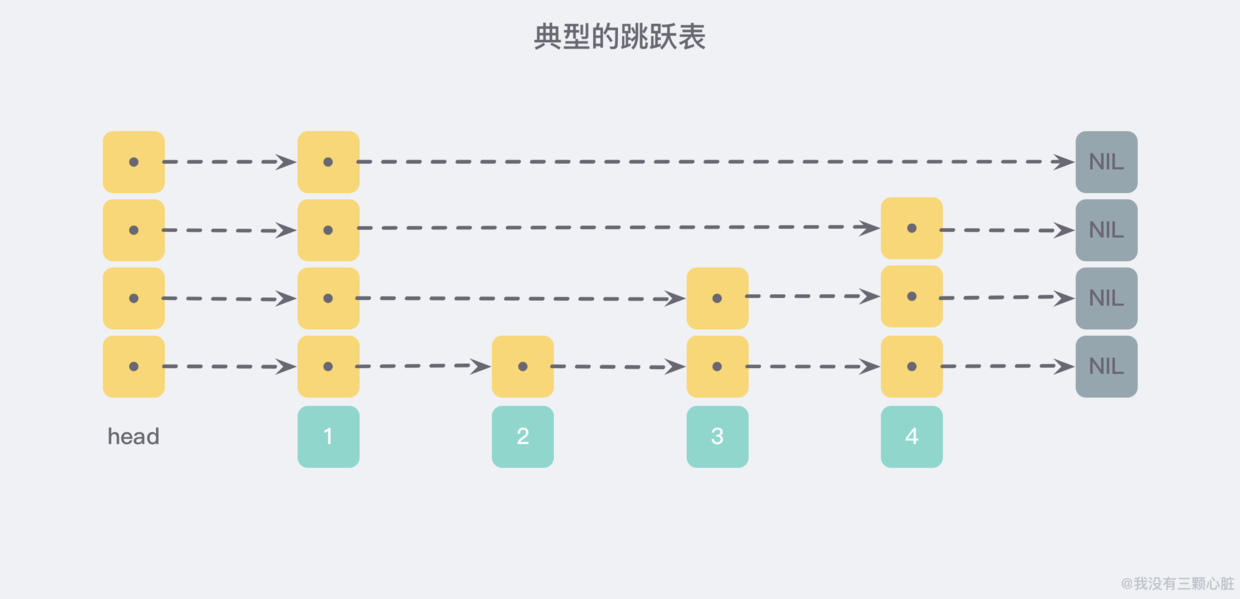
一、跳跃表简介

跳跃表（skiplist）是一种随机化的数据结构，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：



redis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。

二、为什么使用跳跃列表

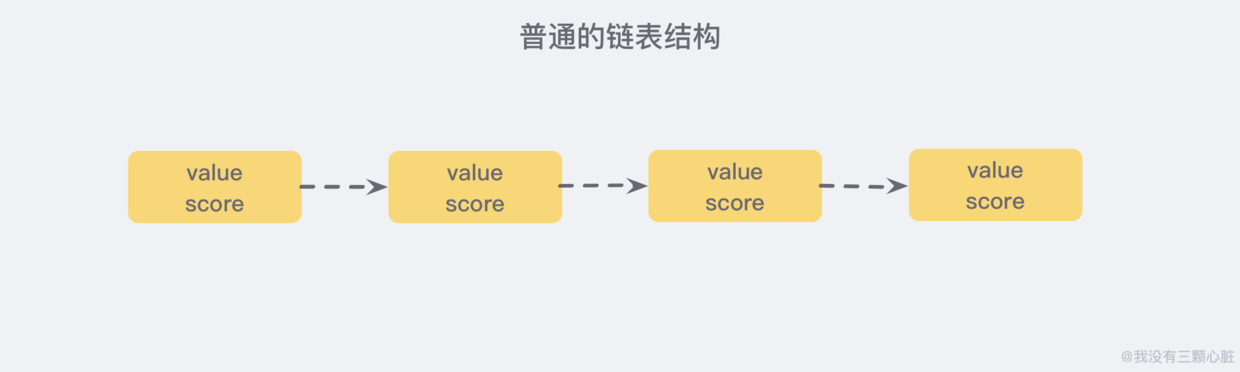
首先，因为zset要支持随机的插入和删除，所以它不宜使用数组来实现，关于排序问题，我们也很容易就想到红黑树/平衡树这样的树形结构，为什么Redis不使用这样一些结构呢？

