1. **基本LRU实现**

参见《95-leetcode-No146LRU设计》

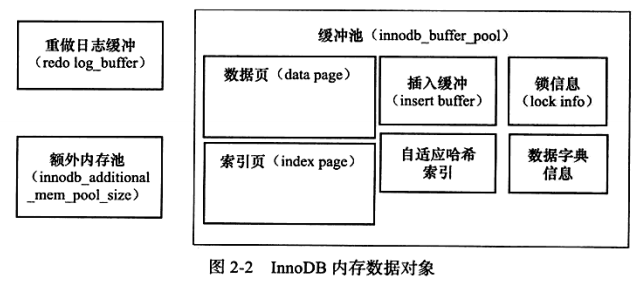
1. **Innodb优化**
2. **使用场景**

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。

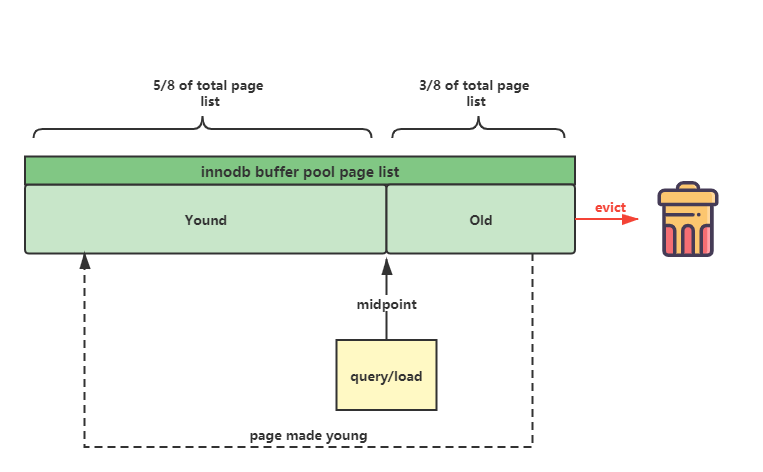
我们可以通过show engine innodb status中的modified db pages可以看出FLUSH列表中的脏页大小。

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. **redis中的lru**
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]：

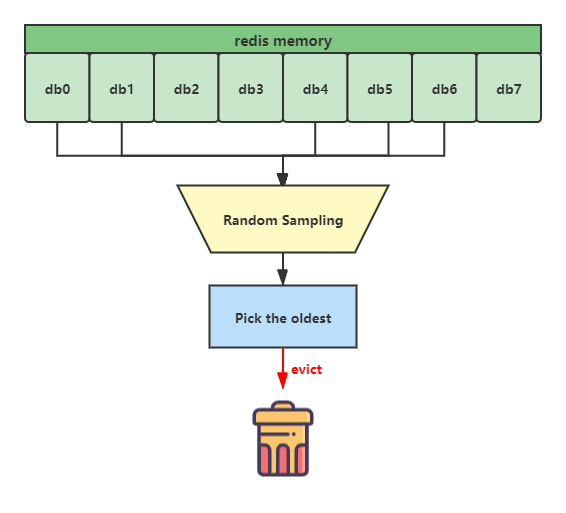
* lru：volatile代表设置了过期时间，allKeys则是所有key
* random：volatile代表设置了过期时间，allKeys则是所有key
* ttl：即将过期

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**

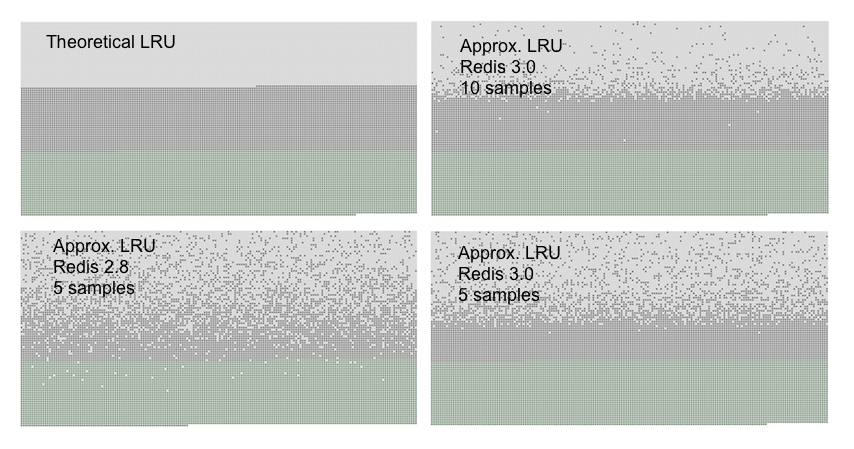
根据设定的maxmemory\_samples，我们从dict数据字典中随机取出maxmemory\_samples个抽样key，然后我们找到这些key中那个最老没有调用的执行删除（samples中的lru）。maxmemory\_samples越大越接近理论上的lru。



