过期的数据该怎么办？接下来，我们一起来通过学习设计、使用缓存的最佳实践，找到这些问题的答案。

选择只读缓存还是读写缓存？

使用缓存，首先你就会面临选择读缓存还是读写缓存的问题。他们唯一的区别就是，在更新数据的时候，是否经过缓存。

我们之前的课中讲到Kafka使用的PageCache，它就是一个非常典型的读写缓存。操作系统会利用系统空闲的物理内存来给文件读写做缓存，这个缓存叫做PageCache。应用程序在写文件的时候，操作系统会先把数据写入到PageCache中，数据在成功写到PageCache之后，对于用户代码来说，写入就结束了。

然后，操作系统再异步地把数据更新到磁盘的文件中。应用程序在读文件的时候，操作系统也是先尝试从PageCache中寻找数据，如果找到就直接返回数据，找不到会触发一个缺页中断，然后操作系统把数据从文件读取到PageCache中，再返回给应用程序。

我们可以看到，在数据写到PageCache中后，它并不是同时就写到磁盘上了，这中间是有一个延迟的。操作系统可以保证，即使是应用程序意外退出了，操作系统也会把这部分数据同步到磁盘上。但是，如果服务器突然掉电了，这部分数据就丢失了。

你需要知道，**读写缓存的这种设计，它天然就是不可靠的，是一种牺牲数据一致性换取性能的设计。**当然，应用程序可以调用sync等系统调用，强制操作系统立即把缓存数据同步到磁盘文件中去，但是这个同步的过程是很慢的，也就失去了缓存的意义。

另外，写缓存的实现是非常复杂的。应用程序不停地更新PageCache中的数据，操作系统需要记录哪些数据有变化，同时还要在另外一个线程中，把缓存中变化的数据更新到磁盘文件中。在提供并发读写的同时来异步更新数据，这个过程中要保证数据的一致性，并且有非常好的性能，实现这些真不是一件容易的事儿。

所以说，一般情况下，不推荐你来使用读写缓存。

那为什么Kafka可以使用PageCache来提升它的性能呢？这是由消息队列的一些特点决定的。

首先，消息队列它的读写比例大致是1：1，因为，大部分我们用消息队列都是一收一发这样使用。这种读写比例，只读缓存既无法给写加速，读的加速效果也有限，并不能提升多少性能。

另外，Kafka它并不是只靠磁盘来保证数据的可靠性，它更依赖的是，在不同节点上的多副本来解决数据可靠性问题，这样即使某个服务器掉电丢失一部分文件内容，它也可以从其他节点上找到正确的数据，不会丢消息。

而且，PageCache这个读写缓存是操作系统实现的，Kafka只要按照正确的姿势来使用就好了，不涉及到实现复杂度的问题。所以，Kafka其实在设计上，充分利用了PageCache这种读写缓存的优势，并且规避了PageCache的一些劣势，达到了一个非常好的效果。

和Kafka一样，大部分其他的消息队列，同样也会采用读写缓存来加速消息写入的过程，只是实现的方式都不一样。

不同于消息队列，我们开发的大部分业务类应用程序，读写比都是严重不均衡的，一般读的数据的频次会都会远高于写数据的频次。从经验值来看，读次数一般都是写次数的几倍到几十倍。这种情况下，使用只读缓存来加速系统才是非常明智的选择。

接下来，我们一起来看一下，在构建一个只读缓存时，应该侧重考虑哪些问题。

保持缓存数据新鲜

对于只读缓存来说，缓存中的数据来源只有一个途径，就是从磁盘上来。当数据需要更新的时候，磁盘中的数据和缓存中的副本都需要进行更新。我们知道，在分布式系统中，除非是使用事务或者一些分布式一致性算法来保证数据一致性，否则，由于节点宕机、网络传输故障等情况的存在，我们是无法保证缓存中的数据和磁盘中的数据是完全一致的。

如果出现数据不一致的情况，数据一定是以磁盘上的那份拷贝为准。我们需要解决的问题就是，尽量让缓存中的数据与磁盘上的数据保持同步。

那选择什么时候来更新缓存中的数据呢？比较自然的想法是，我在更新磁盘中数据的同时，更新一下缓存中的数据不就可以了？这个想法是没有任何问题的，缓存中的数据会一直保持最新。但是，在并发的环境中，实现起来还是不太容易的。

