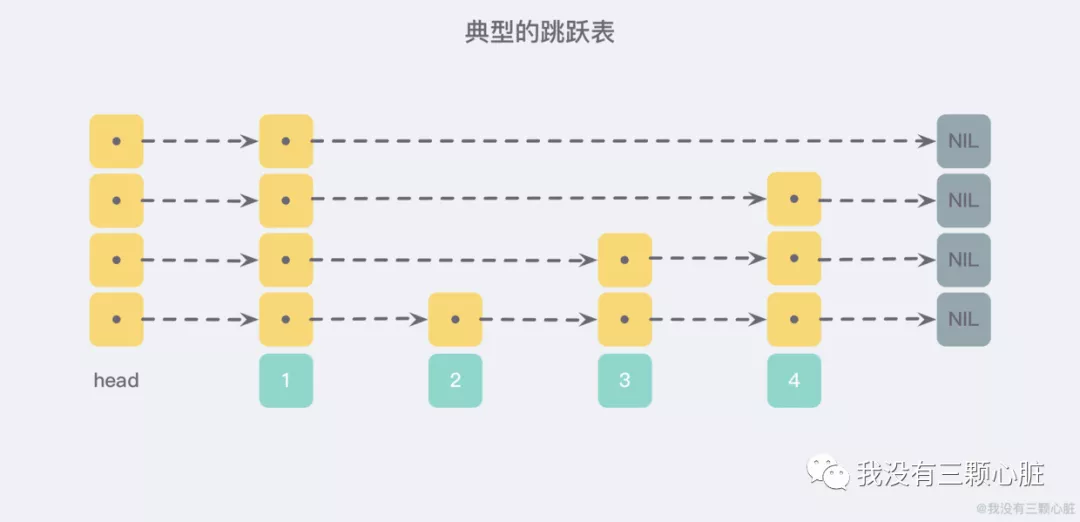
<https://mp.weixin.qq.com/s/gA8CUpk6BivLGrizRRv_Bw>

# Redis(2)——跳跃表

# **一、跳跃表简介**

跳跃表（skiplist）是一种随机化的数据结构，由 **William Pugh** 在论文《Skip lists: a probabilistic alternative to balanced trees》中提出，是一种可以与平衡树媲美的层次化链表结构——查找、删除、添加等操作都可以在对数期望时间下完成，以下是一个典型的跳跃表例子：



我们在上一篇中提到了 Redis 的五种基本结构中，有一个叫做 **有序列表 zset** 的数据结构，它类似于 Java 中的 **SortedSet** 和 **HashMap** 的结合体，一方面它是一个 set 保证了内部 value 的唯一性，另一方面又可以给每个 value 赋予一个排序的权重值 score，来达到 **排序** 的目的。

它的内部实现就依赖了一种叫做 **「跳跃列表」** 的数据结构。

## **为什么使用跳跃表**

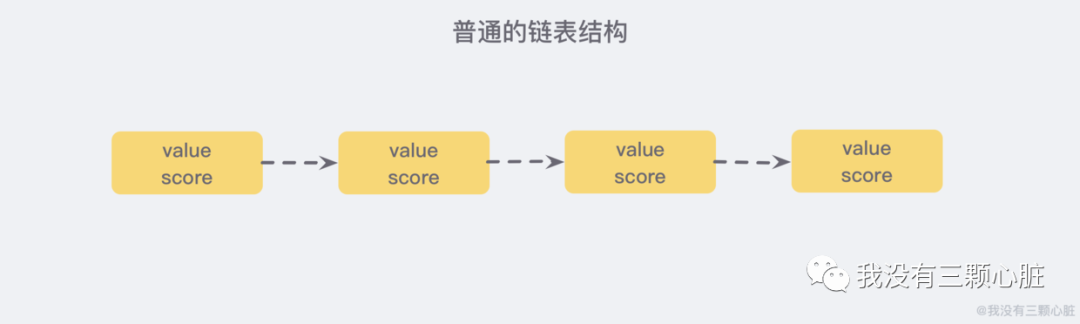
首先，因为 zset 要支持随机的插入和删除，所以它 **不宜使用数组来实现**，关于排序问题，我们也很容易就想到 **红黑树/ 平衡树** 这样的树形结构，为什么 Redis 不使用这样一些结构呢？

1. **性能考虑：** 在高并发的情况下，树形结构需要执行一些类似于 rebalance 这样的可能涉及整棵树的操作，相对来说跳跃表的变化只涉及局部 *(下面详细说)*；
2. **实现考虑：** 在复杂度与红黑树相同的情况下，跳跃表实现起来更简单，看起来也更加直观；

