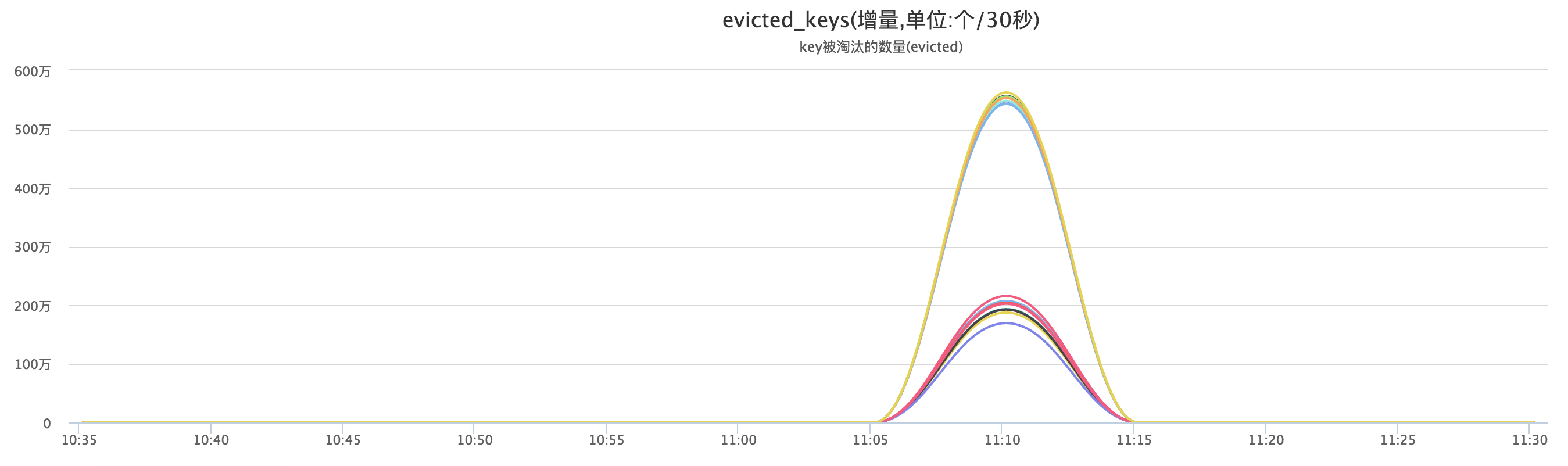
**背景**

Squirrel（松鼠）是京东技术团队基于Redis Cluster打造的缓存系统。经过不断的迭代研发，目前已形成一整套自动化运维体系：涵盖一键运维集群、细粒度的监控、支持自动扩缩容以及热点Key监控等完整的解决方案。同时服务端通过Docker进行部署，最大程度的提高运维的灵活性。分布式缓存Squirrel产品自2014年上线至今，已在京东内部广泛使用，存储容量超过80T，日均调用量也超过万亿次，逐步成为京东目前最主要的缓存系统之一。

随着使用的量和场景不断深入，Squirrel团队也不断发现Redis的若干"坑"和不足，因此也在持续的改进Redis以支撑内部快速发展的业务需求。本文尝试分享在运维过程中踩过的Redis Rehash机制的一些坑以及我们的解决方案，其中在高负载情况下物理机发生丢包的现象和解决方案已经写成博客。感兴趣的同学可以参考：[Redis 高负载下的中断优化](https://blog.csdn.net/u013380694/article/details/82915174)。

**案例**

**Redis 满容状态下由于Rehash导致大量Key驱逐**



我们先来看一张监控图（上图，我们线上真实案例），Redis在满容有驱逐策略的情况下，Master/Slave 均有大量的Key驱逐淘汰，导致Master/Slave 主从不一致。

**Root Cause 定位**

由于Slave内存区域比Master少一个repl-backlog buffer（线上一般配置为128M），正常情况下Master到达满容后根据驱逐策略淘汰Key并同步给Slave。所以Slave这种情况下不会因满容触发驱逐。

按照以往经验，排查思路主要聚焦在造成Slave内存陡增的问题上，包括客户端连接、输入/输出缓冲区、业务数据存取访问、网路抖动等导致Redis内存陡增的所有外部因素，通过Redis监控和业务链路监控均没有定位成功。

于是，通过梳理Redis源码，我们尝试将目光投向了Redis会占用内存开销的一个重要机制——Redis Rehash。

**Redis Rehash 内部实现**

在Redis中，键值对（Key-Value Pair）存储方式是由字典（Dict）保存的，而字典底层是通过哈希表来实现的。通过哈希表中的节点保存字典中的键值对。类似Java中的HashMap，将Key通过哈希函数映射到哈希表节点位置。

接下来我们一步步来分析Redis Dict Reash的机制和过程。

(1) Redis 哈希表结构体：

/\* hash表结构定义 \*/

typedef struct dictht {

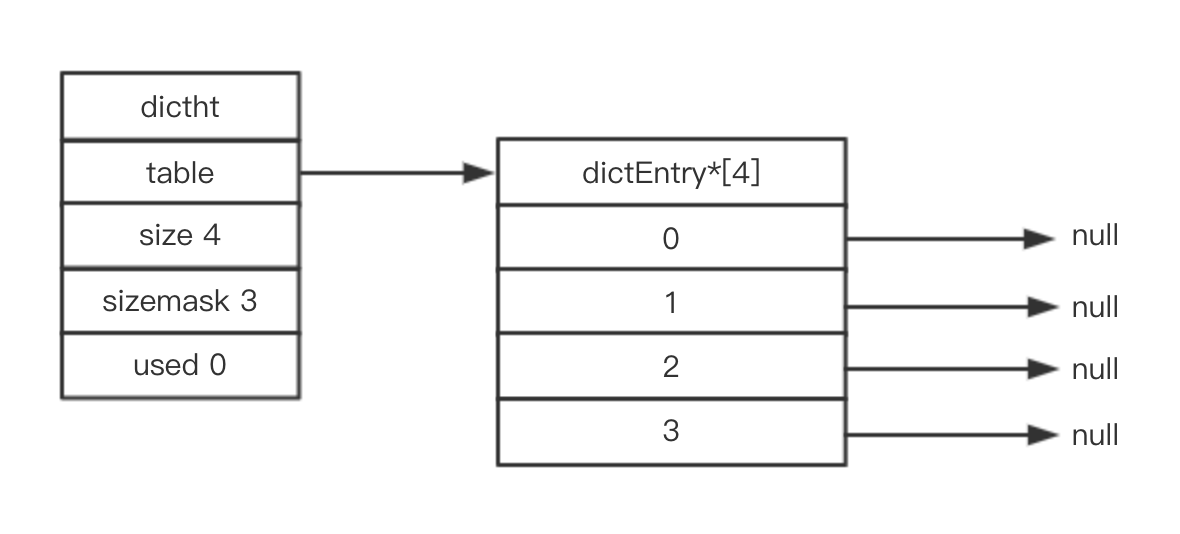
dictEntry \*\*table; // 哈希表数组

unsigned long size; // 哈希表的大小

unsigned long sizemask; // 哈希表大小掩码

unsigned long used; // 哈希表现有节点的数量

} dictht;

实体化一下，如下图所指一个大小为4的空哈希表（Redis默认初始化值为4）：  


(2) Redis 哈希桶  
Redis 哈希表中的table数组存放着哈希桶结构（dictEntry），里面就是Redis的键值对；类似Java实现的HashMap，Redis的dictEntry也是通过链表（next指针）方式来解决hash冲突：

