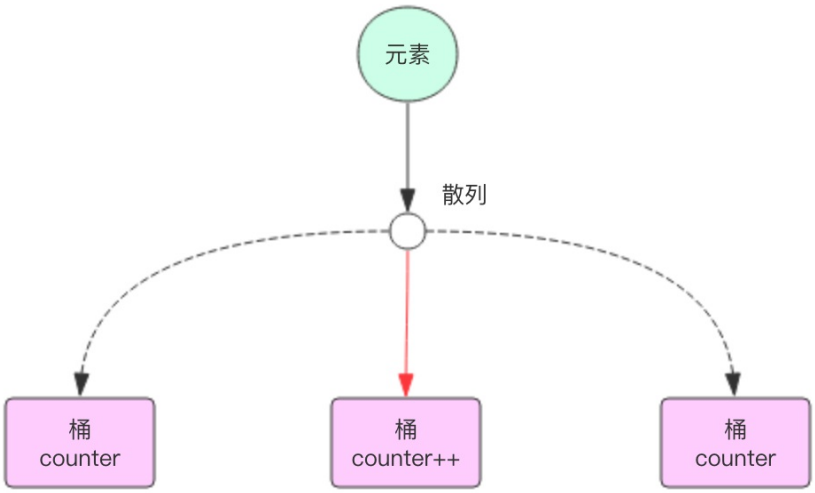
源码 11：见缝插针 —— 探索 HyperLogLog 内部

HyperLogLog算法是一种非常巧妙的近似统计海量去重元素数量的算法。它内部维护了 16384 个桶（bucket）来记录各自桶的元素数量。当一个元素到来时，它会散列到其中一个桶，以一定的概率影响这个桶的计数值。因为是概率算法，所以单个桶的计数值并不准确，但是将所有的桶计数值进行调合均值累加起来，结果就会非常接近真实的计数值。



为了便于理解HyperLogLog算法，我们先简化它的计数逻辑。因为是去重计数，如果是准确的去重，肯定需要用到 set 集合，使用集合来记录所有的元素，然后使用 scard 指令来获取集合大小就可以得到总的计数。因为元素特别多，单个集合会特别大，所以将集合打散成 16384 个小集合。当元素到来时，通过 hash 算法将这个元素分派到其中的一个小集合存储，同样的元素总是会散列到同样的小集合。这样总的计数就是所有小集合大小的总和。使用这种方式精确计数除了可以增加元素外，还可以减少元素。

用 Python 代码描述如下

```python
# coding:utf-8
import hashlib

class ExactlyCounter:

    def __init__(self):
        # 先分配16384个空集合
        self.buckets = []
        for i in range(16384):
            self.buckets.append(set([]))
        # 使用md5哈希算法
        self.hash = lambda x:
int(hashlib.md5(x).hexdigest(), 16)
        self.count = 0

    def add(self, element):
        h = self.hash(element)
        idx = h % len(self.buckets)
        bucket = self.buckets[idx]
        old_len = len(bucket)
        bucket.add(element)
        if len(bucket) > old_len:
            # 如果数量变化了，总数就+1
            self.count += 1

    def remove(self, element):
        h = self.hash(element)
        idx = h % len(self.buckets)
        bucket = self.buckets[idx]
        old_len = len(bucket)
        bucket.remove(element)
        if len(bucket) < old_len:
            # 如果数量变化了，总数-1
```

```
            self.count -= 1


if __name__ == '__main__':
    c = ExactlyCounter()
    for i in range(100000):
        c.add("element_%d" % i)
    print c.count
    for i in range(100000):
        c.remove("element_%d" % i)
    print c.count
```

集合打散并没有什么明显好处，因为总的内存占用并没有减少。HyperLogLog肯定不是这个算法，它需要对这个小集合进行优化，压缩它的存储空间，让它的内存变得非常微小。HyperLogLog算法中每个桶所占用的空间实际上只有 6 个 bit，这 6 个 bit 自然是无法容纳桶中所有元素的，它记录的是桶中元素数量的对数值。

