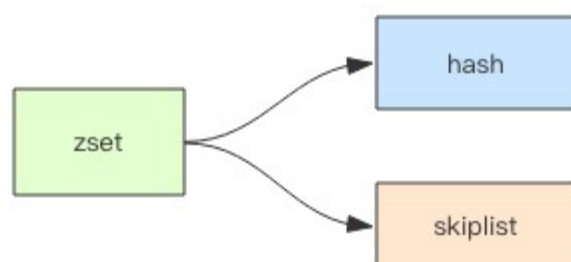




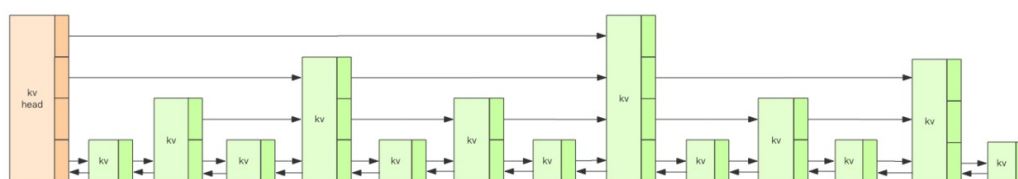
源码 5：极度深寒 —— 探索「跳跃列表」内部结构

Redis 的 zset 是一个复合结构，一方面它需要一个 hash 结构来存储 value 和 score 的对应关系，另一方面需要提供按照 score 来排序的功能，还需要能够指定 score 的范围来获取 value 列表的功能，这就需要另外一个结构「跳跃列表」。



zset 的内部实现是一个 hash 字典加一个跳跃列表 (skiplist)。hash 结构在讲字典结构时已经详细分析过了，它很类似于 Java 语言中的 HashMap 结构。本节我们来讲跳跃列表，它比较复杂，读者要有心理准备。

基本结构



上图就是跳跃列表的示意图，图中只画了四层，Redis 的跳跃表共有 64 层，意味着最多可以容纳 2^{64} 次方个元素。每一个 kv 块对应的结构如下面的代码中的 `zslnode` 结构，kv header 也是这个结构，只不过 value 字段是 null 值——无效的，score 是 `Double.MIN_VALUE`，用来垫底的。kv 之间使用指针

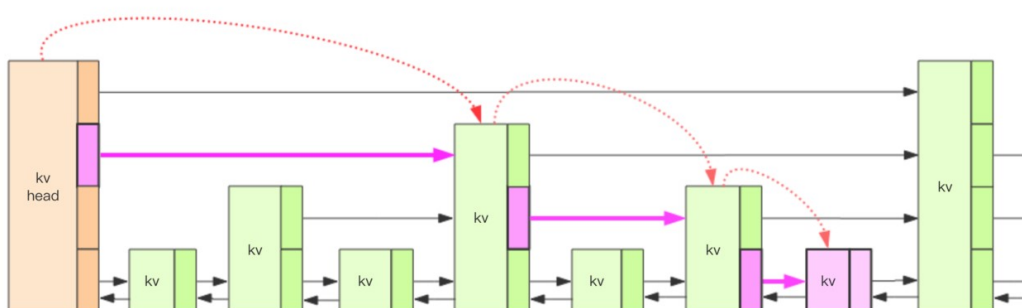


串起来形成了双向链表结构，它们是 **有序** 排列的，从小到大。不同的 kv 层高可能不一样，层数越高的 kv 越少。同一层的 kv 会使用指针串起来。每一个层元素的遍历都是从 kv header 出发。

```
struct zslnode {  
    string value;  
    double score;  
    zslnode*[] forwards; // 多层连接指针  
    zslnode* backward; // 回溯指针  
}  
  
struct zsl {  
    zslnode* header; // 跳跃列表头指针  
    int maxLevel; // 跳跃列表当前的最高层  
    map<string, zslnode*> ht; // hash 结构的所有键值对  
}
```

查找过程

设想如果跳跃列表只有一层会怎样？插入删除操作需要定位到相应的位置节点（定位到最后一个比「我」小的元素，也就是第一个比「我」大的元素的前一个），定位的效率肯定比较差，复杂度将会是 $O(n)$ ，因为需要挨个遍历。也许你会想到二分查找，但是二分查找的结构只能是有序数组。跳跃列表有了多层结构之后，这个定位的算法复杂度将会降到 $O(\lg(n))$ 。