你是选择同步还是异步来更新缓存呢？如果是同步更新，更新磁盘成功了，但是更新缓存失败了，你是不是要反复重试来保证更新成功？如果多次重试都失败，那这次更新是算成功还是失败呢？如果是异步更新缓存，怎么保证更新的时序？

比如，我先把一个文件中的某个数据设置成0，然后又设为1，这个时候文件中的数据肯定是1，但是缓存中的数据可不一定就是1了。因为把缓存中的数据更新为0，和更新为1是两个并发的异步操作，不一定谁会先执行。

这些问题都会导致缓存的数据和磁盘中的数据不一致，而且，在下次更新这条数据之前，这个不一致的问题它是一直存在的。当然，这些问题也不是不能解决的，比如，你可以使用分布式事务来解决，只是付出的性能、实现复杂度等代价比较大。

另外一种比较简单的方法就是，定时将磁盘上的数据同步到缓存中。一般的情况下，每次同步时直接全量更新就可以了，因为是在异步的线程中更新数据，同步的速度即使慢一些也不是什么大问题。如果缓存的数据太大，更新速度慢到无法接受，也可以选择增量更新，每次只更新从上次

缓存同步至今这段时间内变化的数据，代价是实现起来会稍微有些复杂。

如果说，某次同步过程中发生了错误，等到下一个同步周期也会自动把数据纠正过来。这种定时同步缓存的方法，缺点是缓存更新不那么及时，优点是实现起来非常简单，鲁棒性非常好。

还有一种更简单的方法，我们从来不去更新缓存中的数据，而是给缓存中的每条数据设置一个比较短的过期时间，数据过期以后即使它还存在缓存中，我们也认为它不再有效，需要从磁盘上再次加载这条数据，这样就变相地实现了数据更新。

很多情况下，缓存的数据更新不那么及时，我们的系统也是能够接受的。比如说，你刚刚发了一封邮件，收件人过了一会儿才收到。或者说，你改了自己的微信头像，在一段时间内，你的好友看到的你还是旧的头像，这些都是可以接受的。这种对数据一致性没有那么敏感的场景下，你一定要选择后面两种方法。

而像交易类的系统，它对数据的一致性非常敏感。比如，你给别人转了一笔钱，别人查询自己余额却没有变化，这种情况肯定是无法接受的。对于这样的系统，一般来说，都不使用缓存或者使用我们提到的第一种方法，在更新数据的时候同时来更新缓存。

缓存置换策略

在使用缓存的过程中，除了要考虑数据一致性的问题，你还需要关注的另一个重要的问题是，在内存有限的情况下，要优先缓存哪些数据，让缓存的命中率最高。

当应用程序要访问某些数据的时候，如果这些数据在缓存中，那直接访问缓存中的数据就可以了，这次访问的速度是很快的，这种情况我们称为一次缓存命中；如果这些数据不在缓存中，那只能去磁盘中访问数据，就会比较慢。这种情况我们称为“缓存穿透”。显然，缓存的命中率越高，应用程序的总体性能就越好。

那用什么样的策略来选择缓存的数据，能使得缓存的命中率尽量高一些呢？

如果你的系统是那种可以预测未来访问哪些数据的系统，比如说，有的系统它会定期做数据同步，每次同步的数据范围都是一样的，像这样的系统，缓存策略很简单，就是你要访问什么数据，就缓存什么数据，甚至可以做到百分之百的命中。

但是，大部分系统，它并没有办法准确地预测未来会有哪些数据会被访问到，所以只能使用一些策略来尽可能地提高缓存命中率。

一般来说，我们都会在数据首次被访问的时候，顺便把这条数据放到缓存中。随着访问的数据越来越多，总有把缓存占满的时刻，这个时候就需要把缓存中的一些数据删除掉，以便存放新的数据，这个过程称为缓存置换。

到这里，问题就变成了：当缓存满了的时候，删除哪些数据，才能会使缓存的命中率更高一些，也就是采用什么置换策略的问题。

命中率最高的置换策略，一定是根据你的业务逻辑，定制化的策略。比如，你如果知道某些数据已经删除了，永远不会再被访问到，那优先置换这些数据肯定是没问题的。再比如，你的系统是一个有会话的系统，你知道现在哪些用户是在线的，哪些用户已经离线，那优先置换那些已经离线用户的数据，尽量保留在线用户的数据也是一个非常好的策略。

另外一个选择，就是使用通用的置换算法。一个最经典也是最实用的算法就是LRU算法，也叫最近最少使用算法。这个算法它的思想是，最近刚刚被访问的数据，它在将来被访问的可能性也很大，而很久都没被访问过的数据，未来再被访问的几率也不大。