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. **源码分析**

Redis释放内存是由函数freeMemoryIfNeeded完成的，redis用processCommand函数处理每条命令，函数中在真正处理命令之前都会调用freeMemoryIfNeeded函数，这个函数会判断当前使用的内存是否超过了最大使用内存，如果超过，就会根据内存释放策略释放内存。

freeMemoryIfNeeded函数首先会计算出当前使用了多少内存，注意，这里并不会包括slaves 输出缓存以及AOF缓存，源码如下：

int freeMemoryIfNeeded(void) {

size\_t mem\_used, mem\_tofree, mem\_freed;

int slaves = listLength(server.slaves);

/\* Remove the size of slaves output buffers and AOF buffer from the

\* count of used memory.

\*/

//计算占用内存大小时，并不计算slave output buffer和aof buffer，因此maxmemory应该比实际内存小，为这两个buffer留足空间。

mem\_used = zmalloc\_used\_memory();

if (slaves) {

listIter li;

listNode \*ln;

listRewind(server.slaves,&li);

while((ln = listNext(&li))) {

redisClient \*slave = listNodeValue(ln);

unsigned long obuf\_bytes = getClientOutputBufferMemoryUsage(slave);

if (obuf\_bytes > mem\_used)

mem\_used = 0;

else

mem\_used -= obuf\_bytes;

}

}

if (server.appendonly) {

mem\_used -= sdslen(server.aofbuf);

mem\_used -= sdslen(server.bgrewritebuf);

}

//判断已经使用内存是否超过最大使用内存，如果没有超过就返回REDIS\_OK，

/\* Check if we are over the memory limit. \*/

if (mem\_used <= server.maxmemory) return REDIS\_OK;

//当超过了最大使用内存时，就要判断此时redis到底采用的是那种内存释放策略，根据不同的策略，采取不同的手段。

//（1）首先判断是否是为no-enviction策略，如果是，则返回REDIS\_ERR,然后redis就不再接受任何写命令了。

if (server.maxmemory\_policy == REDIS\_MAXMEMORY\_NO\_EVICTION)

return REDIS\_ERR; /\* We need to free memory, but policy forbids. \*/

/\* Compute how much memory we need to free. \*/

mem\_tofree = mem\_used - server.maxmemory;

mem\_freed = 0;

//（2）接下来就判断淘汰策略是基于所有的键还是只是基于设置了过期时间的键，如果是针对所有的键，就从server.db[j].dict中取数据，如果是针对设置了过期时间的键，就从server.db[j].expires中取数据。

while (mem\_freed < mem\_tofree) {

int j, k, keys\_freed = 0;

for (j = 0; j < server.dbnum; j++) {

long bestval = 0; /\* just to prevent warning \*/

sds bestkey = NULL;

struct dictEntry \*de;

redisDb \*db = server.db+j;

dict \*dict;

//（3）然后判断是不是random策略，包括volatile-random 和allkeys-random，这两种策略是最简单的，就是在上面的数据集中随便去一个键，然后删掉。

if (server.maxmemory\_policy == REDIS\_MAXMEMORY\_ALLKEYS\_LRU ||

server.maxmemory\_policy == REDIS\_MAXMEMORY\_ALLKEYS\_RANDOM)

{

dict = server.db[j].dict;

} else {

dict = server.db[j].expires;

}

if (dictSize(dict) == 0) continue;

//接着又判断allkeys-random还是volatile-ttl策略

/\* volatile-random and allkeys-random policy \*/

if (server.maxmemory\_policy == REDIS\_MAXMEMORY\_ALLKEYS\_RANDOM ||

server.maxmemory\_policy == REDIS\_MAXMEMORY\_VOLATILE\_RANDOM)

