# <https://mp.weixin.qq.com/s/IDls_J5ZrqY3cmbSVaZM7A>

# **二、Redis 五种基本数据结构**

**Redis** 有 5 种基础数据结构，它们分别是：**string(字符串)**、**list(列表)**、**hash(字典)**、**set(集合)** 和 **zset(有序集合)**。这 5 种是 Redis 相关知识中最基础、最重要的部分，下面我们结合源码以及一些实践来给大家分别讲解一下。

## **1）字符串 string**

Redis 中的字符串是一种 **动态字符串**，这意味着使用者可以修改，它的底层实现有点类似于 Java 中的 **ArrayList**，有一个字符数组，从源码的 **sds.h/sdshdr 文件** 中可以看到 Redis 底层对于字符串的定义 **SDS**，即 *Simple Dynamic String* 结构：

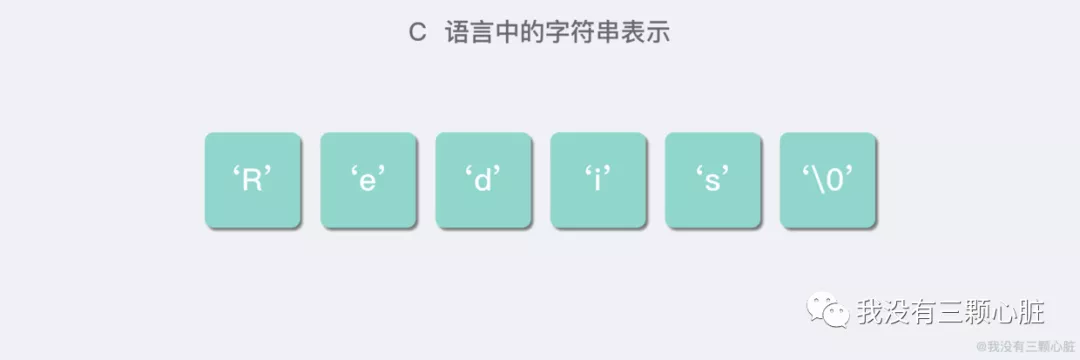


你会发现同样一组结构 Redis 使用泛型定义了好多次，**为什么不直接使用 int 类型呢？**

因为当字符串比较短的时候，len 和 alloc 可以使用 byte 和 short 来表示，**Redis 为了对内存做极致的优化，不同长度的字符串使用不同的结构体来表示。**

### **SDS 与 C 字符串的区别**

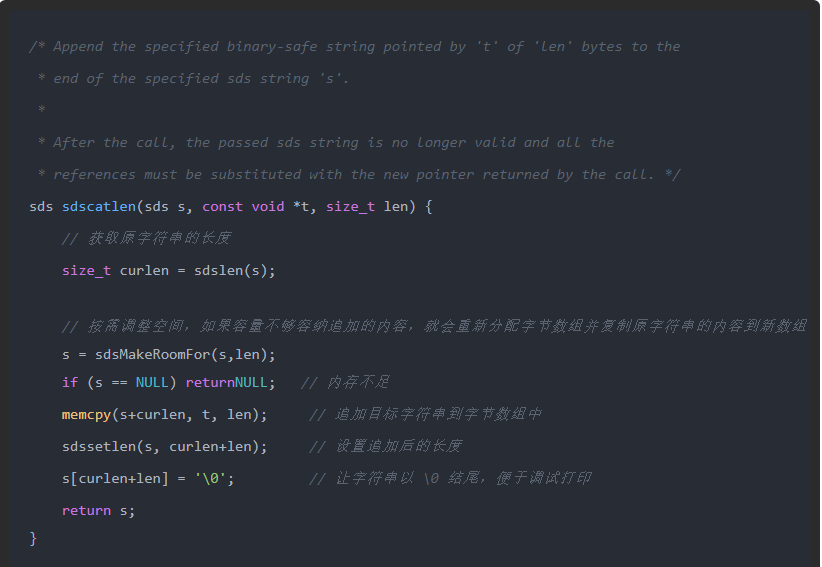
为什么不考虑直接使用 C 语言的字符串呢？因为 C 语言这种简单的字符串表示方式 **不符合 Redis 对字符串在安全性、效率以及功能方面的要求**。我们知道，C 语言使用了一个长度为 N+1 的字符数组来表示长度为 N 的字符串，并且字符数组最后一个元素总是 '\0'。*(下图就展示了 C 语言中值为 "Redis" 的一个字符数组)*