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。
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。

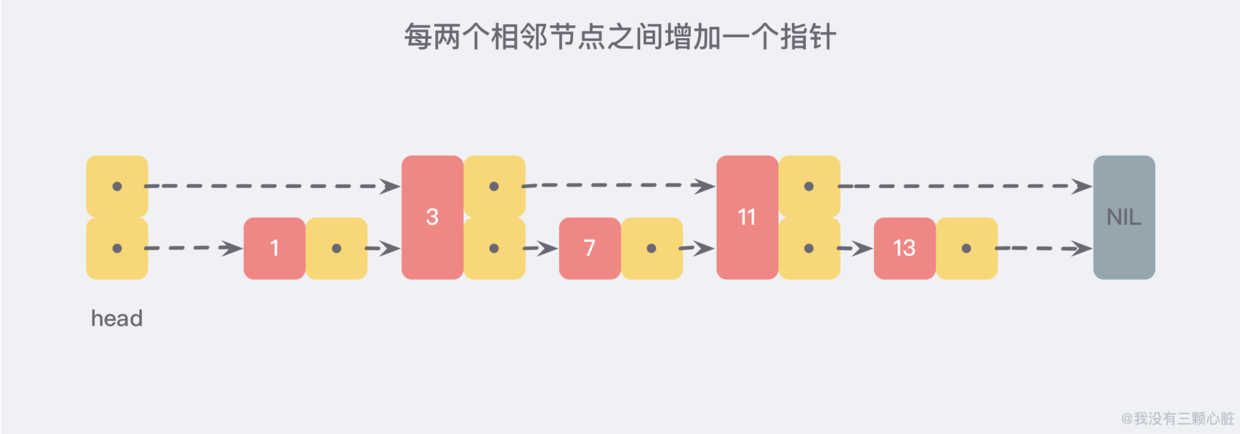
三、本质是解决查找问题

普通链表：



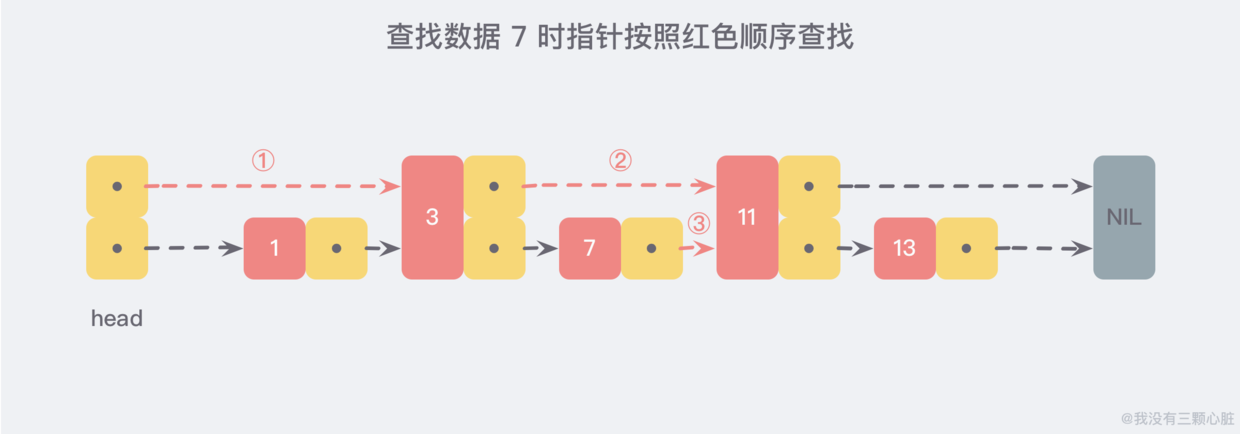
需要这个链表按照score值进行排序，这也就意味着，当我们需要添加新元素时，我们需要定位到插入点，但二分查找是有序数组的，链表没办法进行位置定位，我们除了遍历整个找到第一个比给定数据大的节点为止（时间复杂度O(n)）似乎没有更好的办法。