基于这个思想，**LRU的算法原理非常简单，它总是把最长时间未被访问的数据置换出去。**你别看这个LRU算法这么简单，它的效果是非常非常好的。

Kafka使用的PageCache，是由Linux内核实现的，它的置换算法的就是一种LRU的变种算法：LRU 2Q。我在设计JMQ的缓存策略时，也是采用一种改进的LRU算法。LRU淘汰最近最少使用的页，JMQ根据消息这种流数据存储的特点，在淘汰时增加了一个考量维度：页面位置与尾部的距离。因为越是靠近尾部的数据，被访问的概率越大。

这样综合考虑下的淘汰算法，不仅命中率更高，还能有效地避免“挖坟”问题：例如某个客户端正在从很旧的位置开始向后读取一批历史数据，内存中的缓存很快都会被替换成这些历史数据，相当于大部分缓存资源都被消耗掉了，这样会导致其他客户端的访问命中率下降。加入位置权重后，比较旧的页面会很快被淘汰掉，减少“挖坟”对系统的影响。

小结

这节课我们主要聊了一下，如何使用缓存来加速你的系统，减少磁盘IO。按照读写性质，可以分为读写缓存和只读缓存，读写缓存实现起来非常复杂，并且只在消息队列等少数情况下适用。只读缓存适用的范围更广，实现起来也更简单。

在实现只读缓存的时候，你需要考虑的第一个问题是如何来更新缓存。这里面有三种方法，第一种是在更新数据的同时去更新缓存，第二种是定期来更新全部缓存，第三种是给缓存中的每个数据设置一个有效期，让它自然过期以达到更新的目的。这三种方法在更新的及时性上和实现的复杂度这两方面，都是依次递减的，你可以按需选择。

对于缓存的置换策略，最优的策略一定是你根据业务来设计的定制化的置换策略，当然你也可以考虑LRU这样通用的缓存置换算法。

思考题

课后来写点儿代码吧，实现一个采用LRU置换算法的缓存。

```
/**
 * KV存储抽象
 */
public interface Storage<K,V> {
    /**
     * 根据提供的key来访问数据
     * @param key 数据Key
     * @return 数据值
     */
    V get(K key);
}

/**
 * LRU缓存。你需要继承这个抽象类来实现LRU缓存。
 * @param <K> 数据Key
 * @param <V> 数据值
 */
public abstract class LruCache<K, V> implements Storage<K,V>{
    // 缓存容量
    protected final int capacity;
    // 低速存储，所有的数据都可以从这里读到
    protected final Storage<K,V> lowSpeedStorage;

    public LruCache(int capacity, Storage<K,V> lowSpeedStorage) {
        this.capacity = capacity;
        this.lowSpeedStorage = lowSpeedStorage;
    }
}
```

你需要继承LruCache这个抽象类，实现你自己的LRU缓存。lowSpeedStorage是提供给你可用的低速存储，你不需要实现它。

欢迎你把代码上传到GitHub上，然后在评论区给出访问链接。大家来比一下，谁的算法性能更好。如果你有任何问题，也可以在评论区留言与我交流。

感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给你的朋友。

精选留言

leslie 4

一路跟着三位老师在同时学习相关知识：我想我大致明白了消息队列的特性以及真正适用的场景，周二许老师的课程中刚好提到了相关东西特意为了许老师。

关系型数据库、非关系型数据库、消息队列其实都属于存储中间件，老师今天的话题如何利用缓存来减少磁盘IO其实就是涉及了三种存储中间件；其实老师今天的三种方法我的理解其实就是对应了三种存储中间件：三种存储中间件都和内存有关，如何合理使用缓存、内存、磁盘去做合适的事情才是关键；

老师课程的算法：几门相关的课程同时在学时间还是偏紧，不过刘老师的操作系统、许老师的架构、以及老师的消息队列同时学习，倒是了解什么场景下用以及用哪种消息队列；至少跟着老师的消息队列我应当知道业务中的什么事情让它去承担并且使用什么策略，先用现成的吧；改进只能等第二遍学习时再去做了。

一直跟着学习学到现在发现学好消息队列的前提：1)对业务的数据情况清楚2)对于当下使用的存储中间件情况非常清楚3)对于消息队列所在的系统熟悉：其中就涉及需要非常了解操作系统和当前系统的架构最后就是充分利用当下的消息队列追加一些适合自己业务场景的算法：调整优化当下所用的消息队列。