基于以上的一些考虑，Redis 基于 **William Pugh** 的论文做出一些改进后采用了 **跳跃表** 这样的结构。

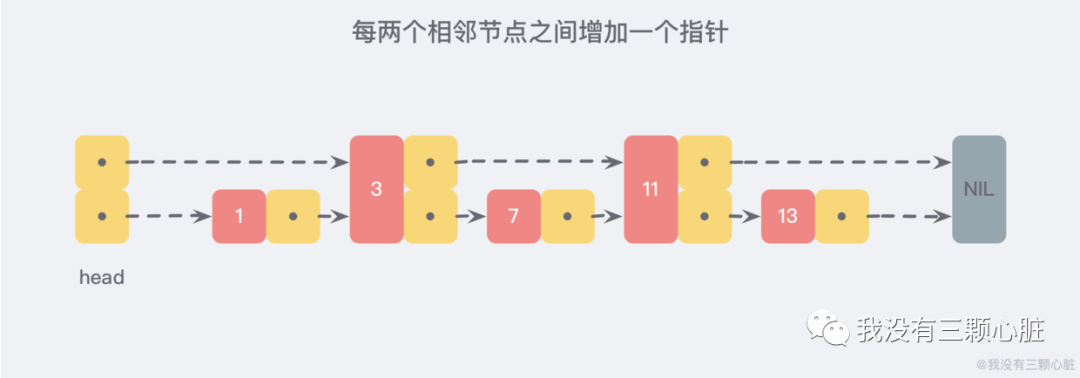
## **本质是解决查找问题**

我们先来看一个普通的链表结构：



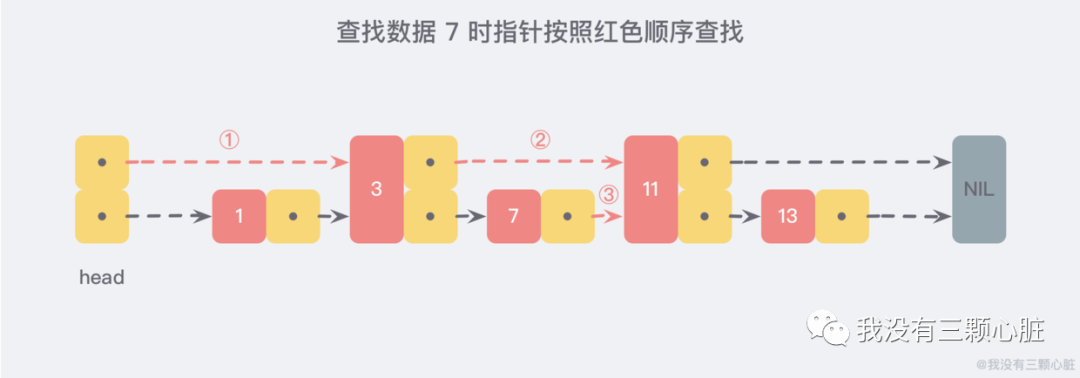
我们需要这个链表按照 score 值进行排序，这也就意味着，当我们需要添加新的元素时，我们需要定位到插入点，这样才可以继续保证链表是有序的，通常我们会使用 **二分查找法**，但二分查找是有序数组的，链表没办法进行位置定位，我们除了遍历整个找到第一个比给定数据大的节点为止 *（时间复杂度 O(n))* 似乎没有更好的办法。

但假如我们每相邻两个节点之间就增加一个指针，让指针指向下一个节点，如下图：



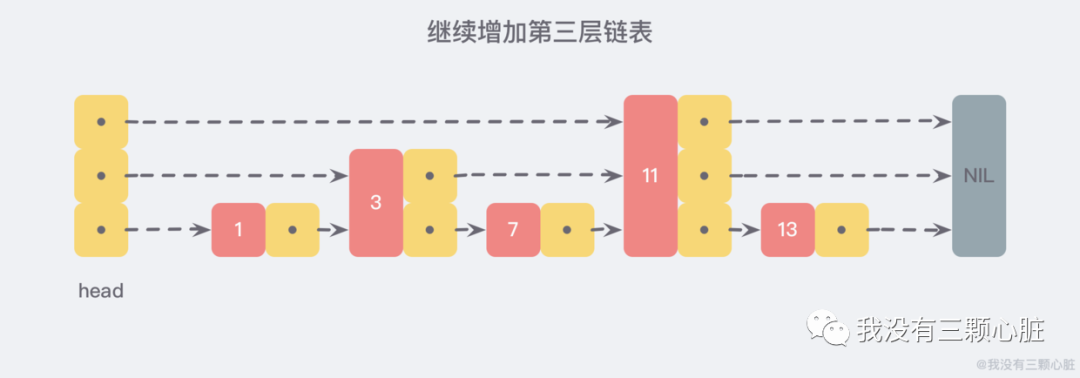
这样所有新增的指针连成了一个新的链表，但它包含的数据却只有原来的一半 *（图中的为 3，11）*。

