

Redis 数据结构 | 小林coding | Java面试学习

 xiaolincoding.com/redis/data_struct/data_struct.html

小林coding

Redis 数据结构

大家好，我是小林。

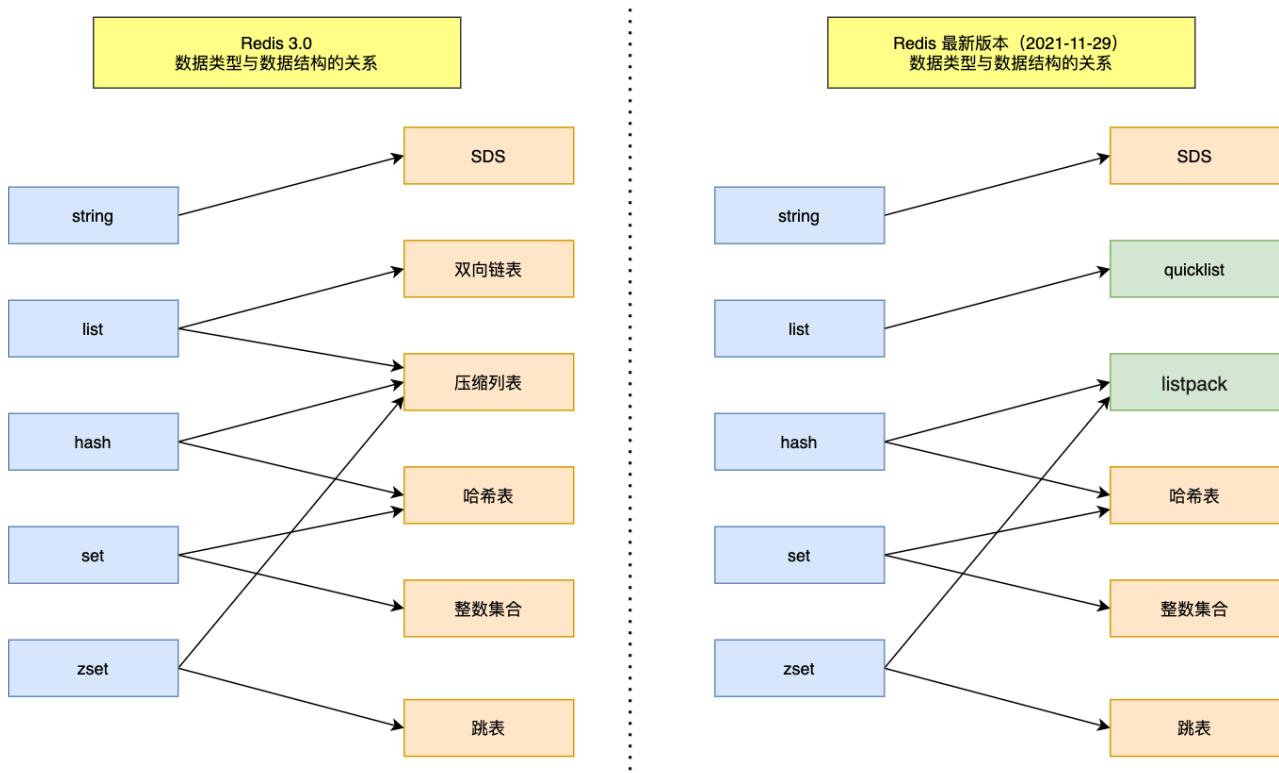
Redis 为什么那么快？

除了它是内存数据库，使得所有的操作都在内存上进行之外，还有一个重要因素，它实现的数据结构，使得我们对数据进行增删查改操作时，Redis 能高效的处理。

因此，这次我们就来好好聊一下 Redis 数据结构，这个在面试中太常问了。

注意，Redis 数据结构并不是指 String（字符串）对象、List（列表）对象、Hash（哈希）对象、Set（集合）对象和 Zset（有序集合）对象，因为这些是 Redis 键值对中值的数据类型，也就是数据的保存形式，这些对象的底层实现的方式就用到了数据结构。

我画了一张 Redis 数据类型（也叫 Redis 对象）和底层数据结构的对应关图，左边是 Redis 3.0 版本的，也就是《Redis 设计与实现》这本书讲解的版本，现在看还是有点过时了，右边是现在 Github 最新的 Redis 代码的（还未发布正式版本）。



可以看到，Redis 数据类型的底层数据结构随着版本的更新也有所不同，比如：

- 在 Redis 3.0 版本中 List 对象的底层数据结构由「双向链表」或「压缩表列表」实现，但是在 3.2 版本之后，List 数据类型底层数据结构是由 quicklist 实现的；

- 在最新的 Redis 代码（还未发布正式版本）中，压缩列表数据结构已经废弃了，交由 listpack 数据结构来实现了。

这次，小林把新旧版本的数据结构说图解一遍，共有 9 种数据结构：**SDS**、双向链表、压缩列表、哈希表、跳表、整数集合、**quicklist**、**listpack**。

不多 BB 了，直接发车！



键值对数据库是怎么实现的？

在开始讲数据结构之前，先给介绍下 Redis 是怎样实现键值对（key-value）数据库的。

Redis 的键值对中的 **key** 就是字符串对象，而 **value** 可以是字符串对象，也可以是集合数据类型的对象，比如 List 对象、Hash 对象、Set 对象和 Zset 对象。

举个例子，我这里列出几种 Redis 新增键值对的命令：

```
> SET name "xiaolinCoding"
OK

> HSET person name "xiaolinCoding" age 18
0

> RPUSH stu "xiaolin" "xiaomei"
(integer) 4
```

这些命令代表着：

- 第一条命令：name 是一个字符串键，因为键的值是一个字符串对象；
- 第二条命令：person 是一个哈希表键，因为键的值是一个包含两个键值对的哈希表对象；
- 第三条命令：stu 是一个列表键，因为键的值是一个包含两个元素的列表对象；

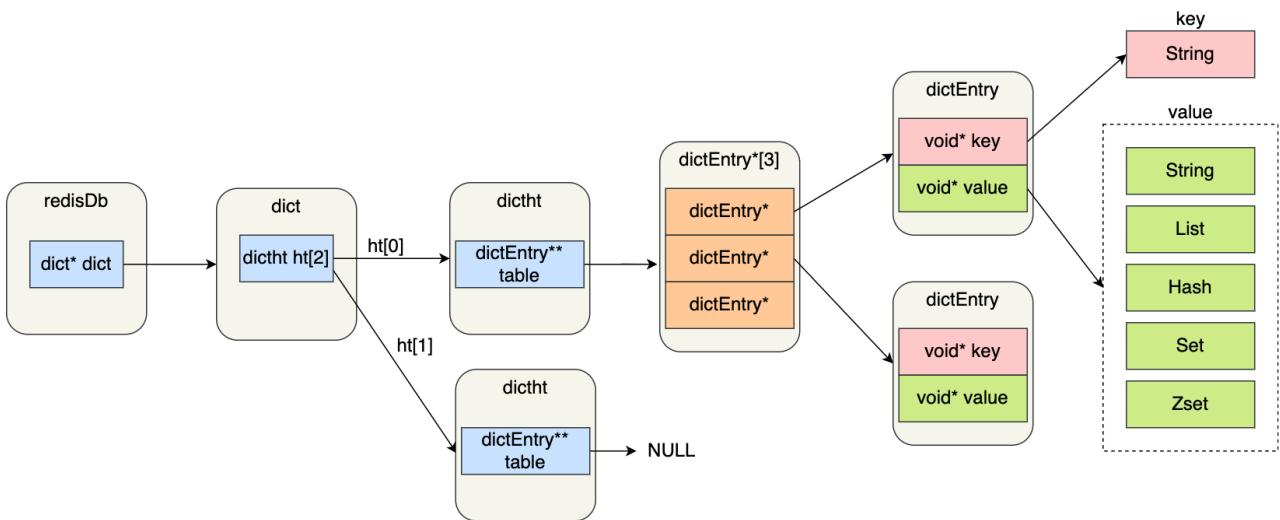
这些键值对是如何保存在 Redis 中的呢？

Redis 是使用了一个「哈希表」保存所有键值对，哈希表的最大好处就是让我们可以用 $O(1)$ 的时间复杂度来快速查找到键值对。哈希表其实就是一个数组，数组中的元素叫做哈希桶。

Redis 的哈希桶是怎么保存键值对数据的呢？

哈希桶存放的是指向键值对数据的指针（`dictEntry*`），这样通过指针就能找到键值对数据，然后因为键值对的值可以保存字符串对象和集合数据类型的对象，所以键值对的数据结构中并不是直接保存值本身，而是保存了 `void * key` 和 `void * value` 指针，分别指向了实际的键对象和值对象，这样一来，即使值是集合数据，也可以通过 `void * value` 指针找到。

我这里画了一张 Redis 保存键值对所涉及到的数据结构。

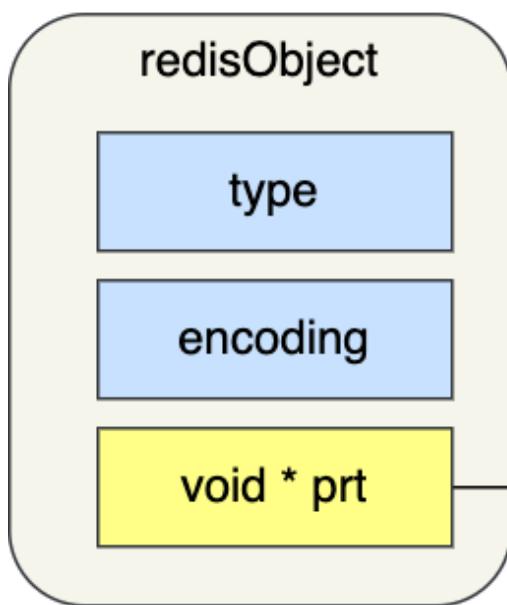


这些数据结构的内部细节，我先不展开讲，后面在讲哈希表数据结构的时候，在详细的说说，因为用到的数据结构是一样的。这里先大概说下图中涉及到的数据结构的名字和用途：

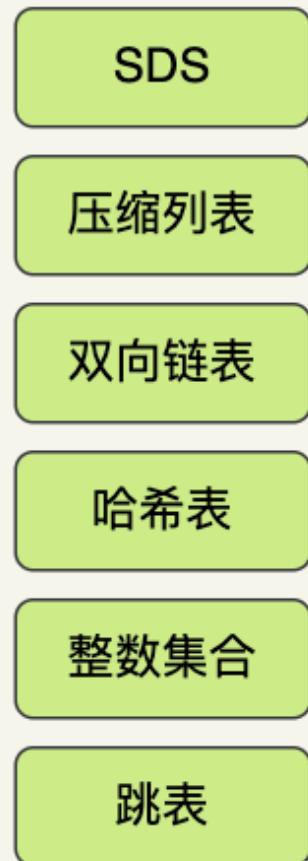
- redisDb 结构，表示 Redis 数据库的结构，结构体里存放了指向了 dict 结构的指针；
- dict 结构，结构体里存放了 2 个哈希表，正常情况下都是用「哈希表1」，「哈希表2」只有在 rehash 的时候才用，具体什么是 rehash，我在本文的哈希表数据结构会讲；
- dictht 结构，表示哈希表的结构，结构里存放了哈希表数组，数组中的每个元素都是指向一个哈希表节点结构 (dictEntry) 的指针；
- dictEntry 结构，表示哈希表节点的结构，结构里存放了 **void * key 和 void * value 指针，key 指向的是 String 对象，而 value 则可以指向 String 对象，也可以指向集合类型的对象，比如 List 对象、Hash 对象、Set 对象和 Zset 对象。

特别说明下，void * key 和 void * value 指针指向的是 **Redis 对象**，Redis 中的每个对象都由 redisObject 结构表示，如下图：

对象结构



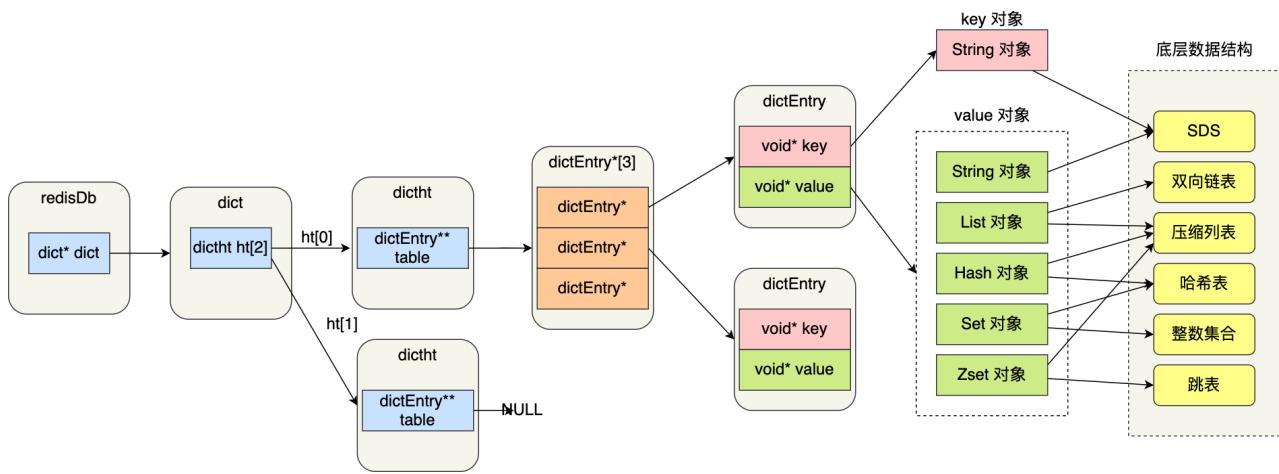
底层数据结构



对象结构里包含的成员变量：

- `type`，标识该对象是什么类型的对象（String 对象、List 对象、Hash 对象、Set 对象和 Zset 对象）；
- `encoding`，标识该对象使用了哪种底层的数据结构；
- `ptr`，指向底层数据结构的指针。