这样简单的数据结构可能会造成以下一些问题：

* **获取字符串长度为 O(N) 级别的操作** → 因为 C 不保存数组的长度，每次都需要遍历一遍整个数组；
* 不能很好的杜绝 **缓冲区溢出/内存泄漏** 的问题 → 跟上述问题原因一样，如果执行拼接 or 缩短字符串的操作，如果操作不当就很容易造成上述问题；
* C 字符串 **只能保存文本数据** → 因为 C 语言中的字符串必须符合某种编码（比如 ASCII），例如中间出现的 '\0' 可能会被判定为提前结束的字符串而识别不了；

我们以追加字符串的操作举例，Redis 源码如下：

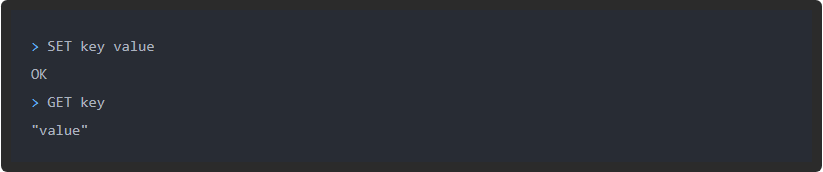


**注：Redis 规定了字符串的长度不得超过 512 MB。**

### **对字符串的基本操作**

安装好 Redis，我们可以使用 redis-cli 来对 Redis 进行命令行的操作，当然 Redis 官方也提供了在线的调试器，你也可以在里面敲入命令进行操作：http://try.redis.io/#run