2019-08-29

樱花落花 3

LRU算法最经典的我觉得还是MySQL的bufferpool的设计，里面按比例分为young和old区，能很好的解决预读问题和老师讲的“挖坟”问题；

2019-09-02

岁月安然

3

”□位置与尾部的距离。因为越是靠近尾部的数据，被访问的概率越□“
不大理解这句话，尾部指的是啥？当某个客户端正在从很旧的位置开始向后读取□批历史数据，是怎么判断与尾部的距离，从而减少这部分的缓存。

2019-08-29

作者回复

与尾部的距离=最后一个一条消息的尾部位置 - 页面的位置

这个值越小，说明请求的数据与尾部越近，置换的时候被留下的概率也就越大。

对于历史数据，由于距离远，这个值会很大，那这些页面在置换的时候被留下的概率就很小，所以很快就会被从内从中置换出去。

2019-08-30

姜戈

2

利用双向链表+哈希表, 支持所有操作时间复杂度都为O(1). https://github.com/djangogao/mqexercise.git
实现了最基础LRU算法， 关于LRU的改进算法：LRU-K和 LRU 2Q，可参考此文章https://www.jianshu.com/p/c4e4d55706ff

2019-08-30

付永强

1

可以通过在进行更新数据库操作时，删除缓存，读取数据库时如果有缓存就直接读，没有缓存则从数据库读取并更新缓存，这样的设计可以确保缓存幂等。

2019-09-03

ponymm

1

“找不到会触发一个缺页中断，然后操作系统把数据从文件读取到 PageCache 中，再返回给应用程序” 这里pagecache中没有数据并不会产生缺页中断，而是alloc page,然后放入lru链表中，接着调用a_ops->readpage()读取数据到page，可以参考kernel的 do_generic_mapping_read 函数

2019-08-29

泛岁月的涟漪

0

```
public class DefaultLruCache<K, V> extends LruCache<K, V> {

private LruCacheImpl<K, V> cache;

public DefaultLruCache(int capacity, Storage<K, V> lowSpeedStorage) {
super(capacity, lowSpeedStorage);
cache = new LruCacheImpl<>(capacity, true);
}
```

```
@Override
public V get(K key) {
V value = cache.get(key);
if (value == null) {
value = lowSpeedStorage.get(key);
cache.put(key, value);
}
return value;
}

class LruCacheImpl<K, V> extends LinkedHashMap<K, V> {

public LruCacheImpl(int capacity, boolean accessOrder) {
super(capacity, 0.75F, accessOrder);
}

@Override
protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
return size() > capacity;
}
}
}
```

2019-09-09

明日 0

Java实现: https://gist.github.com/imgaoxin/ed59397c895b5a8a9572408b98542015

2019-09-03

作者回复

2019-09-04

木木木 0

缓存和db的一致性有很多工作做的。文中三种方法，定时更新全部缓存估计只适合量小的，更新不频繁的配置性数据。定时的时间也有问题，太短影响性能，太长一致性问题。设置缓存时间也是类似的问题。我们业务中实际使用同步更新的方式，当然异步也可以。关键是在业务代码落库时落缓存补偿的记录，里面带序号，用来解决异步问题。更新后还可以进行一致性检查

2019-09-03

平常心 0

期待JMQ开源

2019-09-02

约书亚 0

来晚了，根据jms和kafka这两节，我试着猜想一下jms的机制，顺便提个问题：
1. 老师这节提到的jms实现的缓存机制，都是基于direct buffer自己实现的一个内存池，并实现了变种的LRU对么？这个缓存就是前面提到的journal cache，被writeThread/RelicationThread/FlushThread使用？
2. 这个内存池看起来并没有借助于netty的direict buffer pool是吧？
3. 那原谅我对比一下jms和rocket，jms没有基于mmap去做而选择direct buffer，看起来是为了：
a. 减少GC的压力
b. 比mmap更容易控制，就更容增加缓存的命中率
这样？
4. 另外，有个概念我很模糊，有资料说direct buffer在写磁盘/socket时并不能真的节省一次cpu copy？那这样的话jms可以说并没有利用zero copy？