/\* 哈希桶 \*/

typedef struct dictEntry {

void \*key; // 键定义

// 值定义

union {

void \*val; // 自定义类型

uint64\_t u64; // 无符号整形

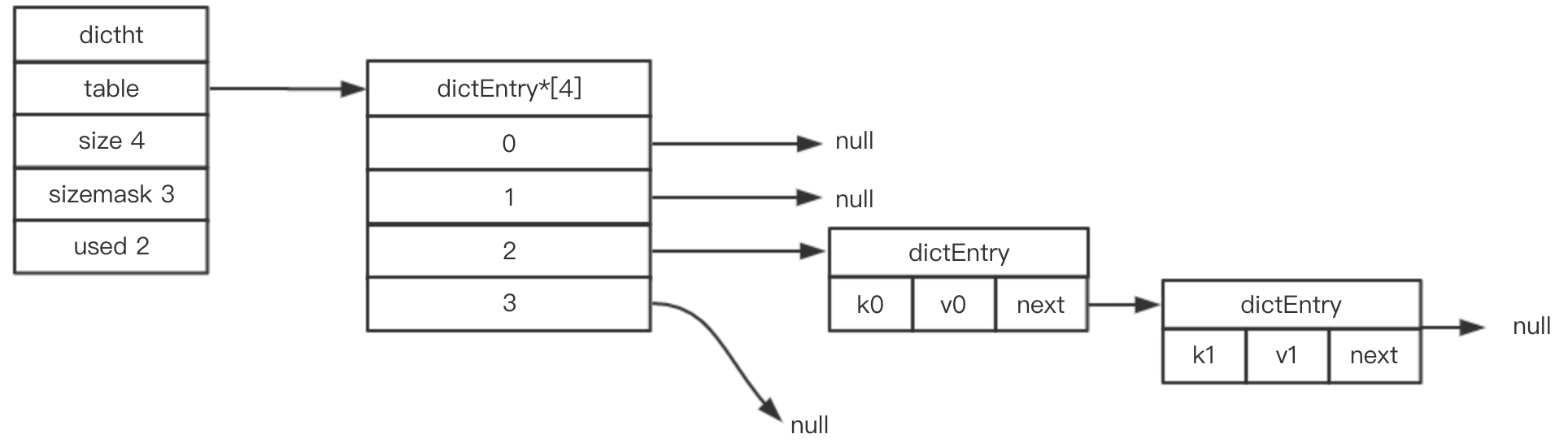
int64\_t s64; // 有符号整形

double d; // 浮点型

} v;

struct dictEntry \*next; //指向下一个哈希表节点

} dictEntry;



(3) 字典  
Redis Dict 中定义了两张哈希表，是为了后续字典的扩展作Rehash之用：

/\* 字典结构定义 \*/

typedef struct dict {

dictType \*type; // 字典类型

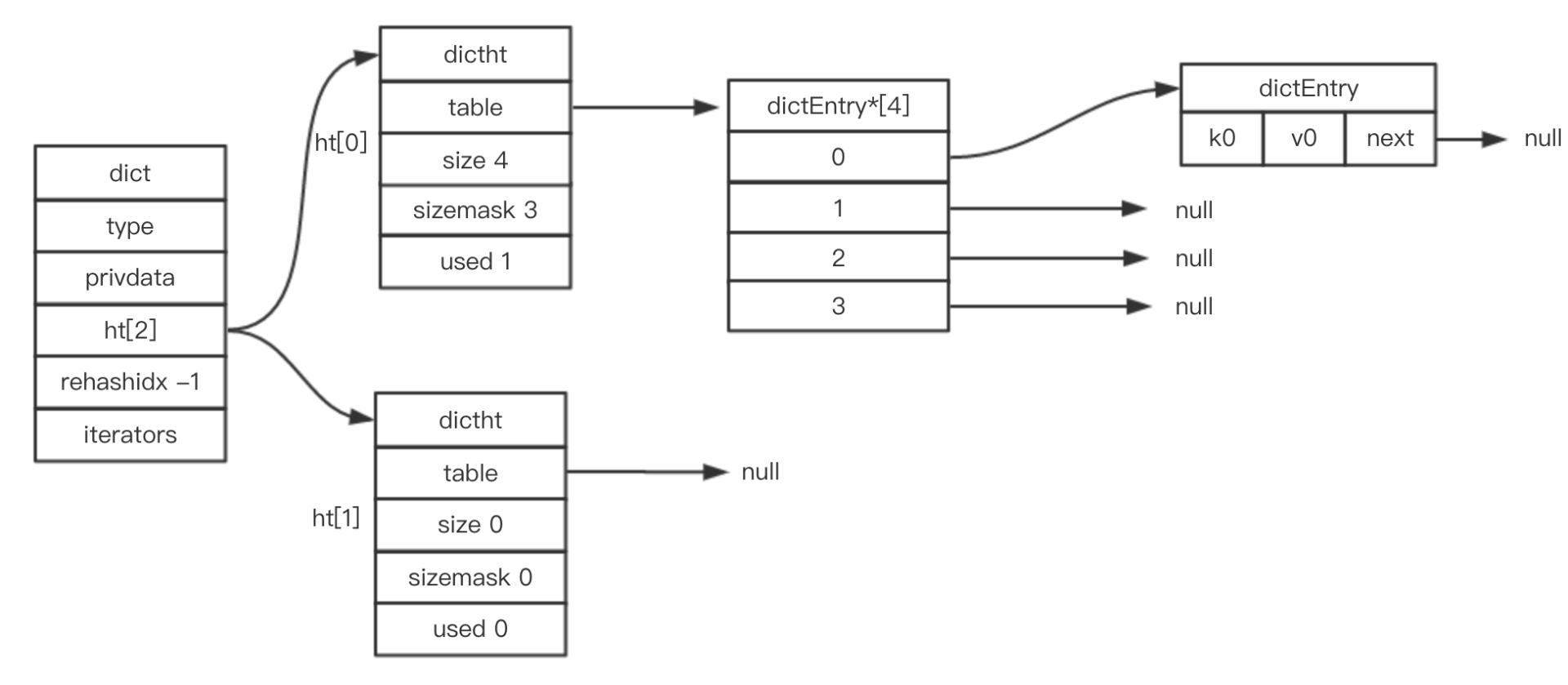
void \*privdata; // 私有数据

dictht ht[2]; // 哈希表[两个]

long rehashidx; // 记录rehash 进度的标志，值为-1表示rehash未进行

int iterators; // 当前正在迭代的迭代器数

} dict;



总结一下：

* 在Cluster模式下，一个Redis实例对应一个RedisDB(db0);
* 一个RedisDB对应一个Dict;
* 一个Dict对应2个Dictht，正常情况只用到ht[0]；ht[1] 在Rehash时使用。

如上，我们回顾了一下Redis KV存储的实现。（Redis内部还有其他结构体，由于跟Rehash不涉及，这里不再赘述）

我们知道当HashMap中由于Hash冲突（负载因子）超过某个阈值时，出于链表性能的考虑，会进行Resize的操作。Redis也一样【Redis中通过dictExpand()实现】，很多好的设计都相互借鉴和参考。我们看一下Redis中的实现方式：

/\* 根据相关触发条件扩展字典 \*/

static int \_dictExpandIfNeeded(dict \*d)

{

if (dictIsRehashing(d)) return DICT\_OK; // 如果正在进行Rehash，则直接返回

if (d->ht[0].size == 0) return dictExpand(d, DICT\_HT\_INITIAL\_SIZE); // 如果ht[0]字典为空，则创建并初始化ht[0]

/\* (ht[0].used/ht[0].size)>=1前提下，

当满足dict\_can\_resize=1或ht[0].used/t[0].size>5时，便对字典进行扩展 \*/

if (d->ht[0].used >= d->ht[0].size &&

(dict\_can\_resize ||

d->ht[0].used/d->ht[0].size > dict\_force\_resize\_ratio))

{

return dictExpand(d, d->ht[0].used\*2); // 扩展字典为原来的2倍

}

return DICT\_OK;

}

...

