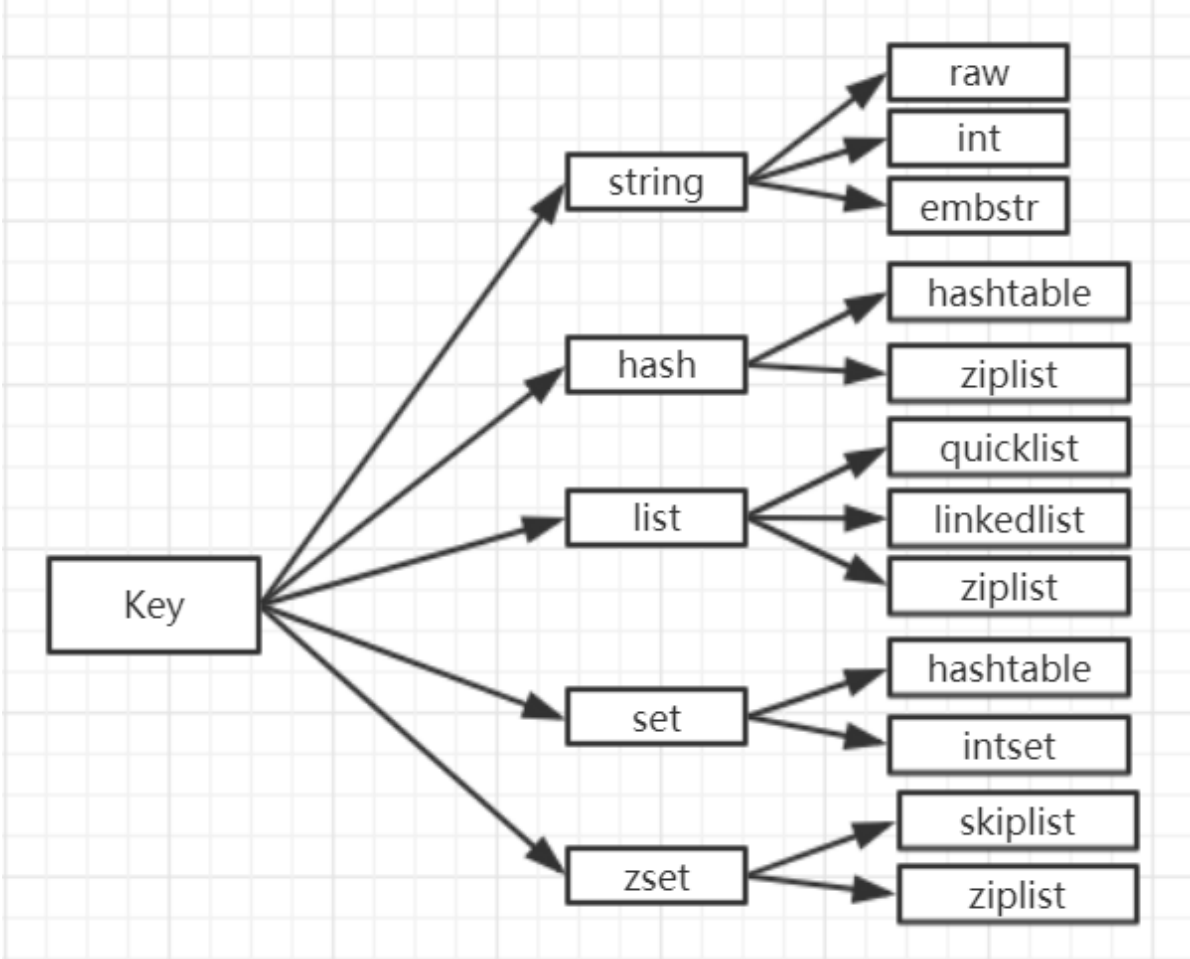
Redis 数据结构



0.内存

> info memory命令可以查看redis内存使用情况
>
> 1）**used_memory****：**Redis分配器分配的内存总量（单位是字节），包括使用的虚拟内存（即swap）
>
> 2）**used_memory_rss****：**Redis进程占据操作系统的内存，除了分配器分配的内存，还包括进程运行本身需要的内存、内存碎片等，但是不包括虚拟内存，虚拟内存用的是磁盘
>
> 3）**mem_fragmentation_ratio**：内存碎片比率，该值是used_memory_rss / used_memory的比值
>
> 该值一般大于1，如果小于1说明使用了虚拟内存，此时就需要注意了
>
> Redis内存包括：基础的数据(即我们存放到redis内存的数据)、进程本身运行所需要的空间、
>
> 缓冲内存(比如AOF缓冲区等)、内存碎片
>
> 寻找大key：
>
> redis-rdb-tools
>
> ```
> rdb -c memory /wjqdata/redis/rdb/dump.rdb --bytes 128 -f
> /wjqdata/redis/rdb/dump_memory.csv
> ```

直接对CSV文件操作或者将CSV文件数据导入数据库

1.SDS(简单动态字符串)

SDS是Redis自己构建的字符串抽象类型，redis的字符串不是使用c的字符串实现的，而是单独构建了SDS来表示字符串值。

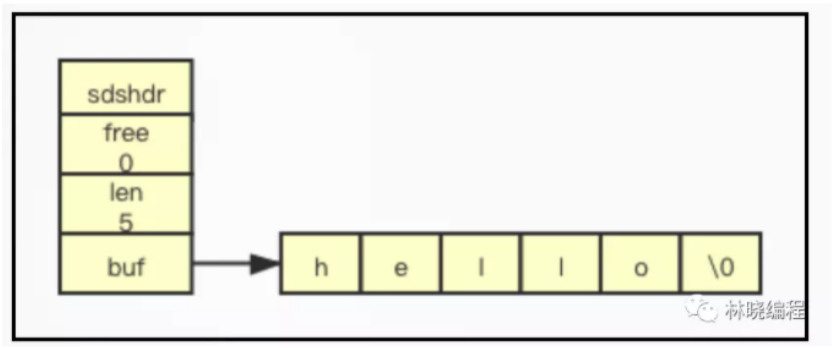
```
redis>SET msg "hello world"
```

以上的redis命令给一个名为msg的键设置value值为hello world，其中代表键(key)的msg是一个字符串对象，这个对面底层既是一个保存着字符串'msg'的SDS；同理，值(value)也是一个字符串对象，底层实现是一个保存着字符串“hello world” 的SDS；

包括列表中存储的一个个字符串对象，集合中存储的一个个字符串对象等涉及到字符串底层都是用SDS实现的。

SDS结构体：

```
struct sdshdr {
//记录 buf 数组中已使用字节的数量,等于 SDS 所保存字符串的长度
int len;
//记录 buf 数组中未使用字节的数量
int free;
//字节数组，用于保存字符串
char buf[];
}
```



真正的字符串是存储在buf数组的，分别存储了各个字符的值。

最后会存储一个\0的空字符，空字符的作用是保持C语言字符串的格式，便于使用C字符底层的函数。 这个空字符不计入长度

buf数组的长度=free+len+1（其中1表示字符串结尾的空字符）

为什么使用SDS?