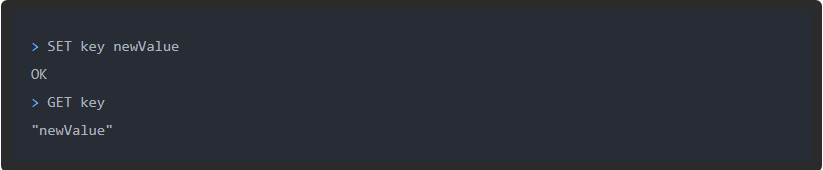
#### **设置和获取键值对**



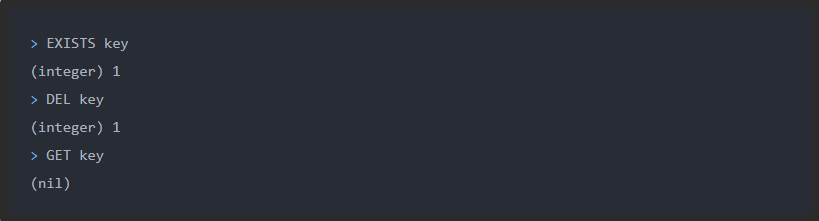
正如你看到的，我们通常使用 SET 和 GET 来设置和获取字符串值。

值可以是任何种类的字符串（包括二进制数据），例如你可以在一个键下保存一张 .jpeg 图片，只需要注意不要超过 512 MB 的最大限度就好了。

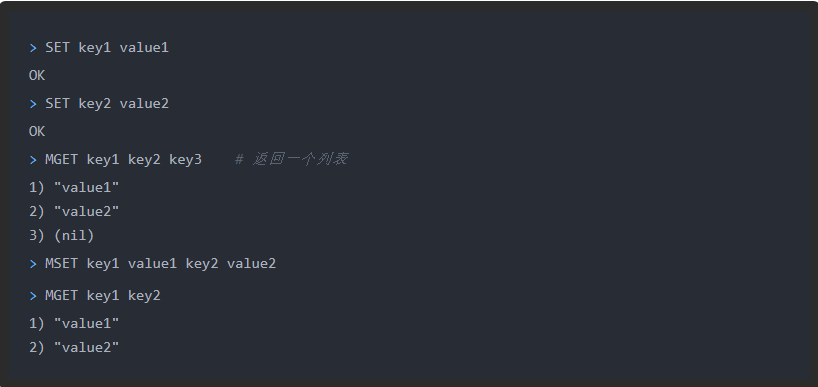
当 key 存在时，SET 命令会覆盖掉你上一次设置的值：



另外你还可以使用 EXISTS 和 DEL 关键字来查询是否存在和删除键值对：

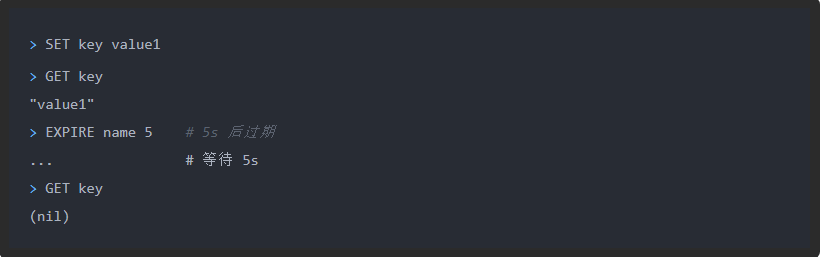


#### **批量设置键值对**

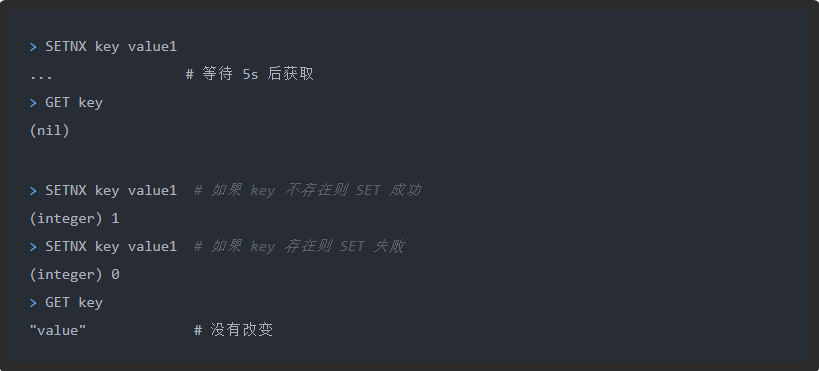


#### **过期和 SET 命令扩展**

可以对 key 设置过期时间，到时间会被自动删除，这个功能常用来控制缓存的失效时间。*(过期可以是任意数据结构)*

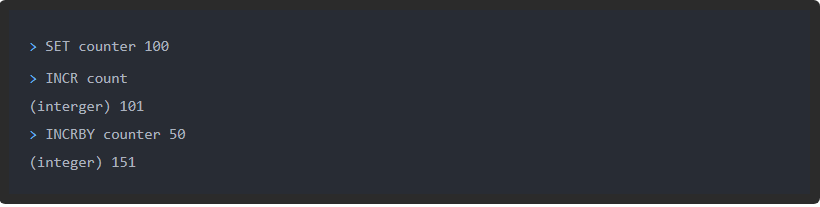


等价于 SET + EXPIRE 的 SETNX 命令：



#### **计数**

如果 value 是一个整数，还可以对它使用 INCR 命令进行 **原子性** 的自增操作，这意味着及时多个客户端对同一个 key 进行操作，也决不会导致竞争的情况：