/\* 计算存储Key的bucket的位置 \*/

static int \_dictKeyIndex(dict \*d, const void \*key)

{

unsigned int h, idx, table;

dictEntry \*he;

/\* 检查是否需要扩展哈希表，不足则扩展 \*/

if (\_dictExpandIfNeeded(d) == DICT\_ERR)

return -1;

/\* 计算Key的哈希值 \*/

h = dictHashKey(d, key);

for (table = 0; table <= 1; table++) {

idx = h & d->ht[table].sizemask; //计算Key的bucket位置

/\* 检查节点上是否存在新增的Key \*/

he = d->ht[table].table[idx];

/\* 在节点链表检查 \*/

while(he) {

if (key==he->key || dictCompareKeys(d, key, he->key))

return -1;

he = he->next;

}

if (!dictIsRehashing(d)) break; // 扫完ht[0]后，如果哈希表不在rehashing，则无需再扫ht[1]

}

return idx;

}

...

/\* 将Key插入哈希表 \*/

dictEntry \*dictAddRaw(dict \*d, void \*key)

{

int index;

dictEntry \*entry;

dictht \*ht;

if (dictIsRehashing(d)) \_dictRehashStep(d); // 如果哈希表在rehashing，则执行单步rehash

/\* 调用\_dictKeyIndex() 检查键是否存在，如果存在则返回NULL \*/

if ((index = \_dictKeyIndex(d, key)) == -1)

return NULL;

ht = dictIsRehashing(d) ? &d->ht[1] : &d->ht[0];

entry = zmalloc(sizeof(\*entry)); // 为新增的节点分配内存

entry->next = ht->table[index]; // 将节点插入链表表头

ht->table[index] = entry; // 更新节点和桶信息

ht->used++; // 更新ht

/\* 设置新节点的键 \*/

dictSetKey(d, entry, key);

return entry;

}

...

/\* 添加新键值对 \*/

int dictAdd(dict \*d, void \*key, void \*val)

{

dictEntry \*entry = dictAddRaw(d,key); // 添加新键

if (!entry) return DICT\_ERR; // 如果键存在，则返回失败

dictSetVal(d, entry, val); // 键不存在，则设置节点值

return DICT\_OK;

}

继续dictExpand的源码实现：

int dictExpand(dict \*d, unsigned long size)

{

dictht n; // 新哈希表

unsigned long realsize = \_dictNextPower(size); // 计算扩展或缩放新哈希表的大小(调用下面函数\_dictNextPower())

/\* 如果正在rehash或者新哈希表的大小小于现已使用，则返回error \*/

if (dictIsRehashing(d) || d->ht[0].used > size)

return DICT\_ERR;

/\* 如果计算出哈希表size与现哈希表大小一样，也返回error \*/

if (realsize == d->ht[0].size) return DICT\_ERR;

/\* 初始化新哈希表 \*/

n.size = realsize;

n.sizemask = realsize-1;

n.table = zcalloc(realsize\*sizeof(dictEntry\*)); // 为table指向dictEntry 分配内存

n.used = 0;

/\* 如果ht[0] 为空，则初始化ht[0]为当前键值对的哈希表 \*/

if (d->ht[0].table == NULL) {

d->ht[0] = n;

return DICT\_OK;

}

/\* 如果ht[0]不为空，则初始化ht[1]为当前键值对的哈希表，并开启渐进式rehash模式 \*/

d->ht[1] = n;

d->rehashidx = 0;

return DICT\_OK;

}

...

static unsigned long \_dictNextPower(unsigned long size) {

unsigned long i = DICT\_HT\_INITIAL\_SIZE; // 哈希表的初始值：4

if (size >= LONG\_MAX) return LONG\_MAX;

/\* 计算新哈希表的大小：第一个大于等于size的2的N 次方的数值 \*/

while(1) {

if (i >= size)

return i;

i \*= 2;