假如我们每相邻两个节点之间就增加一个指针，让指针指向下一节点，如下图：



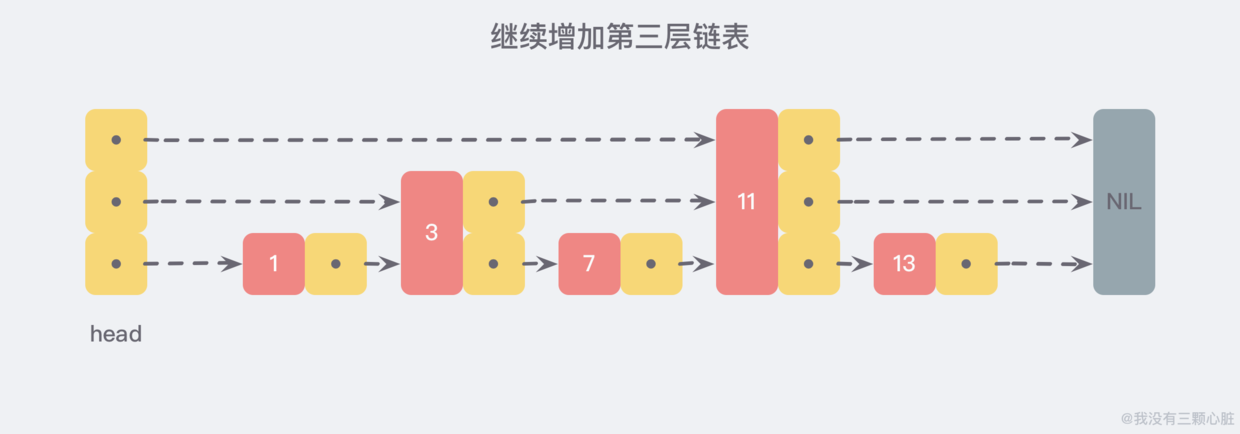
这样所有新增的指针连成一个新的链表，但它包含的数据却只有原来的一半。

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，于是我们就只需要到11后面继续查找，从而一下子跳过11前面的所有节点。