1）空间复杂度

C字符串复杂度O(N)；SDS复杂度O(1)

2）移除问题(内存扩展检测)

C字符串API是不安全的，容易溢出；SDS的API是安全的，不会缓冲溢出。

3）内存重分配次数(预分配)

C字符串修改N次则重分配N次内存空间；SDS修改N次则最高会重分配N次。

sds使用了空间预分配和惰性空间释放机制，说白了就是每次在扩展时是成倍的多分配的，**在缩容是也是先留着并不正式归还给OS**

4）存储数据格式

C字符串只能存储文本数据；SDS可以保存文本或二进制数据。

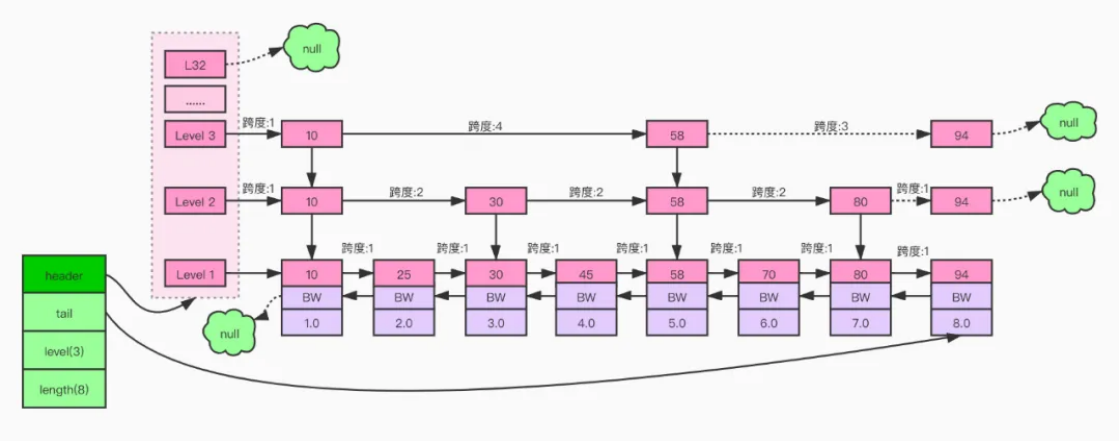
说明：C 字符串中的字符必须符合某种编码，并且除了字符串的末尾之外，字符串里面不能包含空字符，否则最先被程序读入的空字符将被误认为是字符串结尾，这些限制使得C字符串只能保存文本数据，而不能保存像图片，音频，视频，压缩文件这样的二进制数据。

5）函数

C字符串可以使用所有<string.h>函数；SDS可以使用一部分库中的函数。

2.跳跃表(仅仅在有序集合用到)SkipList

每一层都有一个链表作为索引



跳跃表分为两部分：zskiplist与zskilistNode

zskilistNode表示跳跃表节点

zskiplist记录了跳跃表信息：头节点，尾节点，节点数目(除头节点)，节点中层数最高的节点的层数(除头节点)

zskiplist结构：

```
typedef struct zskiplist{
# 表头节点和表尾节点
struct skiplistNode *header, *tail;
# 表中节点的数量
unsigned long length;
# 表中层数最大的节点层数
int level;
}zskiplist;
```

- header 属性，指向跳跃表表头节点
- tail 属性，指向跳跃表的表尾节点
- level 属性，记录目前跳跃表内层数最大的那个节点数（不计表头）
- lenght 属性，记录跳跃表的长度，即节点的数量（不计算表头）

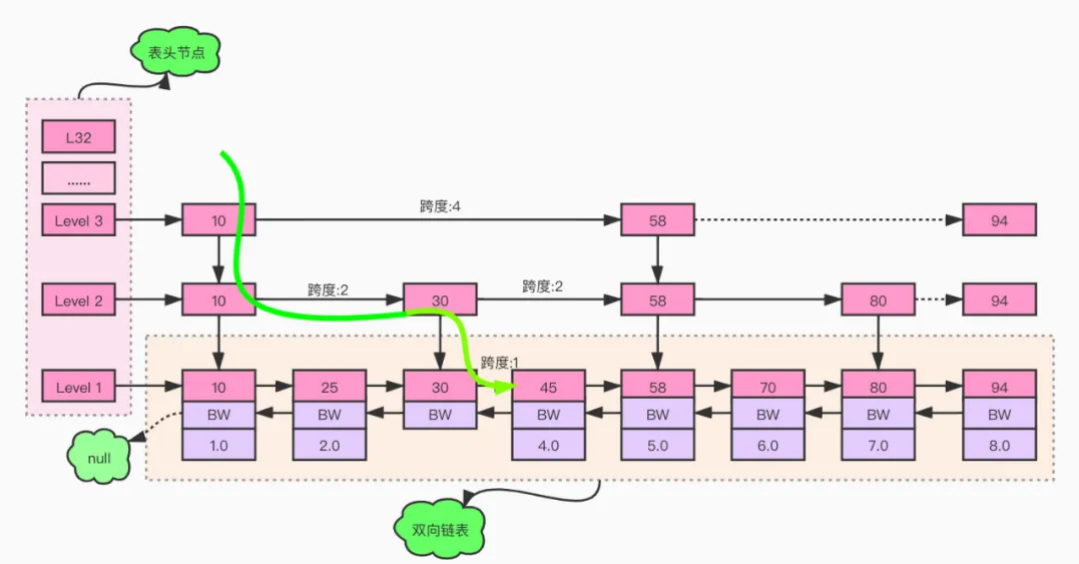
zskilistNode结构：

```
typedef struct zskiplistNode{
   struct zskiplistNode *backward;# 后退指针;
   double score;  # 分值
   object *obj;   # 成员对象
       # Leval层
       struct zskiplistLevel {
           # 前进指针
           struct zsklistNode *forward;
           # 跨度
           unsigned int span;
       }level[]
}zskiplistNode;
```

