lfu

LFU(Least Frequently Used)算法，最近最频繁使用

介绍

在Redis中的LRU算法文中说到，LRU有一个缺陷，在如下情况下：

```
~~~~~A~~~~~A~~~~~A~~~~A~~~~~A~~~~~A~~|
~~B~~B~~B~~B~~B~~B~~B~~B~~B~~B~~B~|
~~~~~~~~~~C~~~~~~~~~C~~~~~~~~~C~~~~~~|
~~~~~D~~~~~~~~~~D~~~~~~~~~D~~~~~~~~~D|
```

会将数据D误认为将来最有可能被访问到的数据。

Redis作者曾想改进LRU算法，但发现Redis的LRU算法受制于随机采样数maxmemory_samples，在maxmemory_samples等于10的情况下已经很接近于理想的LRU算法性能，也就是说，LRU算法本身已经很难再进一步了。

于是，将思路回到原点，淘汰算法的本意是保留那些将来最有可能被再次访问的数据，而LRU算法只是预测最近被访问的数据将来最有可能被访问到。我们可以转变思路，采用一种LFU(Least Frequently Used)算法，也就是最频繁被访问的数据将来最有可能被访问到。在上面的情况中，根据访问频繁情况，可以确定保留优先级：B>A>C=D。

Redis中的LFU思路 在LFU算法中，可以为每个key维护一个计数器。每次key被访问的时候，计数器增大。计数器越大，可以约等于访问越频繁。

上述简单算法存在两个问题：

在LRU算法中可以维护一个双向链表，然后简单的把被访问的节点移至链表开头，但在LFU中是不可行的，节点要严格按照计数器进行排序，新增节点或者更新节点位置时，时间复杂度可能达到O(N)。只是简单的增加计数器的方法并不完美。访问模式是会频繁变化的，一段时间内频繁访问的key一段时间之后可能会很少被访问到，只增加计数器并不能体现这种趋势。 第一个问题很好解决，可以借鉴LRU实现的经验，维护一个待淘汰key的pool。第二个问题的解决办法是，记录key最后一个被访问的时间，然后随着时间推移，降低计数器。

Redis对象的结构如下：

```
typedef struct redisObject {
    unsigned type:4;
    unsigned encoding:4;
    unsigned lru:LRU_BITS; /* LRU time (relative to global lru_clock) or
                            * LFU data (least significant 8 bits frequency
                            * and most significant 16 bits access time). */
    int refcount;
    void *ptr;
} robj;
```

在LRU算法中，24 bits的lru是用来记录LRU time的，在LFU中也可以使用这个字段，不过是分成16 bits与8 bits使用：

```
           16 bits      8 bits
      +----------------+--------+
      + Last decr time | LOG_C  |
      +----------------+--------+
```

高16 bits用来记录最近一次计数器降低的时间ldt，单位是分钟，低8 bits记录计数器数值counter。

LFU配置 Redis4.0之后为maxmemory_policy淘汰策略添加了两个LFU模式：

```
volatile-lfu：对有过期时间的key采用LFU淘汰算法
allkeys-lfu：对全部key采用LFU淘汰算法
```

还有2个配置可以调整LFU算法：

```
lfu-log-factor 10
lfu-decay-time 1
```

lfu-log-factor可以调整计数器counter的增长速度，lfu-log-factor越大，counter增长的越慢。

lfu-decay-time是一个以分钟为单位的数值，可以调整counter的减少速度

源码实现 在lookupKey中:

```
robj *lookupKey(redisDb *db, robj *key, int flags) {
    dictEntry *de = dictFind(db->dict,key->ptr);
    if (de) {
        robj *val = dictGetVal(de);

        /* Update the access time for the ageing algorithm.
         * Don't do it if we have a saving child, as this will trigger
         * a copy on write madness. */
        if (server.rdb_child_pid == -1 &&
            server.aof_child_pid == -1 &&
            !(flags & LOOKUP_NOTOUCH))
        {
            if (server.maxmemory_policy & MAXMEMORY_FLAG_LFU) {
                updateLFU(val);
            } else {
                val->lru = LRU_CLOCK();
            }
        }
        return val;
    } else {
        return NULL;
    }
}
```

当采用LFU策略时，updateLFU更新lru。

```
/* Update LFU when an object is accessed.
 * Firstly, decrement the counter if the decrement time is reached.
```

```
 * Then logarithmically increment the counter, and update the access time.
*/
void updateLFU(robj *val) {
    unsigned long counter = LFUDecrAndReturn(val);
    counter = LFULogIncr(counter);
    val->lru = (LFUGetTimeInMinutes()<<8) | counter;
}
```

降低LFUDecrAndReturn
首先LFUDecrAndReturn对counter进行减少操作：

```
/* If the object decrement time is reached decrement the LFU counter but
 * do not update LFU fields of the object, we update the access time
 * and counter in an explicit way when the object is really accessed.
 * And we will times halve the counter according to the times of
 * elapsed time than server.lfu_decay_time.
 * Return the object frequency counter.
 *
 * This function is used in order to scan the dataset for the best object
 * to fit: as we check for the candidate, we incrementally decrement the
 * counter of the scanned objects if needed. */
unsigned long LFUDecrAndReturn(robj *o) {
    unsigned long ldt = o->lru >> 8;
    unsigned long counter = o->lru & 255;
    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;
}
```

函数首先取得高16 bits的最近降低时间ldt与低8 bits的计数器counter然后根据配置的lfu_decay_time计算应该降低多少。

LFUTimeElapsed用来计算当前时间与ldt的差值：