}

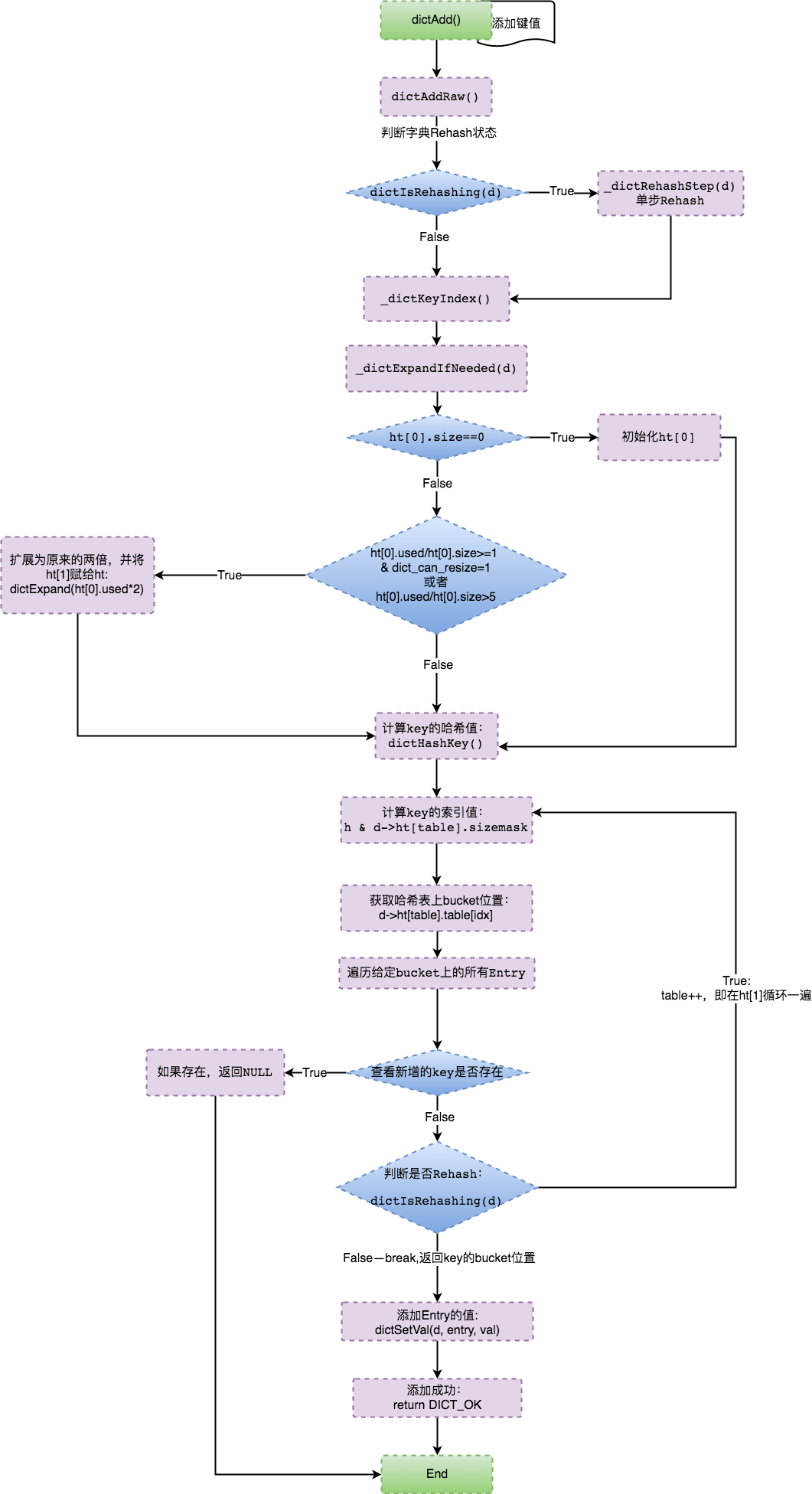
}

总结一下具体逻辑实现：  
v |= ~m1;

v = rev(v);

v++;

v = rev(v);



可以确认当Redis Hash冲突到达某个条件时就会触发dictExpand()函数来扩展HashTable。

DICT\_HT\_INITIAL\_SIZE初始化值为4，通过上述表达式，取当4\*2^n >= ht[0].used\*2的值作为字典扩展的size大小。即为：ht[1].size 的值等于第一个大于等于ht[0].used\*2的2^n的数值。

Redis通过dictCreate()创建词典，在初始化中，table指针为Null，所以两个哈希表ht[0].table和ht[1].table都未真正分配内存空间。只有在dictExpand()字典扩展时才给table分配指向dictEntry的内存。

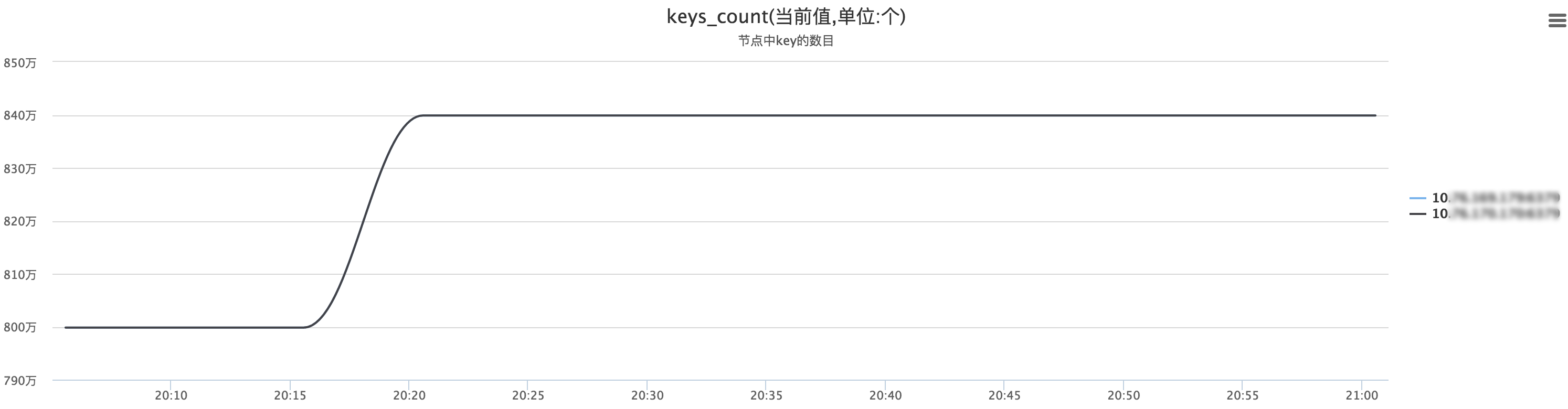
由上可知，当Redis触发Resize后，就会动态分配一块内存，最终由ht[1].table指向，动态分配的内存大小为：realsize\*sizeof(dictEntry\*)，table指向dictEntry\*的一个指针，大小为8bytes（64位OS），即ht[1].table需分配的内存大小为：8\*2\*2^n （n大于等于2）。

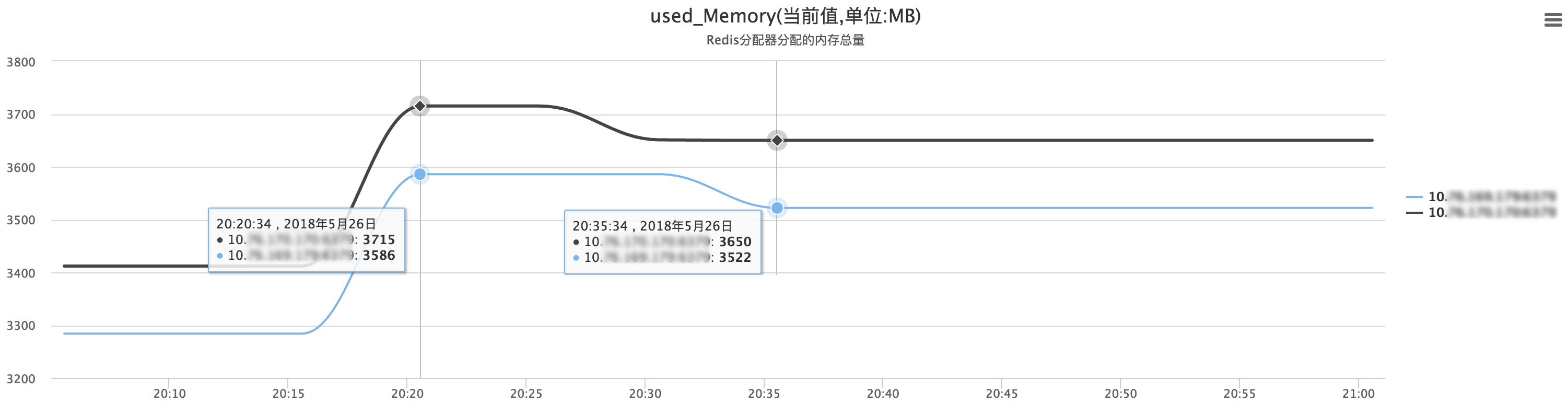
梳理一下哈希表大小和内存申请大小的对应关系：

| **ht[0].size** | **触发Resize时，ht[1]需分配的内存** |
| --- | --- |
| 4 | 64bytes |
| 8 | 128bytes |
| 16 | 256bytes |
| ... | ... |
| 65536 | 1024K |
| ... | ... |
| 8388608 | 128M |
| 16777216 | 256M |
| 33554432 | 512M |
| 67108864 | 1024M |
| ... | ... |

**复现验证**

我们通过测试环境数据来验证一下，当Redis Rehash过程中，内存真正的占用情况。





上述两幅图中，Redis Key个数突破Redis Resize的临界点，当Key总数稳定且Rehash完成后，Redis内存（Slave）从3586M降至为3522M：3586-3522=64M。即验证上述Redis在Resize至完成的中间状态，会维持一段时间内存消耗，且占用内存的值为上文列表相应的内存空间。

进一步观察一下Redis内部统计信息：

/\* Redis节点800万左右Key时候的Dict状态信息:只有ht[0]信息。\*/

"[Dictionary HT]