{

de = dictGetRandomKey(dict);

bestkey = dictGetEntryKey(de);

}//如果是random delete,则从dict中随机选一个key

//然后就是判断是lru策略还是ttl策略，如果是lru策略就采用lru近似算法

/\* volatile-lru and allkeys-lru policy \*/

else if (server.maxmemory\_policy == REDIS\_MAXMEMORY\_ALLKEYS\_LRU ||

server.maxmemory\_policy == REDIS\_MAXMEMORY\_VOLATILE\_LRU)

{

for (k = 0; k < server.maxmemory\_samples; k++) {

sds thiskey;

long thisval;

robj \*o;

de = dictGetRandomKey(dict);

thiskey = dictGetEntryKey(de);

/\* When policy is volatile-lru we need an additonal lookup

\* to locate the real key, as dict is set to db->expires. \*/

if (server.maxmemory\_policy == REDIS\_MAXMEMORY\_VOLATILE\_LRU)

de = dictFind(db->dict, thiskey); //因为dict->expires维护的数据结构里并没有记录该key的最后访问时间

o = dictGetEntryVal(de);

thisval = estimateObjectIdleTime(o);

/\* Higher idle time is better candidate for deletion \*/

if (bestkey == NULL || thisval > bestval) {

bestkey = thiskey;

bestval = thisval;

}

}//为了减少运算量,redis的lru算法和expire淘汰算法一样，都是非最优解，lru算法是在相应的dict中，选择maxmemory\_samples(默认设置是3)份key，挑选其中lru的，进行淘汰

}

//如果是ttl策略。ttl策略很简单，就是取maxmemory\_samples个键，然后比较他们的过期时间，然后从这些键中找到最快过期的那个键，就是我们将要删除的键。

/\* volatile-ttl \*/

else if (server.maxmemory\_policy == REDIS\_MAXMEMORY\_VOLATILE\_TTL) {

for (k = 0; k < server.maxmemory\_samples; k++) {

sds thiskey;

long thisval;

de = dictGetRandomKey(dict);

thiskey = dictGetEntryKey(de);

thisval = (long) dictGetEntryVal(de);

/\* Expire sooner (minor expire unix timestamp) is better

\* candidate for deletion \*/

if (bestkey == NULL || thisval < bestval) {

bestkey = thiskey;

bestval = thisval;

}

}//注意ttl实现和上边一样，都是挑选出maxmemory\_samples份进行挑选

}

//根据不同的策略，我们找到了将要删除的键，下面就是将他们删除的时候了，删除选定的键值对

/\* Finally remove the selected key. \*/

if (bestkey) {

long long delta;

robj \*keyobj = createStringObject(bestkey,sdslen(bestkey));

// 发布数据更新消息，主要是AOF 持久化和从机

propagateExpire(db,keyobj); //将del命令扩散给slaves

// 注意， propagateExpire() 可能会导致内存的分配，

// propagateExpire() 提前执行就是因为redis 只计算

// dbDelete() 释放的内存大小。倘若同时计算dbDelete()

// 释放的内存和propagateExpire() 分配空间的大小，与此

// 同时假设分配空间大于释放空间，就有可能永远退不出这个循环。

// 下面的代码会同时计算dbDelete() 释放的内存和propagateExpire() 分配空间的大小

/\* We compute the amount of memory freed by dbDelete() alone.

\* It is possible that actually the memory needed to propagate

\* the DEL in AOF and replication link is greater than the one

\* we are freeing removing the key, but we can't account for

\* that otherwise we would never exit the loop.

\*

\* AOF and Output buffer memory will be freed eventually so

\* we only care about memory used by the key space. \*/

// 只计算dbDelete() 释放内存的大小

delta = (long long) zmalloc\_used\_memory();

dbDelete(db,keyobj);

delta -= (long long) zmalloc\_used\_memory();

mem\_freed += delta;

server.stat\_evictedkeys++;

decrRefCount(keyobj);

keys\_freed++;

/\* When the memory to free starts to be big enough, we may

\* start spending so much time here that is impossible to

\* deliver data to the slaves fast enough, so we force the

\* transmission here inside the loop. \*/

// 将从机回复空间中的数据及时发送给从机

if (slaves) flushSlavesOutputBuffers();