为了说明这个对数值具体是个什么东西，我们先来考虑一个小问题。一个随机的整数值，这个整数的尾部有一个 0 的概率是 50%，要么是 0 要么是 1。同样，尾部有两个 0 的概率是 25%，有三个零的概率是 12.5%，以此类推，有 k 个 0 的概率是 2^(-k)。如果我们随机出了很多整数，整数的数量我们并不知道，但是我们记录了整数尾部连续 0 的最大数量 K。我们就可以通过这个 K 来近似推断出整数的数量，这个数量就是 2^K。

当然结果是非常不准确的，因为可能接下来你随机了非常多的整数，但是末尾连续零的最大数量 K 没有变化，但是估计值还是 2^K。你也许会想到要是这个 K 是个浮点数就好了，每次随机一个新元素，它都可以稍微往上涨一点点，那么估计值应该会准确很多。

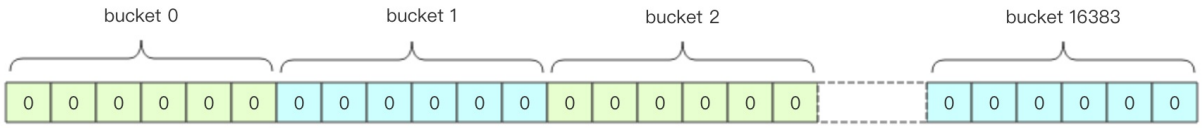
HyperLogLog通过分配 16384 个桶，然后对所有的桶的最大数量 K 进行调合平均来得到一个平均的末尾零最大数量 K# ，K# 是一个浮点数，使用平均后的 2^K# 来估计元素的总量相对而言就会准确

很多。不过这只是简化算法，真实的算法还有很多修正因子，因为涉及到的数学理论知识过于繁多，这里就不再精确描述。

下面我们看看Redis HyperLogLog 算法的具体实现。我们知道一个HyperLogLog实际占用的空间大约是 13684 * 6bit / 8 = 12k 字节。但是在计数比较小的时候，大多数桶的计数值都是零。如果 12k 字节里面太多的字节都是零，那么这个空间是可以适当节约一下的。Redis 在计数值比较小的情况下采用了稀疏存储，稀疏存储的空间占用远远小于 12k 字节。相对于稀疏存储的就是密集存储，密集存储会恒定占用 12k 字节。

密集存储结构

不论是稀疏存储还是密集存储，Redis 内部都是使用字符串位图来存储 HyperLogLog 所有桶的计数值。密集存储的结构非常简单，就是连续 16384 个 6bit 串成的字符串位图。

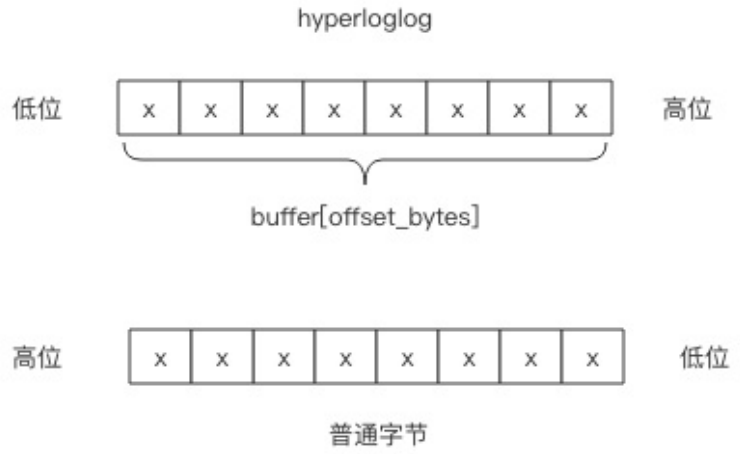


那么给定一个桶编号，如何获取它的 6bit 计数值呢？这 6bit 可能在一个字节内部，也可能会跨越字节边界。我们需要对这一个或者两个字节进行适当的移位拼接才可以得到计数值。