Hash table 0 stats (main hash table):

table size: 8388608

number of elements: 8003582

different slots: 5156314

max chain length: 9

avg chain length (counted): 1.55

avg chain length (computed): 1.55

Chain length distribution:

0: 3232294 (38.53%)

1: 3080243 (36.72%)

2: 1471920 (17.55%)

3: 466676 (5.56%)

4: 112320 (1.34%)

5: 21301 (0.25%)

6: 3361 (0.04%)

7: 427 (0.01%)

8: 63 (0.00%)

9: 3 (0.00%)

"

/\* Redis节点840万左右Key时候的Dict状态信息正在Rehasing中，包含了ht[0]和ht[1]信息。\*/

"[Dictionary HT]

[Dictionary HT]

Hash table 0 stats (main hash table):

table size: 8388608

number of elements: 8019739

different slots: 5067892

max chain length: 9

avg chain length (counted): 1.58

avg chain length (computed): 1.58

Chain length distribution:

0: 3320716 (39.59%)

1: 2948053 (35.14%)

2: 1475756 (17.59%)

3: 491069 (5.85%)

4: 123594 (1.47%)

5: 24650 (0.29%)

6: 4135 (0.05%)

7: 553 (0.01%)

8: 78 (0.00%)

9: 4 (0.00%)

Hash table 1 stats (rehashing target):

table size: 16777216

number of elements: 384321

different slots: 305472

max chain length: 6

avg chain length (counted): 1.26

avg chain length (computed): 1.26

Chain length distribution:

0: 16471744 (98.18%)

1: 238752 (1.42%)

2: 56041 (0.33%)

3: 9378 (0.06%)

4: 1167 (0.01%)

5: 119 (0.00%)

6: 15 (0.00%)

"

/\* Redis节点840万左右Key时候的Dict状态信息(Rehash完成后);ht[0].size从8388608扩展到了16777216。\*/

"[Dictionary HT]

Hash table 0 stats (main hash table):

table size: 16777216

number of elements: 8404060

different slots: 6609691

max chain length: 7

avg chain length (counted): 1.27

avg chain length (computed): 1.27

Chain length distribution:

0: 10167525 (60.60%)

1: 5091002 (30.34%)

2: 1275938 (7.61%)

3: 213024 (1.27%)

4: 26812 (0.16%)

5: 2653 (0.02%)

6: 237 (0.00%)

7: 25 (0.00%)

"

经过Redis Rehash内部机制的深入、Redis状态监控和Redis内部统计信息，我们可以得出结论：  
**当Redis 节点中的Key总量到达临界点后，Redis就会触发Dict的扩展，进行Rehash。申请扩展后相应的内存空间大小。**

如上，Redis在满容驱逐状态下，Redis Rehash是导致Redis Master和Slave大量触发驱逐淘汰的根本原因。

除了导致满容驱逐淘汰，Redis Rehash还会引起其他一些问题：

* 在tablesize级别与现有Keys数量不在同一个区间内，主从切换后，由于Redis全量同步，从库tablesize降为与现有Key匹配值，导致内存倾斜；
* Redis Cluster下的某个分片由于Key数量相对较多提前Resize，导致集群分片内存不均。  
  等等...

**Redis Rehash机制优化**

那么针对在Redis满容驱逐状态下，如何避免因Rehash而导致Redis抖动的这种问题。

* 我们在Redis Rehash源码实现的逻辑上，加上了一个判断条件，如果现有的剩余内存不够触发Rehash操作所需申请的内存大小，即不进行Resize操作；
* 通过提前运营进行规避，比如容量预估时将Rehash占用的内存考虑在内，或者通过监控定时扩容。

Redis Rehash机制除了会影响上述内存管理和使用外，也会影响Redis其他内部与之相关联的功能模块。下面我们分享一下由于Rehash机制而踩到的第二个坑。

**Redis使用Scan清理Key由于Rehash导致清理数据不彻底**

Squirrel平台提供给业务清理Key的API后台逻辑，是通过Scan来实现的。实际线上运行效果并不是每次都能完全清理干净。即通过Scan扫描清理相匹配的Key，较低频率会有遗漏、Key未被全部清理掉的现象。有了前几次的相关经验后，我们直接从原理入手。

**Scan原理**

为了高效地匹配出数据库中所有符合给定模式的Key，Redis提供了Scan命令。该命令会在每次调用的时候返回符合规则的部分Key以及一个游标值Cursor（初始值使用0），使用每次返回Cursor不断迭代，直到Cursor的返回值为0代表遍历结束。

Redis官方定义Scan特点如下：

1. 整个遍历从开始到结束期间， 一直存在于Redis数据集内的且符合匹配模式的所有Key都会被返回；
2. 如果发生了rehash，同一个元素可能会被返回多次，遍历过程中新增或者删除的Key可能会被返回，也可能不会。

**具体实现**

上述提及Redis的Keys是以Dict方式来存储的，正常只要一次遍历Dict中所有Hash桶就可以完整扫描出所有Key。但是在实际使用中，Redis Dict是有状态的，会随着Key的增删不断变化。

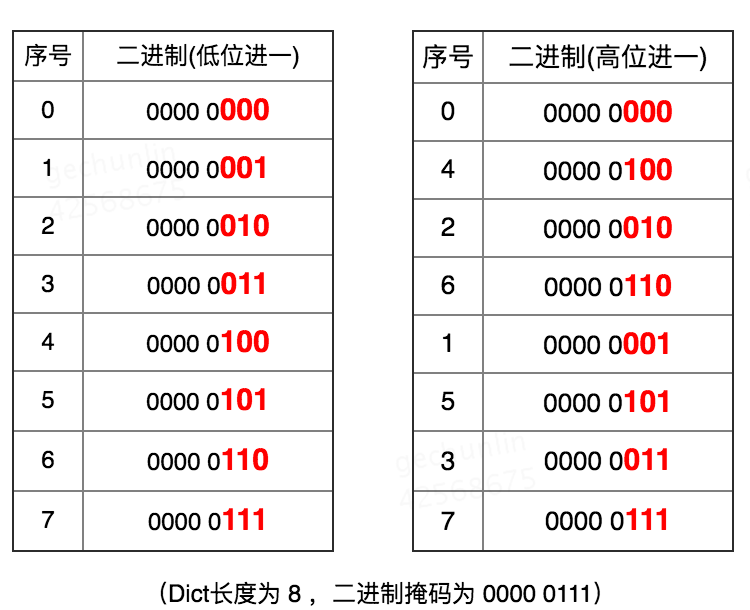
接下来根据Dict四种状态来分析一下Scan的不同实现。

Dict的四种状态场景：

1. 字典tablesize保持不变，没有扩缩容；
2. 字典Resize，Dict扩大了（完成状态）；
3. 字典Resize，Dict缩小了（完成状态）；
4. 字典正在Rehashing（扩展或收缩）。

(1) 字典tablesize保持不变，在Redis Dict稳定的状态下，直接顺序遍历即可；  
(2) 字典Resize，Dict扩大了，如果还是按照顺序遍历，就会导致扫描大量重复Key。比如字典tablesize从8变成了16，假设之前访问的是3号桶，那么表扩展后则是继续访问4~15号桶；但是，原先的0~3号桶中的数据在Dict长度变大后被迁移到8~11号桶中，因此，遍历8~11号桶的时候会有大量的重复Key被返回；  
(3) 字典Resize，Dict缩小了，如果还是按照顺序遍历，就会导致大量的Key被遗漏。比如字典tablesize从8变成了4，假设当前访问的是3号桶，那么下一次则会直接返回遍历结束了；但是之前4~7号桶中的数据在缩容后迁移带可0~3号桶中，因此这部分Key就无法扫描到；  
(4) 字典正在Rehashing，这种情况如(2)和(3)情况一下，要么大量重复扫描、要么遗漏很多Key。