现在假设我们想要查找数据时，可以根据这条新的链表查找，如果碰到比待查找数据大的节点时，再回到原来的链表中进行查找，比如，我们想要查找 7，查找的路径则是沿着下图中标注出的红色指针所指向的方向进行的：



这是一个略微极端的例子，但我们仍然可以看到，通过新增加的指针查找，我们不再需要与链表上的每一个节点逐一进行比较，这样改进之后需要比较的节点数大概只有原来的一半。

利用同样的方式，我们可以在新产生的链表上，继续为每两个相邻的节点增加一个指针，从而产生第三层链表：



在这个新的三层链表结构中，我们试着 **查找 13**，那么沿着最上层链表首先比较的是 11，发现 11 比 13 小，于是我们就知道只需要到 11 后面继续查找，**从而一下子跳过了 11 前面的所有节点。**

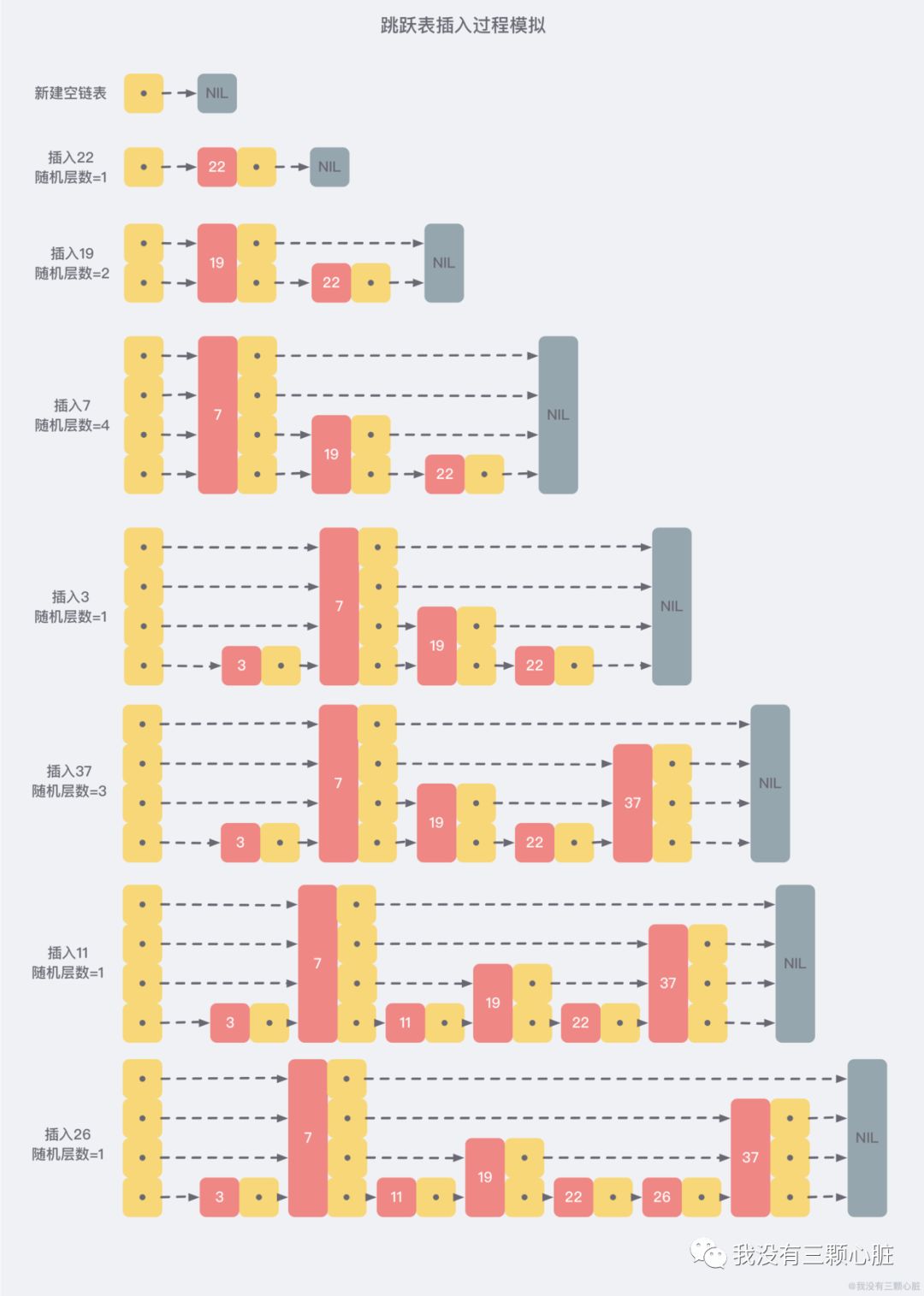
可以想象，当链表足够长，这样的多层链表结构可以帮助我们跳过很多下层节点，从而加快查找的效率。

