1. **背景**

使用guava CACHE缓存命中率逐渐下降。 Why?

1. **guava原理**

参见[《cache-1-guava》](cache-1-guava.docx)

1. **常见缓存算法与实践**
2. **LRU**

* 优点
  + 结构简单
  + 占用内存少
  + 善于处理稀疏请求（Sparse burst）
* 缺点
  + 由于没有统计概念，可能会将一些高频数据错误抛弃

1. **LFU**

* 优点
  + 可以统计高频数据
* 缺点
  + 占用内存多
  + 高频数据难以过期或移除
  + 无法处理稀疏请求（Sparse burst），因为可能始终无法累计到一定数量存入缓存中

1. **Redis中LRU近似算法**
2. **使用场景**

redis过期键有惰性、定期的删除策略，也有maxmemory设置，具体参见[2]。其中maxmemory设置之后，client在执行耗费内存命令redisCommand（flags会标识是否为耗费内存操作）会首先判断是否达到maxmemory，如果达到则需要清理过期键或释放内存。maxmemory释放内存的方式主要有三种[3]：

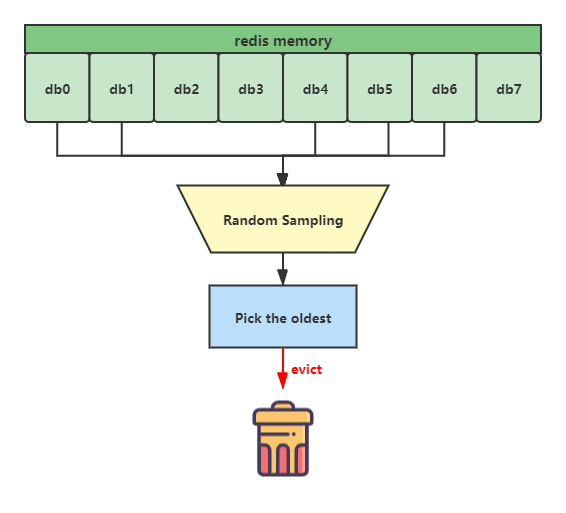
* lru：volatile代表设置了过期时间，allKeys则是所有key
* random：volatile代表设置了过期时间，allKeys则是所有key
* ttl：即将过期

1. **优化原因**

这里的lru其实是近似lru，redis并没有在内存中按照双端链表的形式构建lru列表，而只有原有redisDb中的数据字典redisDb.dict和过期时间字典redisDb.expire，这样只是为了减少内存开销。

1. **优化措施**

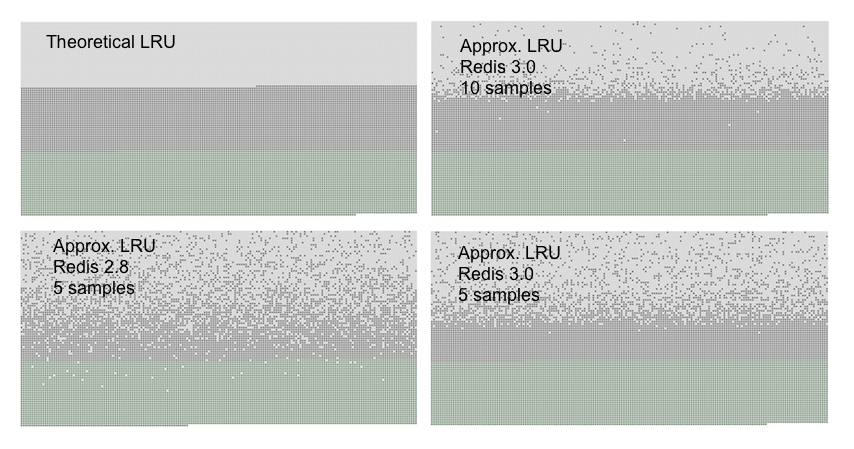
根据设定的maxmemory\_samples，我们从dict数据字典中随机取出maxmemory\_samples个抽样key，然后我们找到这些key中那个最老没有调用的执行删除（samples中的lru）。maxmemory\_samples越大越接近理论上的lru。



1. **测试结果**

Redis 不采用真正的 LRU 实现的原因是为了节约内存使用。虽然不是真正的 LRU 实现，但是它们在应用上几乎是等价的。下图是 Redis 的近似 LRU 实现和理论 LRU 实现的对比：

