源码 8：精益求精 —— LFU vs LRU

在第 27 小节，我们讲到了 Redis 的 LRU 模式，它可以有效的控制 Redis 占用内存大小，将冷数据从内存中淘汰出去。Antirez 在 Redis 4.0 里引入了一个新的淘汰策略 —— LFU 模式，作者认为它比 LRU 更加优秀。

LFU 的全称是Least Frequently Used，表示按最近的访问频率进行淘汰，它比 LRU 更加精准地表示了一个 key 被访问的热度。

如果一个 key 长时间不被访问，只是刚刚偶然被用户访问了一下，那么在使用 LRU 算法下它是不容易被淘汰的，因为 LRU 算法认为当前这个 key 是很热的。而 LFU 是需要追踪最近一段时间的访问频率，如果某个 key 只是偶然被访问一次是不足以变得很热的，它需要在近期一段时间内被访问很多次才有机会被认为很热。

Redis 对象的热度

Redis 的所有对象结构头中都有一个 24bit 的字段，这个字段用来记录对象的「热度」。

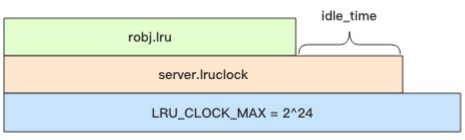
```
// redis 的对象头
typedef struct redisObject {
    unsigned type:4; // 对象类型如 zset/set/hash 等等
    unsigned encoding:4; // 对象编码如 ziplist/intset/skiplist 等等
    unsigned lru:24; // 对象的「热度」
    int refcount; // 引用计数
    void *ptr; // 对象的 body
} robj;
```

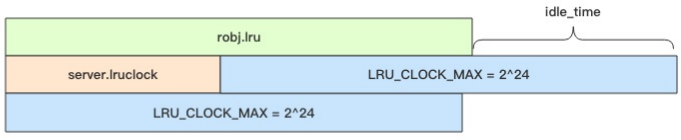
LRU 模式

在 LRU 模式下，lru 字段存储的是 Redis 时钟 server.lruclock，Redis 时钟是一个 24bit 的整数，默认是 Unix 时间戳对 2^24 取模的结果，大约 97 天清零一次。当某个 key 被访问一次，它的对象头的 lru 字段值就会被更新为server.lruclock。

默认 Redis 时钟值每毫秒更新一次，在定时任务serverCron里主动设置。Redis 的很多定时任务都是在serverCron里面完成的，比如大型 hash 表的渐进式迁移、过期 key 的主动淘汰、触发 bgsave、bgaofrewrite 等等。

如果server.lruclock没有折返 (对 2^24 取模)，它就是一直递增的，这意味着对象的 LRU 字段不会超过server.lruclock的值。如果超过了，说明server.lruclock折返了。通过这个逻辑就可以精准计算出对象多长时间没有被访问——对象的空闲时间。



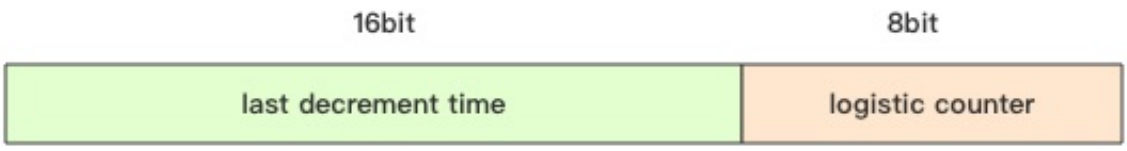


```
// 计算对象的空闲时间，也就是没有被访问的时间，返回结果是毫秒
unsigned long long estimateObjectIdleTime(robj *o) {
    unsigned long long lruclock = LRU_CLOCK(); // 获取 redis 时钟，也就是 server.lruclock 的值
    if (lruclock >= o->lru) {
        // 正常递增
        return (lruclock - o->lru) * LRU_CLOCK_RESOLUTION; // LRU_CLOCK_RESOLUTION 默认是 1000
    } else {
        // 折返了
        return (lruclock + (LRU_CLOCK_MAX - o->lru)) * // LRU_CLOCK_MAX 是 2^24-1
                    LRU_CLOCK_RESOLUTION;
    }
}
```