如图所示，我们要定位到那个紫色的 kv，需要从 header 的最高层开始遍历找到第一个节点（最后一个比「我」小的元素），然后从这个节点开始降一层再遍历找到第二个节点（最后一个比「我」小的元素），然后一直降到最底层进行遍历就找到了期望的节点（最底层的最后一个比我「小」的元素）。



我们将中间经过的一系列节点称之为「搜索路径」，它是从最高层一直到最底层的每一层最后一个比「我」小的元素节点列表。

有了这个搜索路径，我们就可以插入这个新节点了。不过这个插入过程也不是特别简单。因为新插入的节点到底有多少层，得有个算法来分配一下，跳跃列表使用的是随机算法。

随机层数

对于每一个新插入的节点，都需要调用一个随机算法给它分配一个合理的层数。直观上期望的目标是 50% 的 Level1，25% 的 Level2，12.5% 的 Level3，一直到最顶层 2^{-63} ，因为这里每一层的晋升概率是 50%。

```
/* Returns a random level for the new skiplist node we are going to create.
 * The return value of this function is between 1 and ZSKIPLIST_MAXLEVEL
 * (both inclusive), with a powerlaw-alike distribution where higher
 * levels are less likely to be returned. */
int zslRandomLevel(void) {
    int level = 1;
    while ((random() & 0xFFFF) < (ZSKIPLIST_P * 0xFFFF))
        level += 1;
    return (level < ZSKIPLIST_MAXLEVEL) ? level : ZSKIPLIST_MAXLEVEL;
}
```

不过 Redis 标准源码中的晋升概率只有 25%，也就是代码中的 ZSKIPLIST_P 的值。所以官方的跳跃列表更加的扁平化，层高相对较低，在单个层上需要遍历的节点数量会稍多一点。

也正是因为层数一般不高，所以遍历的时候从顶层开始往下遍历会非常浪费。跳跃列表会记录一下当前的最高层数 `maxLevel`，遍历时从这个 `maxLevel` 开始遍历性能就会提高很多。

插入过程

下面是插入过程的源码，它稍微有点长，不过整体的过程还是比较清晰的。



```
/* Insert a new node in the skiplist. Assumes the element does not already
 * exist (up to the caller to enforce that). The skiplist takes ownership
 * of the passed SDS string 'ele'. */
zskiplistNode *zslInsert(zskiplist *zsl, double score, sds ele) {
    // 存储搜索路径
    zskiplistNode *update[ZSKIPLIST_MAXLEVEL], *x;
    // 存储经过的节点跨度
    unsigned int rank[ZSKIPLIST_MAXLEVEL];
    int i, level;

    serverAssert(!isnan(score));
    x = zsl->header;
    // 逐步降级寻找目标节点，得到「搜索路径」
    for (i = zsl->level-1; i >= 0; i--) {
        /* store rank that is crossed to reach the insert position */
        rank[i] = i == (zsl->level-1) ? 0 : rank[i+1];
        // 如果score相等，还需要比较value
        while (x->level[i].forward &&
            (x->level[i].forward->score < score ||
            (x->level[i].forward->score == score &&
            sdscmp(x->level[i].forward->ele,ele) < 0)))
        {
            rank[i] += x->level[i].span;
            x = x->level[i].forward;
        }
        update[i] = x;
    }
    // 正式进入插入过程
    /* we assume the element is not already inside, since we allow duplicated
     * scores, reinserting the same element should never happen since the
     * caller of zslInsert() should test in the hash table if the element is
     * already inside or not. */
    // 随机一个层数
    level = zslRandomLevel();
    // 填充跨度
    if (level > zsl->level) {
        for (i = zsl->level; i < level; i++) {
            rank[i] = 0;
            update[i] = zsl->header;
            update[i]->level[i].span = zsl->length;
        }
        // 更新跳跃列表的层高
        zsl->level = level;
    }
    // 创建新节点
    x = zslCreateNode(level,score,ele);
    // 重排一下前向指针
    for (i = 0; i < level; i++) {
        x->level[i].forward = update[i]->level[i].forward;
```



```
update[i]->level[i].forward = x;

/* update span covered by update[i] as x is inserted here */
x->level[i].span = update[i]->level[i].span - (rank[0] - rank[i]);
update[i]->level[i].span = (rank[0] - rank[i]) + 1;
}

/* increment span for untouched levels */
for (i = level; i < zsl->level; i++) {
    update[i]->level[i].span++;
}

// 重排一下后向指针
x->backward = (update[0] == zsl->header) ? NULL : update[0];
if (x->level[0].forward)
    x->level[0].forward->backward = x;
else
    zsl->tail = x;
zsl->length++;
return x;
}
```