测试开始首先在 Redis 中导入一定数目的 key，然后从第一个 key 依次访问到最后一个key，因此根据 LRU 算法第一个被访问的 key 应该最新被置换，之后再增加 50% 数目的 key，导致 50% 的老的 key 被替换出去。



在上图中你可以看到三种类型的点，组成三种不同的区域：

* 淡灰色的是被置换出去的key
* 灰色的是没有被置换出去的key
* 绿色的是新增加的key

理论 LRU 实现就像我们期待的那样，最旧的 50% 数目的 key 被置换出去，Redis 的 LRU 将一定比例的旧 key 置换出去。可以看到在样本数为 5 的情况下，Redis3.0 要比 Redis2.8 做的好很多，Redis2.8 中有很多应该被置换出去的数据没有置换出去。在样本数为10的情况下，Redis3.0 很接近真正的 LRU 实现。

LRU 是一个预测未来我们会访问哪些数据的模型，如果我们访问数据的形式接近我们预想——幂律，那么近似 LRU 算法实现将能处理的很好。在模拟测试中我们可以发现，在幂律访问模式下，理论 LRU 和 Redis 近似 LRU 的差距很小或者就不存在差距。

如果你将 maxmemory-samples 设置为 10，那么 Redis 将会增加额外的 CPU 开销以保证接近真正的 LRU 性能，可以通过检查命中率来查看有什么不同。

1. **Mysql innodb**

参见[2]《mysql之innodb-9-lru优化》

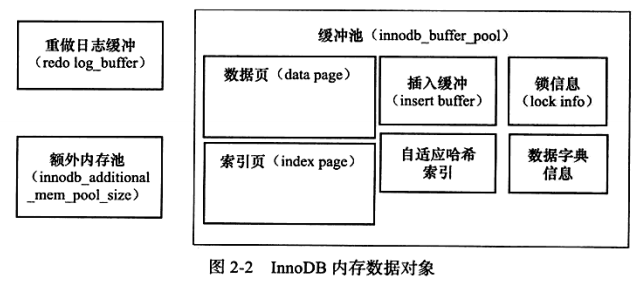
1. **使用场景**

Innodb是基于磁盘存储的，在CPU中，使用缓冲池几乎来提高数据库整体性能。

缓存池简单来说就是一块内存区域，在数据库中读取页的操作，首先将该页放置于缓冲池中，下次再读取相同的页时，就可以通过缓冲池读取。

对于数据页中的修改操作，会先修改缓冲池中的页为脏页，再以一定频率刷新到磁盘上，其刷新频率由checkpoint机制决定（参见3））。这样可以减少随机读写，提高顺序读写。

具体来看缓冲池缓冲的数据页有：索引页、数据页、undo页、insert buffer、自适应哈希索引和Innodb的锁信息、数据字典信息、double write buffer(2MB)等。



缓冲池的更新一般使用LRU列表来管理缓冲页，一般需要保证100个空页，否则触发LRU机制。如果LRU列表中的页被修改，被称为脏页，flush列表中的页就是脏页。LRU列表用于管理缓冲池页的可用性，flush列表用于将脏页刷会磁盘，二者互不影响，脏页既存在于LRU列表，也存在于FLUSH列表。

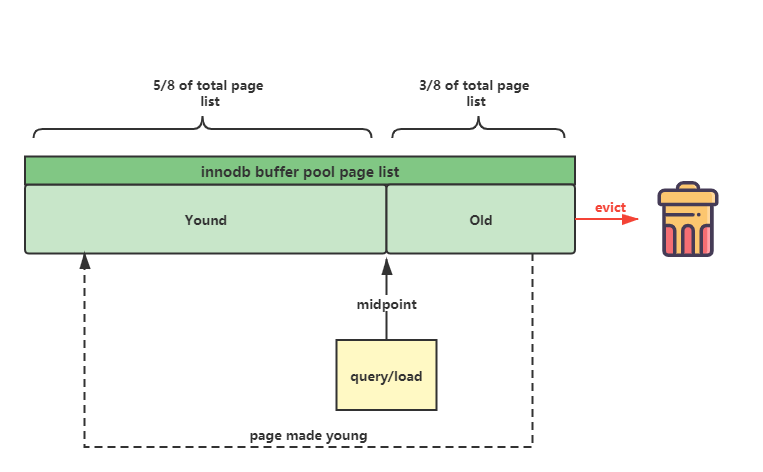
我们可以通过show engine innodb status中的modified db pages可以看出FLUSH列表中的脏页大小。