我画了一张 Redis 键值对数据库的全景图，你就能清晰知道 Redis 对象和数据结构的关系了：



接下里，就好好聊一下底层数据结构！

SDS

字符串在 Redis 中是很常用的，键值对中的键是字符串类型，值有时也是字符串类型。

Redis 是用 C 语言实现的，但是它没有直接使用 C 语言的 `char*` 字符数组来实现字符串，而是自己封装了一个名为简单动态字符串（simple dynamic string，SDS）的数据结构来表示字符串，也就是 Redis 的 String 数据类型的底层数据结构是 SDS。

既然 Redis 设计了 SDS 结构来表示字符串，肯定是 C 语言的 `char*` 字符数组存在一些缺陷。

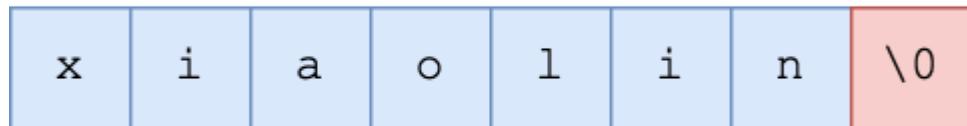
要了解这一点，得先来看看 `char*` 字符数组的结构。

C 语言字符串的缺陷

C 语言的字符串其实就是一个字符数组，即数组中每个元素是字符串中的一个字符。

比如，下图就是字符串“xiaolin”的 `char*` 字符数组的结构：

char* name = "xiaolin"



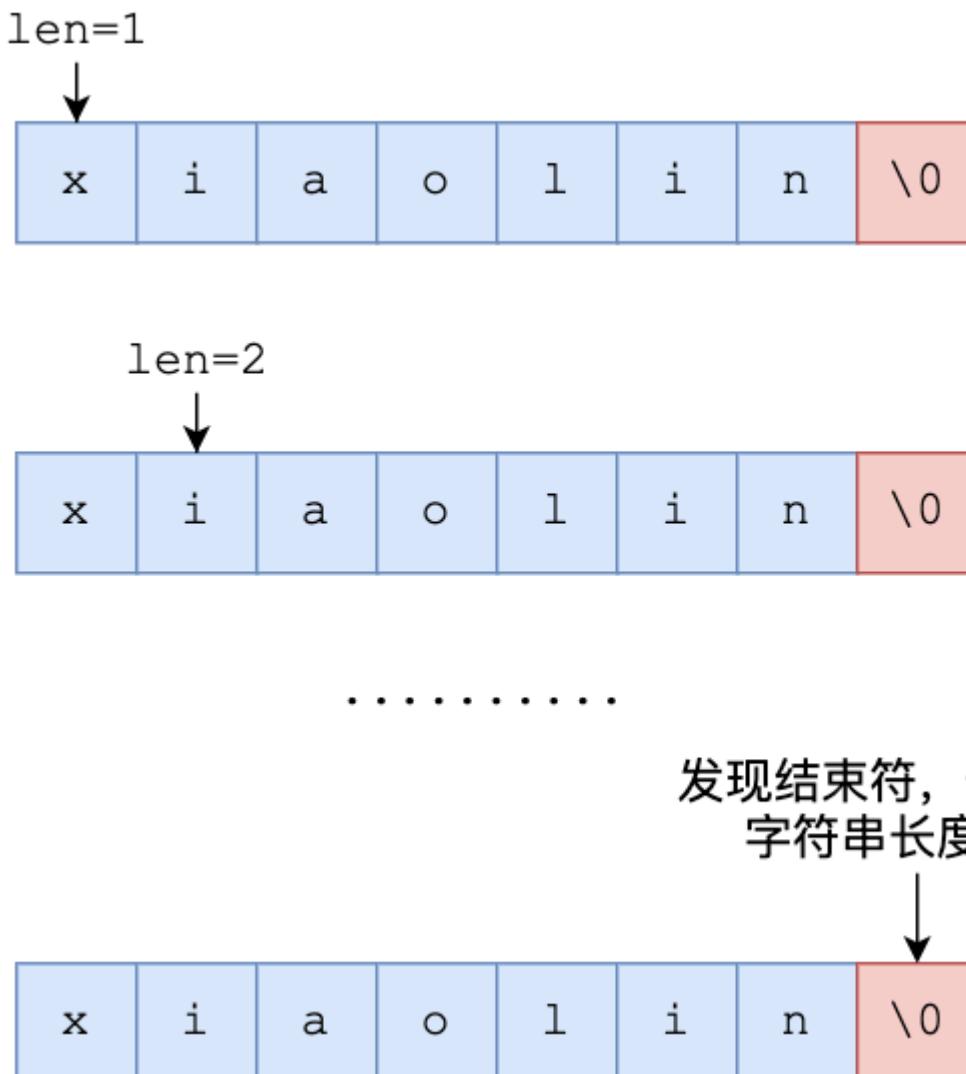
字符串“xiaolin”的字符数组结构

没学过 C 语言的同学，可能会好奇为什么最后一个字符是“\0”？

在 C 语言里，对字符串操作时，`char *` 指针只是指向字符数组的起始位置，而字符数组的结尾位置就用“\0”表示，意思是指字符串的结束。

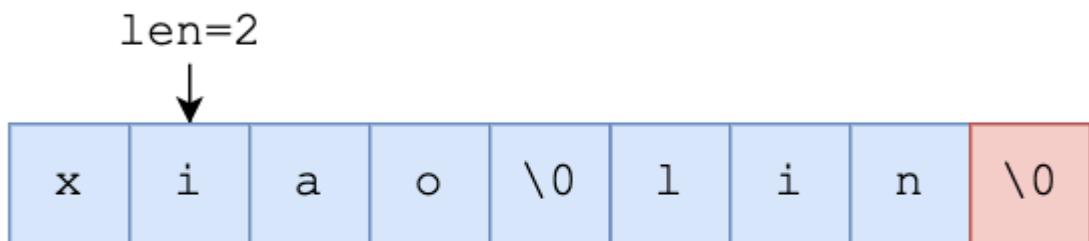
因此，C 语言标准库中的字符串操作函数就通过判断字符是不是 “\0” 来决定要不要停止操作，如果当前字符不是 “\0”，说明字符串还没结束，可以继续操作，如果当前字符是 “\0” 是则说明字符串结束了，就要停止操作。

举个例子，C 语言获取字符串长度的函数 `strlen`，就是通过字符数组中的每一个字符，并进行计数，等遇到字符为 “\0” 后，就会停止遍历，然后返回已经统计到的字符个数，即为字符串长度。下图显示了 `strlen` 函数的执行流程：



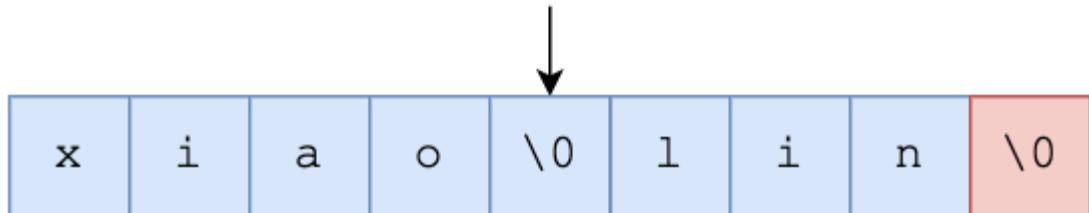
很明显，C 语言获取字符串长度的时间复杂度是 $O(N)$ （这是一个可以改进的地方）

C 语言字符串用 “\0” 字符作为结尾标记有个缺陷。假设有个字符串中有个 “\0” 字符，这时在操作这个字符串时就会提早结束，比如 “xiao\0lin” 字符串，计算字符串长度的时候则会是 4，如下图：



.....

发现结束符，停止计数
字符串长度为 4



因此，除了字符串的末尾之外，字符串里面不能含有“\0”字符，否则最先被程序读入的“\0”字符将被误认为是字符串结尾，这个限制使得 C 语言的字符串只能保存文本数据，不能保存像图片、音频、视频这样的二进制数据（这也是一个可以改进的地方）

另外，C 语言标准库中字符串的操作函数是很不安全的，对程序员很不友好，稍微一不注意，就会导致缓冲区溢出。

举个例子，`strcat` 函数是可以将两个字符串拼接在一起。

```
//将 src 字符串拼接到 dest 字符串后面
char *strcat(char *dest, const char* src);
```

C 语言的字符串是不会记录自身的缓冲区大小的，所以 `strcat` 函数假定程序员在执行这个函数时，已经为 `dest` 分配了足够多的内存，可以容纳 `src` 字符串中的所有内容，而一旦这个假定不成立，就会发生缓冲区溢出将可能会造成程序运行终止，（这是一个可以改进的地方）。

而且，`strcat` 函数和 `strlen` 函数类似，时间复杂度也很高，也都需要先通过遍历字符串才能得到目标字符串的末尾。然后对于 `strcat` 函数来说，还要再遍历源字符串才能完成追加，对字符串的操作效率不高。

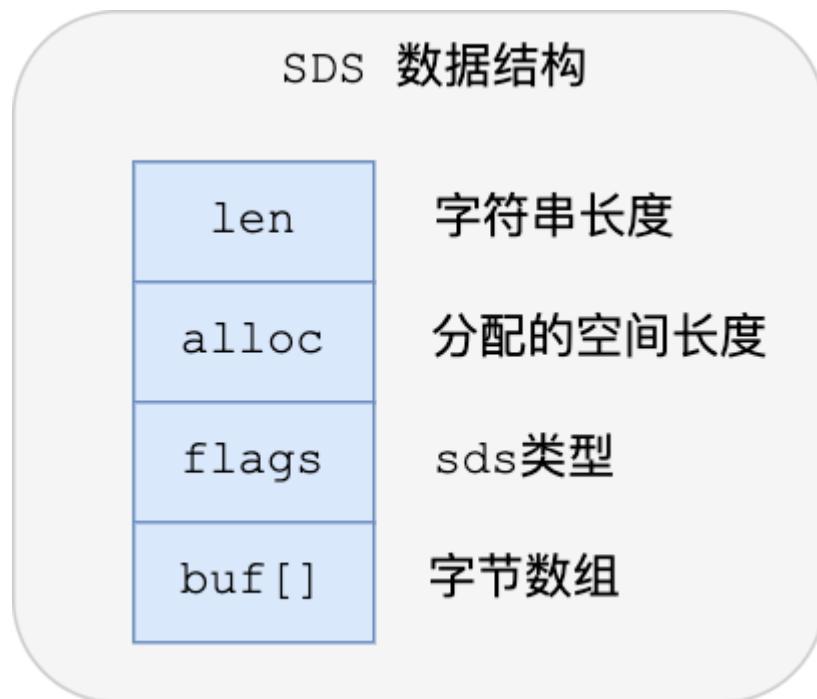
好了，通过以上的分析，我们可以得知 C 语言的字符串不足之处以及可以改进的地方：

- 获取字符串长度的时间复杂度为 $O(N)$ ；
- 字符串的结尾是以“\0”字符标识，字符串里面不能包含有“\0”字符，因此不能保存二进制数据；
- 字符串操作函数不高效且不安全，比如有缓冲区溢出的风险，有可能会造成程序运行终止；

Redis 实现的 SDS 的结构就把上面这些问题解决了，接下来我们一起看看 Redis 是如何解决的。

SDS 结构设计

下图就是 Redis 5.0 的 SDS 的数据结构：



结构中的每个成员变量分别介绍下：

- **len**，记录了字符串长度。这样获取字符串长度的时候，只需要返回这个成员变量值就行，时间复杂度只需要 $O(1)$ 。
- **alloc**，分配给字符数组的空间长度。这样在修改字符串的时候，可以通过 `alloc - len` 计算出剩余的空间大小，可以用来判断空间是否满足修改需求，如果不满足的话，就会自动将 SDS 的空间扩展至执行修改所需的大小，然后才执行实际的修改操作，所以使用 SDS 既不需要手动修改 SDS 的空间大小，也不会出现前面所说的缓冲区溢出的问题。
- **flags**，用来表示不同类型的 SDS。一共设计了 5 种类型，分别是 `sdshdr5`、`sdshdr8`、`sdshdr16`、`sdshdr32` 和 `sdshdr64`，后面在说明区别之处。
- **buf[]**，字节数组，用来保存实际数据。不仅可以保存字符串，也可以保存二进制数据。