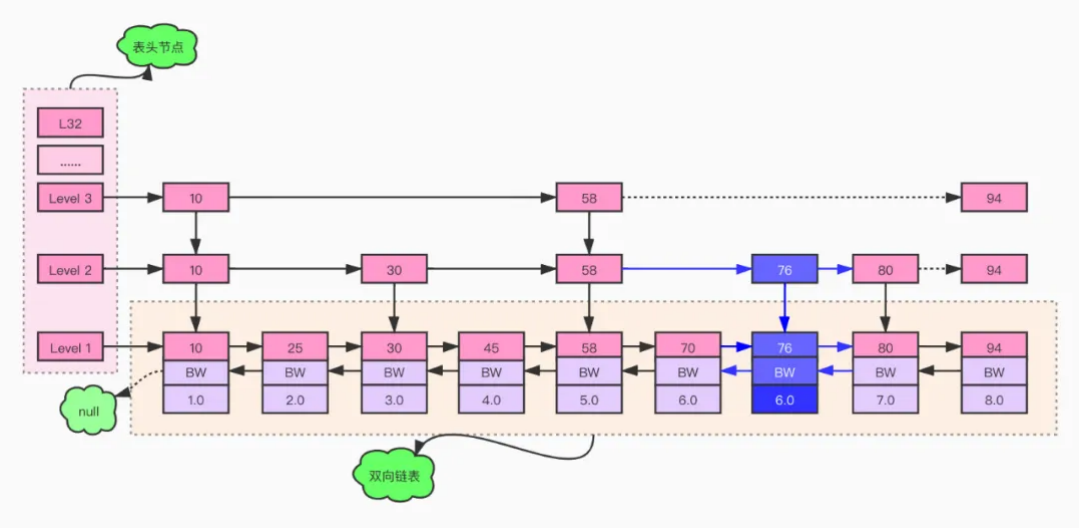
- level 层属性，节点中用level 字样标记的各个层。每个层有包含两个属性：前进指针，和跨度。前进指针用于访问位于表尾方向的其他节点，而跨度则记录了前进指针所指向节点和当前节点的距离。
- backward后退指针， 节点中用BW字样标记节点的后退指针，它指向位于当前节点的前一个节点。用于从表尾向表头遍历时使用。
- score 分值属性，各个节点中的1.0、2.0、3.0 是节点所保存的分值，按分值从小到大排序的。在跳跃表中，多个节点的分值是可以重复的，但是每个节点的对象值必须唯一。
- obj 成员对象属性，各个节点的【10、25、30...】是节点所保存的成员对象值。

注意：表头节点和其他节点的构造是一样的，只是表头节点的一些属性不会用到，则可以忽略不计。Redis中的跳跃表，表头节点的层高共有32层

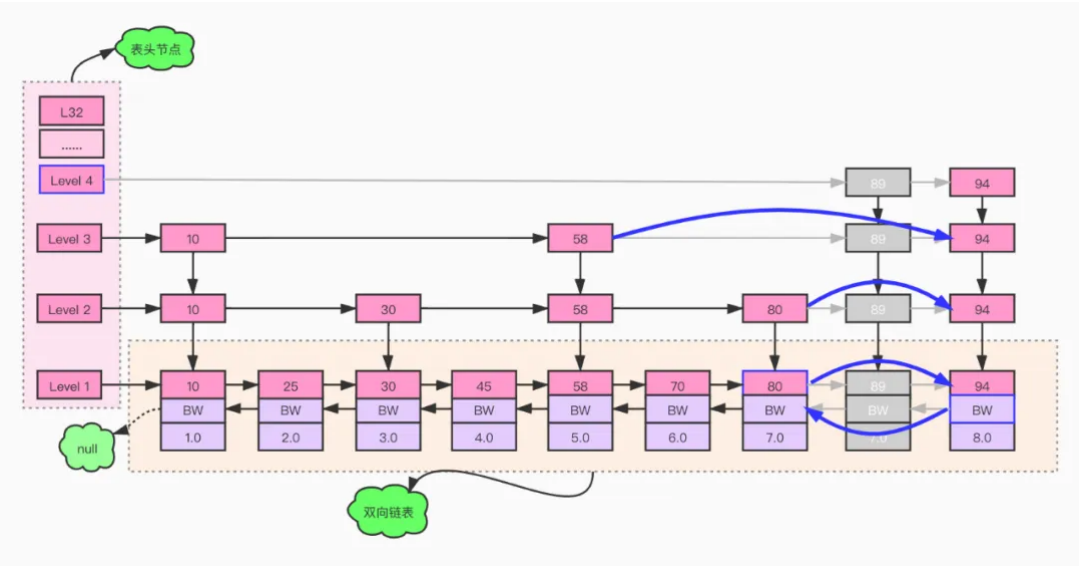
跳跃表的查询



跳跃表的插入



跳跃表的删除



跳跃表的插入与删除都是从高层开始向底层执行

跳跃表查询的时间复杂度为o(logN) 如果要删除M个元素，复杂度就是O(log(N) + M)