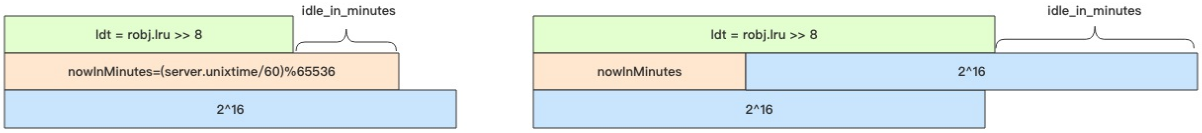
有了对象的空闲时间，就可以相互之间进行比较谁新谁旧，随机 LRU 算法靠的就是比较对象的空闲时间来决定谁该被淘汰了。

LFU 模式

在 LFU 模式下，lru 字段 24 个 bit 用来存储两个值，分别是ldt(last decrement time)和logc(logistic counter)。



logc 是 8 个 bit，用来存储访问频次，因为 8 个 bit 能表示的最大整数值为 255，存储频次肯定远远不够，所以这 8 个 bit 存储的是频次的对数值，并且这个值还会随时间衰减。如果它的值比较小，那么就很容易被回收。为了确保新创建的对象不被回收，新对象的这 8 个 bit 会初始化为一个大于零的值，默认是LFU_INIT_VAL=5。



ldt 是 16 个位，用来存储上一次 logc 的更新时间，因为只有 16 位，所以精度不可能很高。它取的是分钟时间戳对 2^16 进行取模，大约每隔 45 天就会折返。同 LRU 模式一样，我们也可以使用这个逻辑计算出对象的空闲时间，只不过精度是分钟级别的。图中的 server.unixtime 是当前 redis 记录的系统时间戳，和 server.lruclock 一样，它也是每毫秒更新一次。

```
// nowInMinutes
// server.unixtime 为 redis 缓存的系统时间戳
unsigned long LFUGetTimeInMinutes(void) {
    return (server.unixtime/60) & 65535;
}

// idle_in_minutes
unsigned long LFUTimeElapsed(unsigned long ldt) {
    unsigned long now = LFUGetTimeInMinutes();
    if (now >= ldt) return now-ldt; // 正常比较
    return 65535-ldt+now; // 折返比较
}
```

ldt 的值和 LRU 模式的 lru 字段不一样的是 ldt 不是在对象被访问时更新的。它在 Redis 的淘汰逻辑进行时进行更新，淘汰逻辑只会在内存达到 maxmemory 的设置时才会触发，在每一个指令的执行之前都会触发。每次淘汰都是采用随机策略，随机挑选若干个 key，更

新这个 key 的「热度」，淘汰掉「热度」最低的。因为 Redis 采用的是随机算法，如果 key 比较多的话，那么 ldt 更新的可能会比较慢。不过既然它是分钟级别的精度，也没有必要更新的过于频繁。

ldt 更新的同时也会一同衰减 logc 的值，衰减也有特定的算法。它将现有的 logc 值减去对象的空闲时间 (分钟数) 除以一个衰减系数，默认这个衰减系数`lfu_decay_time`是 1。如果这个值大于 1，那么就会衰减的比较慢。如果它等于零，那就表示不衰减，它是可以通过配置参数`lfu-decay-time`进行配置。

```
// 衰减 logc
unsigned long LFUDecrAndReturn(robj *o) {
    unsigned long ldt = o->lru >> 8; // 前 16bit
    unsigned long counter = o->lru & 255; // 后
8bit 为 logc
    // num_periods 为即将衰减的数量
    unsigned long num_periods =
server.lfu_decay_time ? LFUTimeElapsed(ldt) /
server.lfu_decay_time : 0;
    if (num_periods)
        counter = (num_periods > counter) ? 0 :
counter - num_periods;
    return counter;
}
```

