redis4.0之基于LFU的热点key发现机制

前言

业务中存在访问热点是在所难免的，redis也会遇到这个问题，然而如何发现热点key一直困扰着许多用户，redis4.0为我们带来了许多新特性，其中便包括基于LFU的热点key发现机制。

Least Frequently Used

Least Frequently Used——简称LFU，意为最不经常使用，是redis4.0新增的一类内存逐出策略，关于内存逐出可以参考文章《Redis数据过期和淘汰策略详解》。

从LFU的字面意思我们很容易联想到key的访问频率，但是4.0最初版本仅用来做内存逐出，对于访问频率并没有很好的记录，那么经过一番改造，redis于4.0.3版本开始正式支持基于LFU的热点key发现机制。

LFU算法介绍

在redis中每个对象都有24 bits空间来记录LRU/LFU信息：

```
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;
```

当这24 bits用作LFU时，其被分为两部分：

1. 高16位用来记录访问时间（单位为分钟）
2. 低8位用来记录访问频率，简称counter

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

counter：基于概率的对数计数器

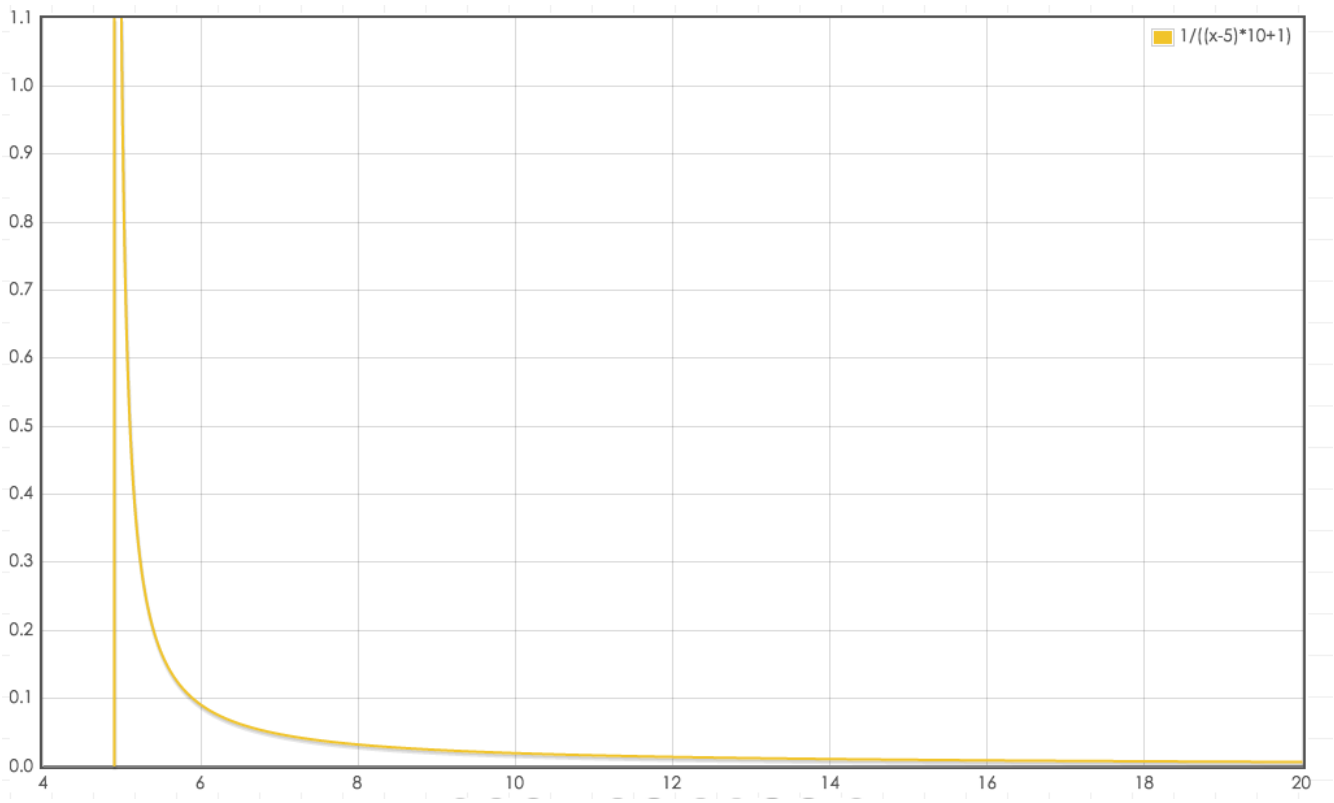
这里读者可能会有疑问，8 bits最大值也就是255，只用8位来记录访问频率够用吗？对于counter，redis用了一个trick的手段，counter并不是一个简单的线性计数器，而是用基于概率的对数计数器来实现，算法如下：

```
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;
}
```