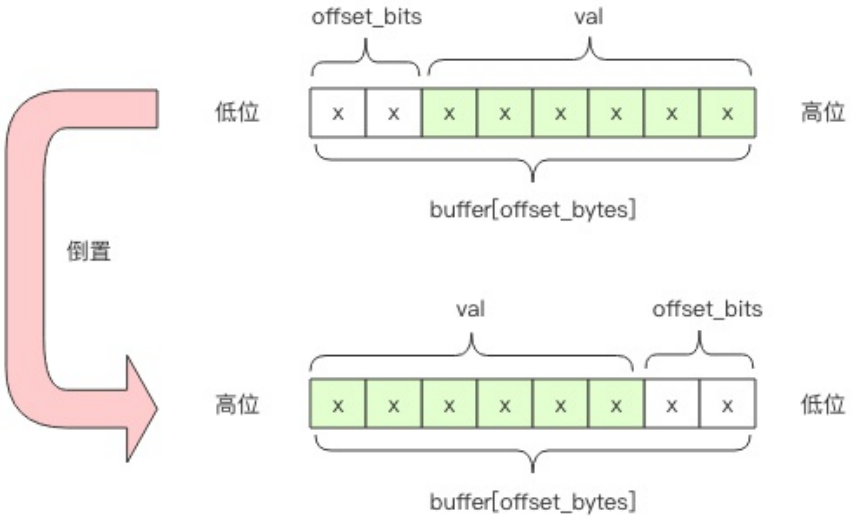
假设桶的编号为idx，这个 6bit 计数值的起始字节位置偏移用 offset_bytes表示，它在这个字节的起始比特位置偏移用 offset_bits 表示。我们有

```
offset_bytes = (idx * 6) / 8
offset_bits = (idx * 6) % 8
```

前者是商，后者是余数。比如 bucket 2 的字节偏移是 1，也就是第 2 个字节。它的位偏移是4，也就是第 2 个字节的第 5 个位开始是 bucket 2 的计数值。需要注意的是字节位序是左边低位右边高位，而通常我们使用的字节都是左边高位右边低位，我们需要在脑海中进行倒置。

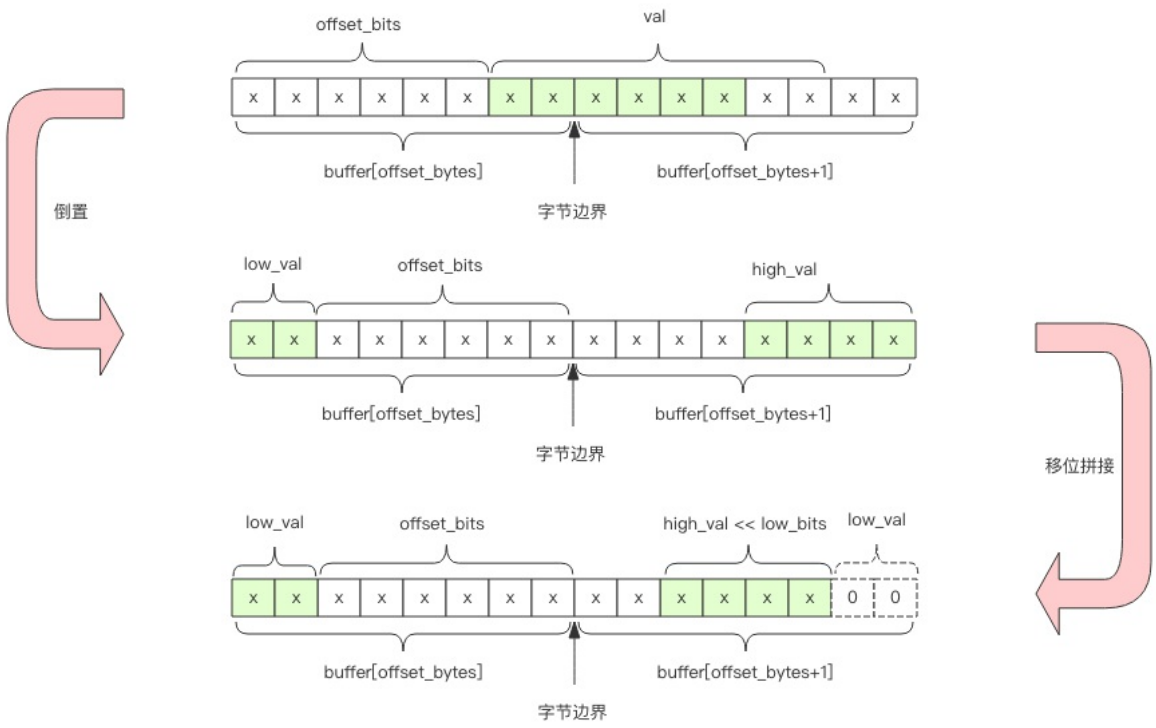


如果 offset_bits 小于等于 2，那么这 6bit 在一个字节内部，可以直接使用下面的表达式得到计数值 val




```
val = buffer[offset_bytes] >> offset_bits  # 向右移位
```