## **更进一步的跳跃表**

**跳跃表 skiplist** 就是受到这种多层链表结构的启发而设计出来的。按照上面生成链表的方式，上面每一层链表的节点个数，是下面一层的节点个数的一半，这样查找过程就非常类似于一个二分查找，使得查找的时间复杂度可以降低到 *O(logn)*。

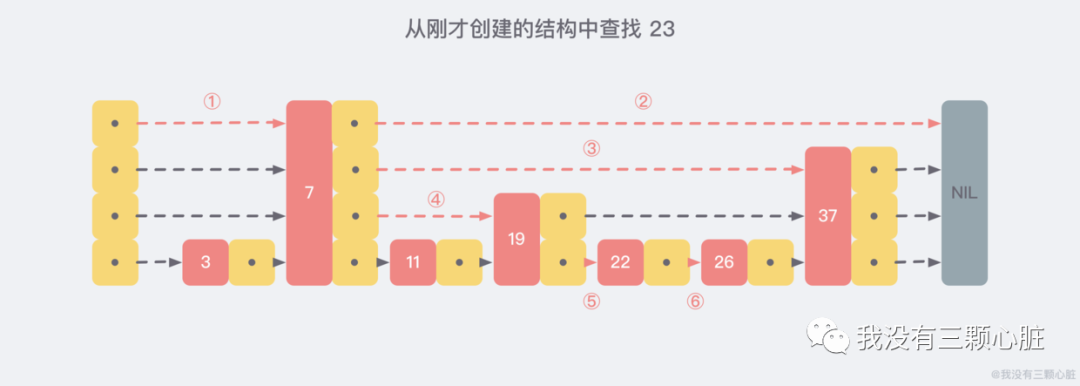
但是，这种方法在插入数据的时候有很大的问题。新插入一个节点之后，就会打乱上下相邻两层链表上节点个数严格的 2:1 的对应关系。如果要维持这种对应关系，就必须把新插入的节点后面的所有节点 *（也包括新插入的节点）* 重新进行调整，这会让时间复杂度重新蜕化成 *O(n)*。删除数据也有同样的问题。

**skiplist** 为了避免这一问题，它不要求上下相邻两层链表之间的节点个数有严格的对应关系，而是 **为每个节点随机出一个层数(level)**。比如，一个节点随机出的层数是 3，那么就把它链入到第 1 层到第 3 层这三层链表中。为了表达清楚，下图展示了如何通过一步步的插入操作从而形成一个 skiplist 的过程：