}

}//在所有的db中遍历一遍，然后判断删除的key释放的空间是否足够，未能释放空间，且此时redis 使用的内存大小依旧超额，失败返回

if (!keys\_freed) return REDIS\_ERR; /\* nothing to free... \*/

}

return REDIS\_OK;

}

1. **linux中的lru**
2. **linux的LRU设计**

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**

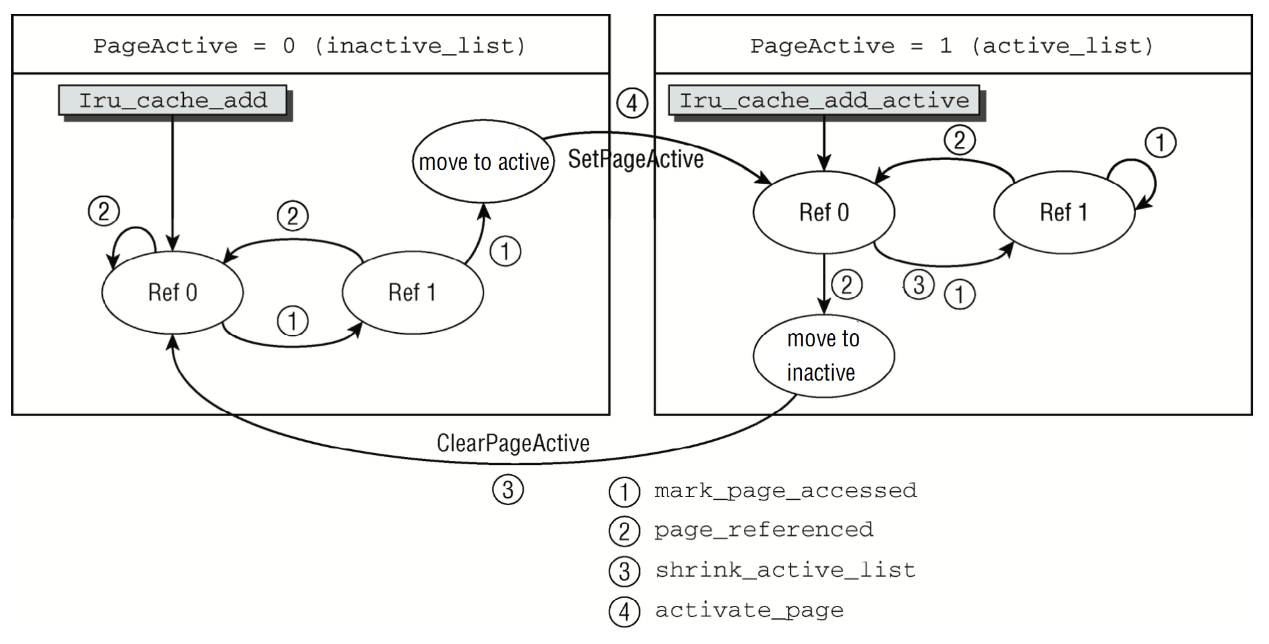
页面访问时，如果发现当前页位于inactive链表，且PG\_referenced标识未置位，则PG\_referenced标识置位，还存放在inactive。

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. **linux的内存回收**
2. **回收手段**

linux系统的缓存分为两大类：

* 匿名缓存页anno：堆，栈，数据段等，不是以文件形式存在，因此无法和磁盘文件交换，但是可以通过磁盘划分的swap交换分区进行swap。
* 文件缓存页file-backed：就是我们常用的page cache，缓存磁盘读写内容，包含脏页，所以其回收主要通过回写脏页后直接回收。