缓冲池中的页大小默认16 kb。操作系统的页大小可能是2kb或4kb。

1. **优化原因**

数据库中经常有根据索引或数据的扫描操作，这类操作通常需要访问表中许多页，而这些操作的数据通常又只在这次查询使用，不是活跃的热点数据，如果直接把读到的页放到lru首部，显然会将真正的热点缓存页刷出去。

1. **优化措施**



innodb将LRU进行分代，midpoint位于距离tail 3/8，以midpoint为界，向上到head处为热点数据，表示为new，线下到tail处为old代。

新数据插入到lru双向链表的midpoint处，而不是队列head，同时引入innodb\_old\_blocks\_time，用于表示新页读取到midpoint处多久才被放置到lru首部。当页从old部分加入到new部分，称为page made young。如果因为innodb\_old\_blocks\_time设置导致没有从old部分移动到new部分称之为page not made young。

1. **Linux 页缓存**

linux中lru是一对双向链表：active和inactive，这两个链表是进行页面回收的关键数据结构，经常使用且活跃的位于active链表，不活跃的可以被回收的位于inactive链表。那些最近最少使用的页面会被逐个放到 inactive 链表的尾部。进行页面回收的时候，Linux 操作系统会从 inactive 链表的尾部开始进行回收。页面会在这两个链表上来回移动，至于是否获取的判断标识如下：

* PG\_active：标识该页为活跃页；
* PG\_referenced：标识该页最近是否被访问，每次被访问都会被置位，如果一段时间后页面没有被访问，系统会清除页面的PG\_referenced。

Linux 必须同时使用这两个标志符来判断页面的活跃程度，判断标准如下：

1. **进入inactive**

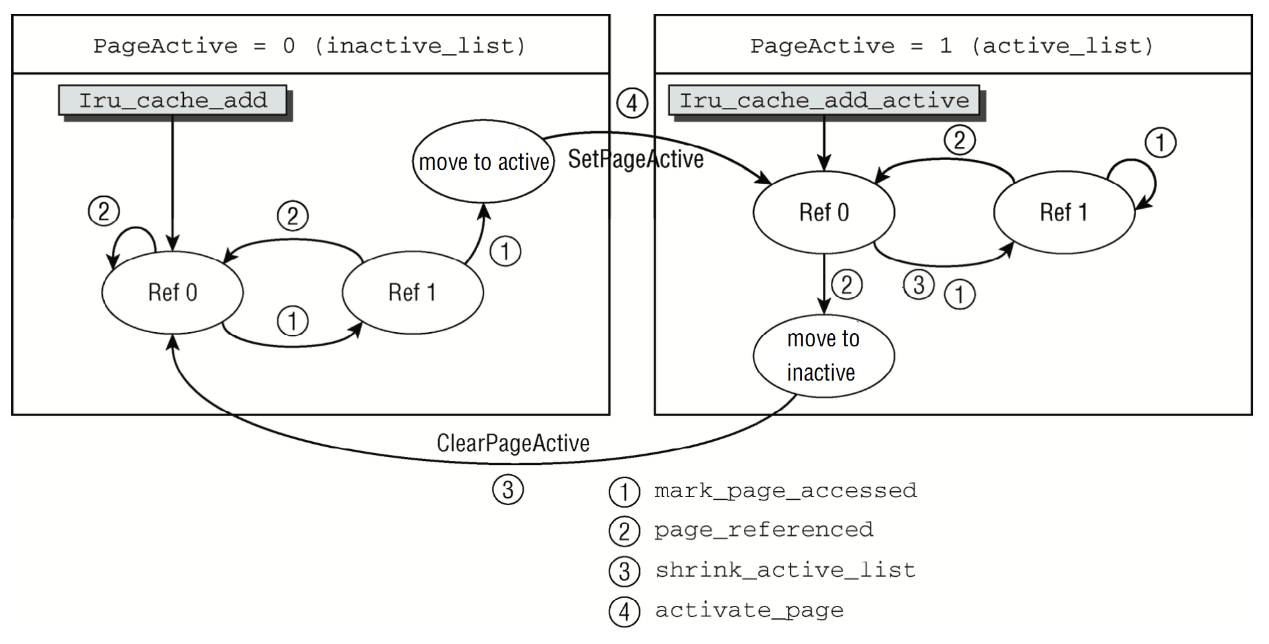
页面访问时，如果发现当前页位于inactive链表，且PG\_referenced标识未置位，则PG\_referenced标识置位，还存放在inactive。