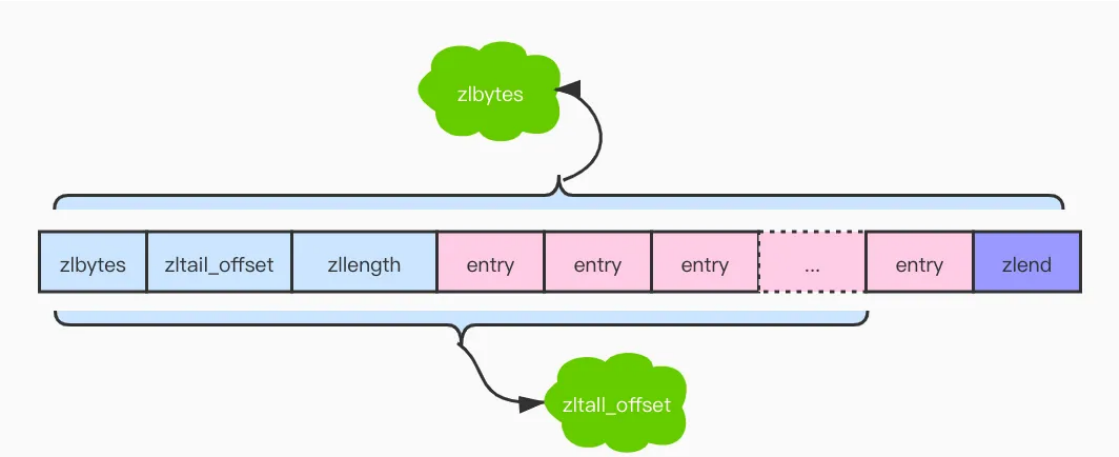
压缩列表ZipList

压缩列表是Redis为了节约内存而开发的，是由一系列特殊编码的**连续内存块组成的顺序型数据结构**。和数组数据结构不同的是，**大小可变、类型可变**。

压缩列表最擅长的是，存储字符串相对较短，元素个数相对较少的数据场景。毕竟，压缩列表底层存储的是一块连续的内存空间，除了所存储的元素以为，没有任何多余的空闲空间。

压缩列表结构：

```
struct ziplist {
# 整个压缩列表占用的字节数
int32 zlbytes;
# 最后一个元素距离压缩列表起始位置的偏移量，用户快速定位最后一个节点
int32 zlbail_offset;
# 元素个数
int16 zllength;
# 元素内容列表、依次紧凑存储
T[] entries;
# 标志压缩列表的结束，值为OxFF
int8 zlend;
}
```



zlbytes 属性记录着整个压缩列表，占用的内存字节数，对压缩列表进行重分配，或计算zlend的位置时使用。

zltail 属性记录压缩列表，表尾节点距离压缩列表的起始地址有多少字节，确定表尾节点地址。

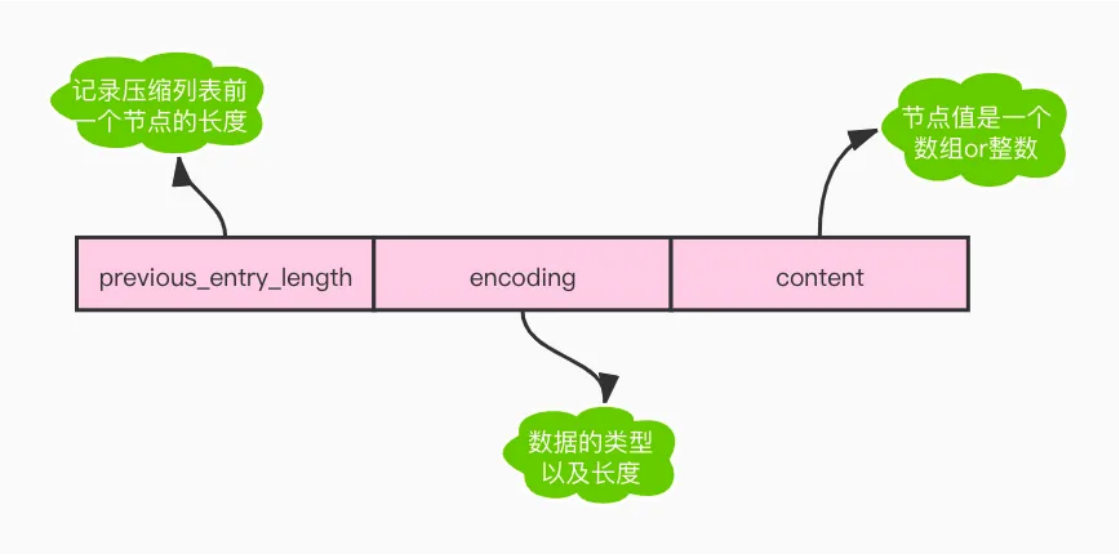
zllen 属性记录压缩列表，包含节点的数量，当这个属性值小于UINT16_MAX(65535)时可信，否则需要遍历整个压缩列表获得最新的长度。

entryX 属性是压缩列表，包含的各个节点，节点的长度由节点保存的内容决定。

zlend 属性特殊值0xff(十进制255) 用于标记压缩列表的末端。

entry结构：

```
struct entry {
# 前一个entry的字节长度
int prevlen;
# 元素类型编码
int encoding;
# 元素内容
optional byte[] content;
}
```



如果前一节点的长度小于254字节那么previous_entry_length属性的长度为1字节 如果前一节点的长度大于等于254字节previous_entry_length属性的长度为5字节

根据当前节点的地址和previous_entry_length的值来计算出前一个节点的地址

整数集合(intset)

整数集合是Redis用于保存整数值的集合抽象数据结构。它支持编码为 int16、int32、int64 的整数值。存储元素的值必须唯一。

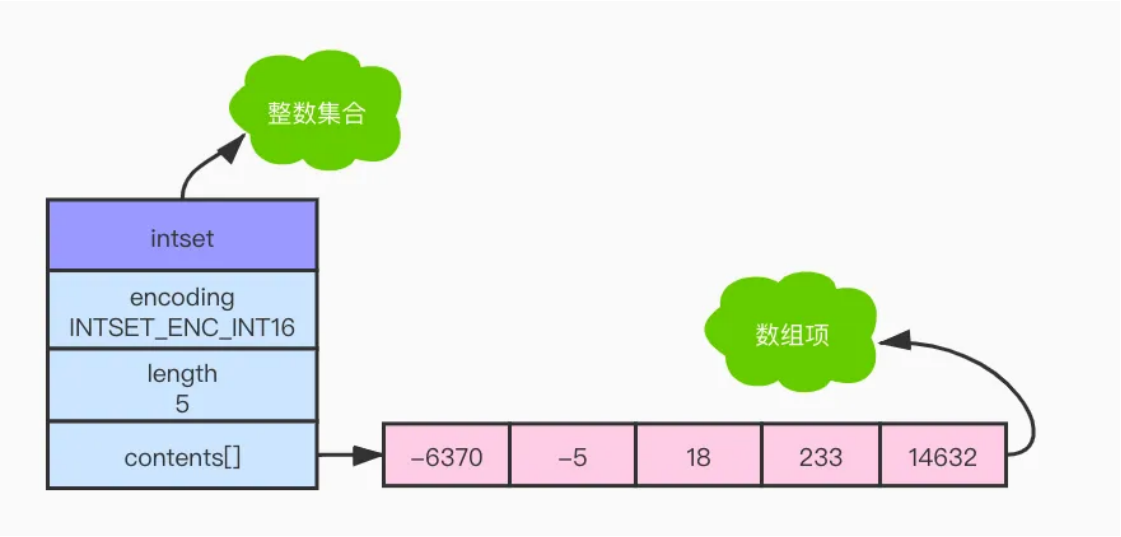
结构:

```
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;
```