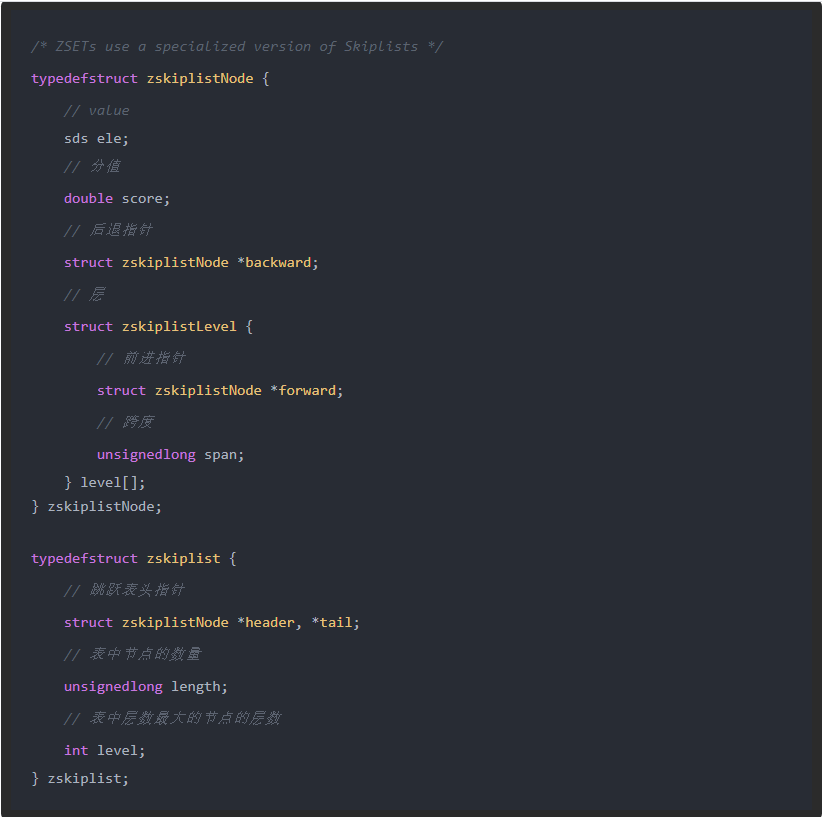
从上面的创建和插入的过程中可以看出，每一个节点的层数（level）是随机出来的，而且新插入一个节点并不会影响到其他节点的层数，因此，**插入操作只需要修改节点前后的指针，而不需要对多个节点都进行调整**，这就降低了插入操作的复杂度。

现在我们假设从我们刚才创建的这个结构中查找 23 这个不存在的数，那么查找路径会如下图：



# **二、跳跃表的实现**

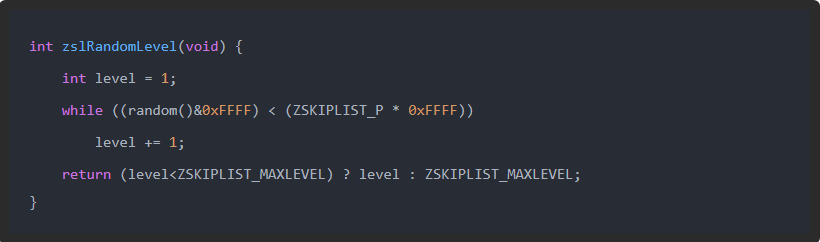
Redis 中的跳跃表由 server.h/zskiplistNode 和 server.h/zskiplist 两个结构定义，前者为跳跃表节点，后者则保存了跳跃节点的相关信息，同之前的 集合 list 结构类似，其实只有 zskiplistNode 就可以实现了，但是引入后者是为了更加方便的操作：



正如文章开头画出来的那张标准的跳跃表那样。

## **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，源码在 t\_zset.c/zslRandomLevel(void) 中被定义：



直观上期望的目标是 50% 的概率被分配到 Level 1，25% 的概率被分配到 Level 2，12.5% 的概率被分配到 Level 3，以此类推...有 2-63 的概率被分配到最顶层，因为这里每一层的晋升率都是 50%。

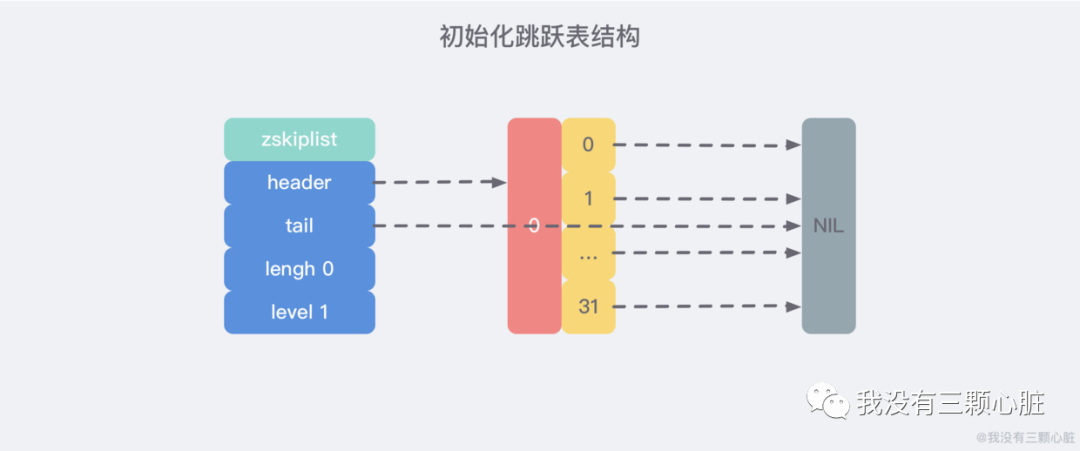
**Redis 跳跃表默认允许最大的层数是 32**，被源码中 ZSKIPLIST\_MAXLEVEL 定义，当 Level[0] 有 264 个元素时，才能达到 32 层，所以定义 32 完全够用了。