那么在Dict非稳定状态，即发生Rehash的情况下，Scan要如何保证原有的Key都能遍历出来，又尽少可能重复扫描呢？Redis Scan通过Hash桶掩码的高位顺序访问来解决。

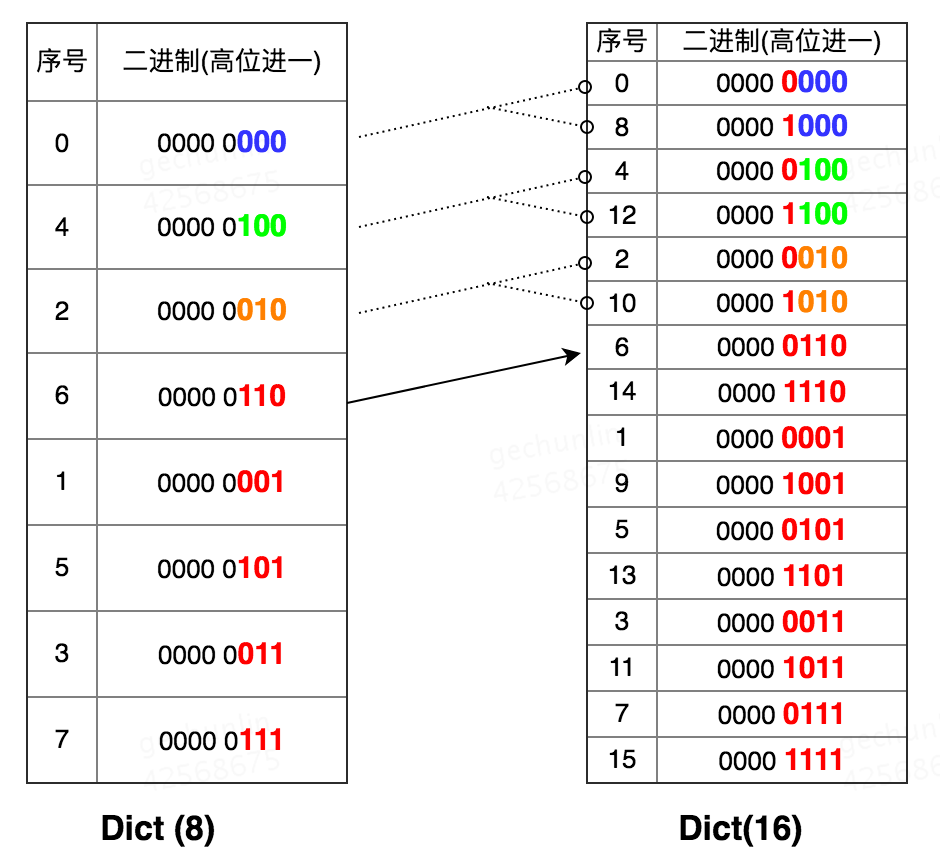


高位顺序访问即按照Dict sizemask（掩码），在有效位（上图中Dict sizemask为3）上从高位开始加一枚举；低位则按照有效位的低位逐步加一访问。  
低位序：0→1→2→3→4→5→6→7  
高位序：0→4→2→6→1→5→3→7

Scan采用高位序访问的原因，就是为了实现Redis Dict在Rehash时尽可能少重复扫描返回Key。

举个例子，如果Dict的tablesize从8扩展到了16，梳理一下Scan扫描方式:

1. Dict(8) 从Cursor 0开始扫描；
2. 准备扫描Cursor 6时发生Resize，扩展为之前的2倍，并完成Rehash；
3. 客户端这时开始从Dict(16)的Cursor 6继续迭代；
4. 这时按照 6→14→1→9→5→13→3→11→7→15 Scan完成。



可以看出，高位序Scan在Dict Rehash时即可以避免重复遍历，又能完整返回原始的所有Key。同理，字典缩容时也一样，字典缩容可以看出是反向扩容。

上述是Scan的理论基础，我们看一下Redis源码如何实现。

(1) 非Rehashing 状态下的实现：

if (!dictIsRehashing(d)) { // 判断是否正在rehashing，如果不在则只有ht[0]

t0 = &(d->ht[0]); // ht[0]

m0 = t0->sizemask; // 掩码

/\* Emit entries at cursor \*/

de = t0->table[v & m0]; // 目标桶

while (de) {

fn(privdata, de);

de = de->next; // 遍历桶中所有节点，并通过回调函数fn()返回

}

...

/\* 反向二进制迭代算法具体实现逻辑——游标实现的精髓 \*/

/\* Set unmasked bits so incrementing the reversed cursor

\* operates on the masked bits of the smaller table \*/

v |= ~m0;

/\* Increment the reverse cursor \*/

v = rev(v);

v++;

v = rev(v);

return v;

}

源码中Redis将Cursor的计算通过Reverse Binary Iteration（反向二进制迭代算法）来实现上述的高位序扫描方式。

(2) Rehashing 状态下的实现：

...

else { // 否则说明正在rehashing，就存在两个哈希表ht[0]、ht[1]

t0 = &d->ht[0];

t1 = &d->ht[1]; // 指向两个哈希表

/\* Make sure t0 is the smaller and t1 is the bigger table \*/

if (t0->size > t1->size) { 确保t0小于t1

t0 = &d->ht[1];

t1 = &d->ht[0];

}

m0 = t0->sizemask;

m1 = t1->sizemask; // 相对应的掩码

/\* Emit entries at cursor \*/

/\* 迭代(小表)t0桶中的所有节点 \*/

de = t0->table[v & m0];

while (de) {

fn(privdata, de);

de = de->next;

}

/\* Iterate over indices in larger table that are the expansion

\* of the index pointed to by the cursor in the smaller table \*/

/\* \*/

do {

/\* Emit entries at cursor \*/

/\* 迭代(大表)t1 中所有节点，循环迭代，会把小表没有覆盖的slot全部扫描一遍 \*/

de = t1->table[v & m1];

while (de) {

fn(privdata, de);

de = de->next;

}

/\* Increment bits not covered by the smaller mask \*/

v = (((v | m0) + 1) & ~m0) | (v & m0);

/\* Continue while bits covered by mask difference is non-zero \*/

} while (v & (m0 ^ m1));

}

/\* Set unmasked bits so incrementing the reversed cursor

\* operates on the masked bits of the smaller table \*/

v |= ~m0;

/\* Increment the reverse cursor \*/

v = rev(v);