总的来说，Redis 的 SDS 结构在原本字符数组之上，增加了三个元数据：`len`、`alloc`、`flags`，用来解决 C 语言字符串的缺陷。

O (1) 复杂度获取字符串长度

C 语言的字符串长度获取 `strlen` 函数，需要通过遍历的方式来统计字符串长度，时间复杂度是 $O(N)$ 。

而 Redis 的 SDS 结构因为加入了 `len` 成员变量，那么获取字符串长度的时候，直接返回这个成员变量的值就行，所以复杂度只有 $O(1)$ 。

二进制安全

因为 SDS 不需要用“\0”字符来标识字符串结尾了，而是有个专门的 `len` 成员变量来记录长度，所以可存储包含“\0”的数据。但是 SDS 为了兼容部分 C 语言标准库的函数，SDS 字符串结尾还是会加上“\0”字符。

因此，SDS 的 API 都是以处理二进制的方式来处理 SDS 存放在 `buf[]` 里的数据，程序不会对其中的数据做任何限制，数据写入的时候是什么样的，它被读取时就是什么样的。

通过使用二进制安全的 SDS，而不是 C 字符串，使得 Redis 不仅可以保存文本数据，也可以保存任意格式的二进制数据。

不会发生缓冲区溢出

C 语言的字符串标准库提供的字符串操作函数，大多数（比如 `strcat` 追加字符串函数）都是不安全的，因为这些函数把缓冲区大小是否满足操作需求的工作交由开发者来保证，程序内部并不会判断缓冲区大小是否足够用，当发生了缓冲区溢出就有可能造成程序异常结束。

所以，Redis 的 SDS 结构里引入了 `alloc` 和 `len` 成员变量，这样 SDS API 通过 `alloc - len` 计算，可以算出剩余可用的空间大小，这样在对字符串做修改操作的时候，就可以由程序内部判断缓冲区大小是否足够用。

而且，当判断出缓冲区大小不够用时，Redis 会自动将扩大 SDS 的空间大小，以满足修改所需的大小。

SDS 扩容的规则代码如下：

```

hisds hi_sdsMakeRoomFor(hisds s, size_t addlen)
{
    ...
    // s目前的剩余空间已足够，无需扩展，直接返回
    if (avail >= addlen)
        return s;
    //获取目前s的长度
    len = hi_sdslen(s);
    sh = (char *)s - hi_sdsHdrSize(oldtype);
    //扩展之后 s 至少需要的长度
    newlen = (len + addlen);
    //根据新长度，为s分配新空间所需要的大小
    if (newlen < HI_SDS_MAX_PREALLOC)
        //新长度<HI_SDS_MAX_PREALLOC 则分配所需空间*2的空间
        newlen *= 2;
    else
        //否则，分配长度为目前长度+1MB
        newlen += HI_SDS_MAX_PREALLOC;
    ...
}

```

- 如果所需的 sds 长度小于 1 MB，那么最后的扩容是按照翻倍扩容来执行的，即 2 倍的 newlen
- 如果所需的 sds 长度超过 1 MB，那么最后的扩容长度应该是 newlen + 1MB。

在扩容 SDS 空间之前，SDS API 会优先检查未使用空间是否足够，如果不够的话，API 不仅会为 SDS 分配修改所必须要的空间，还会给 SDS 分配额外的「未使用空间」。

这样的好处是，下次在操作 SDS 时，如果 SDS 空间够的话，API 就会直接使用「未使用空间」，而无须执行内存分配，有效的减少内存分配次数。

所以，使用 SDS 即不需要手动修改 SDS 的空间大小，也不会出现缓冲区溢出的问题。

节省内存空间

SDS 结构中有个 flags 成员变量，表示的是 SDS 类型。

Redis 一共设计了 5 种类型，分别是 sdshdr5、sdshdr8、sdshdr16、sdshdr32 和 sdshdr64。

这 5 种类型的主要区别就在于，它们数据结构中的 len 和 alloc 成员变量的数据类型不同。

比如 sdshdr16 和 sdshdr32 这两个类型，它们的定义分别如下：

```

struct __attribute__((__packed__)) sdshdr16 {
    uint16_t len;
    uint16_t alloc;
    unsigned char flags;
    char buf[];
};

struct __attribute__((__packed__)) sdshdr32 {
    uint32_t len;
    uint32_t alloc;
    unsigned char flags;
    char buf[];
};

```

可以看到：

- `sdshdr16` 类型的 `len` 和 `alloc` 的数据类型都是 `uint16_t`，表示字符数组长度和分配空间大小不能超过 2 的 16 次方。
- `sdshdr32` 则都是 `uint32_t`，表示字符数组长度和分配空间大小不能超过 2 的 32 次方。

之所以 SDS 设计不同类型的结构体，是为了能灵活保存不同大小的字符串，从而有效节省内存空间。比如，在保存小字符串时，结构头占用空间也比较少。

除了设计不同类型的结构体，Redis 在编程上还使用了专门的编译优化来节省内存空间，即在 `struct` 声明了 `__attribute__((packed))`，它的作用是：告诉编译器取消结构体在编译过程中的优化对齐，按照实际占用字节数进行对齐。

比如，`sdshdr16` 类型的 SDS，默认情况下，编译器会按照 2 字节对齐的方式给变量分配内存，这意味着，即使一个变量的大小不到 2 个字节，编译器也会给它分配 2 个字节。

举个例子，假设下面这个结构体，它有两个成员变量，类型分别是 `char` 和 `int`，如下所示：

```

#include <stdio.h>

struct test1 {
    char a;
    int b;
} test1;

int main() {
    printf("%lu\n", sizeof(test1));
    return 0;
}

```

大家猜猜这个结构体大小是多少？我先直接说答案，这个结构体大小计算出来是 8。



字节对齐方式的内存分配

这是因为默认情况下，编译器是使用「字节对齐」的方式分配内存，虽然 char 类型只占一个字节，但是由于成员变量里有 int 类型，它占用了 4 个字节，所以在成员变量为 char 类型分配内存时，会分配 4 个字节，其中这多余的 3 个字节是为了字节对齐而分配的，相当于有 3 个字节被浪费掉了。

如果不希望编译器使用字节对齐的方式进行分配内存，可以采用了 `__attribute__((packed))` 属性定义结构体，这样一来，结构体实际占用多少内存空间，编译器就分配多少空间。

比如，我用 `__attribute__((packed))` 属性定义下面的结构体，同样包含 char 和 int 两个类型的成员变量，代码如下所示：

```
#include <stdio.h>

struct __attribute__((packed)) test2 {
    char a;
    int b;
} test2;

int main() {
    printf("%lu\n", sizeof(test2));
    return 0;
}
```

这时打印的结果是 5 (1 个字节 char + 4 字节 int)。

可以看得出，这是按照实际占用字节数进行分配内存的，这样可以节省内存空间。

链表

大家最熟悉的数据结构除了数组之外，我相信就是链表了。

Redis 的 List 对象的底层实现之一就是链表。C 语言本身没有链表这个数据结构的，所以 Redis 自己设计了一个链表数据结构。

链表节点结构设计

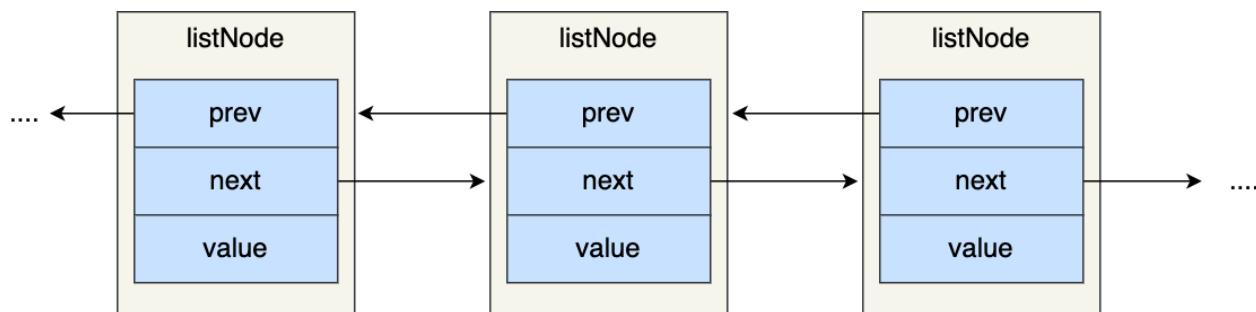
先来看看「链表节点」结构的样子：

```

typedef struct listNode {
    //前置节点
    struct listNode *prev;
    //后置节点
    struct listNode *next;
    //节点的值
    void *value;
} listNode;

```

有前置节点和后置节点，可以看的出，这个是一个双向链表。



链表结构设计

不过，Redis 在 listNode 结构体基础上又封装了 list 这个数据结构，这样操作起来会更方便，链表结构如下：

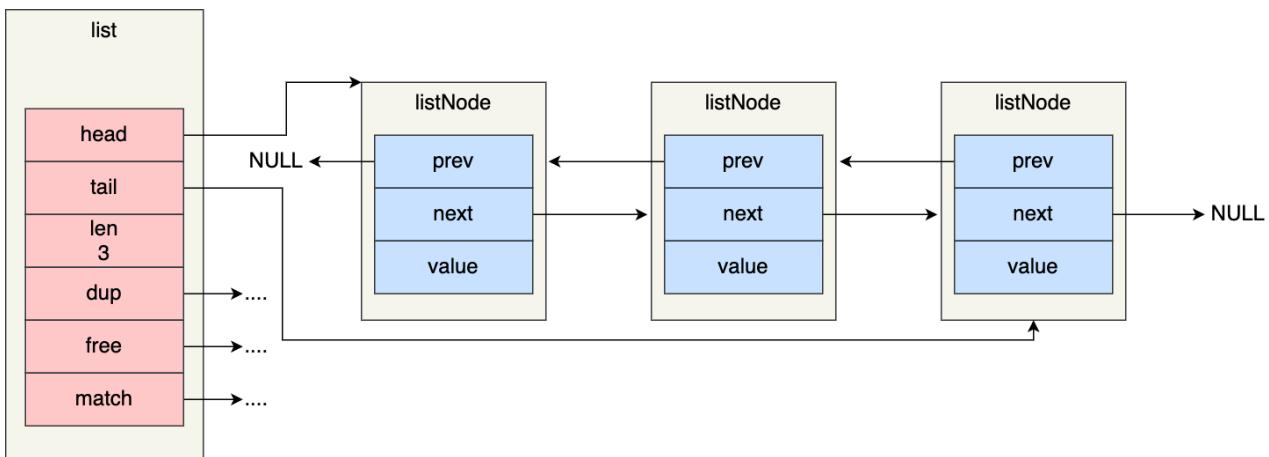
```

typedef struct list {
    //链表头节点
    listNode *head;
    //链表尾节点
    listNode *tail;
    //节点值复制函数
    void *(*dup)(void *ptr);
    //节点值释放函数
    void (*free)(void *ptr);
    //节点值比较函数
    int (*match)(void *ptr, void *key);
    //链表节点数量
    unsigned long len;
} list;

```

list 结构为链表提供了链表头指针 head、链表尾节点 tail、链表节点数量 len、以及可以自定义实现的 dup、free、match 函数。

举个例子，下面是由 list 结构和 3 个 listNode 结构组成的链表。



链表的优势与缺陷

Redis 的链表实现优点如下：

- `listNode` 链表节点的结构里带有 `prev` 和 `next` 指针，**获取某个节点的前置节点或后置节点的时间复杂度只需O(1)**，而且这两个指针都可以指向 `NULL`，所以链表是无环链表；
- `list` 结构因为提供了表头指针 `head` 和表尾节点 `tail`，所以**获取链表的表头节点和表尾节点的时间复杂度只需O(1)**；
- `list` 结构因为提供了链表节点数量 `len`，所以**获取链表中的节点数量的时间复杂度只需O(1)**；
- `listNode` 链表节使用 `void*` 指针保存节点值，并且可以通过 `list` 结构的 `dup`、`free`、`match` 函数指针为节点设置该节点类型特定的函数，因此**链表节点可以保存各种不同类型的值**；

链表的缺陷也是有的：

- 链表每个节点之间的内存都是不连续的，意味着**无法很好利用 CPU 缓存**。能很好利用 CPU 缓存的数据结构就是数组，因为数组的内存是连续的，这样就可以充分利用 CPU 缓存来加速访问。
- 还有一点，保存一个链表节点的值都需要一个链表节点结构头的分配，**内存开销较大**。

因此，Redis 3.0 的 List 对象在数据量比较少的情况下，会采用「压缩列表」作为底层数据结构的实现，它的优势是节省内存空间，并且是内存紧凑型的数据结构。