如果 offset_bits 大于 2，那么就会跨越字节边界，这时需要拼接两个字节的位片段。




```
# 低位值
low_val = buffer[offset_bytes] >> offset_bits
# 低位个数
low_bits = 8 - offset_bits
# 拼接，保留低6位
val = (high_val << low_bits | low_val) & 0b111111
```

不过下面 Redis 的源码要晦涩一点，看形式它似乎只考虑了跨越字节边界的情况。这是因为如果 6bit 在单个字节内，上面代码中的 high_val 的值是零，所以这一份代码可以同时照顾单字节和双字节。

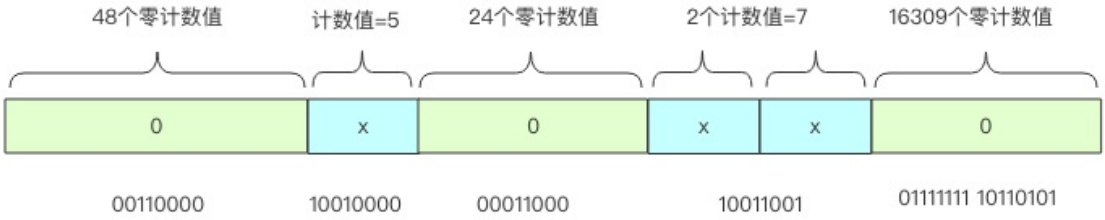
```
// 获取指定桶的计数值
#define HLL_DENSE_GET_REGISTER(target,p,regnum)
do { \
    uint8_t *_p = (uint8_t*) p; \
    unsigned long _byte = regnum*HLL_BITS/8; \
    unsigned long _fb = regnum*HLL_BITS&7; \  #
%8 = &7
    unsigned long _fb8 = 8 - _fb; \
    unsigned long b0 = _p[_byte]; \
    unsigned long b1 = _p[_byte+1]; \
    target = ((b0 >> _fb) | (b1 << _fb8)) &
HLL_REGISTER_MAX; \
} while(0)

// 设置指定桶的计数值
#define HLL_DENSE_SET_REGISTER(p,regnum,val) do {
\
    uint8_t *_p = (uint8_t*) p; \
    unsigned long _byte = regnum*HLL_BITS/8; \
    unsigned long _fb = regnum*HLL_BITS&7; \
    unsigned long _fb8 = 8 - _fb; \
    unsigned long _v = val; \
    _p[_byte] &= ~(HLL_REGISTER_MAX << _fb); \
    _p[_byte] |= _v << _fb; \
    _p[_byte+1] &= ~(HLL_REGISTER_MAX >> _fb8); \
    _p[_byte+1] |= _v >> _fb8; \
} while(0)
```

稀疏存储结构

稀疏存储适用于很多计数值都是零的情况。下图表示了一般稀疏存储计数值的状态。

当多个连续桶的计数值都是零时，Redis 使用了一个字节来表示接下来有多少个桶的计数值都是零：00xxxxxx。前缀两个零表示接下来的 6bit 整数值加 1 就是零值计数器的数量，注意这里要加 1 是因为数量如果为零是没有意义的。比如 00010101表示连续 22 个零值计数器。6bit 最多只能表示连续 64 个零值计数器，所以 Redis 又设计了连续多个多于 64 个的连续零值计数器，它使用两个字节来表示：01xxxxxx yyyyyyyy，后面的 14bit 可以表示最多连续 16384 个零值计数器。这意味着 HyperLogLog 数据结构中 16384 个桶的初始状态，所有的计数器都是零值，可以直接使用 2 个字节来表示。