## **创建跳跃表**

这个过程比较简单，在源码中的 t\_zset.c/zslCreate 中被定义：



即执行完之后创建了如下结构的初始化跳跃表：



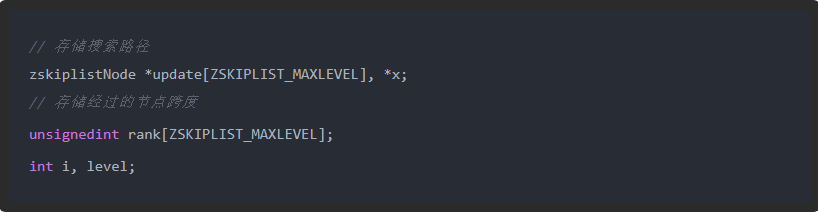
## **插入节点实现**

这几乎是最重要的一段代码了，但总体思路也比较清晰简单，如果理解了上面所说的跳跃表的原理，那么很容易理清楚插入节点时发生的几个动作 *（几乎跟链表类似）*：

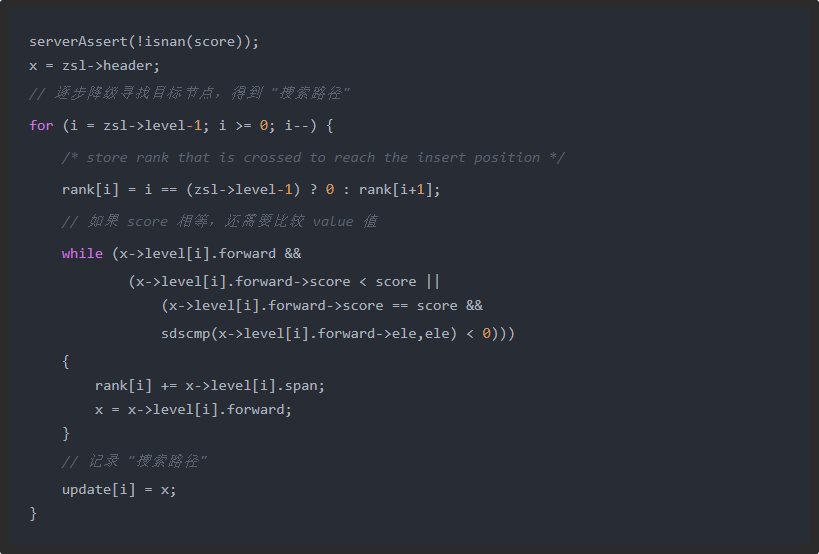
1. 找到当前我需要插入的位置 *（其中包括相同 score 时的处理）*；
2. 创建新节点，调整前后的指针指向，完成插入；