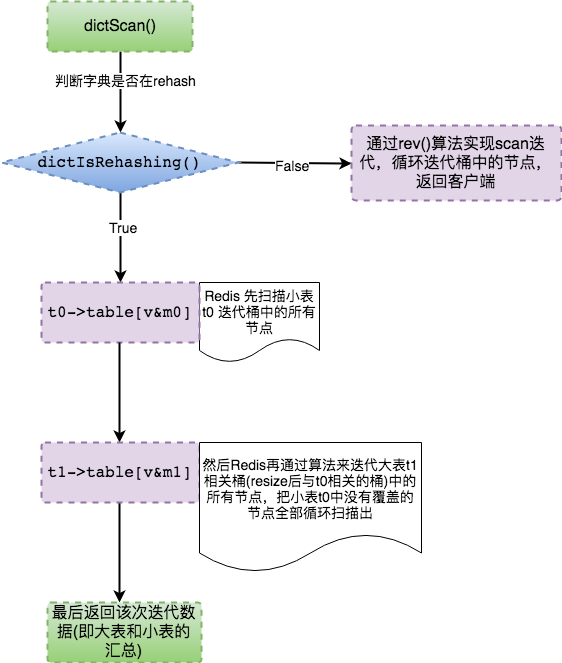
v++;

v = rev(v);

return v;

如上Rehashing时，Redis 通过else分支实现该过程中对两张Hash表进行扫描访问。

梳理一下逻辑流程：



Redis在处理dictScan()时，上面细分的四个场景的实现分成了两个逻辑：

1. 此时不在Rehashing的状态：  
   这种状态，即Dict是静止的。针对这种状态下的上述三种场景，Redis采用上述的Reverse Binary Iteration（反向二进制迭代算法）：  
   Ⅰ. 首先对游标（Cursor）二进制位翻转；  
   Ⅱ. 再对翻转后的值加1；  
   Ⅲ. 最后再次对Ⅱ的结果进行翻转。

通过穷举高位，依次向低位推进的方式（即高位序访问的实现）来确保所有元素都会被遍历到。

这种算法已经尽可能减少重复元素的返回，但是实际实现和逻辑中还是会有可能存在重复返回，比如在Dict缩容时，高位合并到低位桶中，低位桶中的元素就会被重复取出。

1. 正在Rehashing的状态：  
   Redis在Rehashing状态的时候，dictScan()实现通过一次性扫描现有的两种字典表，避免中间状态无法维护。  
   具体实现就是在遍历完小表Cursor位置后，将小表Cursor位置可能Rehash到的大表所有位置全部遍历一遍，然后再返回遍历元素和下一个小表遍历位置。

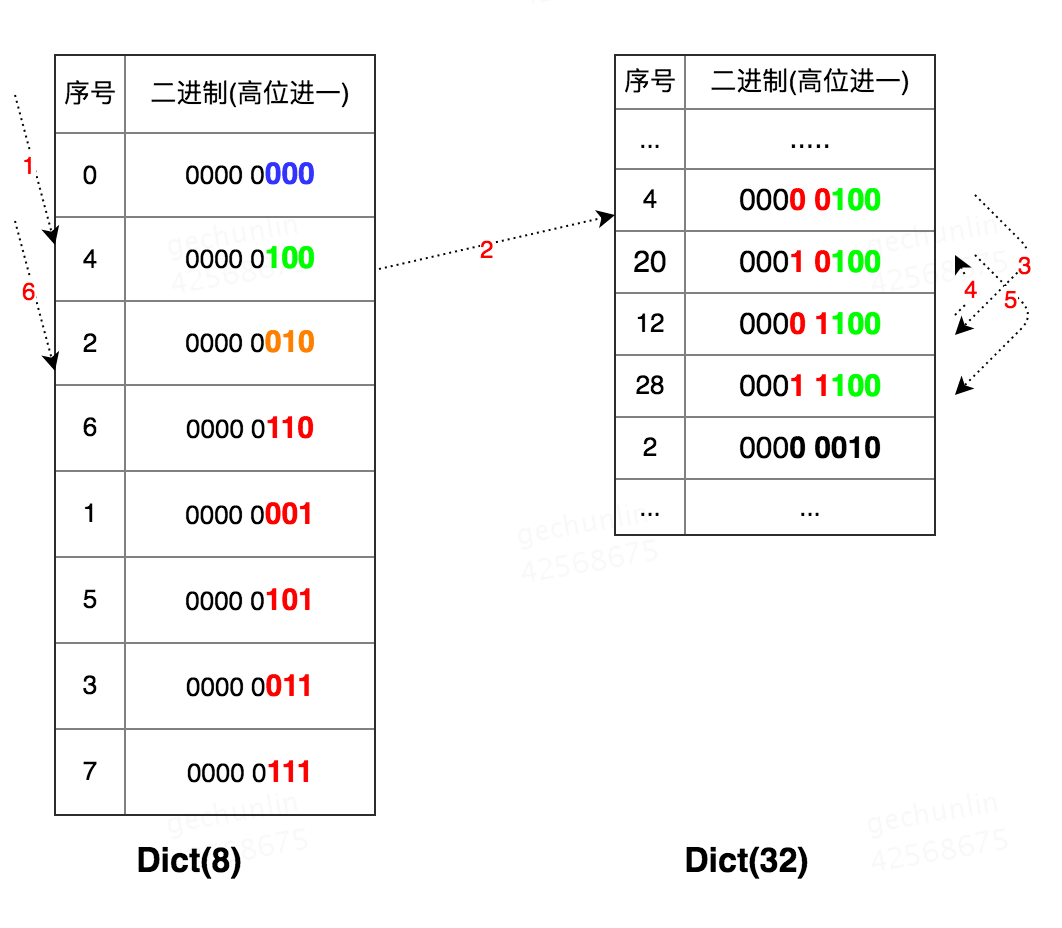
**Root Cause 定位**

Rehashing状态时，游标迭代主要逻辑代码实现：

/\* Increment bits not covered by the smaller mask \*/

v = (((v | m0) + 1) & ~m0) | (v & m0); //BUG

Ⅰ. v低位加1向高位进位；  
Ⅱ. 去掉v最前面和最后面的部分，只保留v相较于m0的高位部分；  
Ⅲ. 保留v的低位，高位不断加1。即低位不变，高位不断加1，实现了小表到大表桶的关联。

