Redis 数据结构

SDS

SDS 为 Simple Daynamic String 的缩写，表示简单字符串，Redis 在原有的 C 字符串之上进行了一层封装

```
struct sdshdr {
    // 保存字符串长度
    int len;

    // 剩余空间
    int free;

    // 字节数组
    char buf[];
}
```

优点
1. O(1) 时间内获取字符串的长度
2. 可以有效的防止缓冲区溢出
3. 减少修改字符串时带来的内存重新分配次数

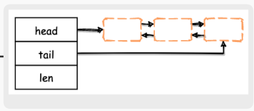
均采用结构体成员 len 和 free 实现

缺点：相较于原有 C 字符串，额外了多出了 8 字节的存储空间

链表

Redis 和 Linux 内核一样，也是采用双向链表实现，并没有使用单链表来节省内存

list 这一结构体主要包含了 3 个成员: 头指针、尾指针，以及双向链表所包含的节点数量



链表结构的应用非常广泛，包括数据量较小时的列表、发布订阅、慢查询和监视器等

字典

采用哈希表实现，并且处理哈希冲突的方式为拉链法

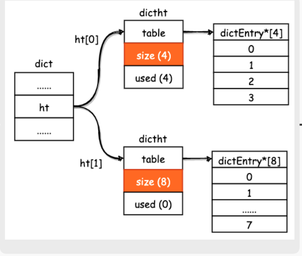
渐进式 rehash

为什么使用渐进式 rehash?

Redis 底层网络模型为单线程+epoll，也就是说，发送至 Redis 的命令是串行执行的

若采用直接扩容的方式，那么当遇到巨大的哈希表时，数据的复制将花费很长时间，势必会阻塞其它命令的执行，从而导致整体性能下降

过程



在渐进式 rehash 过程中，会同时存在 ht[0]、ht[1] 两张哈希表，并且同时对外提供服务

1. 当我们新增一个 key 时，新的 key-value 将会保存在 ht[1] 中，也就是那个容量更大的哈希表中
2. 当我们查找、更新以及删除某一个 key 时，会在两张哈希表上进行，而不仅仅只是对 ht[0] 或者是 ht[1] 中的某一个哈希表进行操作

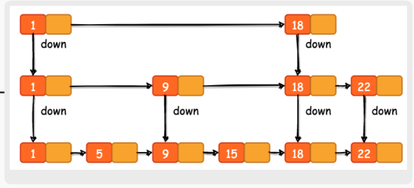
将扩容过程均摊

Redis 本身的数据库也是使用哈希表实现的，所以才能在 O(1) 时间内查找到某一个 key

跳跃表

跳跃表是一种有序的数据结构，可以简单的认为它是一种多层链表，支持平均 O(logn)、最坏 O(n) 复杂度的节点查找

跳跃表是实现有序集合底层数据结构之一，当有序集合中的元素较多时，Redis 就会选择跳跃表来实现该集合



整数集合

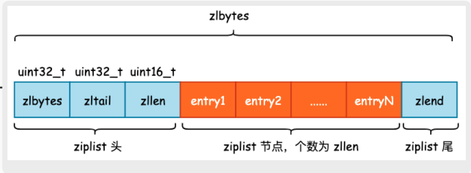
整数集合的底层实现为数组，并且元素在该数组中以有序、无重复的方式排列，通过名字我们就知道它能够实现集合，并且是整数集合

整数集合中我们可以保存 3 种类型的整数，分别为 16 位、32 位和 64 位的整数

这么实现的原因还是那个初衷: 节省内存。因为在一般情况下，程序不知道用户会保存多大的整数，所以为了避免整数溢出，可能会直接使用 64 位来保存。如果用户保存的都是小整数的话，那么就会有很多的内存浪费

Redis 出于这个考虑对 intset 进行动态编码，如果当前集合能够用 int16_t 来保存，那就用 int16_t。如果用户后续新增了一个 32 位整数，那么 Redis 会对 intset 进行升级，改用 int32_t 来保存

压缩列表



压缩列表，ziplist 由上图所示组件构成，其目的就是为了节省内存。当一个 LIST 中只包含少量的元素，并且每一个元素要么是小正数，要么是长度很短的字符串，Redis 就会采用压缩列表来实现

理解压缩列表的一个关键点就在于，entry 中保存的数据大小是不固定的，也就是说，对于一个 ziplist 而言，我们既可以往里面扔 int，也可以往里面儿扔 string

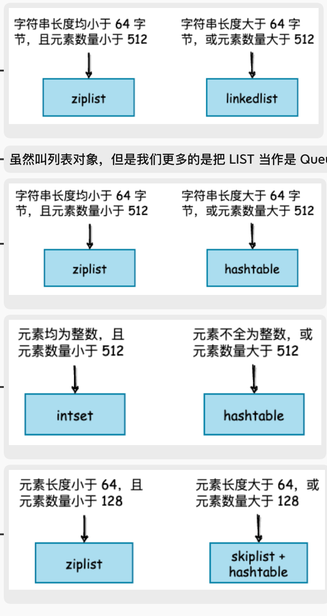
RPUSH ziplist 1 2 3 "Hello" "World"

entry 中保存的是数据，而非指针

因此，如果我们向 ziplist 中添加、删除、查询等操作时，平均时间复杂度均为 O(N)，并且添加和删除这两个操作的最坏时间复杂度为 O(N^2)。不过没关系，反正 ziplist 只会在小数据量时使用，即便是 O(N^2) 也不会有太大的性能问题

Redis 对象与数据结构

与其说 Redis 是一个内存数据库，倒不如说是一个数据结构工厂。Redis 对外暴露的列表、哈希、集合以及有序集合等对象的底层实现会因为数据规模和数据特点的不同而不同



总之，ziplist 除了不能实现集合以外，其它什么都能干