logc 的更新和 LRU 模式的 lru 字段一样，都会在 key 每次被访问的时候更新，只不过它的更新不是简单的+1，而是采用概率法进行递增，因为 logc 存储的是访问计数的对数值，不能直接+1。

```
/* Logarithmically increment a counter. The
greater is the current counter value
 * the less likely is that it gets really
implemented. Saturate it at 255. */
// 对数递增计数值
uint8_t LFULogIncr(uint8_t counter) {
    if (counter == 255) return 255; // 到最大值了，
不能在增加了
    double baseval = counter - LFU_INIT_VAL; // 减
去新对象初始化的基数值 (LFU_INIT_VAL 默认是 5)
    // baseval 如果小于零，说明这个对象快不行了，不过本
次 incr 将会延长它的寿命
    if (baseval < 0) baseval = 0;
    // 当前计数越大，想要 +1 就越困难
    // lfu_log_factor 为困难系数，默认是 10
    // 当 baseval 特别大时，最大是 (255-5)，p 值会非常
小，很难会走到 counter++ 这一步
    // p 就是 counter 通往 [+1] 权力的门缝，baseval
越大，这个门缝越窄，通过就越艰难
    double p =
1.0/(baseval*server.lfu_log_factor+1);
    // 开始随机看看能不能从门缝挤进去
    double r = (double)rand()/RAND_MAX; // 0 < r
< 1
    if (r < p) counter++;
    return counter;
}
```

为什么 Redis 要缓存系统时间戳?

我们平时使用系统时间戳时，常常是不假思索地使用System.currentTimeInMillis或者time.time()来获取系统的毫秒时间戳。Redis 不能这样，因为每一次获取系统时间戳都是一次系统调用，系统调用相对来说是比较费时间的，作为单线程的 Redis 表示承受不起，所以它需要对时间进行缓存，获取时间都直接从缓存中直接拿。

redis 为什么在获取 lruclock 时使用原子操作?

我们知道 Redis 是单线程的，那为什么 lruclock 要使用原子操作 atomicGet 来获取呢?

```
unsigned int LRU_CLOCK(void) {
    unsigned int lruclock;
    if (1000/server.hz <= LRU_CLOCK_RESOLUTION) {
        // 这里原子操作，通常会走这里，我们只需要注意这里
        atomicGet(server.lruclock,lruclock);
    } else {
        // 直接通过系统调用获取时间戳，hz 配置的太低 (一般不会这么干)，lruclock 更新不及时，需要实时获取系统时间戳
        lruclock = getLRUClock();
    }
    return lruclock;
}
```

因为 Redis 实际上并不是单线程，它背后还有几个异步线程也在默默工作。这几个线程也要访问 Redis 时钟，所以 lruclock 字段是需要支持多线程读写的。使用 atomic 读写能保证多线程 lruclock 数据的一致性。

如何打开 LFU 模式?

Redis 4.0 给淘汰策略配置参数maxmemory-policy增加了 2 个选项，分别是 volatile-lfu 和 allkeys-lfu，分别是对带过期时间的 key 进行 lfu 淘汰以及对所有的 key 执行 lfu 淘汰算法。打开了这个选项之后，就可以使用 object freq指令获取对象的 lfu 计数值了。

```
> config set maxmemory-policy allkeys-lfu
OK
> set codehole yeahyeahyeah
OK
// 获取计数值，初始化为 LFU_INIT_VAL=5
> object freq codehole
(integer) 5
// 访问一次
> get codehole
"yeahyeahyeah"
// 计数值增加了
> object freq codehole
(integer) 6
```

思考题

1. 你能尝试使用 py 或者 Java 写一个简单的 LFU 算法么?
2. 如果一开始使用 LRU 模式，突然改变配置变成了 LFU 模式，想象一下 Redis 对象头的 lru 字段值，会对现有的对象产生什么影响?