不过，压缩列表存在性能问题（具体什么问题，下面会说），所以 Redis 在 3.2 版本设计了新的数据结构 quicklist，并将 List 对象的底层数据结构改由 quicklist 实现。

然后在 Redis 5.0 设计了新的数据结构 listpack，沿用了压缩列表紧凑型的内存布局，最终在最新的 Redis 版本，将 Hash 对象和 Zset 对象的底层数据结构实现之一的压缩列表，替换成由 listpack 实现。

压缩列表

压缩列表的最大特点，就是它被设计成一种内存紧凑型的数据结构，占用一块连续的内存空间，不仅可以利用 CPU 缓存，而且会针对不同长度的数据，进行相应编码，这种方法可以有效地节省内存开销。

但是，压缩列表的缺陷也是有的：

- 不能保存过多的元素，否则查询效率就会降低；
- 新增或修改某个元素时，压缩列表占用的内存空间需要重新分配，甚至可能引发连锁更新的问题。

因此，Redis 对象（List 对象、Hash 对象、Zset 对象）包含的元素数量较少，或者元素值不大的情况才会使用压缩列表作为底层数据结构。

接下来，就跟大家详细聊下压缩列表。

压缩列表结构设计

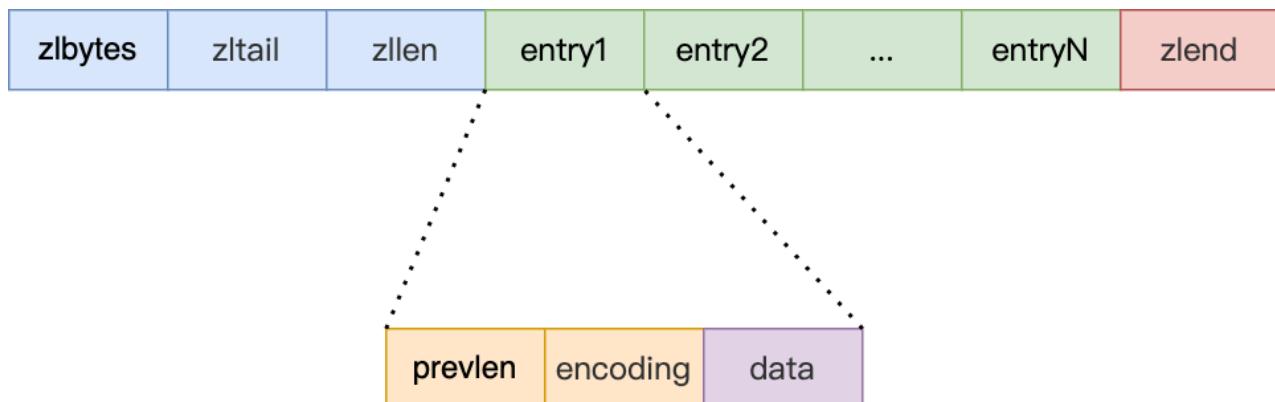
压缩列表是 Redis 为了节约内存而开发的，它是由连续内存块组成的顺序型数据结构，有点类似于数组。

压缩列表在表头有三个字段：

- zbytes**，记录整个压缩列表占用对内存字节数；
- zltail**，记录压缩列表「尾部」节点距离起始地址由多少字节，也就是列表尾的偏移量；
- zllen**，记录压缩列表包含的节点数量；
- zlend**，标记压缩列表的结束点，固定值 0xFF（十进制 255）。

在压缩列表中，如果我们要查找定位第一个元素和最后一个元素，可以通过表头三个字段（zllen）的长度直接定位，复杂度是 O(1)。而查找其他元素时，就没有这么高效了，只能逐个查找，此时的复杂度就是 O(N) 了，因此压缩列表不适合保存过多的元素。

另外，压缩列表节点（entry）的构成如下：



压缩列表节点包含三部分内容：

- prevlen**，记录了「前一个节点」的长度，目的是为了实现从后向前遍历；

- **encoding**，记录了当前节点实际数据的「类型和长度」，类型主要有两种：字符串和整数。
- **data**，记录了当前节点的实际数据，类型和长度都由 **encoding** 决定；

当我们往压缩列表中插入数据时，压缩列表就会根据数据类型是字符串还是整数，以及数据的大小，会使用不同空间大小的 prevlen 和 encoding 这两个元素里保存的信息，这种根据数据大小和类型进行不同的空间大小分配的设计思想，正是 Redis 为了节省内存而采用的。

分别说下，prevlen 和 encoding 是如何根据数据的大小和类型来进行不同的空间大小分配。

压缩列表里的每个节点中的 prevlen 属性都记录了「前一个节点的长度」，而且 prevlen 属性的空间大小跟前一个节点长度值有关，比如：

- 如果前一个节点的长度小于 254 字节，那么 prevlen 属性需要用 1 字节的空间来保存这个长度值；
- 如果前一个节点的长度大于等于 254 字节，那么 prevlen 属性需要用 5 字节的空间来保存这个长度值；

encoding 属性的空间大小跟数据是字符串还是整数，以及字符串的长度有关，如下图（下图中的 content 表示的是实际数据，即本文的 data 字段）：

本图来源于《Redis 5设计与源码分析》

表4-1 压缩列表元素的编码

encoding 编码	encoding 长度	content 类型
00 bbbbb (6 比特表示 content 长度)	1 字节	最大长度为 63 的字节数组
01 bbbbb xxxx (14 比特表示 content 长度)	2 字节	最大长度为 $2^{14}-1$ 的字节数组
10——aaaaaaaa bbbbbbbb cccccccc dddddddd (32 比特表示 content 长度)	5 字节	最大长度为 $2^{32}-1$ 的字节数组
11 00 0000	1 字节	int16 整数
11 01 0000	1 字节	int32 整数
11 10 0000	1 字节	int64 整数
11 11 0000	1 字节	24 位整数
11 11 1110	1 字节	8 位整数
11 11 xxxx	1 字节	没有 content 字段； xxxx 表示 0~12 的整数

<https://blog.csdn.net/Edidaugther>

- 如果当前节点的数据是整数，则 encoding 会使用 1 字节的空间进行编码，也就是 encoding 长度为 1 字节。通过 encoding 确认了整数类型，就可以确认整数数据的实际大小了，比如如果 encoding 编码确认了数据是 int16 整数，那么 data 的长度就是 int16 的大小。
- 如果当前节点的数据是字符串，根据字符串的长度大小，encoding 会使用 1 字节/2 字节/5 字节的空间进行编码，encoding 编码的前两个 bit 表示数据的类型，后续的其他 bit 标识字符串数据的实际长度，即 data 的长度。

连锁更新

压缩列表除了查找复杂度高的问题，还有一个问题。

压缩列表新增某个元素或修改某个元素时，如果空间不够，压缩列表占用的内存空间就需要重新分配。而当新插入的元素较大时，可能会导致后续元素的 `prevlen` 占用空间都发生变化，从而引起「连锁更新」问题，导致每个元素的空间都要重新分配，造成访问压缩列表性能的下降。

前面提到，压缩列表节点的 `prevlen` 属性会根据前一个节点的长度进行不同的空间大小分配：

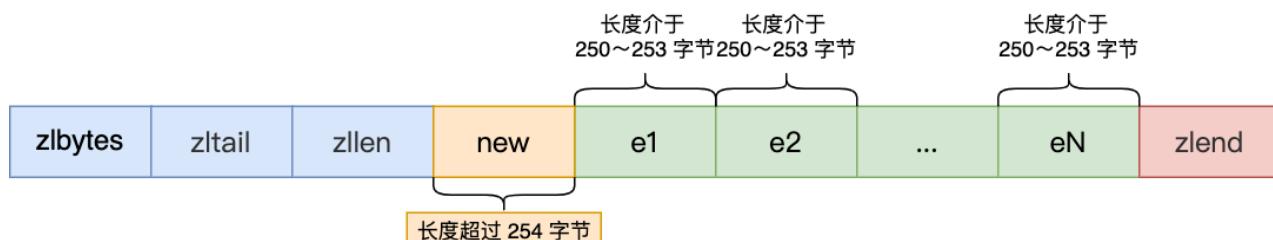
- 如果前一个节点的长度小于 254 字节，那么 `prevlen` 属性需要用 1 字节的空间来保存这个长度值；
- 如果前一个节点的长度大于等于 254 字节，那么 `prevlen` 属性需要用 5 字节的空间来保存这个长度值；

现在假设一个压缩列表中有多个连续的、长度在 250~253 之间的节点，如下图：



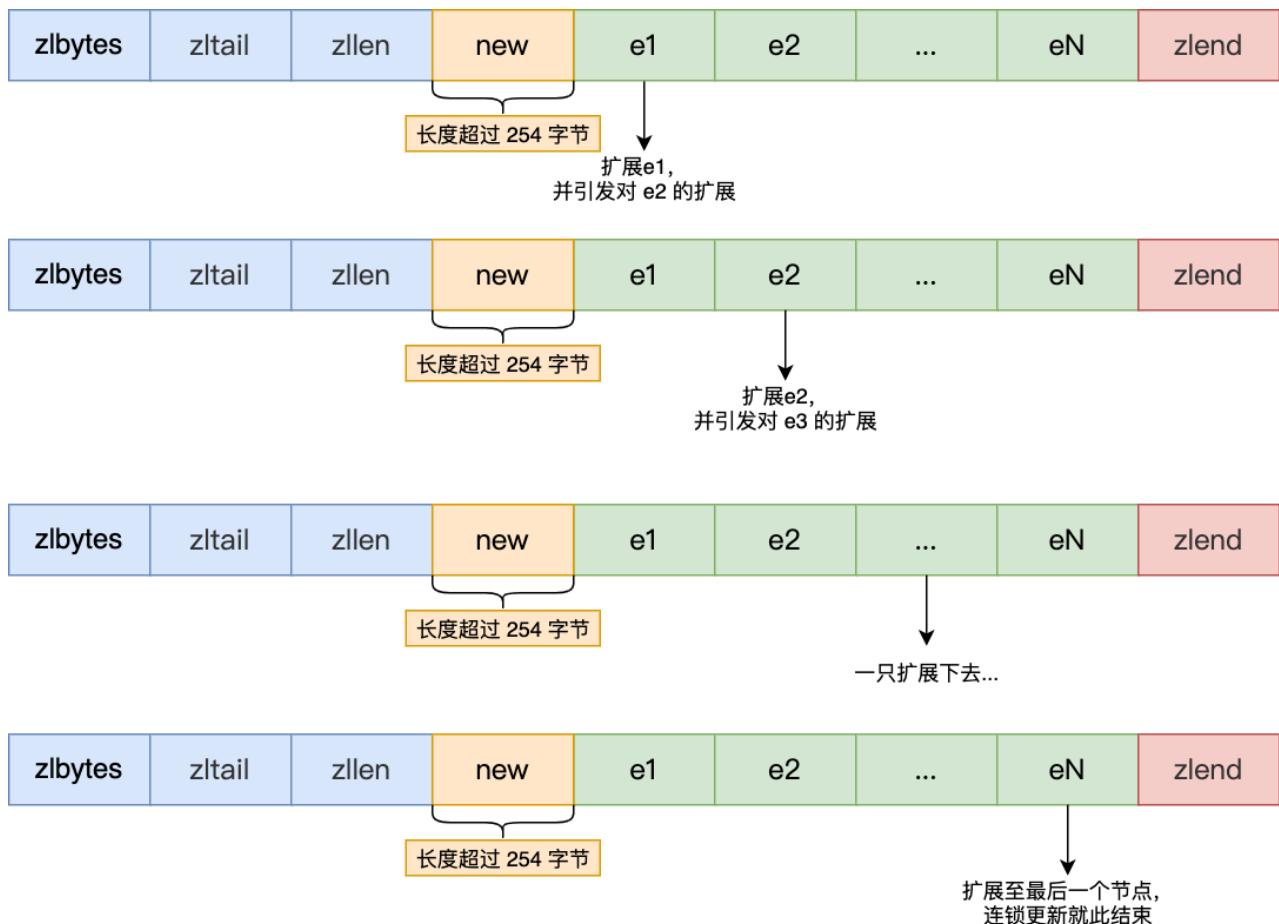
因为这些节点长度值小于 254 字节，所以 `prevlen` 属性需要用 1 字节的空间来保存这个长度值。

这时，如果将一个长度大于等于 254 字节的新节点加入到压缩列表的表头节点，即新节点将成为 `e1` 的前置节点，如下图：



因为 `e1` 节点的 `prevlen` 属性只有 1 个字节大小，无法保存新节点的长度，此时就需要对压缩列表的空间重分配操作，并将 `e1` 节点的 `prevlen` 属性从原来的 1 字节大小扩展为 5 字节大小。

多米诺牌的效应就此开始。



e1 原本的长度在 250 ~ 253 之间，因为刚才的扩展空间，此时 e1 的长度就大于等于 254 了，因此原本 e2 保存 e1 的 prevlen 属性也必须从 1 字节扩展至 5 字节大小。