这两种类型的缓存页都有各自的active和inactive链表。

所以内存回收主要有两大策略，都是对inactive链表中的页进行处理。

* 针对匿名缓存页，主要回收手段是swap；
* 针对文件缓存页，主要释放手段是写回和清空。

脏页的回写也会根据脏页比例和过期时间来判断。

1. **回收过程**

Linux通过watermark来描述内存压力情况，分为high、low、min：

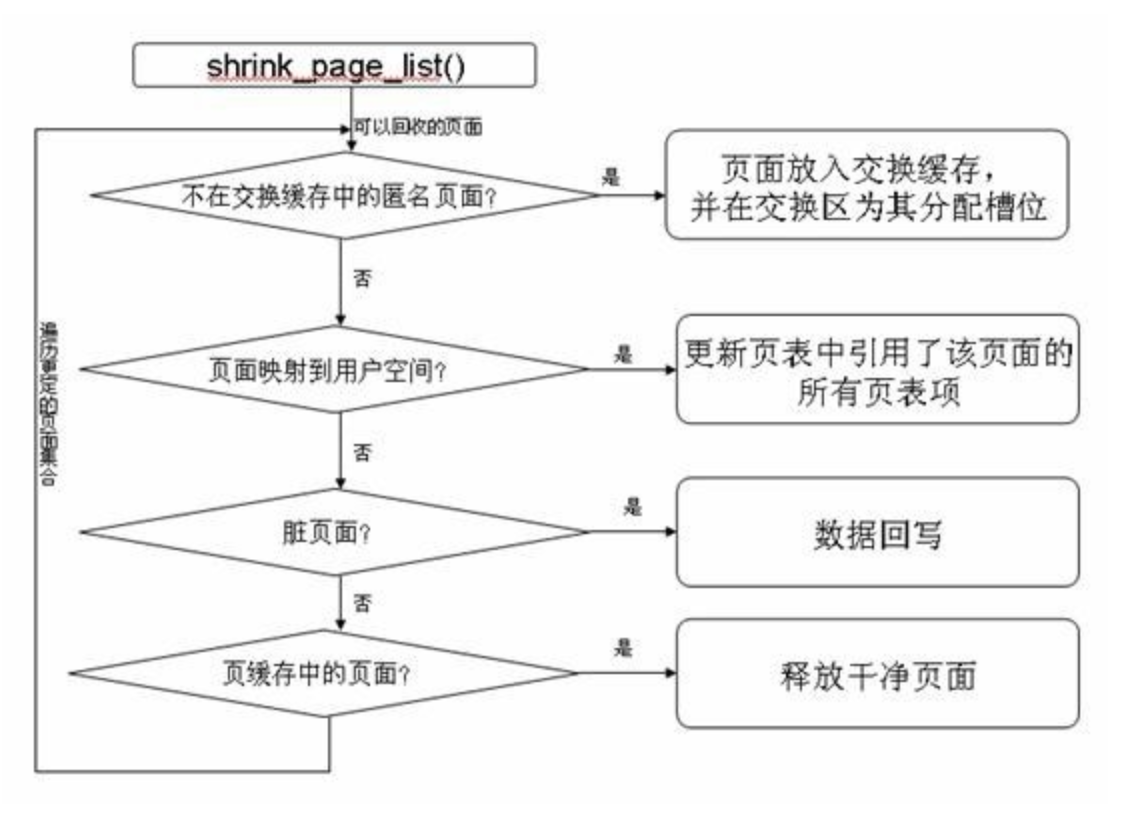
* 剩余内存在High以上说明内存压力不大；
* 剩余内存在[low, high]说明存在一定压力，但还不需要定时回收；
* 剩余内存在[min, low]说明内存压力较大，此时启动kswapd周期检查进行内存回收，知道剩余内存达到high以上kswapd停止；
* 剩余内存小于min，则会触发直接回收。

所以内存回收从总体上有两种方式：kswapd周期检查和直接回收。

同时，在全局回收状态下，如果满足*zonefile + zonefree <= high\_wmark\_pages(zone)* 则需要进行swap匿名页，其中zonefile是内存中文件映射总量。此时说明要让剩余内存回收到high只靠文件缓存页回写和清理是不可能实现的，所以必须执行匿名页swap。内核一般认为，如果zonefile还有的话，就可以尽量通过清空文件缓存获得部分内存，而不必只使用swap方式对匿名页的内存进行交换。所以即使swappiness=0，也会执行swap。

shrink\_zone()是Linux执行内存回收的关键函数，它主要做了两个工作：

* 将某些页从active移动到inactive；
* 从inactive中选取一定页面执行shrink\_page\_list()进行回收。



1. **回收策略swappiness**

swappiness是一个百分比，代表执行匿名页swap操作和文件页回写清理操作在系统看来的权重。值越高，内核就会越积极使用swap，反之亦然。

如果swappiness=100，则代表匿名页swap操作和文件页回写清理操作权重相等；

如果swappiness=0，则代表不满足*zonefile + zonefree <= high\_wmark\_pages(zone)* 的情况下不会使用swap，并不是字面意义上关闭swap。关闭swap对应指令swapoff。如果关闭swap的情况下发生内存不足，则内核可能执行OOM killer来杀死一些进程来回收其所有缓存页。