```
/* Return the current time in minutes, just taking the least significant
 * 16 bits. The returned time is suitable to be stored as LDT (last
decrement
 * time) for the LFU implementation. */
unsigned long LFUGetTimeInMinutes(void) {
    return (server.unixtime/60) & 65535;
}

/* Given an object last access time, compute the minimum number of minutes
 * that elapsed since the last access. Handle overflow (ldt greater than
 * the current 16 bits minutes time) considering the time as wrapping
 * exactly once. */
unsigned long LFUTimeElapsed(unsigned long ldt) {
    unsigned long now = LFUGetTimeInMinutes();
    if (now >= ldt) return now-ldt;
    return 65535-ldt+now;
```

```
}
```

具体是当前时间转化成分钟数后取低16 bits，然后计算与ldt的差值now-ldt，当ldt > now时，默认为过了一个周期(16 bits，最大65535)，取值65535-ldt+now。

然后用差值与配置lfu_decay_time相除，LFUTimeElapsed(ldt) / server.lfu_decay_time，已过去n个lfu_decay_time，则将counter减少n，counter - num_periods。

增长LFULogIncr 增长函数LFULogIncr如下：

```
/* Logarithmically increment a counter. The greater is the current counter
value
 * the less likely is that it gets really implemented. Saturate it at 255.
*/
uint8_t LFULogIncr(uint8_t counter) {
    if (counter == 255) return 255;
    double r = (double)rand()/RAND_MAX;
    double baseval = counter - LFU_INIT_VAL;
    if (baseval < 0) baseval = 0;
    double p = 1.0/(baseval*server.lfu_log_factor+1);
    if (r < p) counter++;
    return counter;
}
```

counter并不是简单的访问一次就+1，而是采用了一个0-1之间的p因子控制增长，counter最大值为255。取一个0-1之间的随机数r与p比较，当r<p时，才增加counter，这和比特币中控制产出的策略类似，p取决于当前counter值与lfu_log_factor因子，counter值与lfu_log_factor因子越大，p越小，r<p的概率也越小，counter增长的概率也就越小。增长情况如下：

```
+--------+------------+------------+------------+------------+------------+
| factor | 100 hits   | 1000 hits  | 100K hits  | 1M hits    | 10M hits   |
+--------+------------+------------+------------+------------+------------+
| 0      | 104        | 255        | 255        | 255        | 255        |
+--------+------------+------------+------------+------------+------------+
| 1      | 18         | 49         | 255        | 255        | 255        |
+--------+------------+------------+------------+------------+------------+
| 10     | 10         | 18         | 142        | 255        | 255        |
+--------+------------+------------+------------+------------+------------+
| 100    | 8          | 11         | 49         | 143        | 255        |
+--------+------------+------------+------------+------------+------------+
```

可见counter增长与访问次数呈现对数增长的趋势，随着访问次数越来越大，counter增长的越来越慢。

新生key策略 另外一个问题是，当创建新对象的时候，对象的counter如果为0，很容易就会被淘汰掉，还需要为新生key设置一个初始counter，createObject:

```
robj *createObject(int type, void *ptr) {
    robj *o = zmalloc(sizeof(*o));
    o->type = type;
    o->encoding = OBJ_ENCODING_RAW;
    o->ptr = ptr;
```

```
    o->refcount = 1;

    /* Set the LRU to the current lruclock (minutes resolution), or
     * alternatively the LFU counter. */
    if (server.maxmemory_policy & MAXMEMORY_FLAG_LFU) {
        o->lru = (LFUGetTimeInMinutes()<<8) | LFU_INIT_VAL;
    } else {
        o->lru = LRU_CLOCK();
    }
    return o;
}
```

counter会被初始化为LFU_INIT_VAL默认5。

pool pool算法就与LRU算法一致了：

```
        if (server.maxmemory_policy &
(MAXMEMORY_FLAG_LRU|MAXMEMORY_FLAG_LFU) ||
            server.maxmemory_policy == MAXMEMORY_VOLATILE_TTL)
```

计算idle时有所不同：

```
        } else if (server.maxmemory_policy & MAXMEMORY_FLAG_LFU) {
            /* When we use an LRU policy, we sort the keys by idle time
             * so that we expire keys starting from greater idle time.
             * However when the policy is an LFU one, we have a frequency
             * estimation, and we want to evict keys with lower frequency
             * first. So inside the pool we put objects using the inverted
             * frequency subtracting the actual frequency to the maximum
             * frequency of 255. */
            idle = 255-LFUDecrAndReturn(o);
```

使用了255-LFUDecrAndReturn(o)当做排序的依据。

参考链接

http://antirez.com/news/109

https://redis.io/topics/lru-cache

https://www.cnblogs.com/linxiyue/p/10955533.html