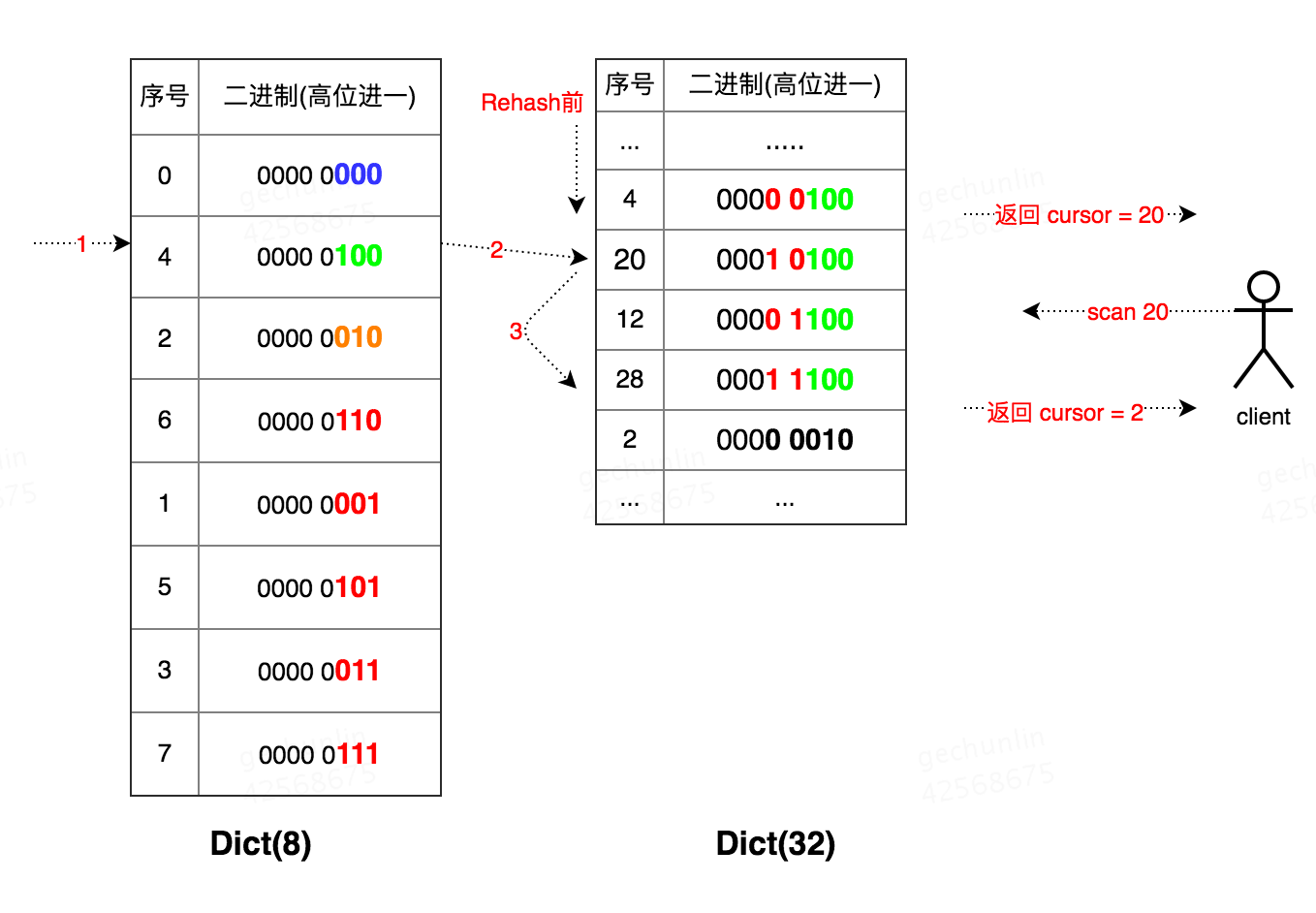


举个例子，如果Dict的tablesize从8扩展到了32，梳理一下Scan扫描方式:

1. Dict(8) 从Cursor 0开始扫描；
2. 准备扫描Cursor 4时发生Resize，扩展为之前的4倍，Rehashing；
3. 客户端先访问Dict(8)中的4号桶；
4. 然后再到Dict(32)上访问:4→12→20→28。

这里可以看到大表的相关桶的顺序并非是按照之前所述的二进制高位序，实际上是按照低位序来遍历大表中高出小表的有效位。

大表t1高位都是向低位加1计算得出的，扫描的顺序却是从低位加1，向高位进位。Redis针对Rehashing时这种逻辑实现在扩容时是可以运行正常的，但是在缩容时高位序和低位序的遍历在大小表上的混用在一定条件下会出现问题。



再次示例，Dict的tablesize从32缩容到8：

1. Dict(32) 从Cursor 0开始扫描；
2. 准备扫描Cursor 20时发生Resize，缩容至原来的四分之一即tablesize为8，Rehashing；
3. 客户端发起Cursor 20,首先访问Dict(8)中的4号桶；
4. 再到Dict(32)上访问:20→28;
5. 最后返回Cursor = 2。

可以看出大表中的12号桶没有被访问到，即遍历大表时，按照低位序访问会遗漏对某些桶的访问。

上述这种情况发生需要具备一定的条件：

1. 在Dict缩容Rehash时Scan;
2. Dict缩容至至少原Dict tablesize的四分之一，只有在这种情况下，大表相对小表的有效位才会高出二位以上，从而触发跳过某个桶的情况；
3. 如果在Rehash开始前返回的Cursor是在小表能表示的范围内（即不超过7），那么在进行高位有效位的加一操作时，必然都是从0开始计算，每次加一也必然能够访问的全所有的相关桶；如果在Rehash开始前返回的cursor不在小表能表示的范围内（比如20），那么在进行高位有效位加一操作的时候，就有可能跳过 ，或者重复访问某些桶的情况。

可见，只有满足上述三种情况才会发生Scan遍历过程中漏掉了一些Key的情况。在执行清理Key的时候，如果清理的Key数量很大，导致了Redis内部的Hash表缩容至少原Dict tablesize的四分之一，就可能存在一些Key被漏掉的风险。

**Scan源码优化**

修复逻辑就是全部都从高位开始增加进行遍历，即大小表都使用高位序访问，修复源码如下：

unsigned long dictScan(dict \*d,

unsigned long v,

dictScanFunction \*fn,

dictScanBucketFunction\* bucketfn,

void \*privdata)

{

dictht \*t0, \*t1;

const dictEntry \*de, \*next;

unsigned long m0, m1;

if (dictSize(d) == 0) return 0;

if (!dictIsRehashing(d)) {

t0 = &(d->ht[0]);

m0 = t0->sizemask;

/\* Emit entries at cursor \*/

if (bucketfn) bucketfn(privdata, &t0->table[v & m0]);

de = t0->table[v & m0];

while (de) {

next = de->next;

fn(privdata, de);

de = next;

}

/\* Set unmasked bits so incrementing the reversed cursor

\* operates on the masked bits \*/

v |= ~m0;

/\* Increment the reverse cursor \*/

v = rev(v);

v++;

v = rev(v);

} else {

t0 = &d->ht[0];

t1 = &d->ht[1];

/\* Make sure t0 is the smaller and t1 is the bigger table \*/

if (t0->size > t1->size) {

t0 = &d->ht[1];

t1 = &d->ht[0];

}

m0 = t0->sizemask;

m1 = t1->sizemask;

/\* Emit entries at cursor \*/

if (bucketfn) bucketfn(privdata, &t0->table[v & m0]);

de = t0->table[v & m0];

while (de) {

next = de->next;

fn(privdata, de);

de = next;

}

/\* Iterate over indices in larger table that are the expansion

\* of the index pointed to by the cursor in the smaller table \*/

do {

/\* Emit entries at cursor \*/

if (bucketfn) bucketfn(privdata, &t1->table[v & m1]);

de = t1->table[v & m1];

while (de) {

next = de->next;

fn(privdata, de);

de = next;

}

/\* Increment the reverse cursor not covered by the smaller mask.\*/

v |= ~m1;

v = rev(v);

v++;

v = rev(v);

/\* Continue while bits covered by mask difference is non-zero \*/

} while (v & (m0 ^ m1));

}

return v;

}

我们团队已经将此PR Push到Redis官方：[Fix dictScan(): It can't scan all buckets when dict is shrinking](https://github.com/antirez/redis/pull/4907)，并已经被官方Merge。

至此，基于Redis Rehash以及Scan实现中涉及Rehash的两个机制已经基本了解和优化完成。

**总结**

本文主要阐述了因Redis的Rehash机制踩到的两个坑，从现象到原理进行了详细的介绍。这里简单总结一下，第一个案例会造成线上集群进行大量淘汰，而且产生主从不一致的情况，在业务层面也会发生大量超时，影响业务可用性，问题严重，非常值得大家关注；第二个案例会造成数据清理无法完全清理，但是可以再利用Scan清理一遍也能够清理完毕。  
注：本文中源码基于Redis 3.2.8。