正如扩展 e1 引发了对 e2 扩展一样，扩展 e2 也会引发对 e3 的扩展，而扩展 e3 又会引发对 e4 的扩展.... 一直持续到结尾。

这种在特殊情况下产生的连续多次空间扩展操作就叫做「链锁更新」，就像多米诺牌的效应一样，第一张牌倒下了，推动了第二张牌倒下；第二张牌倒下，又推动了第三张牌倒下....，

压缩列表的缺陷

空间扩展操作也就是重新分配内存，因此链锁更新一旦发生，就会导致压缩列表占用的内存空间要多次重新分配，这就会直接影响到压缩列表的访问性能。

所以说，虽然压缩列表紧凑型的内存布局能节省内存开销，但是如果保存的元素数量增加了，或是元素变大了，会导致内存重新分配，最糟糕的是会有「链锁更新」的问题。

因此，压缩列表只会用于保存的节点数量不多的场景，只要节点数量足够小，即使发生链锁更新，也是能接受的。

虽说如此，Redis 针对压缩列表在设计上的不足，在后来的版本中，新增设计了两种数据结构：quicklist（Redis 3.2 引入）和 listpack（Redis 5.0 引入）。这两种数据结构的设计目标，就是尽可能地保持压缩列表节省内存的优势，同时解决压缩列表的「链锁更新」的问题。

哈希表

哈希表是一种保存键值对 (key-value) 的数据结构。

哈希表中的每一个 key 都是独一无二的，程序可以根据 key 查找到与之关联的 value，或者通过 key 来更新 value，又或者根据 key 来删除整个 key-value 等等。

在讲压缩列表的时候，提到过 Redis 的 Hash 对象的底层实现之一是压缩列表（最新 Redis 代码已将压缩列表替换成 listpack）。Hash 对象的另外一个底层实现就是哈希表。

哈希表优点在于，它能以 **O(1)** 的复杂度快速查询数据。怎么做到的呢？将 key 通过 Hash 函数的计算，就能定位数据在表中的位置，因为哈希表实际上是数组，所以可以通过索引值快速查询到数据。

但是存在的风险也是有，在哈希表大小固定的情况下，随着数据不断增多，那么哈希冲突的可能性也会越高。

解决哈希冲突的方式，有很多种。

Redis 采用了「链式哈希」来解决哈希冲突，在不扩容哈希表的前提下，将具有相同哈希值的数据串起来，形成链接起，以便这些数据在表中仍然可以被查询到。

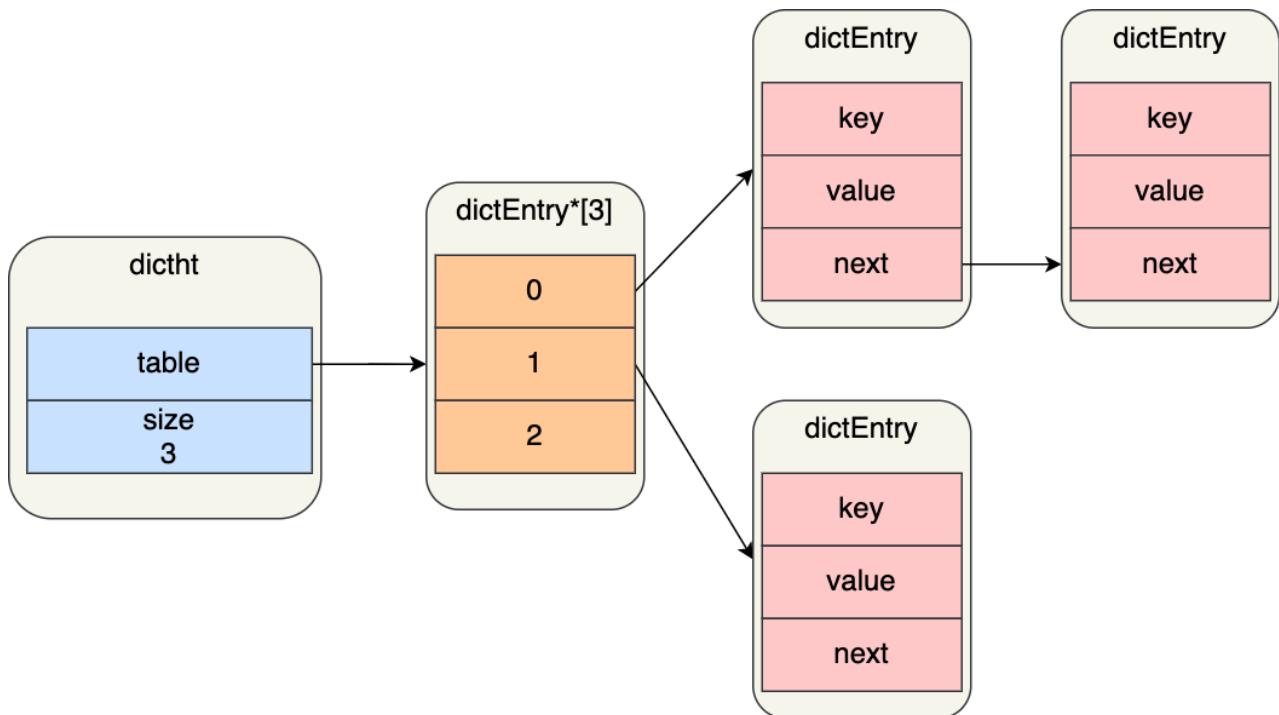
接下来，详细说说哈希表。

哈希表结构设计

Redis 的哈希表结构如下：

```
typedef struct dictht {
    //哈希表数组
    dictEntry **table;
    //哈希表大小
    unsigned long size;
    //哈希表大小掩码，用于计算索引值
    unsigned long sizemask;
    //该哈希表已有的节点数量
    unsigned long used;
} dictht;
```

可以看到，哈希表是一个数组 (dictEntry **table)，数组的每个元素是一个指向「哈希表节点 (dictEntry)」的指针。



哈希表节点的结构如下：

```
typedef struct dictEntry {
    //键值对中的键
    void *key;

    //键值对中的值
    union {
        void *val;
        uint64_t u64;
        int64_t s64;
        double d;
    } v;
    //指向下一个哈希表节点，形成链表
    struct dictEntry *next;
} dictEntry;
```

dictEntry 结构里不仅包含指向键和值的指针，还包含了指向下一个哈希表节点的指针，这个指针可以将多个哈希值相同的键值对链接起来，以此来解决哈希冲突的问题，这就是链式哈希。

另外，这里还跟你提一下，dictEntry 结构里键值对中的值是一个「联合体 v」定义的，因此，键值对中的值可以是一个指向实际值的指针，或者是一个无符号的 64 位整数或有符号的 64 位整数或double 类的值。这么做的好处是可以节省内存空间，因为当「值」是整数或浮点数时，就可以将值的数据内嵌在 dictEntry 结构里，无需再用一个指针指向实际的值，从而节省了内存空间。

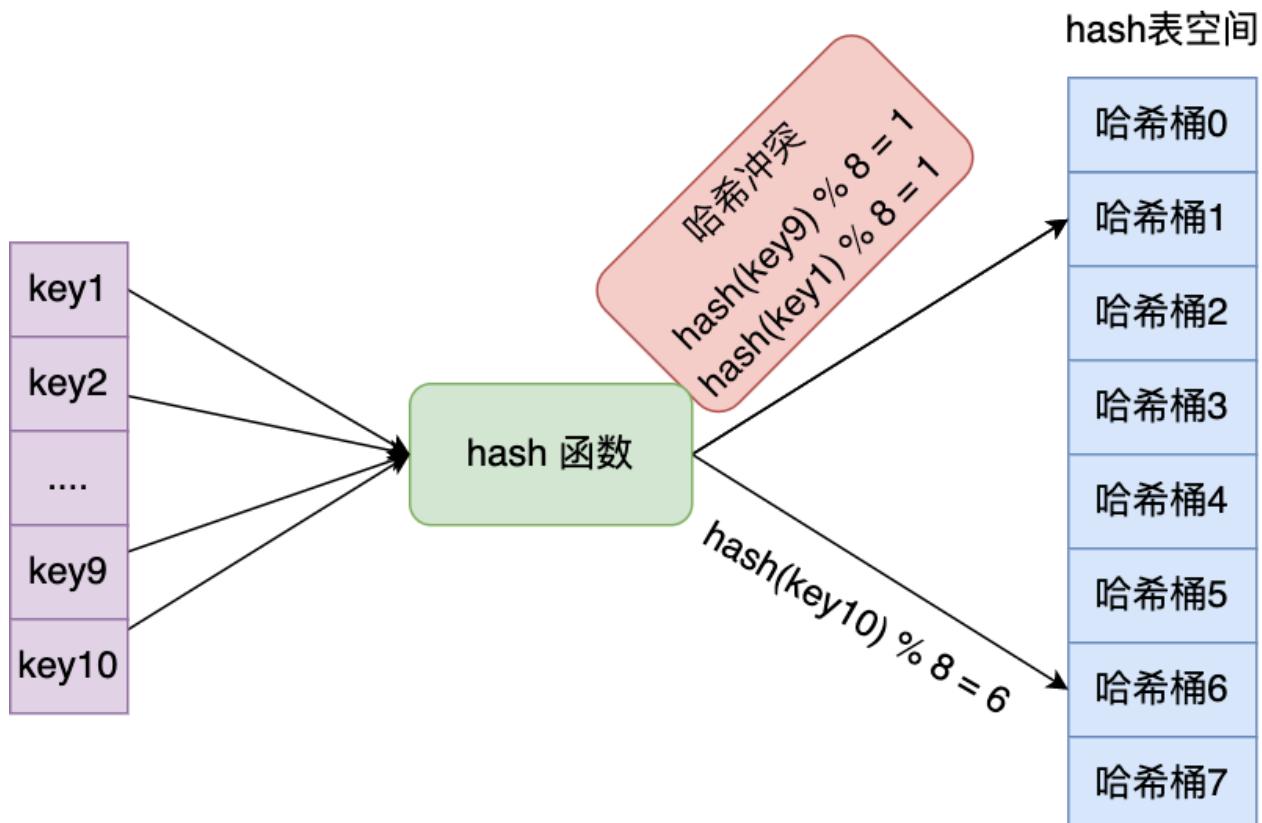
哈希冲突

哈希表实际上是一个数组，数组里每一个元素就是一个哈希桶。

当一个键值对的键经过 Hash 函数计算后得到哈希值，再将(哈希值 % 哈希表大小)取模计算，得到的结果值就是该 key-value 对应的数组元素位置，也就是第几个哈希桶。

什么是哈希冲突呢？

举个例子，有一个可以存放 8 个哈希桶的哈希表。key1 经过哈希函数计算后，再将「哈希值 % 8」进行取模计算，结果值为 1，那么就对应哈希桶 1，类似的，key9 和 key10 分别对应哈希桶 1 和桶 6。



此时，key1 和 key9 对应到了相同的哈希桶中，这就发生了哈希冲突。

因此，当有两个以上数量的 key 被分配到了哈希表中同一个哈希桶上时，此时称这些 key 发生了冲突。

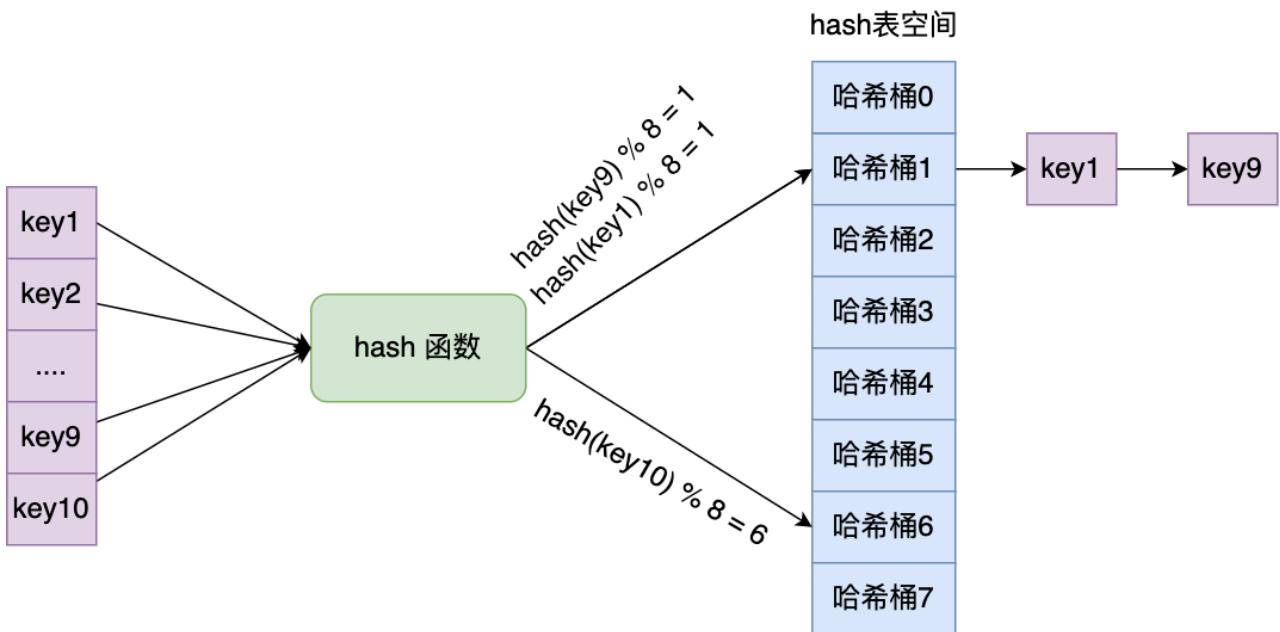
链式哈希

Redis 采用了「链式哈希」的方法来解决哈希冲突。

链式哈希是怎么实现的？

实现的方式就是每个哈希表节点都有一个 next 指针，用于指向下一个哈希表节点，因此多个哈希表节点可以用 next 指针构成一个单项链表，被分配到同一个哈希桶上的多个节点可以用这个单项链表连接起来，这样就解决了哈希冲突。