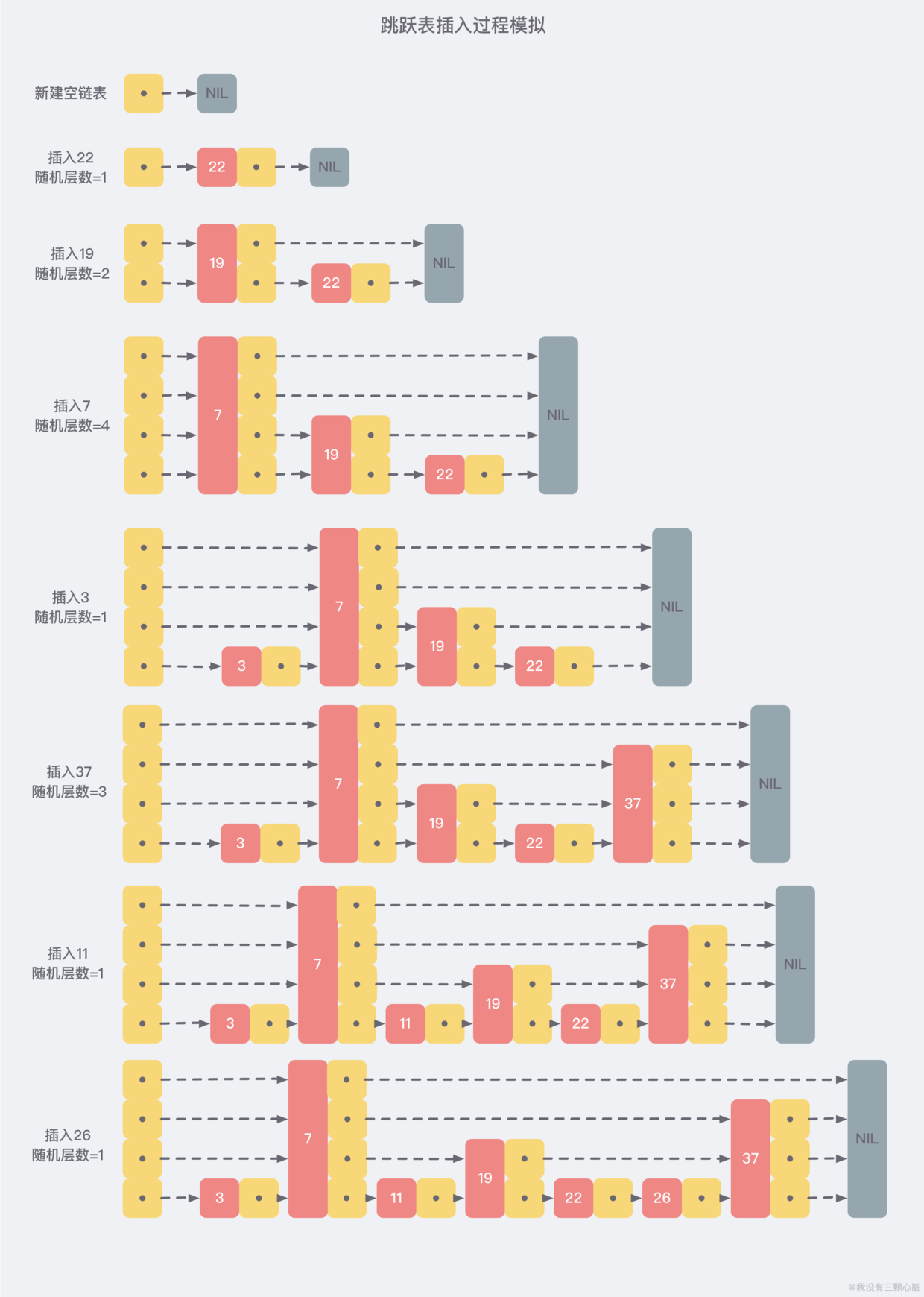
可以想象，当链表足够长，这样的多层链表结构可以帮助我们跳过很多下层节点，从而加快查找的效率。

四、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)\_。

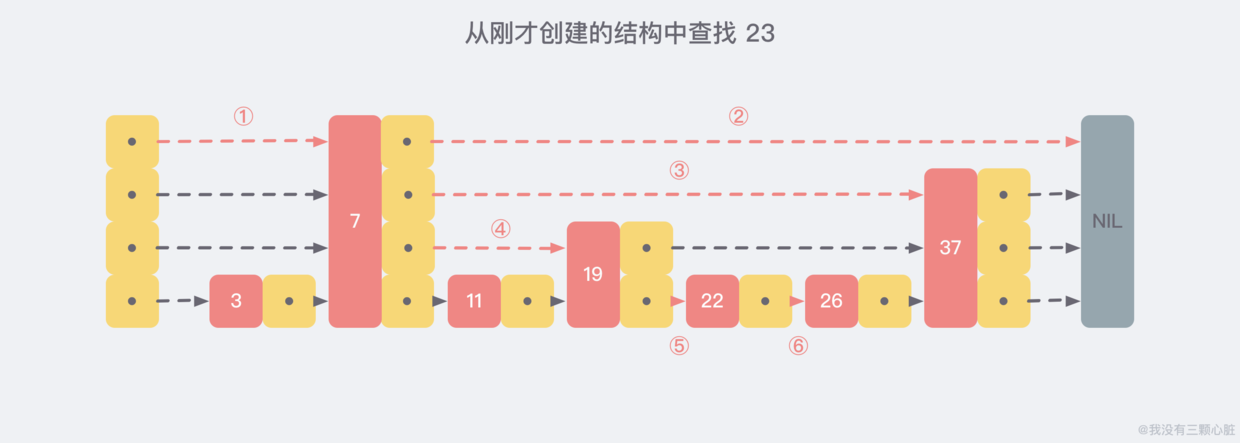
但是，这种方法在插入数据的时候有很大的问题。新插入一个节点之后，就会打乱上下相邻两层链表上节点个数严格的2:1的对应关系。如果要维持这种对应关系，就必须把新插入的节点后面的所有节点（也包括新插入的节点）重新进行调整，这会让时间复杂度重新蜕化成\_O(n)\_。删除数据也有同样的问题。