如果连续几个桶的计数值非零，那就使用形如 1vvvvvxx 这样的一个字节来表示。中间 5bit 表示计数值，尾部 2bit 表示连续几个桶。它的意思是连续 （xx +1） 个计数值都是 （vvvvv + 1）。比如 10101011 表示连续 4 个计数值都是 11。注意这两个值都需要加 1，因为任意一个是零都意味着这个计数值为零，那就应该使用零计数值的形式来表示。注意计数值最大只能表示到32，而 HyperLogLog 的密集存储单个计数值用 6bit 表示，最大可以表示到 63。当稀疏存储的某个计数值需要调整到大于 32 时，Redis 就会立即转换 HyperLogLog 的存储结构，将稀疏存储转换成密集存储。

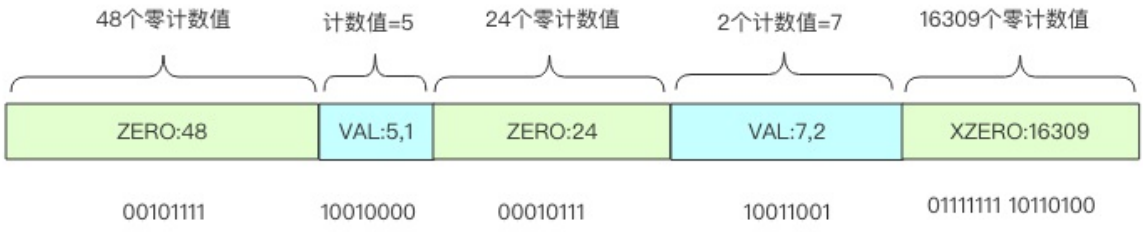


Redis 为了方便表达稀疏存储，它将上面三种字节表示形式分别赋予了一条指令。

1. ZERO:len 单个字节表示 00[len-1]，连续最多64个零计数值
2. VAL:value,len 单个字节表示 1[value-1][len-1]，连续 len 个值为 value 的计数值
3. XZERO:len 双字节表示 01[len-1]，连续最多16384个零计数值

```
#define HLL_SPARSE_XZERO_BIT 0x40 /* 01xxxxxx */
#define HLL_SPARSE_VAL_BIT 0x80 /* 1vvvvvxx */
#define HLL_SPARSE_IS_ZERO(p) (((*(p)) & 0xc0) ==
0) /* 00xxxxxx */
#define HLL_SPARSE_IS_XZERO(p) (((*(p)) & 0xc0)
== HLL_SPARSE_XZERO_BIT)
#define HLL_SPARSE_IS_VAL(p) ((*(p)) &
HLL_SPARSE_VAL_BIT)
#define HLL_SPARSE_ZERO_LEN(p) (((*(p)) &
0x3f)+1)
#define HLL_SPARSE_XZERO_LEN(p) (((((*(p)) &
0x3f) << 8) | (*((p)+1)))+1)
#define HLL_SPARSE_VAL_VALUE(p) ((((*(p)) >> 2) &
0x1f)+1)
#define HLL_SPARSE_VAL_LEN(p) (((*(p)) & 0x3)+1)
#define HLL_SPARSE_VAL_MAX_VALUE 32
#define HLL_SPARSE_VAL_MAX_LEN 4
#define HLL_SPARSE_ZERO_MAX_LEN 64
#define HLL_SPARSE_XZERO_MAX_LEN 16384
```

上图可以使用指令形式表示如下



存储转换

当计数值达到一定程度后，稀疏存储将会不可逆一次性转换为密集存储。转换的条件有两个，任意一个满足就会立即发生转换

1. 任意一个计数值从 32 变成 33，因为VAL指令已经无法容纳，它能表示的计数值最大为 32
2. 稀疏存储占用的总字节数超过 3000 字节，这个阈值可以通过 hll_sparse_max_bytes 参数进行调整。

计数缓存

前面提到 HyperLogLog 表示的总计数值是由 16384 个桶的计数值进行调和平均后再基于因子修正公式计算得出来的。它需要遍历所有的桶进行计算才可以得到这个值，中间还涉及到很多浮点运算。这个计算量相对来说还是比较大的。

所以 Redis 使用了一个额外的字段来缓存总计数值，这个字段有 64bit，最高位如果为 1 表示该值是否已经过期，如果为 0， 那么剩下的 63bit 就是计数值。