首先我们在搜索合适插入点的过程中将「搜索路径」摸出来了，然后就可以开始创建新节点了，创建的时候需要给这个节点随机分配一个层数，再将搜索路径上的节点和这个新节点通过前向后向指针串起来。如果分配的新节点的高度高于当前跳跃列表的最大高度，就需要更新一下跳跃列表的最大高度。

删除过程

删除过程和插入过程类似，都需先把这个「搜索路径」找出来。然后对于每个层的相关节点都重排一下前向后向指针就可以了。同时还要注意更新一下最高层数 `maxLevel`。

更新过程

当我们调用 `zadd` 方法时，如果对应的 `value` 不存在，那就是插入过程。如果这个 `value` 已经存在了，只是调整一下 `score` 的值，那就需要走一个更新的流程。假设这个新的 `score` 值不会带来排序位置上的改变，那么就不需要调整位置，直接修改元素的 `score` 值就可以了。但是如果排序位置改变了，那就要调整位置。那该如何调整位置呢？



```
/* Remove and re-insert when score changes. */
if (score != curscore) {
    zskiplistNode *node;
    serverAssert(zslDelete(zs->zsl,curscore,ele,&node));
    znode = zslInsert(zs->zsl,score,node->ele);
    /* We reused the node->ele SDS string, free the node now
     * since zslInsert created a new one. */
    node->ele = NULL;
    zslFreeNode(node);
    /* Note that we did not removed the original element from
     * the hash table representing the sorted set, so we just
     * update the score. */
    dictGetVal(de) = &znode->score; /* Update score ptr. */
    *flags |= ZADD_UPDATED;
}
return 1;
```

一个简单的策略就是先删除这个元素，再插入这个元素，需要经过两次路径搜索。Redis 就是这么干的。不过 Redis 遇到 score 值改变了就直接删除再插入，不会去判断位置是否需要调整，从这点看，Redis 的 zadd 的代码似乎还有优化空间。关于这一点，读者们可以继续讨论。

如果 score 值都一样呢？

在一个极端的情况下，zset 中所有的 score 值都是一样的，zset 的查找性能会退化为 $O(n)$ 么？Redis 作者自然考虑到了这一点，所以 zset 的排序元素不只看 score 值，如果 score 值相同还需要再比较 value 值 (字符串比较)。

元素排名是怎么算出来的？

前面我们啰嗦了一堆，但是有一个重要的属性没有提到，那就是 zset 可以获取元素的排名 rank。那这个 rank 是如何算出来的？如果仅仅使用上面的结构，rank 是不能算出来的。Redis 在 skiplist 的 forward 指针上进行了优化，给每一个 forward 指针都增加了 span 属性，span 是「跨度」的意思，表示从前一个节点沿着当前层的 forward 指针跳到当前这个节点中间会跳过多少个节点。Redis 在插入删除操作时会小心翼翼地更新 span 值的大小。



java

```
struct zslforward {  
    zlnode* item;  
    long span; // 跨度  
}  
  
struct zsl {  
    String value;  
    double score;  
    zslforward*[] forwards; // 多层连接指针  
    zlnode* backward; // 回溯指针  
}
```

这样当我们要计算一个元素的排名时，只需要将「搜索路径」上的经过的所有节点的跨度 span 值进行叠加就可以算出元素的最终 rank 值。

思考

文中我们提到当 score 值的变化微小，不会带来位置上的调整时，是不是可以直接修改 score 后就返回？

请读者们对这个问题进行讨论。如果确实如此，可以考虑向 Redis 作者 Antirez 提 issue 了。