为了方便阅读，我把源码 t\_zset.c/zslInsert 定义的插入函数拆成了几个部分

### **第一部分：声明需要存储的变量**



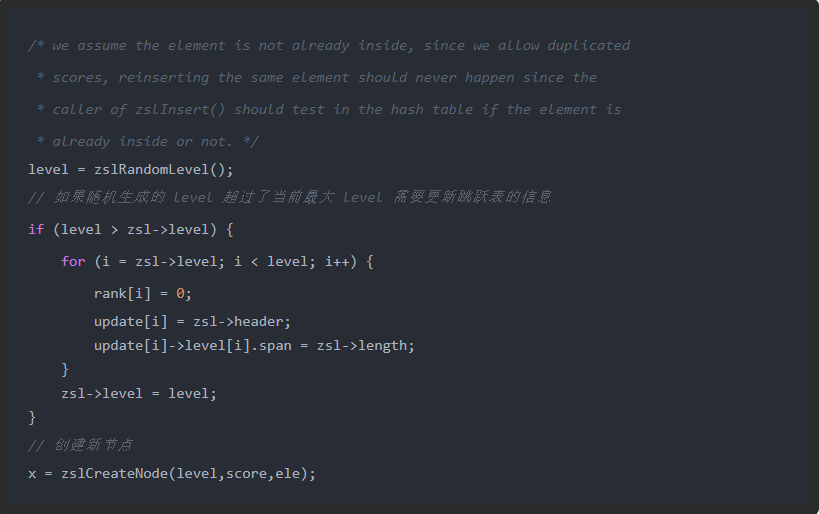
### **第二部分：搜索当前节点插入位置**



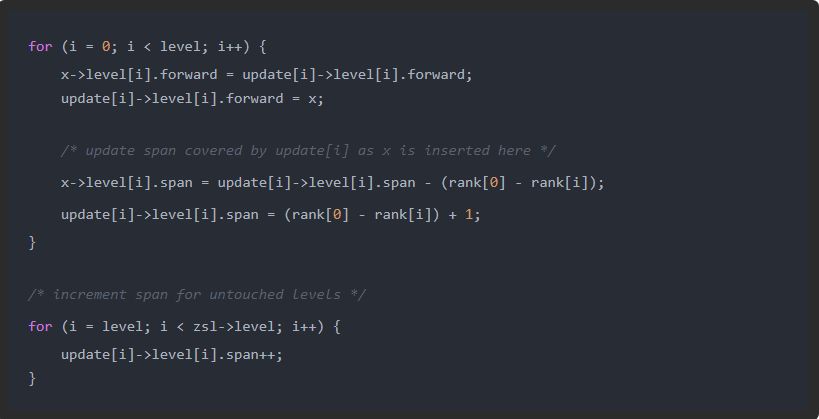
**讨论：** 有一种极端的情况，就是跳跃表中的所有 score 值都是一样，zset 的查找性能会不会退化为 O(n) 呢？