1. **inactive -> active**

页面访问时，如果发现当前页位于inactive链表，且PG\_referenced标识已置位，则认为该页经常访问，将其置位PG\_active，移动到active链表并清除PG\_referenced标识；

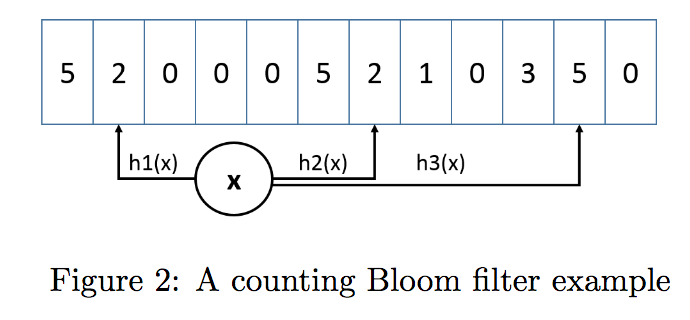
1. **active -> inactive**

在active链表上，其PG\_active被置位，如果给定时间PG\_referenced未被置位，则清除PG\_active标识，将其移动到inactive中。



1. **基于LRU和LFU相结合的现代缓存构想**
2. **解决LFU占用内存：BloomFilter**

我们不希望记录LFU中每个key对应的count，那么可以参考布隆过滤器，利用k次hash，将对应的hash槽加1，那么对于key的多个hash槽的count，取最小值作为lfu近似count。这样可以大大减少空间复杂度。这种算法称为CountMin Sketch。



1. **解决LFU数据过期：reset**

TinyLFU使用reset机制来保证sketch中的数据尽可能最新。每增加一个新的元素到approximation sketches，会增加一个计数值size，一旦计数值size达到了一个预设的采样尺寸（W）：

* 就会将频率采样（CBF）维护的所有计数值除以2（可以使用高效的寄存器位移来实现）
* 同时，size=size/2

论文也花了较大的篇幅通过数学归纳法来证明了这种Reset机制的正确性，且评估了其存在的截断错误（3会被reset为1，而非1.5），并且得出了以下结论：

* reset在constant distribution频率分布下完全正确，且可以应对流量频率的变化（数学归纳法证明，感兴趣的可以参考原文3.3.1）。
* 采样数W越大，截断错误的带来的影响越小。

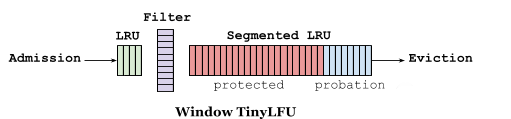
1. **解决稀疏流量：window cache**

有了tinyLFU，我们就可以使用其作为Lru过期淘汰策略。同时为了保证不会有突发稀疏流量，这部分流量可能无法累计足够的count而进入tinyLFU缓存，从而导致缓存失败。所以在tinyLFU之前我们加入了一个window cache（lru，占用整个w-tinyLFU内存1%）用于处理LRU适合的burst场景

1. **Cache分代**

window cache（lru，占用整个w-tinyLFU内存1%）用于处理LRU适合的burst场景，tinyLFU作为filter来执行淘汰策略，主缓存则是一个分段缓存：segmented cache（lru，占用整个w-tinyLFU内存99%）。这个分段缓存分为两个区域，和linux中的active和inactive链表类似。

* probation A1，占用整个分段缓存20%的存储空间，主要用于**执行淘汰**。
  + 如果这里的元素被访问，则将其升级至protected段，如果protected满了则而将protected段最老元素会降级到probation段。
  + 如果从window cache有淘汰元素进入，则需要将probation中的最老元素与其进行count比较来判断到底淘汰谁。
* protected A2，占用整个分段缓存80%的存储空间，主要存储有价值的缓存数据。