#### **返回原值的 GETSET 命令**

对字符串，还有一个 GETSET 比较让人觉得有意思，它的功能跟它名字一样：为 key 设置一个值并返回原值：

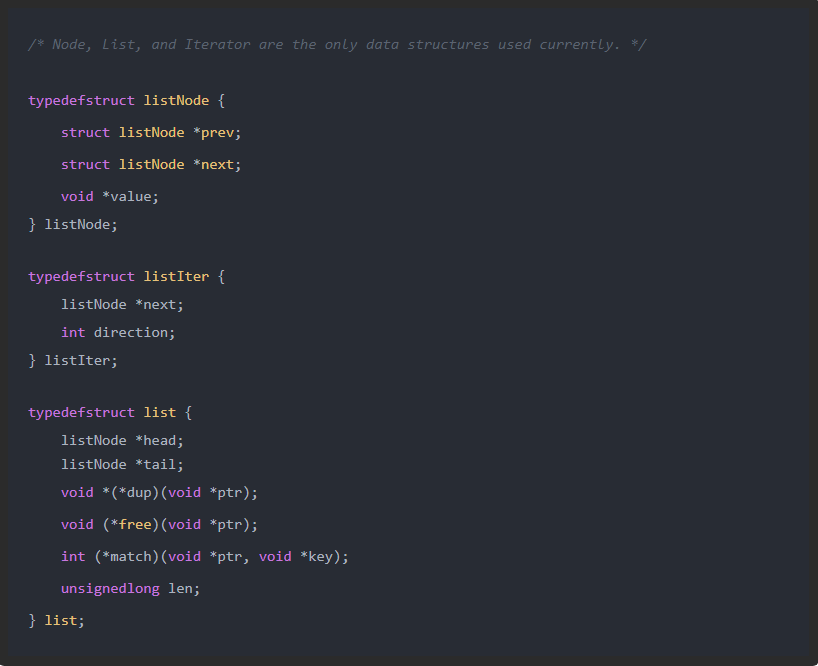


这可以对于某一些需要隔一段时间就统计的 key 很方便的设置和查看，例如：系统每当由用户进入的时候你就是用 INCR 命令操作一个 key，当需要统计时候你就把这个 key 使用 GETSET 命令重新赋值为 0，这样就达到了统计的目的。

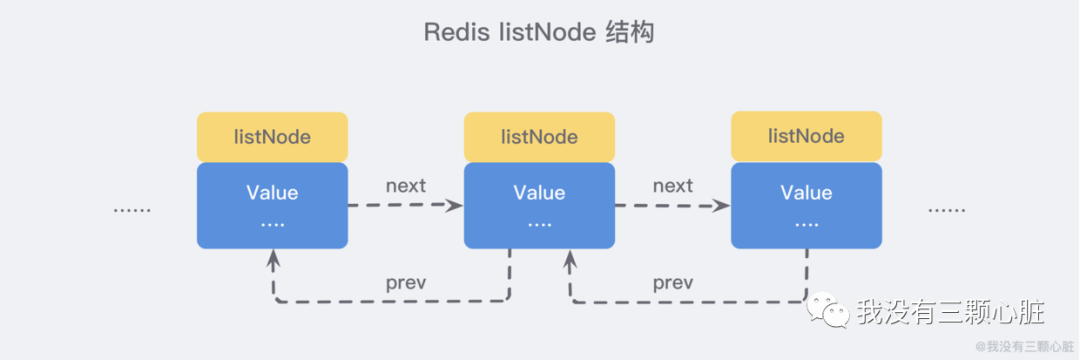
## **2）列表 list**

Redis 的列表相当于 Java 语言中的 **LinkedList**，注意它是链表而不是数组。这意味着 list 的插入和删除操作非常快，时间复杂度为 O(1)，但是索引定位很慢，时间复杂度为 O(n)。