contents 数组是整数集合的底层实现，整数集合的每个元素都是contents数组的一个数组项。各个项从小到大的排序，并且每个项元素值唯一。

length 属性记录了整数集合包含元素的数量，即contents的长度。

encoding 属性的值为 INTSET_ENC_INT16、INTSET_ENC_INT32，INTSET_ENC_INT64



对象的属性

redis 的键和值都是一个对象，每个对象都有以下五个属性：类型、编码、指针、引用计数、空转时长。

```
typedef struct redisObject {
unsigned type: 4; #类型
unsigned encoding:4; #编码
int refcount; #引用计数
unsigned lru:22; #空转时长
void *ptr; #指向底层实现数据结构的指针     ...
}
```

type 属性，可为以下五种的其中一种：字符串、列表、哈希、集合、有序集合

refcount 属性，用于记录该对象被引用的次数，当引用计数为0时，对象会释放

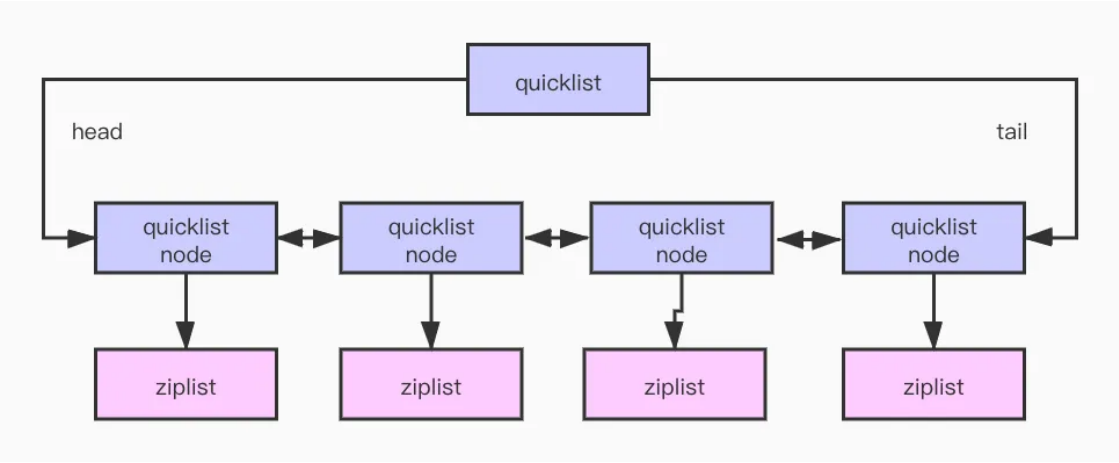
lru 属性，用于记录对象最后一次访问的时间，若访问的时间过久，对象会释放

ptr 属性，用于指向对象的底层实现的数据结构，而数据结构是由encoding决定的

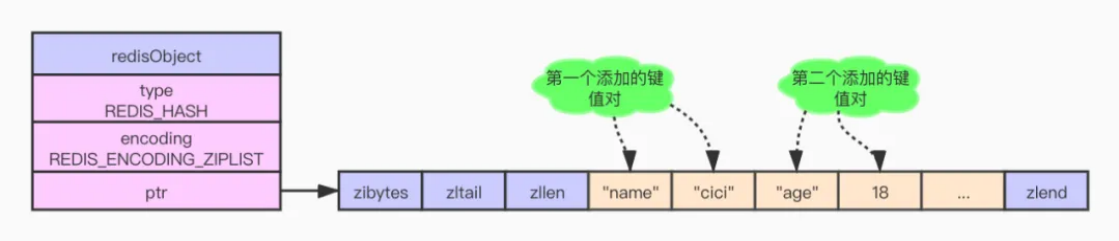
encoding 属性，记录了对象所使用的编码，也就是说，对象底层使用了哪种数据结构作为对象的底层实现。属性值可以是以下表格中的一个。

REDIS_ENCODING_INT	long类型的整数
REDIS_ENCODING_EMBSTR	embstr编码的简单动态字符串
REDIS_ENCODING_RAW	简单动态字符串
REDIS_ENCODING_HT	字典
REDIS_ENCODING_LINKEDLIST	双端链表
REDIS_ENCODING_ZIPLIST	压缩列表
REDIS_ENCODING_INTSET	整数集合
REDIS_ENCODING_SKIPLIST	跳跃表和字典

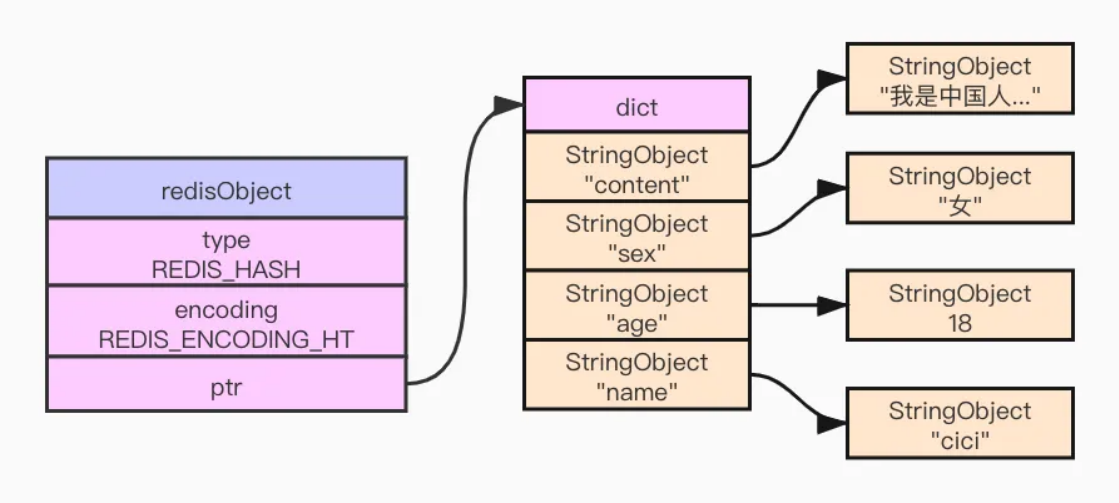
quicklist 是 ziplist 和 linkedlist 的混合体，它将 linkedlist 按段切分，每一段使用 ziplist 存储，多个 ziplist 之间使用双向指针串接起来。如下图所示。