匿名页swap操作和文件页回写清理操作优先级计算公式如下：

*anno\_prio = swappiness*

*file\_prio = 200 – anno\_prio*

1. **zRAM替代swappiness**

不用swap分区，也可以用zRAM机制来缓解内存紧张： 从内存里拿出一段内存空间（compressed block），作为交换空间模拟硬盘的交换分区，用来交换匿名页，并且让kernel看到的物理内存大小不包括这段内存。而这段交换空间自带透明压缩功能，即交换到这块zRAM分区时，Linux会自动将这块匿名页压缩存放。系统访问这块页面的内容时，产生page fault后从交换分区去拿，这时Linux给你透明解压再交换出来。

使用zRAM的好处，就是访存比访问硬盘或flash的速度提高很多，且不用考虑寿命问题，并且由于这段内存是压缩后存储的，因此可以存更多的数据，虽然占用了一段内存，但实际可以存更多的数据，也达到了增加内存的效果。缺点就是压缩要占用CPU时间。

Android里面普遍使用了zRAM技术，由于zRAM牺牲了CPU时间，所以交换次数还是越少越好。像Android和windows，内存越大越好，因为发生交换的几率就小。这样两个进程相互切换（如微博和微信）时就会变得流畅，因为内存足够的话，后台进程无需被换进swap分区或被OOM杀掉。

1. **现代缓存——caffine**
2. **lru和lfu的缺陷**

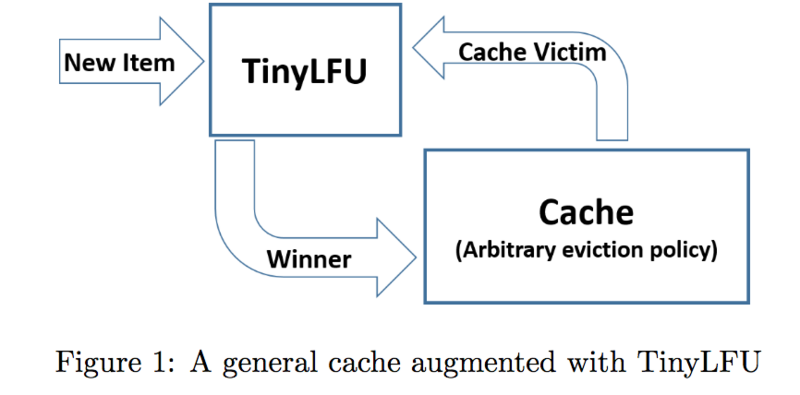
lru是最常见的缓存，认为最近访问的优先级最高，优点是存储空间小，缺点也很明显：可能被突发的稀疏流量冲击，导致真正缓存的数据被evict。其优化方案如Innodb和linux中的分代Lru，但是对缓存优先级定义仍然有缺陷。

lfu认为访问频次越高优先级越高，理论上来说对缓存的定义比lru更贴切，但是缺点也很明显：

* 无法应对数据过期，新来的数据由于频次不及过期的高频数据，无法加入到缓存中导致缓存失效；
* lfu具体实现上需要三个Map：key-value，key-count以及count-keys，占用很大内存空间，所以大部分缓存失效都不会用这种完备的Lfu。

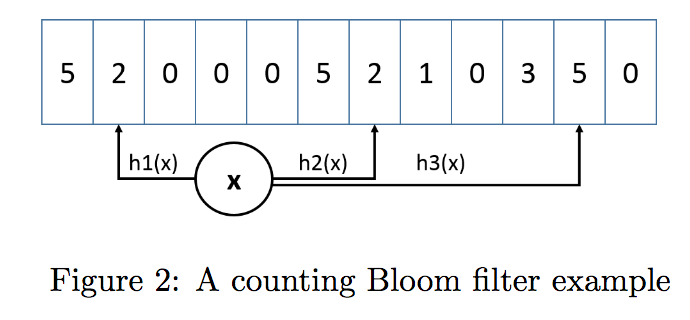
1. **lfu优化——tinyLFU**
2. **基本结构**

TinyLFU的淘汰策略是：一个新元素加入，判断能否替换一个旧元素，判断的依据是能否提升缓存命中率。显然，我们试图利用LFU来辅助Cache的缓存的LRU淘汰。



1. **countMin Sketch**

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. **lfu的保鲜机制**

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. **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**