对应的概率分布计算公式为：

```
1/((counter-LFU_INIT_VAL)*server.lfu_log_factor+1)
```

其中LFU_INIT_VAL为5，我们看下概率分布图会有一个更直观的认识，以默认server.lfu_log_factor=10为例：



从上图可以看到，counter越大，其增加的概率越小，8 bits也足够记录很高的访问频率，下表是不同概率因子server.lfu_log_factor与访问频率counter的对应关系：

```
# +--------+------------+------------+------------+------------+------------+
# | 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        |
# +--------+------------+------------+------------+------------+------------+
```

也就是说，默认server.lfu_log_factor为10的情况下，8 bits的counter可以表示1百万的访问频率。

counter的衰减因子

从上一小节的counter增长函数LFULogIncr中我们可以看到，随着key的访问量增长，counter最终都会收敛为255，这就带来一个问题，如果counter只增长不衰减就无法区分热点key。

为了解决这个问题，redis提供了衰减因子server.lfu_decay_time，其单位为分钟，计算方法也很简单，如果一个key长时间没有访问那么它的计数器counter就要减少，减少的值由衰减因子来控制：

```
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;
}
```

默认为1的情况下也就是N分钟内没有访问，counter就要减N。

概率因子和衰减因子均可配置，推荐使用redis的默认值即可：

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

热点key发现

介绍完LFU算法，接下来就是我们关心的热点key发现机制。

其核心就是在每次对key进行读写访问时，更新LFU的24 bits域代表的访问时间和counter，这样每个key就可以获得正确的LFU值：

```
void updateLFU(robj *val) {
    unsigned long counter = LFUDecrAndReturn(val);
    counter = LFULogIncr(counter);
    val->lru = (LFUGetTimeInMinutes()<<8) | counter;
}
```

那么用户如何获取访问频率呢？redis提供了OBJECT FREQ子命令来获取LFU信息，但是要注意需要先把内存逐出策略设置为allkeys-lfu或者volatile-lfu，否则会返回错误：

```
127.0.0.1:6379> config get maxmemory-policy
1) "maxmemory-policy"
2) "noeviction"
127.0.0.1:6379> object freq counter:000000006889
(error) ERR An LFU maxmemory policy is not selected, access frequency not tracked. Please note that
when switching between policies at runtime LRU and LFU data will take some time to adjust.

127.0.0.1:6379> config set maxmemory-policy allkeys-lfu
OK
127.0.0.1:6379> object freq counter:000000006889
(integer) 3
```

使用scan命令遍历所有key，再通过OBJECT FREQ获取访问频率并排序，即可得到热点key。为了方便用户使用，redis 4.0.3同时也提供了redis-cli的热点key发现功能，执行redis-cli时加上--hotkeys选项即可，示例如下：

```
$./redis-cli --hotkeys

# Scanning the entire keyspace to find hot keys as well as
# average sizes per key type.  You can use -i 0.1 to sleep 0.1 sec
# per 100 SCAN commands (not usually needed).

[00.00%] Hot key 'counter:000000000002' found so far with counter 87
[00.00%] Hot key 'key:000000000001' found so far with counter 254
[00.00%] Hot key 'mylist' found so far with counter 107
[00.00%] Hot key 'key:000000000000' found so far with counter 254
[45.45%] Hot key 'counter:000000000001' found so far with counter 87
[45.45%] Hot key 'key:000000000002' found so far with counter 254
[45.45%] Hot key 'myset' found so far with counter 64
[45.45%] Hot key 'counter:000000000000' found so far with counter 93

-------- summary -------

Sampled 22 keys in the keyspace!
hot key found with counter: 254    keyname: key:000000000001
hot key found with counter: 254    keyname: key:000000000000
hot key found with counter: 254    keyname: key:000000000002
hot key found with counter: 107    keyname: mylist
hot key found with counter: 93    keyname: counter:000000000000
hot key found with counter: 87    keyname: counter:000000000002
hot key found with counter: 87    keyname: counter:000000000001
hot key found with counter: 64    keyname: myset
```

可以看到，排在前几位的即是热点key。