1. **tinyLFU的内存优化**
2. **counter**

对tinyLFU，其reset采样大小为w，那么显然counter就不会无限扩大，其大小最大为w，对应数据大小为log(w) bit。设缓存大小为c，对于新的需要进入cache的元素，其计数器大小本身不需要达到log(w)，因为我们只需要比较在缓存中需要丢弃元素的计数（频率大于等于1/c才能保持在缓存里，而reset又会降低一半的频率），与新元素计数即可，而新元素的计数器只需要保证最大为w/c即可。例如w=100, c= 10，那么新元素的计数器只需要限制在10以内即可。

1. **doorKeeper**

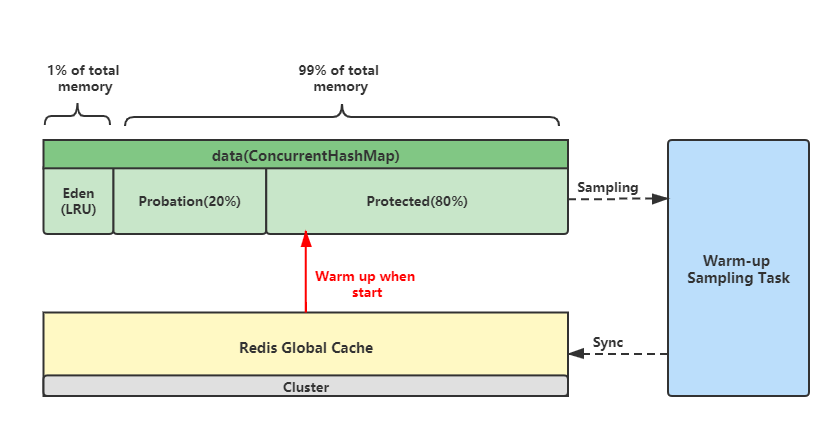
通常会有一些大量只出现一次的对象进入tinyLFU计数，显然这些对象利用count min sketch计算也会增加counter的长度（counter的数据类型可以动态增加）。那么我们可以在进入tinyLFU计算前，加入一个名为doorKeeper的标准布隆过滤器。只有在doorKeeper的元素，才会真正计算器tinyLFU，否则只停留在doorKeeper中，即count为1。

reset会清空doorKeeper。

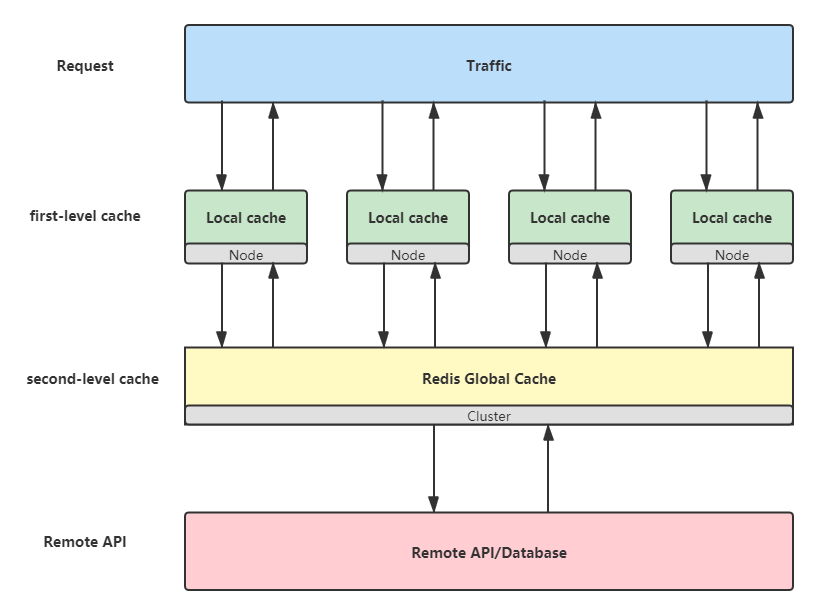
1. **Caffeine的实现**

参见《[cache-2-caffeine](cache-2-caffeine.docx)》

1. **进一步思考**
2. **本地缓存热启动**



1. **全局二级缓存**



**参考**

1. [cache-1-guava](cache-1-guava.docx)
2. [mysql之innodb-9-lru优化](mysql之innodb-9-lru优化.docx)
3. [cache-2-caffeine](cache-2-caffeine.docx)