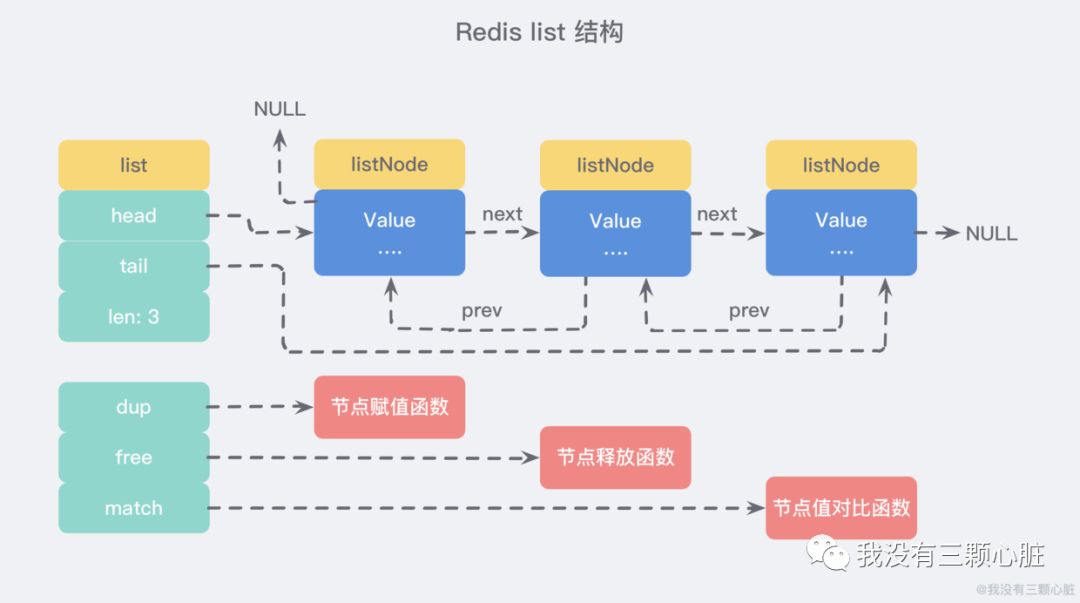
我们可以从源码的 adlist.h/listNode 来看到对其的定义：



可以看到，多个 listNode 可以通过 prev 和 next 指针组成双向链表：



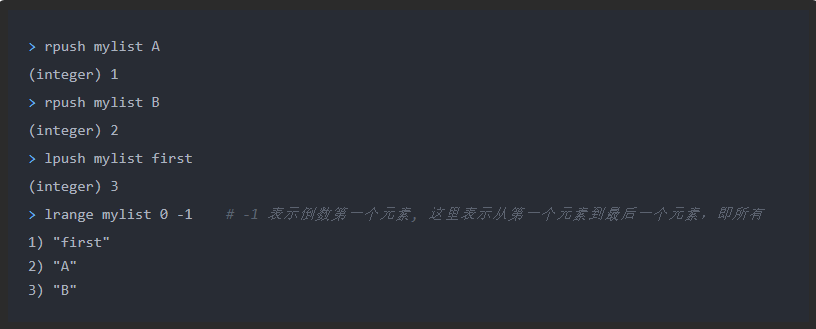
虽然仅仅使用多个 listNode 结构就可以组成链表，但是使用 adlist.h/list 结构来持有链表的话，操作起来会更加方便：



### **链表的基本操作**

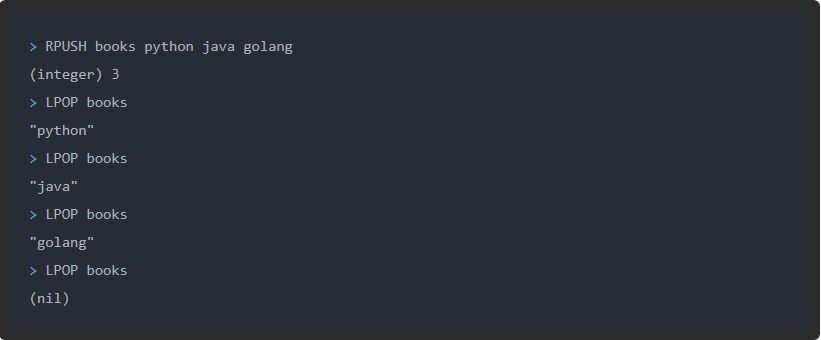
* LPUSH 和 RPUSH 分别可以向 list 的左边（头部）和右边（尾部）添加一个新元素；
* LRANGE 命令可以从 list 中取出一定范围的元素；
* LINDEX 命令可以从 list 中取出指定下表的元素，相当于 Java 链表操作中的 get(int index) 操作；