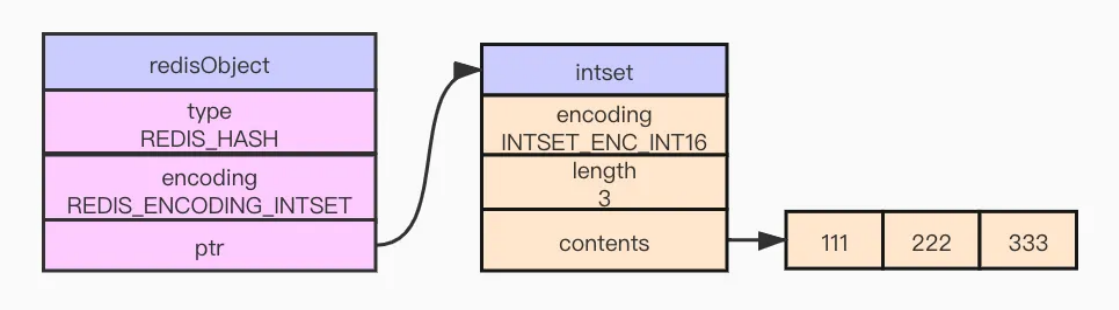
压缩列表作为Hash的编码方式



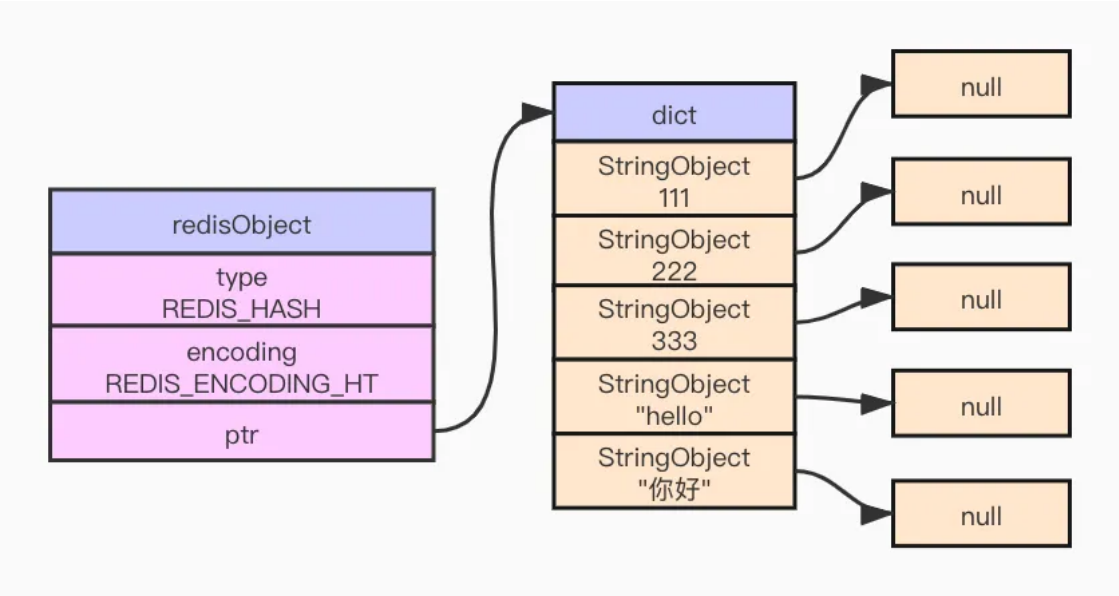
哈希作为Hash的编码方式



intset作为set的编码方式

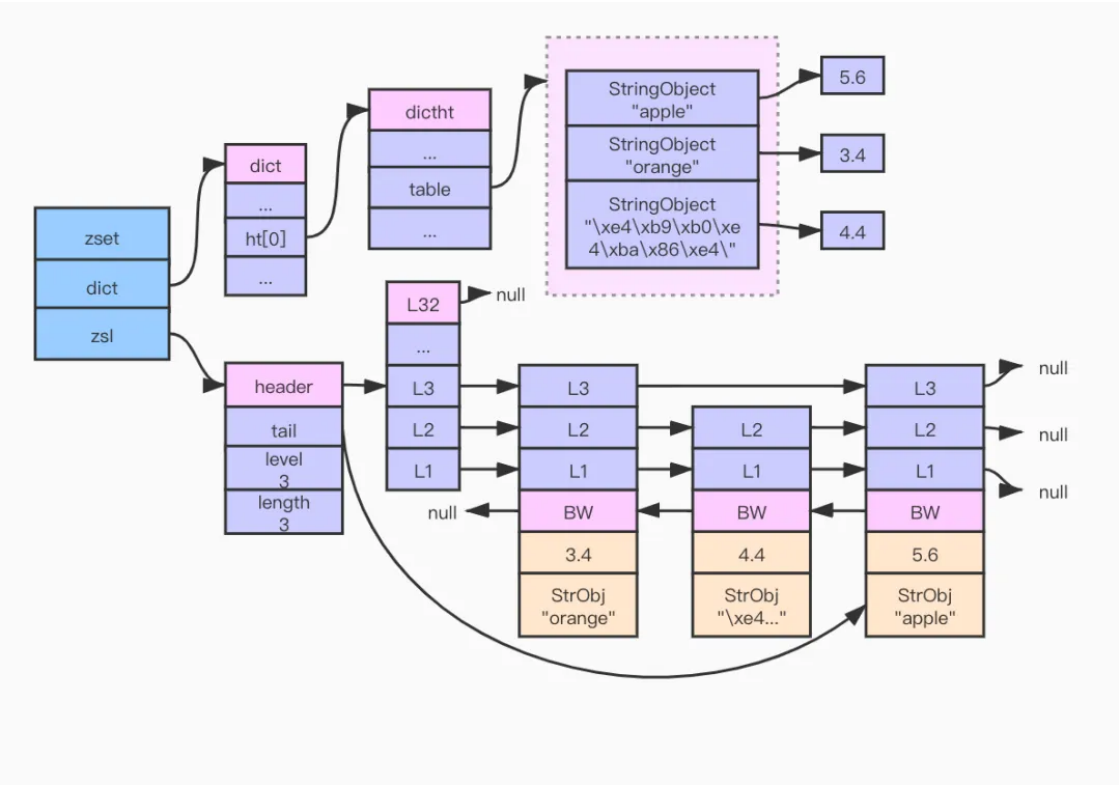


哈希作为set编码方式



压缩列表作为sortset的编码方式

skipList作为sortset的编码方式

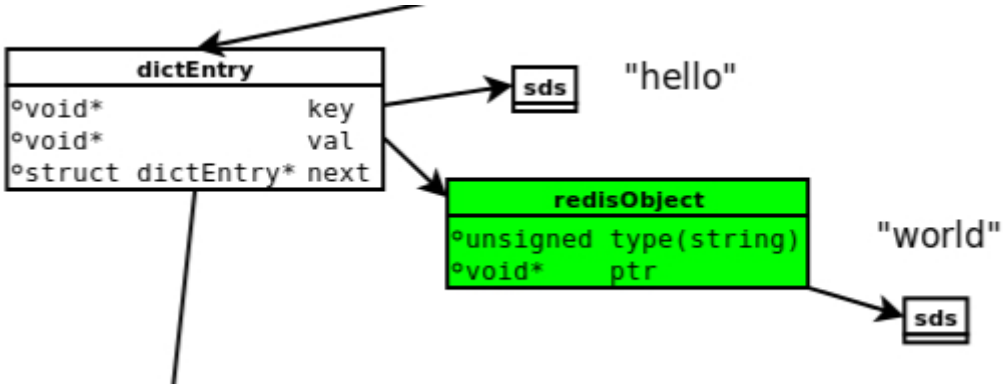


当有序集合使用了 skiplist 编码方式，其实底层采用了zset结构来存储了数据内容。

zset 结构分别用了一个字典 和 一个跳跃表来完成底层实现。

字典的优点，在于它以时间复杂度为O(1) 的速度取值。

跳跃表的优点，在于它以分值进行从小到大的排序。结合二者的优点作为 zset 的整体结构来完成了有序集合的底层实现。

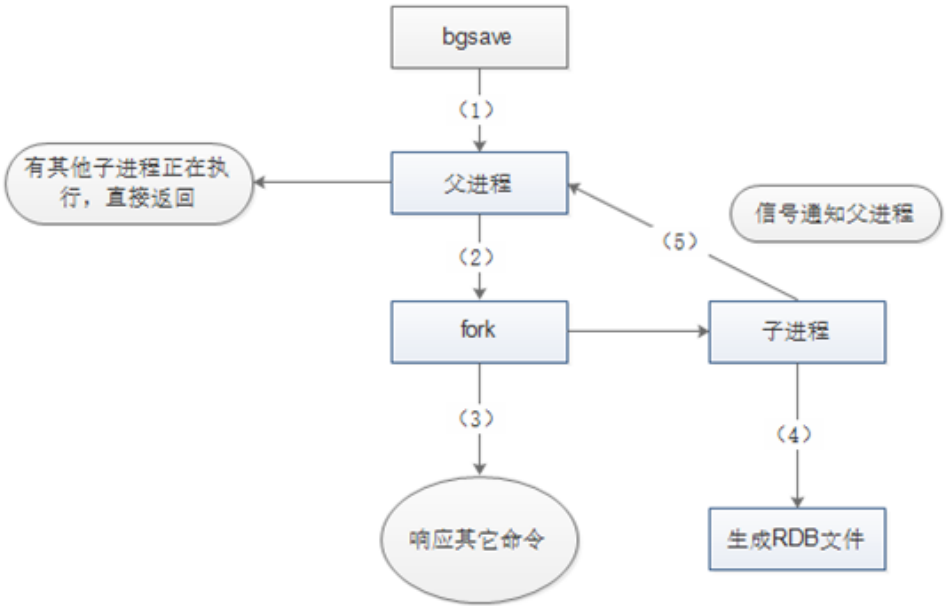


1）dictEntry：Redis是Key-Value数据库，因此对每个键值对都会有一个dictEntry，里面存储了指向Key和Value的指针；next指向下一个dictEntry，与本Key-Value无关。

（2）Key：图中右上角可见，Key（”hello”）并不是直接以字符串存储，而是存储在SDS结构中。

（3）redisObject：Value(“world”)既不是直接以字符串存储，也不是像Key一样直接存储在SDS中，而是存储在redisObject中。**实际上，不论Value是5种类型的哪一种，都是通过redisObject来存储的**；而redisObject中的type字段指明了Value对象的类型，ptr字段则指向对象所在的地址。不过可以看出，字符串对象虽然经过了redisObject的包装，但仍然需要通过SDS存储。

Redis持久化机制