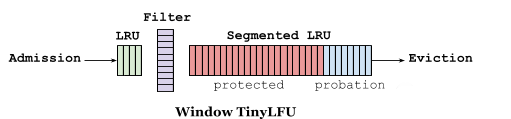
通常会有一些大量只出现一次的对象进入tinyLFU计数，显然这些对象利用count min sketch计算也会增加counter的长度（counter的数据类型可以动态增加）。那么我们可以在进入tinyLFU计算前，加入一个名为doorKeeper的标准布隆过滤器。只有在doorKeeper的元素，才会真正计算器tinyLFU，否则只停留在doorKeeper中，即count为1。

reset会清空doorKeeper。

1. **w-tinyLFU**

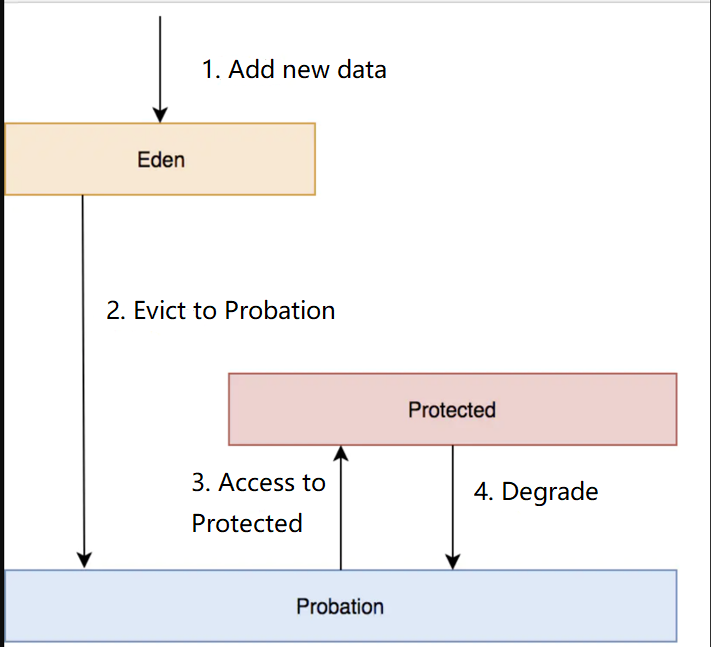
有了tinyLFU，我们就可以使用其作为Lru过期淘汰策略。同时为了保证不会有突发稀疏流量，这部分流量可能无法累计足够的count而进入tinyLFU缓存，从而导致缓存失败。所以在tinyLFU之前我们加入了一个window cache（lru，占用整个w-tinyLFU内存1%）用于处理LRU适合的burst场景，tinyLFU作为filter来执行淘汰策略，主缓存则是一个分段缓存：segmented cache（lru，占用整个w-tinyLFU内存99%）。这个分段缓存分为两个区域，和linux中的active和inactive链表类似。

* 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. **基本流程**

* 新元素首先插入window cache。
* window cache满了则执行降级到probation队尾。如果probation满了，则执行**淘汰**。
* 如果访问了probation中的元素，则将其升级到protected；
* protected满了则将其最老元素降级到probation；



1. **淘汰算法**

probation队列中队首是最Lru中最老的元素，队尾是刚从window cache淘汰来的元素，那么会根据tinyLFU的频度来判断到底淘汰谁（只有这种情况会出现淘汰）：

* 如果队尾元素的频度大于队首，那么就直接淘汰队首，
* 当队尾频度小于等于队首，且频度小于5的时候，直接将队尾淘汰
* 当队尾频度小于等于队首，且频度大于5的时候，通过随机的方式进行淘汰任意一个键。

**参考**

1. redis <https://www.cnblogs.com/alsf/p/9399009.html>
2. 《redis原理-2-数据库结构与持久化》
3. 《redis原理-4-服务器命令处理与启动》
4. linux <https://www.cnblogs.com/muahao/p/10109712.html>
5. linux swap <https://zhuanlan.zhihu.com/p/102181254>
6. zRam https://blog.csdn.net/jasonchen\_gbd/article/details/79462014