示范：



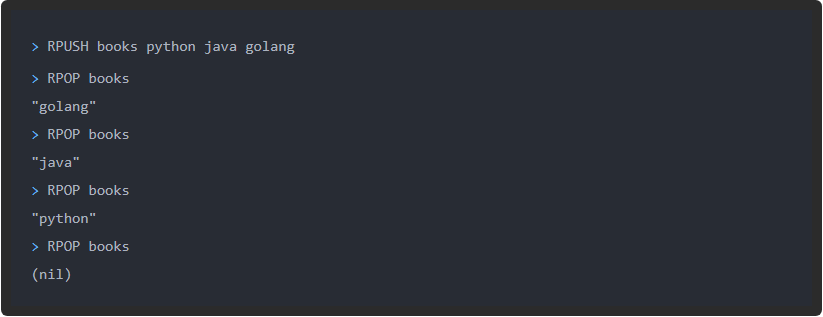
#### **list 实现队列**

队列是先进先出的数据结构，常用于消息排队和异步逻辑处理，它会确保元素的访问顺序：



#### **list 实现栈**

栈是先进后出的数据结构，跟队列正好相反：

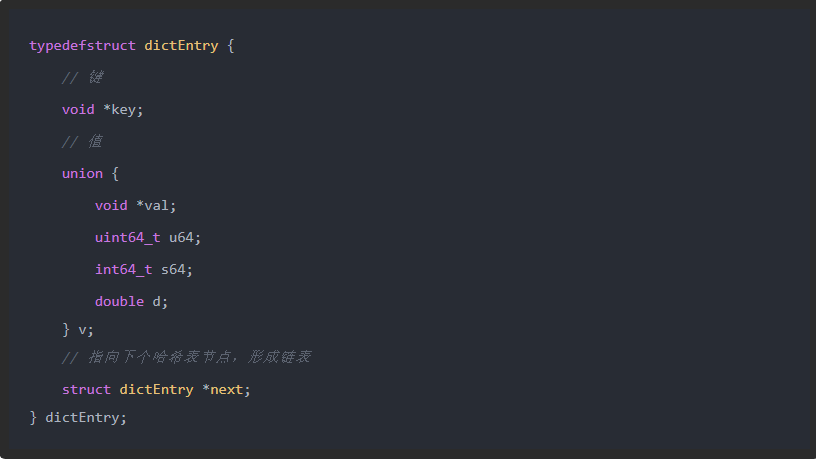


## **3）字典 hash**

Redis 中的字典相当于 Java 中的 **HashMap**，内部实现也差不多类似，都是通过 **"数组 + 链表"** 的链地址法来解决部分 **哈希冲突**，同时这样的结构也吸收了两种不同数据结构的优点。源码定义如 dict.h/dictht 定义：



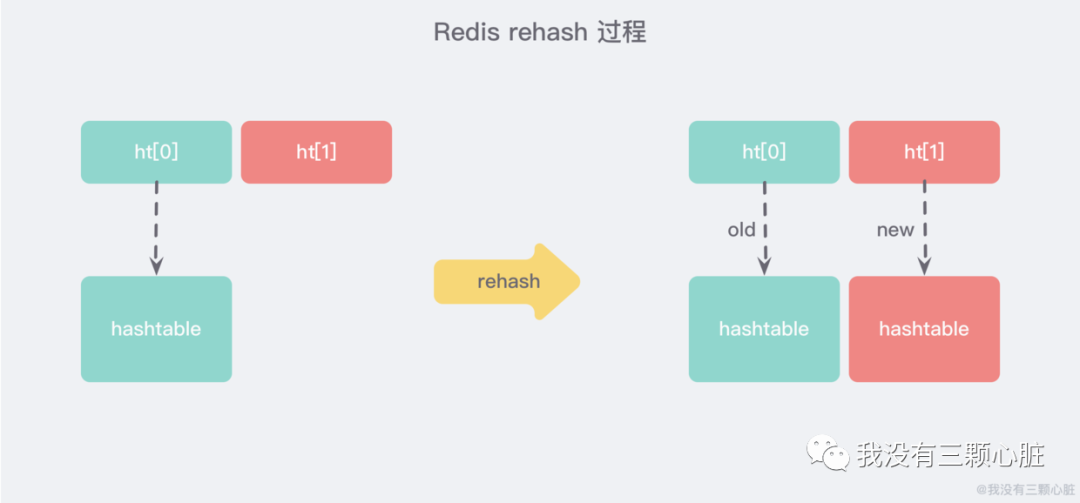
table 属性是一个数组，数组中的每个元素都是一个指向 dict.h/dictEntry 结构的指针，而每个 dictEntry 结构保存着一个键值对：



可以从上面的源码中看到，**实际上字典结构的内部包含两个 hashtable**，通常情况下只有一个 hashtable 是有值的，但是在字典扩容缩容时，需要分配新的 hashtable，然后进行 **渐进式搬迁** *(下面说原因)*。

### **渐进式 rehash**

大字典的扩容是比较耗时间的，需要重新申请新的数组，然后将旧字典所有链表中的元素重新挂接到新的数组下面，这是一个 O(n) 级别的操作，作为单线程的 Redis 很难承受这样耗时的过程，所以 Redis 使用 **渐进式 rehash** 小步搬迁：



渐进式 rehash 会在 rehash 的同时，保留新旧两个 hash 结构，如上图所示，查询时会同时查询两个 hash 结构，然后在后续的定时任务以及 hash 操作指令中，循序渐进的把旧字典的内容迁移到新字典中。当搬迁完成了，就会使用新的 hash 结构取而代之。