还是用前面的哈希冲突例子，key1 和 key9 经过哈希计算后，都落在同一个哈希桶，链式哈希的话，key1 就会通过 next 指针指向 key9，形成一个单向链表。



不过，链式哈希局限性也很明显，随着链表长度的增加，在查询这一位置上的数据的耗时就会增加，毕竟链表的查询的时间复杂度是 $O(n)$ 。

要想解决这一问题，就需要进行 rehash，也就是对哈希表的大小进行扩展。

接下来，看看 Redis 是如何实现的 rehash 的。

rehash

哈希表结构设计的这一小节，我给大家介绍了 Redis 使用 dictht 结构体表示哈希表。不过，在实际使用哈希表时，Redis 定义一个 dict 结构体，这个结构体里定义了两个哈希表（ht[2]）。

```
typedef struct dict {
    ...
    //两个Hash表，交替使用，用于rehash操作
    dictht ht[2];
    ...
} dict;
```

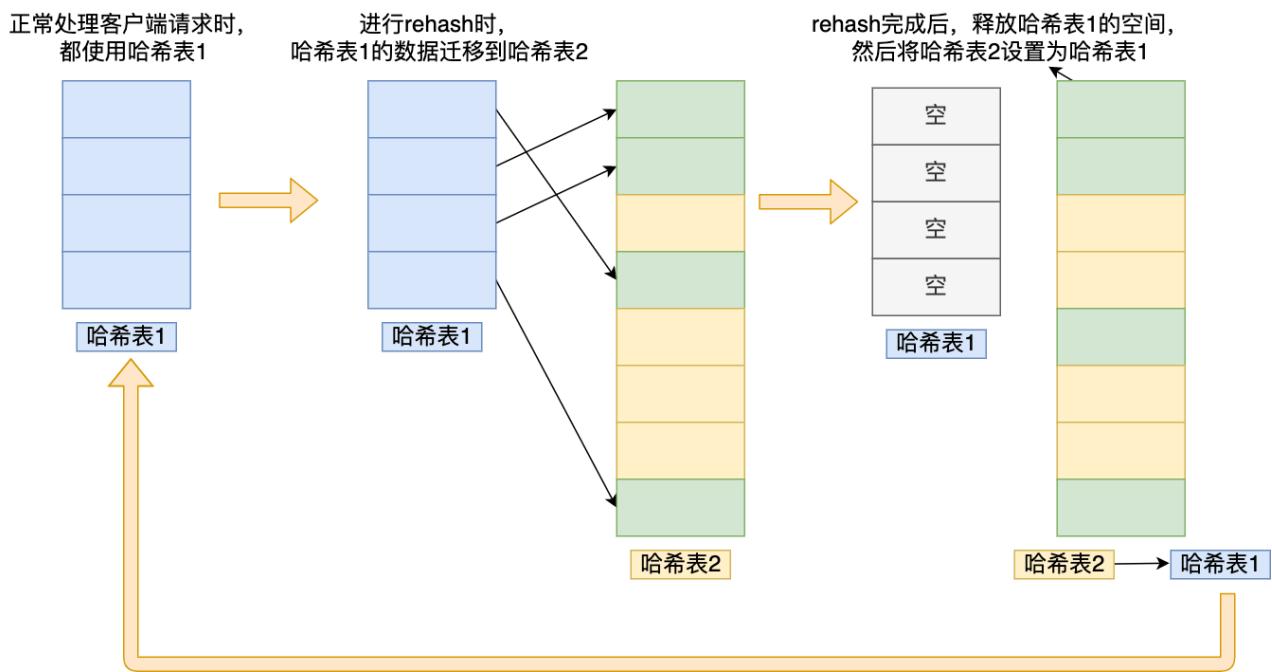
之所以定义了 2 个哈希表，是因为进行 rehash 的时候，需要用上 2 个哈希表了。

在正常服务请求阶段，插入的数据，都会写入到「哈希表 1」，此时的「哈希表 2」并没有被分配空间。

随着数据逐步增多，触发了 rehash 操作，这个过程分为三步：

- 给「哈希表 2」分配空间，一般会比「哈希表 1」大一倍（两倍的意思）；
- 将「哈希表 1」的数据迁移到「哈希表 2」中；
- 迁移完成后，「哈希表 1」的空间会被释放，并把「哈希表 2」设置为「哈希表 1」，然后在「哈希表 2」新创建一个空白的哈希表，为下次 rehash 做准备。

为了方便你理解，我把 rehash 这三个过程画在了下面这张图：



这个过程看起来简单，但是其实第二步很有问题，如果「哈希表 1」的数据量非常大，那么在迁移至「哈希表 2」的时候，因为会涉及大量的数据拷贝，此时可能会对 Redis 造成阻塞，无法服务其他请求。

渐进式 rehash

为了避免 rehash 在数据迁移过程中，因拷贝数据的耗时，影响 Redis 性能的情况，所以 Redis 采用了渐进式 rehash，也就是将数据的迁移的工作不再是一次性迁移完成，而是分多次迁移。

渐进式 rehash 步骤如下：

- 给「哈希表 2」分配空间；
- 在 rehash 进行期间，每次哈希表元素进行新增、删除、查找或者更新操作时，Redis 除了会执行对应的操作之外，还会顺序将「哈希表 1」中索引位置上的所有 key-value 迁移到「哈希表 2」上；
- 随着处理客户端发起的哈希表操作请求数量越多，最终在某个时间点会把「哈希表 1」的所有 key-value 迁移到「哈希表 2」，从而完成 rehash 操作。

这样就巧妙地把一次性大量数据迁移工作的开销，分摊到了多次处理请求的过程中，避免了一次性 rehash 的耗时操作。

在进行渐进式 rehash 的过程中，会有两个哈希表，所以在渐进式 rehash 进行期间，哈希表元素的删除、查找、更新等操作都会在这两个哈希表进行。

比如，查找一个 key 的值的话，先会在「哈希表 1」里面进行查找，如果没找到，就会继续到哈希表 2 里面进行找到。

另外，在渐进式 rehash 进行期间，新增一个 key-value 时，会被保存到「哈希表 2」里面，而「哈希表 1」则不再进行任何添加操作，这样保证了「哈希表 1」的 key-value 数量只会减少，随着 rehash 操作的完成，最终「哈希表 1」就会变成空表。

rehash 触发条件

介绍了 rehash 那么多，还没说什么时情况下会触发 rehash 操作呢？

rehash 的触发条件跟**负载因子（load factor）**有关系。

负载因子可以通过下面这个公式计算：

$$\text{负载因子} = \text{哈希表已保存节点数量} / \text{哈希表大小}$$

触发 rehash 操作的条件，主要有两个：

- 当负载因子大于等于 1，并且 Redis 没有在执行 bgsave 命令或者 bgrewriteaof 命令，也就是没有执行 RDB 快照或没有进行 AOF 重写的时候，就会进行 rehash 操作。
- 当负载因子大于等于 5 时，此时说明哈希冲突非常严重了，不管有没有在执行 RDB 快照或 AOF 重写，都会强制进行 rehash 操作。

整数集合

整数集合是 Set 对象的底层实现之一。当一个 Set 对象只包含整数值元素，并且元素数量不大时，就会使用整数集这个数据结构作为底层实现。

整数集合结构设计

整数集合本质上是一块连续内存空间，它的结构定义如下：

```
typedef struct intset {  
    //编码方式  
    uint32_t encoding;  
    //集合包含的元素数量  
    uint32_t length;  
    //保存元素的数组  
    int8_t contents[];  
} intset;
```

可以看到，保存元素的容器是一个 contents 数组，虽然 contents 被声明为 int8_t 类型的数组，但是实际上 contents 数组并不保存任何 int8_t 类型的元素，contents 数组的真正类型取决于 intset 结构体里的 encoding 属性的值。比如：

- 如果 encoding 属性值为 INTSET_ENC_INT16，那么 contents 就是一个 int16_t 类型的数组，数组中每一个元素的类型都是 int16_t；
- 如果 encoding 属性值为 INTSET_ENC_INT32，那么 contents 就是一个 int32_t 类型的数组，数组中每一个元素的类型都是 int32_t；
- 如果 encoding 属性值为 INTSET_ENC_INT64，那么 contents 就是一个 int64_t 类型的数组，数组中每一个元素的类型都是 int64_t；

不同类型的 contents 数组，意味着数组的大小也会不同。

整数集合的升级操作

整数集合会有一个升级规则，就是当我们将一个新元素加入到整数集合里面，如果新元素的类型 (int32_t) 比整数集合现有所有元素的类型 (int16_t) 都要长时，整数集合需要先进行升级，也就是按新元素的类型 (int32_t) 扩展 contents 数组的空间大小，然后才能将新元素加入到整数集合里，当然升级的过程中，也要维持整数集合的有序性。

整数集合升级的过程不会重新分配一个新类型的数组，而是在原本的数组上扩展空间，然后在将每个元素按间隔类型大小分割，如果 encoding 属性值为 INTSET_ENC_INT16，则每个元素的间隔就是 16 位。

举个例子，假设有一个整数集合里有 3 个类型为 int16_t 的元素。

16 位长度			
位:	0~15位	16~31位	32~47位
元素:	1	2	3

contents 数组

现在，往这个整数集合中加入一个新元素 65535，这个新元素需要用 int32_t 类型来保存，所以整数集合要进行升级操作，首先需要为 contents 数组扩容，在原本空间的大小之上再扩容多 80 位 ($4 \times 32 - 3 \times 16 = 80$)，这样就能保存下 4 个类型为 int32_t 的元素。

16 位长度				
位:	0~15位	16~31位	32~47位	48~127位
元素:	1	2	3	(新扩容的空间)

contents 数组

扩容完 contents 数组空间大小后，需要将之前的三个元素转换为 int32_t 类型，并将转换后的元素放置到正确的位上面，并且需要维持底层数组的有序性不变，整个转换过程如下：

1. 将元素 3 从 <code>int16_t</code> 类型转换成 <code>int32_t</code> 类型				32 位长度		32 位长度					
位:	0~15位	16~31位	32~47位	48~63位	64~95位	96~127位					
元素:	1	2	3	(新分配的空间)	3	(新分配的空间)					
2. 将元素 2 从 <code>int16_t</code> 类型转换成 <code>int32_t</code> 类型											
位:	0~15位	16~31位	32~63位		64~95位	96~127位					
元素:	1	2	2		3	(新分配的空间)					
3. 将元素 1 从 <code>int16_t</code> 类型转换成 <code>int32_t</code> 类型											
位:	0~31位		32~63位		64~95位	96~127位					
元素:	1		2		3	(新分配的空间)					
4. 添加新元素 65535											
位:	0~31位		32~63位		64~95位	96~127位					
元素:	1		2		3	65535					

| 整数集合升级有什么好处呢？

如果要让一个数组同时保存 `int16_t`、`int32_t`、`int64_t` 类型的元素，最简单做法就是直接使用 `int64_t` 类型的数组。不过这样的话，当如果元素都是 `int16_t` 类型的，就会造成内存浪费的情况。

整数集合升级就能避免这种情况，如果一直向整数集合添加 `int16_t` 类型的元素，那么整数集合的底层实现就一直是用 `int16_t` 类型的数组，只有在我们要将 `int32_t` 类型或 `int64_t` 类型的元素添加到集合时，才会对数组进行升级操作。

因此，整数集合升级的好处是节省内存资源。

| 整数集合支持降级操作吗？

不支持降级操作，一旦对数组进行了升级，就会一直保持升级后的状态。比如前面的升级操作的例子，如果删除了 65535 元素，整数集合的数组还是 `int32_t` 类型的，并不会因此降级为 `int16_t` 类型。

跳表

Redis 只有 Zset 对象的底层实现用到了跳表，跳表的优势是能支持平均 $O(\log N)$ 复杂度的节点查找。

Zset 结构体里有两个数据结构：一个是跳表，一个是哈希表。这样的好处是既能进行高效的范围查询，也能进行高效单点查询。

```
typedef struct zset {
    dict *dict;
    zskiplist *zsl;
} zset;
```