从上面的源码中我们可以发现 zset 的排序元素不只是看 score 值，也会比较 value 值 *（字符串比较）*

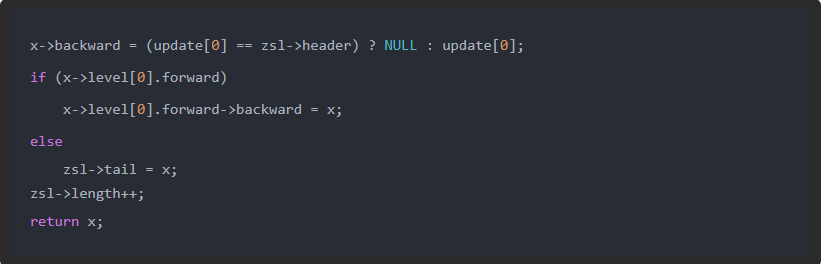
### **第三部分：生成插入节点**



### **第四部分：重排前向指针**

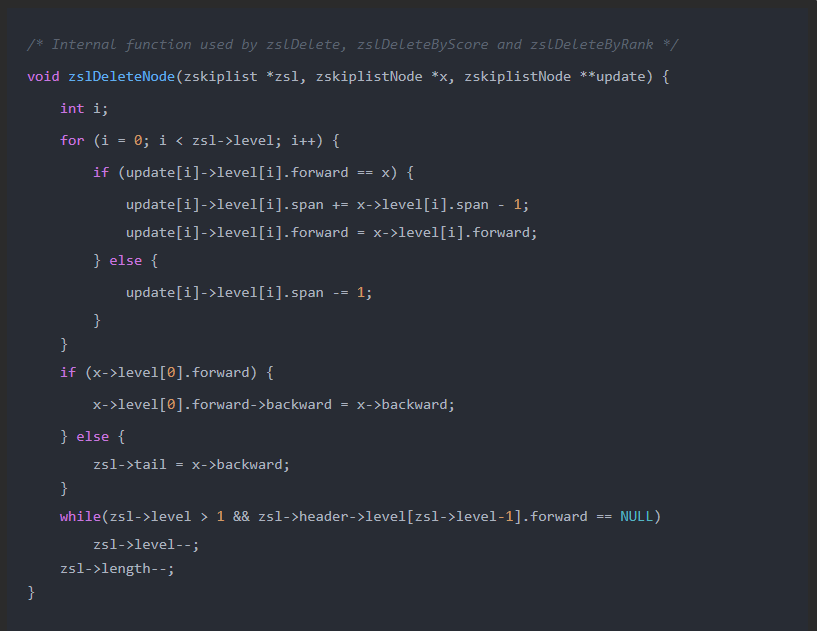


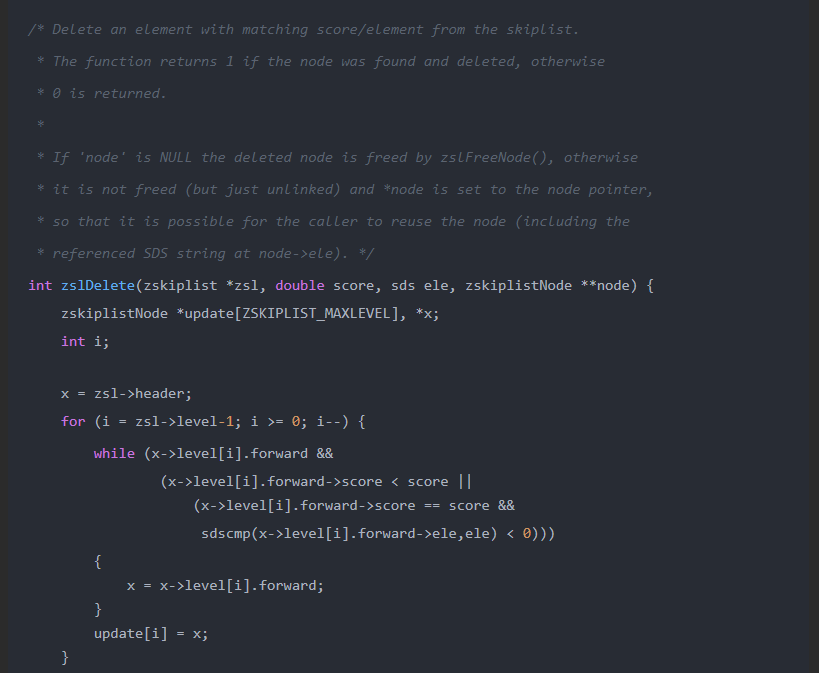
### **第五部分：重排后向指针并返回**

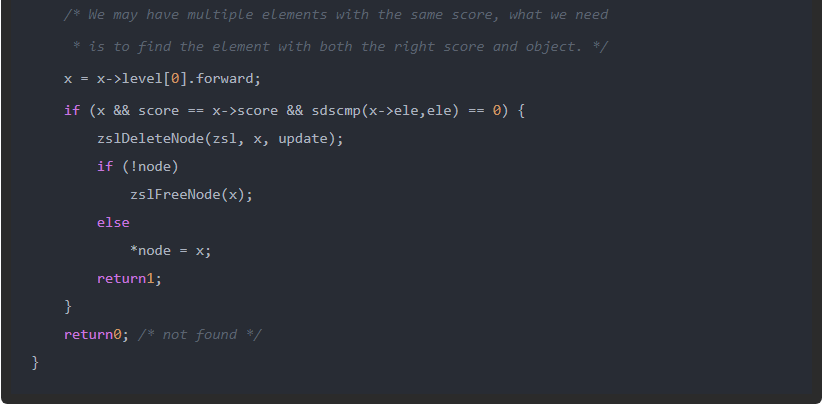


## **节点删除实现**

删除过程由源码中的 t\_zset.c/zslDeleteNode 定义，和插入过程类似，都需要先把这个 **"搜索路径"** 找出来，然后对于每个层的相关节点重排一下前向后向指针，同时还要注意更新一下最高层数 maxLevel，直接放源码 *(如果理解了插入这里还是很容易理解的)*：





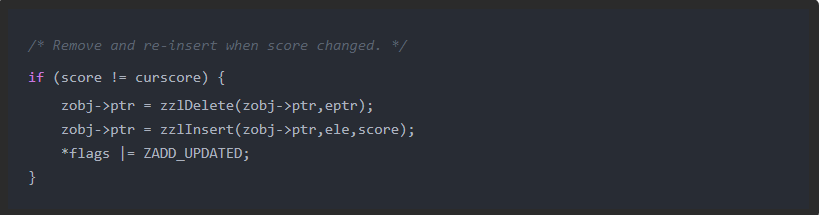


## **节点更新实现**

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，只是调整一下 score 的值，那就需要走一个更新流程。

假设这个新的 score 值并不会带来排序上的变化，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，那就需要调整位置，该如何调整呢？

从源码 t\_zset.c/zsetAdd 函数 1350 行左右可以看到，Redis 采用了一个非常简单的策略：



**把这个元素删除再插入这个**，需要经过两次路径搜索，从这一点上来看，Redis 的 ZADD 代码似乎还有进一步优化的空间。

## **元素排名的实现**

跳跃表本身是有序的，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，用来 **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**。在上面的源码中我们也可以看到 Redis 在插入、删除操作时都会小心翼翼地更新 span 值的大小。

所以，沿着 **"搜索路径"**，把所有经过节点的跨度 span 值进行累加就可以算出当前元素的最终 rank 值了：