> 持久化是最简单的高可用方法(有时甚至不被归为高可用的手段)，主要作用是数据备份，即将数据存储在硬盘，保证数据不会因进程退出而丢失。
>
> Redis持久化分为RDB持久化和AOF持久化：**前者将当前数据保存到硬盘，后者则是将每次执行的写命令保存到硬盘（类似于MySQL的binlog）；**由于AOF持久化的实时性更好，即当进程意外退出时丢失的数据更少，因此AOF是目前主流的持久化方式，不过RDB持久化仍然有其用武之地。
>
> **RDB持久化：**
>
> 
>
>
> 注意：父进程执行fork操作创建子进程，这个过程中父进程是阻塞的，Redis不能执行来自客户端的任何命令，子进程创建结束则父进程可以重新接受外部请求
>
> **AOF持久化**

文件写入与文件同步机制：

为了提高文件写入效率，在现代操作系统中，当用户调用write函数将数据写入文件时，操作系统通常会将数据暂存到一个内存缓冲区里，当缓冲区被填满或超过了指定时限后，才真正将缓冲区的数据写入到硬盘里。这样的操作虽然提高了效率，但也带来了安全问题：如果计算机停机，内存缓冲区中的数据还没有刷新到磁盘则数据会丢失；因此系统同时提供了fsync、fdatasync等同步函数，可以强制操作系统立刻将缓冲区中的数据写入到硬盘里，从而确保数据的安全性。

AOF缓存区的同步文件策略由参数appendfsync控制，各个值的含义如下：

always：命令写入aof_buf后立即调用系统fsync操作同步到AOF文件，fsync完成后线程返回。

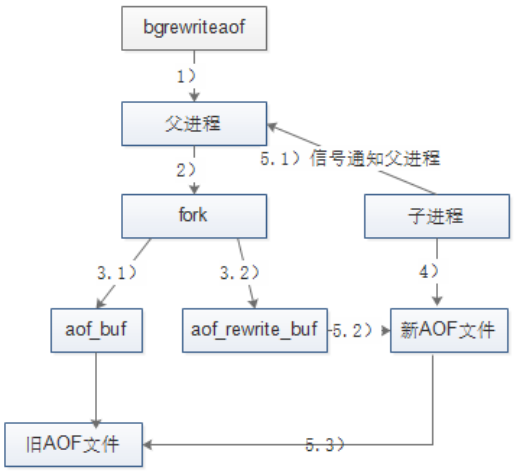
no：命令写入aof_buf后调用系统write操作，不对AOF文件做fsync同步；同步由操作系统负责，通常同步周期为30秒。

everysec：命令写入aof_buf后调用系统write操作，write完成后线程返回；fsync同步文件操作由专门的线程每秒调用一次。

文件重写(rewrite)

AOF重写是把Redis进程内的数据转化为写命令，同步到新的AOF文件；不会对旧的AOF文件进行任何读取、写入操作!