Zset 对象在执行数据插入或是数据更新的过程中，会依次在跳表和哈希表中插入或更新相应数据，从而保证了跳表和哈希表中记录的信息一致。

Zset 对象能支持范围查询（如 ZRANGEBYSCORE 操作），这是因为它的数据结构设计采用了跳表，而又能以常数复杂度获取元素权重（如 ZSCORE 操作），这是因为它同时采用了哈希表进行索引。

可能很多人会奇怪，为什么我开头说 Zset 对象的底层数据结构是「压缩列表」或者「跳表」，而没有说哈希表呢？

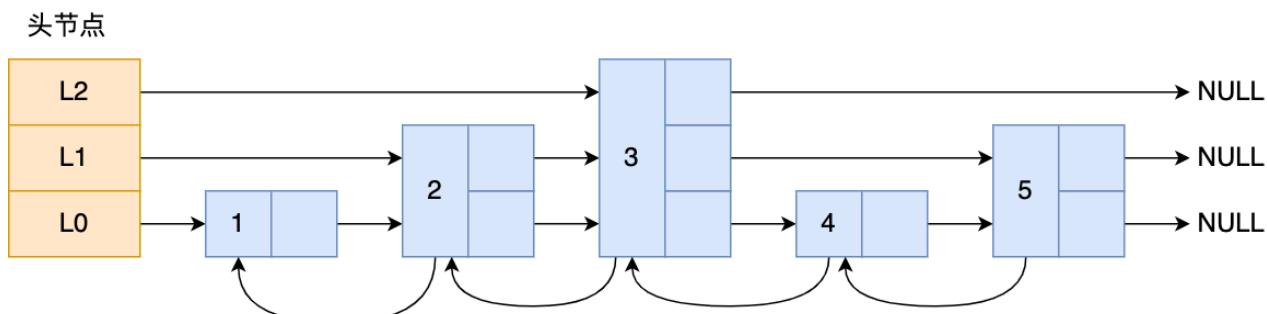
Zset 对象在使用跳表作为数据结构的时候，是使用由「哈希表+跳表」组成的 struct zset，但是我们讨论的时候，都会说跳表是 Zset 对象的底层数据结构，而不会提及哈希表，是因为 struct zset 中的哈希表只是用于以常数复杂度获取元素权重，大部分操作都是跳表实现的。

接下来，详细的说下跳表。

跳表结构设计

链表在查找元素的时候，因为需要逐一查找，所以查询效率非常低，时间复杂度是 $O(N)$ ，于是就出现了跳表。跳表是在链表基础上改进过来的，实现了一种「多层」的有序链表，这样的好处是能快读定位数据。

那跳表长什么样呢？我这里举个例子，下图展示了一个层级为 3 的跳表。



图中头节点有 L0~L2 三个头指针，分别指向了不同层级的节点，然后每个层级的节点都通过指针连接起来：

- L0 层级共有 5 个节点，分别是节点 1、2、3、4、5；
- L1 层级共有 3 个节点，分别是节点 2、3、5；
- L2 层级只有 1 个节点，也就是节点 3。

如果我们要在链表中查找节点 4 这个元素，只能从头开始遍历链表，需要查找 4 次，而使用了跳表后，只需要查找 2 次就能定位到节点 4，因为可以在头节点直接从 L2 层级跳到节点 3，然后再往前遍历找到节点 4。

可以看到，这个查找过程就是在多个层级上跳来跳去，最后定位到元素。当数据量很大时，跳表的查找复杂度就是 $O(\log N)$ 。

那跳表节点是怎么实现多层级的呢？这就需要看「跳表节点」的数据结构了，如下：

```
typedef struct zskiplistNode {
    //Zset 对象的元素值
    sds ele;
    //元素权重值
    double score;
    //后向指针
    struct zskiplistNode *backward;

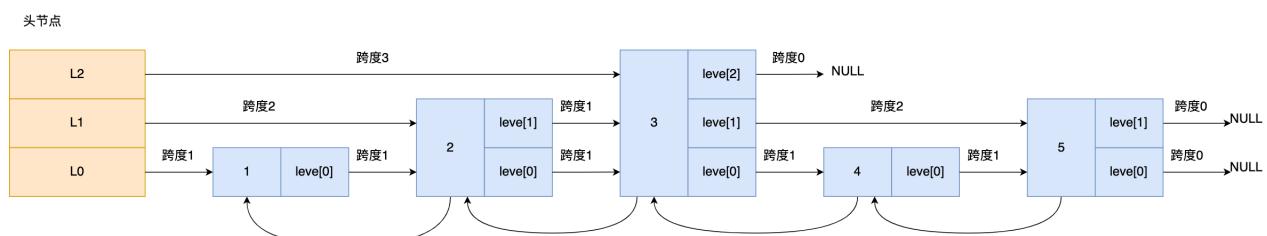
    //节点的level数组，保存每层上的前向指针和跨度
    struct zskiplistLevel {
        struct zskiplistNode *forward;
        unsigned long span;
    } level[];
} zskiplistNode;
```

Zset 对象要同时保存「元素」和「元素的权重」，对应到跳表节点结构里就是 sds 类型的 ele 变量和 double 类型的 score 变量。每个跳表节点都有一个后向指针（struct zskiplistNode *backward），指向下一个节点，目的是为了方便从跳表的尾节点开始访问节点，这样倒序查找时很方便。

跳表是一个带有层级关系的链表，而且每一层级可以包含多个节点，每一个节点通过指针连接起来，实现这一特性就是靠跳表节点结构体中的 **zskiplistLevel** 结构体类型的 level 数组。

level 数组中的每一个元素代表跳表的一层，也就是由 zskiplistLevel 结构体表示，比如 leve[0] 就表示第一层，leve[1] 就表示第二层。zskiplistLevel 结构体里定义了「指向下一个跳表节点的指针」和「跨度」，跨度时用来记录两个节点之间的距离。

比如，下面这张图，展示了各个节点的跨度。



第一眼看到跨度的时候，以为是遍历操作有关，实际上并没有任何关系，遍历操作只需要用前向指针（`struct zskiplistNode *forward`）就可以完成了。

跨度实际上是为了计算这个节点在跳表中的排位。具体怎么做的呢？因为跳表中的节点都是按序排列的，那么计算某个节点排位的时候，从头节点点到该结点的查询路径上，将沿途访问过的所有层的跨度累加起来，得到的结果就是目标节点在跳表中的排位。

举个例子，查找图中节点 3 在跳表中的排位，从头节点开始查找节点 3，查找的过程只经过了一个层（L2），并且层的跨度是 3，所以节点 3 在跳表中的排位是 3。

另外，图中的头节点其实也是 `zskiplistNode` 跳表节点，只不过头节点的后向指针、权重、元素值都没有用到，所以图中省略了这部分。

问题来了，由谁定义哪个跳表节点是头节点呢？这就介绍「跳表」结构体了，如下所示：

```
typedef struct zskiplist {
    struct zskiplistNode *header, *tail;
    unsigned long length;
    int level;
} zskiplist;
```

跳表结构里包含了：

- 跳表的头尾节点，便于在O(1)时间复杂度内访问跳表的头节点和尾节点；
- 跳表的长度，便于在O(1)时间复杂度获取跳表节点的数量；
- 跳表的最大层数，便于在O(1)时间复杂度获取跳表中层高最大的那个节点的层数量；

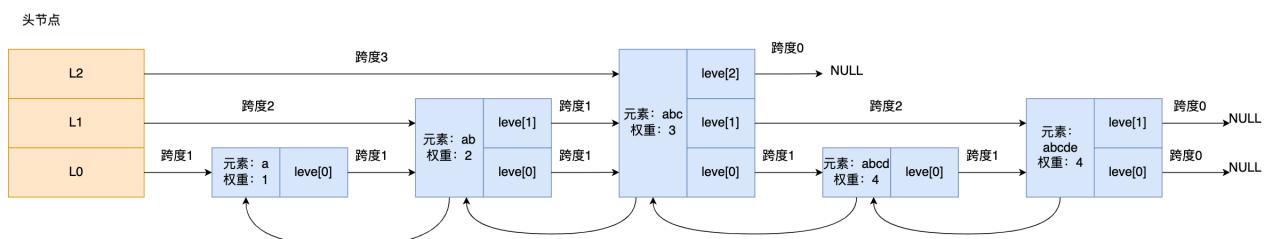
跳表节点查询过程

查找一个跳表节点的过程时，跳表会从头节点的最高层开始，逐一遍历每一层。在遍历某一层的跳表节点时，会用跳表节点中的 SDS 类型的元素和元素的权重来进行判断，共有两个判断条件：

- 如果当前节点的权重「小于」要查找的权重时，跳表就会访问该层上的下一个节点。
- 如果当前节点的权重「等于」要查找的权重时，并且当前节点的 SDS 类型数据「小于」要查找的数据时，跳表就会访问该层上的下一个节点。

如果上面两个条件都不满足，或者下一个节点为空时，跳表就会使用目前遍历到的节点的 `level` 数组里的下一层指针，然后沿着下一层指针继续查找，这就相当于跳到了下一层接着查找。

举个例子，下图有个 3 层级的跳表。



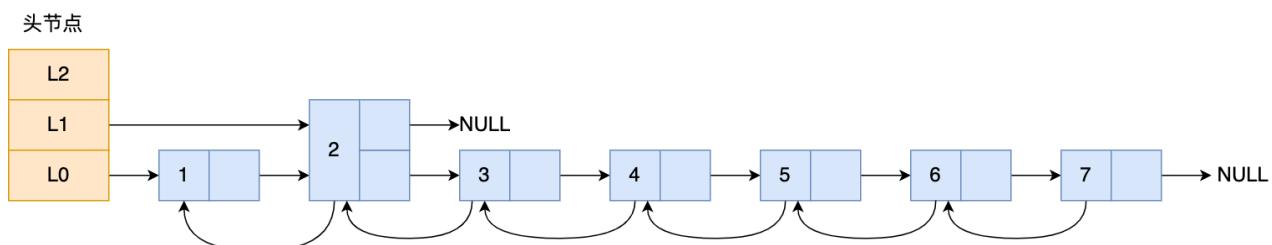
如果要查找「元素：abcd，权重：4」的节点，查找的过程是这样的：

- 先从头节点的最高层开始，L2 指向了「元素：abc，权重：3」节点，这个节点的权重比要查找节点的小，所以要访问该层上的下一个节点；
- 但是该层的下一个节点是空节点（leve[2]指向的是空节点），于是就会跳到「元素：abc，权重：3」节点的下一层去找，也就是 leve[1]；
- 「元素：abc，权重：3」节点的 leve[1] 的下一个指针指向了「元素：abcde，权重：4」的节点，然后将其和要查找的节点比较。虽然「元素：abcde，权重：4」的节点的权重和要查找的权重相同，但是当前节点的 SDS 类型数据「大于」要查找的数据，所以会继续跳到「元素：abc，权重：3」节点的下一层去找，也就是 leve[0]；
- 「元素：abc，权重：3」节点的 leve[0] 的下一个指针指向了「元素：abcd，权重：4」的节点，该节点正是要查找的节点，查询结束。

跳表节点层数设置

跳表的相邻两层的节点数量的比例会影响跳表的查询性能。

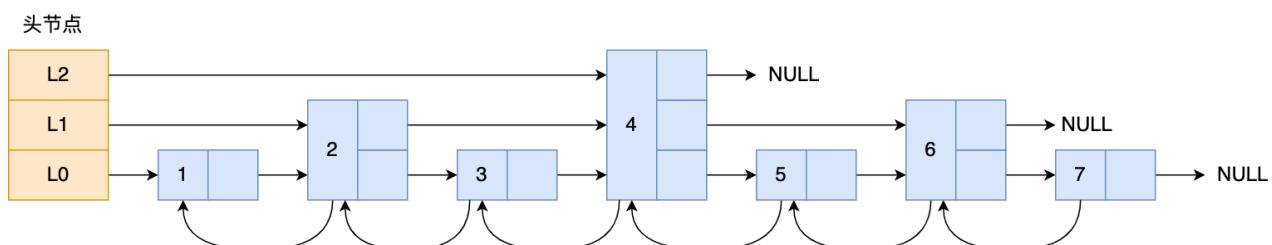
举个例子，下图的跳表，第二层的节点数量只有 1 个，而第一层的节点数量有 6 个。



这时，如果想要查询节点 6，那基本就跟链表的查询复杂度一样，就需要在第一层的节点中依次顺序查找，复杂度就是 $O(N)$ 了。所以，为了降低查询复杂度，我们就需要维持相邻层结点数间的关系。

跳表的相邻两层的节点数量最理想的比例是 2:1，查找复杂度可以降低到 $O(\log N)$ 。

下图的跳表就是，相邻两层的节点数量的比例是 2 : 1。



| 那怎样才能维持相邻两层的节点数量的比例为 2 : 1 呢？

如果采用新增节点或者删除节点时，来调整跳表节点以维持比例的方法的话，会带来额外的开销。

Redis 则采用一种巧妙的方法是，跳表在创建节点的时候，随机生成每个节点的层数，并没有严格维持相邻两层的节点数量比例为 2 : 1 的情况。

具体的做法是，跳表在创建节点时候，会生成范围为[0-1]的一个随机数，如果这个随机数小于 0.25（相当于概率 25%），那么层数就增加 1 层，然后继续生成下一个随机数，直到随机数的结果大于 0.25 结束，最终确定该节点的层数。