skiplist为了避免这一问题额，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：



从上面的创建和插如的过程中可以肯出，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。

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



五、跳跃表的Java实现

#### 1、空的跳跃表是头尾相连的双向链表



#### 2、向链表中放入key-value，假如key是1



此时，不断投掷硬币，如果是反面，则不提升该节点，停止投掷硬币；否则不断提升该节点，并且垂直方向进行节点连接。



#### 3、如果上一步没有提升，再插入key-value，key等于2

然后不断投掷硬币，发现是投了一次正面。需要提升一次。但第二次投的是反面，不再提升。



#### 4、再插入key-value，key等于3

然后不断投掷硬币，发现第一次就投了反面，不提升。



这样的规则一直持续下去。由于连续投掷正面的概率是0.5，0.5\*0.5......，所以某一个节点提升很多层的概率是很低的，这也是为什么说跳跃表是一种概率型数据结构的来源。

跳跃表的查询也比较简单。例如要查找key是3的节点，则从最上层开始查找，直到找到大于或等于3的位置，然后返回上一个节点，再往下一层继续向右寻找。例如三层的跳跃表查询路径如下。



这样就跳过了很多节点，所以叫“跳跃表”。

#### 5、Java代码实现

*/\*\**

*\* <p>  
 \* 跳跃表的节点，包括key-value和上下左右4个指针  
 \* <p>  
 \*  
 \** ***@author*** *chance  
 \** ***@since*** *2020-04-09  
 \*/*public class SkipListNode<T> {  
 public int key;  
 public T value;  
  
 */\*\*  
 \* 上下左右 四个指针  
 \*/* public SkipListNode<T> up, down, left, right;  
  
 */\*\*  
 \* 负无穷  
 \*/* public static final int *HEAD\_KEY* = Integer.*MIN\_VALUE*;  
 */\*\*  
 \* 正无穷  
 \*/* public static final int *TAIL\_KEY* = Integer.*MAX\_VALUE*;  
  
 public SkipListNode(int k, T v) {  
 key = k;  
 value = v;  
 }  
  
 public int getKey() {  
 return key;  
 }  
  
 public void setKey(int key) {  
 this.key = key;  
 }  
  
 public T getValue() {  
 return value;  
 }  
  
 public void setValue(T value) {  
 this.value = value;  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) {  
 return true;  
 }  
 if (o == null) {  
 return false;  
 }  
 if (!(o instanceof SkipListNode<?>)) {  
 return false;  
 }  
 SkipListNode<T> ent;  
 try {  
 // 检测类型  
 ent = (SkipListNode<T>) o;  
 } catch (ClassCastException ex) {  
 return false;  
 }  
 return (ent.getKey() == key) && (ent.getValue() == value);  
 }  
  
 @Override  
 public String toString() {  
 return "key-value:" + key + "-" + value;  
 }  
  
}

六、redis中跳跃表的实现

* 声明需要存储的变量
* 搜索当前节点插入位置
* 生成插入节点
* 重排前向指针
* 重排后向指针
* 节点删除实现
* 节点更新实现
* 元素排名的实现