当 HyperLogLog 中任意一个桶的计数值发生变化时，就会将计数缓存设为过期，但是不会立即触发计算。而是要等到用户显示调用 pfcount 指令时才会触发重新计算刷新缓存。缓存刷新在密集存储时需要遍历 16384 个桶的计数值进行调和平均，但是稀疏存储时没有这么大的计算量。也就是说只有当计数值比较大时才可能产生较大的计算量。另一方面如果计数值比较大，那么大部分 pfadd 操作根本不会导致桶中的计数值发生变化。

这意味着在一个极具变化的 HLL 计数器中频繁调用 pfcount 指令可能会有少许性能问题。关于这个性能方面的担忧在 Redis 作者 antirez 的博客中也提到了。不过作者做了仔细的压力的测试，发现这是无需担心的，pfcount 指令的平均时间复杂度就是 O(1)。

> After this change even trying to add elements at maximum speed using a pipeline of 32 elements with 50 simultaneous clients, PFCOUNT was able to perform as well as any other O(1) command with very small constant times.

对象头

HyperLogLog 除了需要存储 16384 个桶的计数值之外，它还有一些附加的字段需要存储，比如总计数缓存、存储类型。所以它使用了一个额外的对象头来表示。

```
struct hllhdr {
    char magic[4];      /* 魔术字符串"HYLL" */
    uint8_t encoding;   /* 存储类型 HLL_DENSE or
HLL_SPARSE. */
    uint8_t notused[3]; /* 保留三个字节未来可能会使用
*/
    uint8_t card[8];    /* 总计数缓存 */
    uint8_t registers[]; /* 所有桶的计数器 */
};
```

所以 HyperLogLog 整体的内部结构就是 HLL 对象头 加上 16384 个桶的计数值位图。它在 Redis 的内部结构表现就是一个字符串位图。你可以把 HyperLogLog 对象当成普通的字符串来进行处理。

```
127.0.0.1:6379> pfadd codehole python java golang
(integer) 1
127.0.0.1:6379> get codehole
"HYLL\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x80C\x03\x84MK\x80P\xb8\x80^\xf3"
```

但是不可以使用 HyperLogLog 指令来操纵普通的字符串，因为它需要检查对象头魔术字符串是否是 "HYLL"。

```
127.0.0.1:6379> set codehole python
OK
127.0.0.1:6379> pfadd codehole java golang
(error) WRONGTYPE Key is not a valid HyperLogLog
string value.
```

但是如果字符串以 "HYLL\x00" 或者 "HYLL\x01" 开头，那么就可以使用 HyperLogLog 的指令。

```
127.0.0.1:6379> set codehole
"HYLL\x01whatmagicthing"
OK
127.0.0.1:6379> get codehole
"HYLL\x01whatmagicthing"
127.0.0.1:6379> pfadd codehole python java golang
(integer) 1
```

也许你会感觉非常奇怪，这是因为 HyperLogLog 在执行指令前需要对内容进行格式检查，这个检查就是查看对象头的 magic 魔术字符串是否是 "HYLL" 以及 encoding 字段是否是 HLL_SPARSE=0 或者 HLL_DENSE=1 来判断当前的字符串是否是 HyperLogLog 计数器。如果是密集存储，还需要判断字符串的长度是否恰好等于密集计数器存储的长度。

```
int isHLLObjectOrReply(client *c, robj *o) {
    ...
    /* Magic should be "HYLL". */
    if (hdr->magic[0] != 'H' || hdr->magic[1] !=
'Y' ||
        hdr->magic[2] != 'L' || hdr->magic[3] !=
'L') goto invalid;

    if (hdr->encoding > HLL_MAX_ENCODING) goto
invalid;

    if (hdr->encoding == HLL_DENSE &&
        stringObjectLen(o) != HLL_DENSE_SIZE)
goto invalid;

    return C_OK;

invalid:
    addReplySds(c,
        sdsnew("-WRONGTYPE Key is not a valid "
               "HyperLogLog string value.\r\n"));
    return C_ERR;
}
```

HyperLogLog 和 字符串的关系就好比 Geo 和 zset 的关系。你也可以使用任意 zset 的指令来访问 Geo 数据结构，因为 Geo 内部存储就是使用了一个纯粹的 zset来记录元素的地理位置。