这样的做法，相当于每增加一层的概率不超过 25%，层数越高，概率越低，层高最大限制是 64。

虽然我前面讲解跳表的时候，图中的跳表的「头节点」都是 3 层高，但是其实如果层高最大限制是 64，那么在创建跳表「头节点」的时候，就会直接创建 64 层高的头节点。

如下代码，创建跳表时，头节点的 level 数组有 ZSKIPLIST_MAXLEVEL 个元素（层），节点不存储任何 member 和 score 值，level 数组元素的 forward 都指向 NULL，span 值都为 0。

```
/* Create a new skip list. */
zskiplist *zslCreate(void) {
    int j;
    zskiplist *zsl;

    zsl = zmalloc(sizeof(*zsl));
    zsl->level = 1;
    zsl->length = 0;
    zsl->header = zslCreateNode(ZSKIPLIST_MAXLEVEL, 0, NULL);
    for (j = 0; j < ZSKIPLIST_MAXLEVEL; j++) {
        zsl->header->level[j].forward = NULL;
        zsl->header->level[j].span = 0;
    }
    zsl->header->backward = NULL;
    zsl->tail = NULL;
    return zsl;
}
```

其中，ZSKIPLIST_MAXLEVEL 定义的是最高的层数，Redis 7.0 定义为 32，Redis 5.0 定义为 64，Redis 3.0 定义为 32。

为什么用跳表而不用平衡树？

这里插一个常见的面试题：为什么 Zset 的实现用跳表而不用平衡树（如 AVL 树、红黑树等）？

对于[这个问题](#)，Redis 的作者 @antirez 是怎么说的：

There are a few reasons:

1. They are not very memory intensive. It's up to you basically. Changing parameters about the probability of a node to have a given number of levels will make them less memory intensive than btrees.
2. A sorted set is often target of many ZRANGE or ZREVRANGE operations, that is, traversing the skip list as a linked list. With this operation the cache locality of skip lists is at least as good as with other kind of balanced trees.
3. They are simpler to implement, debug, and so forth. For instance thanks to the skip list simplicity I received a patch (already in Redis master) with augmented skip lists implementing ZRANK in O(log(N)). It required little changes to the code.

简单翻译一下，主要是从内存占用、对范围查找的支持、实现难易程度这三方面总结的原因：

- 它们不是非常内存密集型的。基本上由你决定。改变关于节点具有给定级别数的概率的参数将使其比 btree 占用更少的内存。
- Zset 经常需要执行 ZRANGE 或 ZREVRANGE 的命令，即作为链表遍历跳表。通过此操作，跳表的缓存局部性至少与其他类型的平衡树一样好。
- 它们更易于实现、调试等。例如，由于跳表的简单性，我收到了一个补丁（已经在 Redis master 中），其中扩展了跳表，在 O(log(N)) 中实现了 ZRANK。它只需要对代码进行少量修改。

我再详细补充点：

- **从内存占用上来比较，跳表比平衡树更灵活一些。** 平衡树每个节点包含 2 个指针（分别指向左右子树），而跳表每个节点包含的指针数目平均为 $1/(1-p)$ ，具体取决于参数 p 的大小。如果像 Redis 里的实现一样，取 $p=1/4$ ，那么平均每个节点包含 1.33 个指针，比平衡树更有优势。
- **在做范围查找的时候，跳表比平衡树操作要简单。** 在平衡树上，我们找到指定范围的小值之后，还需要以中序遍历的顺序继续寻找其它不超过大值的节点。如果不对平衡树进行一定的改造，这里的中序遍历并不容易实现。而在跳表上进行范围查找就非常简单，只需要在找到小值之后，对第 1 层链表进行若干步的遍历就可以实现。
- **从算法实现难度上来比较，跳表比平衡树要简单得多。** 平衡树的插入和删除操作可能引发子树的调整，逻辑复杂，而跳表的插入和删除只需要修改相邻节点的指针，操作简单又快速。

quicklist

在 Redis 3.0 之前，List 对象的底层数据结构是双向链表或者压缩列表。然后在 Redis 3.2 的时候，List 对象的底层改由 quicklist 数据结构实现。

其实 quicklist 就是「双向链表 + 压缩列表」组合，因为一个 quicklist 就是一个链表，而链表中的每个元素又是一个压缩列表。

在前面讲压缩列表的时候，我也提到了压缩列表的不足，虽然压缩列表是通过紧凑型的内存布局节省了内存开销，但是因为它的结构设计，如果保存的元素数量增加，或者元素变大了，压缩列表会有「连锁更新」的风险，一旦发生，会造成性能下降。

quicklist 解决办法，通过控制每个链表节点中的压缩列表的大小或者元素个数，来规避连锁更新的问题。因为压缩列表元素越少或越小，连锁更新带来的影响就越小，从而提供了更好的访问性能。

quicklist 结构设计

quicklist 的结构体跟链表的结构体类似，都包含了表头和表尾，区别在于 quicklist 的节点是 quicklistNode。

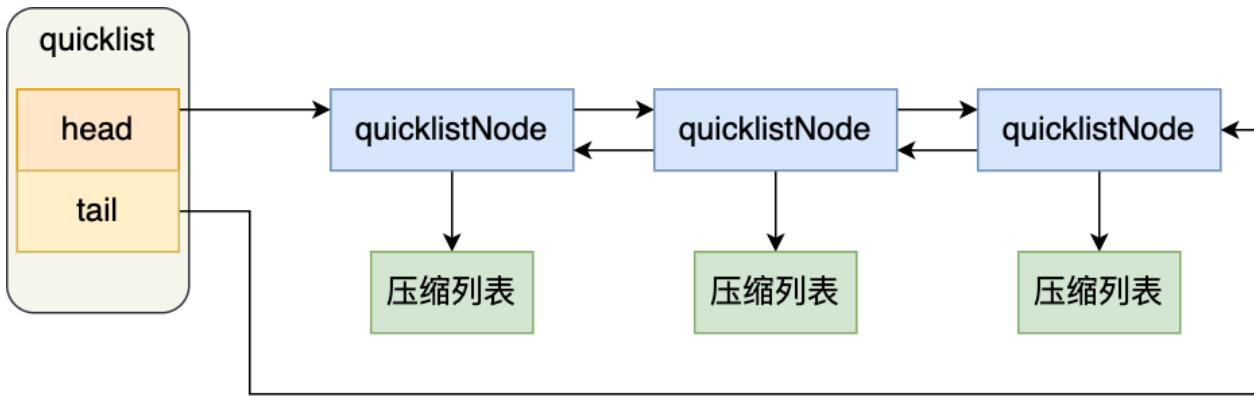
```
typedef struct quicklist {
    //quicklist的链表头
    quicklistNode *head;          //quicklist的链表头
    //quicklist的链表尾
    quicklistNode *tail;
    //所有压缩列表中的总元素个数
    unsigned long count;
    //quicklistNodes的个数
    unsigned long len;
    ...
} quicklist;
```

接下来看看，quicklistNode 的结构定义：

```
typedef struct quicklistNode {
    //前一个quicklistNode
    struct quicklistNode *prev;    //前一个quicklistNode
    //下一个quicklistNode
    struct quicklistNode *next;    //后一个quicklistNode
    //quicklistNode指向的压缩列表
    unsigned char *z1;
    //压缩列表的字节大小
    unsigned int sz;
    //压缩列表的元素个数
    unsigned int count : 16;       //ziplist中的元素个数
    ...
} quicklistNode;
```

可以看到，quicklistNode 结构体里包含了前一个节点和下一个节点指针，这样每个 quicklistNode 形成了一个双向链表。但是链表节点的元素不再是单纯保存元素值，而是保存了一个压缩列表，所以 quicklistNode 结构体里有个指向压缩列表的指针 *z1。

我画了一张图，方便你理解 quicklist 数据结构。



在向 quicklist 添加一个元素的时候，不会像普通的链表那样，直接新建一个链表节点。而是会检查插入位置的压缩列表是否能容纳该元素，如果能容纳就直接保存到 quicklistNode 结构里的压缩列表，如果不能容纳，才会新建一个新的 quicklistNode 结构。

quicklist 会控制 quicklistNode 结构里的压缩列表的大小或者元素个数，来规避潜在的连锁更新的风险，但是这并没有完全解决连锁更新的问题。

listpack

quicklist 虽然通过控制 quicklistNode 结构里的压缩列表的大小或者元素个数，来减少连锁更新带来的性能影响，但是并没有完全解决连锁更新的问题。

因为 quicklistNode 还是用了压缩列表来保存元素，压缩列表连锁更新的问题，来源于它的结构设计，所以要想彻底解决这个问题，需要设计一个新的数据结构。

于是，Redis 在 5.0 新设计一个数据结构叫 listpack，目的是替代压缩列表，它最大特点是 listpack 中每个节点不再包含前一个节点的长度了，压缩列表每个节点正因为需要保存前一个节点的长度字段，就会有连锁更新的隐患。

我看了 Redis 的 Github，在最新 6.2 发行版本中，Redis Hash 对象、ZSet 对象的底层数据结构的压缩列表还未被替换成 listpack，而 Redis 的最新代码（还未发布版本）已经将所有用到压缩列表底层数据结构的 Redis 对象替换成 listpack 数据结构来实现，估计不久将来，Redis 就会发布一个将压缩列表为 listpack 的发行版本。

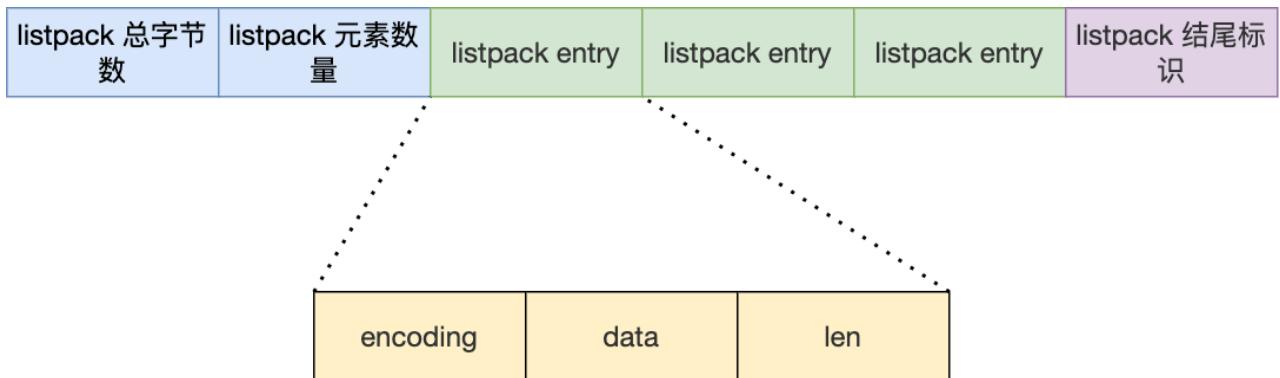
listpack 结构设计

listpack 采用了压缩列表的很多优秀的设计，比如还是用一块连续的内存空间来紧凑地保存数据，并且为了节省内存的开销，listpack 节点会采用不同的编码方式保存不同大小的数据。

我们先看看 listpack 结构：

listpack 头包含两个属性，分别记录了 listpack 总字节数和元素数量，然后 listpack 末尾也有个结尾标识。图中的 listpack entry 就是 listpack 的节点了。

每个 listpack 节点结构如下：



主要包含三个方面内容：

- encoding，定义该元素的编码类型，会对不同长度的整数和字符串进行编码；
 - data，实际存放的数据；
 - len，encoding+data的总长度；

可以看到，listpack 没有压缩列表中记录前一个节点长度的字段了，listpack 只记录当前节点的长度，当我们向 listpack 加入一个新元素的时候，不会影响其他节点的长度字段的变化，从而避免了压缩列表的连锁更新问题。

参考资料：

- 《Redis设计与实现》
 - 《Redis 源码剖析与实战》

总结

终于完工了，松一口气。

好久没写那么长的图解技术文啦，这次潇潇洒洒写了 1.5 万字 + 画了 40 多张图，花费了不少时间，又是看书，又是看源码。

希望这篇文章，能帮你破除 Redis 数据结构的迷雾！

读者问答

压缩列表的entry为什么要保存prevlen呢？listpack改成len之后不会影响功能吗？

压缩列表的 entry 保存 prevlen 是为了实现节点从后往前遍历，知道前一个节点的长度，就可以计算前一个节点的偏移量。

`listpack` 一样可以支持从后往前遍历的。详细的算法可以看：

<https://github.com/antirez/listpack/blob/master/listpack.c> 里的 `lpDecodeBacklen` 函数，`lpDecodeBacklen` 函数就可以从当前列表项起始位置的指针开始，向左逐个字节解析，得到前一项的 `entry-len` 值。