望解惑

2019-09-01

作者回复

A1：是的。

A2：是的。

对于问题3和4，你可以看一下关于JMQ的这篇文章：

https://www.jiqizhixin.com/articles/2019-01-21-19

2019-09-03

 0

有一个问题：java 内部实现缓存很容易导致fullgc频繁发生

2019-08-30

大男孩 0

献丑了，实现了一个 LruCache，核心思路是用 Map 来做本地高速缓存，用 LinkedList 存储最近被访问的 Key 列表，代码如下：https://github.com/jeffreylyp/geektime-cache.git，核心优化点是对 Key 列表的更新操作全部异步化了，由单独的线程负责处理。

2019-08-30

A9 0

将key-value维护成一个双向链表，head为新鲜的数据，tail处为老一些的数据
高速缓存使用HashMap或其他查询实现
在容量满时，一次删除 size/10+1个元素，避免重复触发删除操作
https://gist.github.com/WangYangA9/90b3b05140bfef526eb7cb595b62c71a

2019-08-30

humor 0

/**
* 不知道这样实现可不可以，肯定有问题哈哈
* @param <K>

```
* @param <V>
*/
public class LruCacheImpl<K,V> extends LruCache<K,V> {
/**
* 高速缓存
*/
private Map<K,V> highSpeedStorage = new HashMap<>(capacity);
/**
* lru链表
*/
private LinkedList<K> lruList = new LinkedList<>();

public LruCacheImpl(int capacity, Storage lowSpeedStorage) {
super(capacity, lowSpeedStorage);
}

@Override
public V get(K key) {
//先从高速缓存中获取
V value = highSpeedStorage.get(key);
if (value != null) {
//调整元素位置
K lastKey = lruList.peekLast();
if (!key.equals(lastKey)) {
lruList.remove(key);
lruList.addLast(key);
}
return value;
}
//加锁访问低速缓存
synchronized (this) {
value = highSpeedStorage.get(key);
if (value != null) {
return value;
}
//再从低速缓存中获取
value = lowSpeedStorage.get(key);
if (value == null) {
return null;
}
//写入高速缓存
if (highSpeedStorage.size()>=capacity) {
K removeKey = lruList.removeFirst();
```

```
highSpeedStorage.remove(removeKey);
}
highSpeedStorage.put(key, value);
lruList.addLast(key);
}
return value;
}
}
```

2019-08-30

humor 0

我感觉读写缓存和只读缓存的第一种更新策略(更新数据的同时更新缓存)是一样的吧？因为它们都需要同时更新数据和缓存，区别可能是读写缓存以更新缓存为主，只读缓存的第一种更新策略是以更新数据为主吗

2019-08-30

作者回复

还有就是，读写缓存可以为写入加速，但牺牲了数据可靠性。

2019-08-31

微微一笑 0

老师好，希望您有时间能回答下我上一章节遗留的问题，感谢@老杨同志@冰激凌的眼泪@linqw

小伙伴的留言解答~~~

问题是：

①rocketMq有consumeQueue，存储着offset，然后通过offset去commitlog找到对应的Message。通过看rocketmq的开发文档，通过offset去查询消息属于【随机读】，offset不是存储着消息在磁盘中的位置吗？为什么属于随机读呢？

②rocketMq的某个topic下指定的消息队列数，指的是consumeQueue的数量吗？

③性能上，顺序读优于随机读。rocketMq的实现上，在消费者与commitlog之间设计了consumeQueue的数据结构，导致不能顺序读，只能随机读。我的疑惑是，rocketMq为什么不像kafka那样设计，通过顺序读取消息，然后再根据topic、tag平均分配给不同的消费者实例,，这样消息积压的时候，直接增加消费者实例就可以了，不需要增加consumeQueue，这样也可以去除consumeQueue的存在呀？我在想consumeQueue存在的意义是什么呢？

哈哈，我的理解可能有些问题，希望老师指点迷津~

2019-08-29

作者回复

我已经回复了，你可以去上一节看一下，如果还有问题可以继续留言提问。

2019-08-30

许童童 0

Leetcode上面有一道LRU算法的题目，大家可以去做一做。

2019-08-29

coffee 0

依mybatis的lrucache画了个瓢。采用linkedhashmap来实现的lru。插入应该是近似o(1).查找最好是o(1) ,最坏是o(n) .代码见https://github.com/swgithub1006/mqlearning

2019-08-29

安静的boy

0

课后习题做了一下，大概的实现思路是用链表存储数据，如果数据查询的数据不存在，从低速存储中拿到数据存到链表头部（要判断数据是否已满），如果数据存在，则从链表中取出数据，然后再放到链表头部。github地址：https://github.com/wutianqi/geek-pratice.git

2019-08-29