### **扩缩容的条件**

正常情况下，当 hash 表中 **元素的个数等于第一维数组的长度时**，就会开始扩容，扩容的新数组是 **原数组大小的 2 倍**。不过如果 Redis 正在做 bgsave(持久化命令)，为了减少内存也得过多分离，Redis 尽量不去扩容，但是如果 hash 表非常满了，**达到了第一维数组长度的 5 倍了**，这个时候就会 **强制扩容**。

当 hash 表因为元素逐渐被删除变得越来越稀疏时，Redis 会对 hash 表进行缩容来减少 hash 表的第一维数组空间占用。所用的条件是 **元素个数低于数组长度的 10%**，缩容不会考虑 Redis 是否在做 bgsave。

### **字典的基本操作**

hash 也有缺点，hash 结构的存储消耗要高于单个字符串，所以到底该使用 hash 还是字符串，需要根据实际情况再三权衡：



## **4）集合 set**

Redis 的集合相当于 Java 语言中的 **HashSet**，它内部的键值对是无序、唯一的。它的内部实现相当于一个特殊的字典，字典中所有的 value 都是一个值 NULL。

### **集合 set 的基本使用**

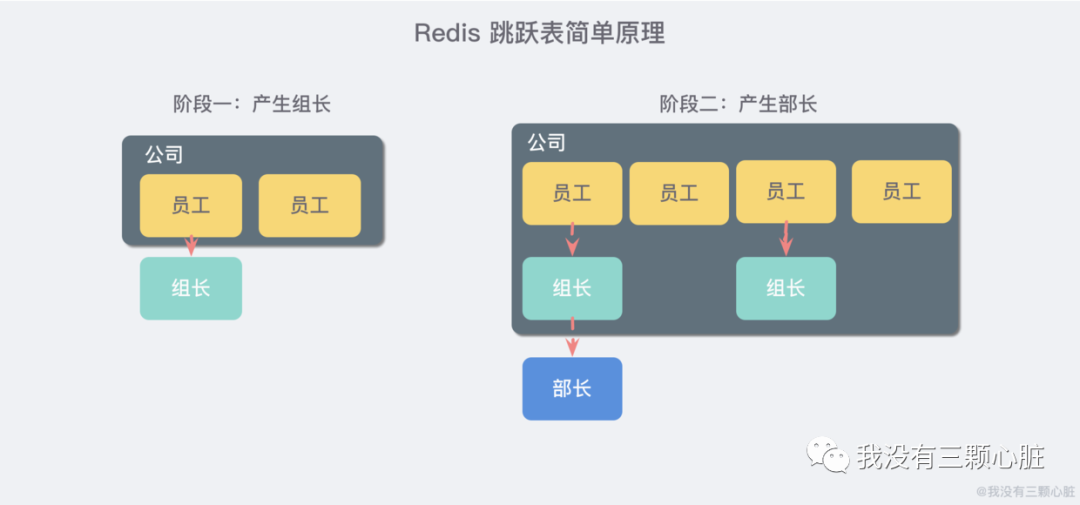
由于该结构比较简单，我们直接来看看是如何使用的：



## **5）有序列表 zset**

这可能是 Redis 最具特色的一个数据结构了，它类似于 Java 中 **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，另一方面它可以为每个 value 赋予一个 score 值，用来代表排序的权重。

它的内部实现用的是一种叫做 **「跳跃表」** 的数据结构，由于比较复杂，所以在这里简单提一下原理就好了：



想象你是一家创业公司的老板，刚开始只有几个人，大家都平起平坐。后来随着公司的发展，人数越来越多，团队沟通成本逐渐增加，渐渐地引入了组长制，对团队进行划分，于是有一些人**又是员工又有组长的身份**。

再后来，公司规模进一步扩大，公司需要再进入一个层级：部门。于是每个部门又会从组长中推举一位选出部长。

跳跃表就类似于这样的机制，最下面一层所有的元素都会串起来，都是员工，然后每隔几个元素就会挑选出一个代表，再把这几个代表使用另外一级指针串起来。然后再在这些代表里面挑出二级代表，再串起来。**最终形成了一个金字塔的结构。**

想一下你目前所在的地理位置：亚洲 > 中国 > 某省 > 某市 > ....，**就是这样一个结构！**

### **有序列表 zset 基础操作**