文件重写可以压缩AOF文件，原因包括：已过期的数据不再写入；多条命令可以合并为一条；无效的命令不再写入



关于文件重写的流程，有两点需要特别注意：(1)重写由父进程fork子进程进行；(2)重写期间Redis执行的写命令，需要追加到新的AOF文件中，为此Redis引入了aof_rewrite_buf缓存。

父进程执行fork操作创建子进程，这个过程中父进程是阻塞的。

由于fork操作使用写时复制技术，子进程只能共享fork操作时的内存数据。**由于父进程依然在响应命令，因此Redis使用AOF重写缓冲区(图中的aof_rewrite_buf)保存这部分数据，防止新AOF文件生成期间丢失这部分数据。也就是说，bgrewriteaof执行期间，Redis的写命令同时追加到aof_buf和aof_rewirte_buf两个缓冲区。**

主从复制

主从复制，是指将一台Redis服务器的数据，复制到其他的Redis服务器。前者称为主节点(master)，后者称为从节点(slave)；数据的复制是单向的，只能由主节点到从节点。

目的：数据冗余，故障恢复，负载均衡，高可用基石(哨兵机制和集群的基础

主从复制的原理：主从复制包括了连接建立阶段、数据同步阶段、命令传播阶段；其中数据同步阶段，有全量复制和部分复制两种数据同步方式；命令传播阶段(主节点将自己执行的写命令发送给从节点，从节点接收命令并执行，从而保证主从节点数据的一致性，主从节点之间有PING和REPLCONF ACK命令互相进行心跳检测)。

> 数据同步阶段是主从复制最核心的阶段，根据主从节点当前状态的不同，可以分为全量复制和部分复制
>
> 在Redis2.8以前，从节点向主节点发送sync命令请求同步数据，此时的同步方式是全量复制；在Redis2.8及以后，从节点可以发送psync命令请求同步数据，此时根据主从节点当前状态的不同，同步方式可能是全量复制或部分复制。
>
> 1. 全量复制：用于初次复制或其他无法进行部分复制的情况，将主节点中的所有数据都发送给从节点，是一个非常重型的操作。
>
> 主节点收到全量复制的命令后，执行bgsave，在后台生成RDB文件，并使用一个缓冲区（称为复制缓冲区）记录从现在开始执行的所有写命令;
>
> 主节点的bgsave执行完成后，将RDB文件发送给从节点；**从节点首先清除自己的旧数据，然后载入接收的RDB文件**，将数据库状态更新至主节点执行bgsave时的数据库状态
>
> 主节点将前述复制缓冲区中的所有写命令发送给从节点，从节点执行这些写命令，将数据库状态更新至主节点的最新状态
>
> 2. 部分复制：用于网络中断等情况后的复制，只将中断期间主节点执行的写命令发送给从节点，与全量复制相比更加高效。
>
> 每个Redis节点(无论主从)，在启动时都会自动生成一个随机ID(每次启动都不一样)，由40个随机的十六进制字符组成；runid用来唯一识别一个Redis节点。主从节点初次复制时，主节点将自己的runid发送给从节点，从节点将这个runid保存起来；当断线重连时，从节点会将这个runid发送给主节点；主节点根据runid判断能否进行部分复制，如果从节点发送的runid跟主节点一致，则说明上次同步的时候就是该主节点，那么判断offset是否可以进行部分复制；如果从节点发送的runid跟主节点不一致，说明从节点在断线前同步的Redis节点并不是当前的主节点，只能进行全量复制。
>
> 主节点和从节点分别维护一个复制偏移量（offset），代表的是**主节点向从节点传递的字节数**；主节点每次向从节点传播N个字节数据时，主节点的offset增加N；从节点每次收到主节点传来的N个字节数据时，从节点的offset增加N。offset用于判断主从节点的数据库状态是否一致：如果二者offset相同，则一致；如果offset不同，则不一致，此时可以根据两个offset找出从节点缺少的那部分数据。由于该缓冲区长度固定且有限，因此可以备份的写命令也有限，**当主从节点offset的差距过大超过缓冲区长度时，将无法执行部分复制，只能执行全量复制。**
>
> 可以进行部分复制的条件：从节点保存的runid跟当前连接的主节点的runid一致并且主从节点维护的offset的差值未超出复制积压缓冲区的长度。

哨兵

在复制的基础上，哨兵实现了自动化的故障恢复。缺陷：写操作无法负载均衡；存储能力受到单机的限制。**哨兵的核心功能是主节点的自动故障转移**

哨兵会不断地检查主节点和从节点是否运作正常。当主节点不能正常工作时，哨兵会开始自动故障转移操作，它会将失效主节点的其中一个从节点升级为新的主节点，并让其他从节点改为复制新的主节点。

哨兵节点：哨兵系统由一个或多个哨兵节点组成，哨兵节点是特殊的redis节点，不存储数据。

需要特别注意的是，客观下线是主节点才有的概念；如果从节点和哨兵节点发生故障，被哨兵主观下线后，不会再有后续的客观下线和故障转移操作。

主观下线：在心跳检测的定时任务中，如果其他节点超过一定时间没有回复，哨兵节点就会将其进行主观下线。

客观下线：哨兵节点在对主节点进行主观下线后，会通过sentinel is-master-down-by-addr命令询问其他哨兵节点该主节点的状态；如果判断主节点下线的哨兵数量达到一定数值，则对该主节点进行客观下线。

Redis-Cluster

集群由多个节点(Node)组成，Redis的数据分布在这些节点中。集群中的节点分为主节点和从节点：只有主节点负责读写请求和集群信息的维护；从节点只进行主节点数据和状态信息的复制。

集群的作用，可以归纳为两点：

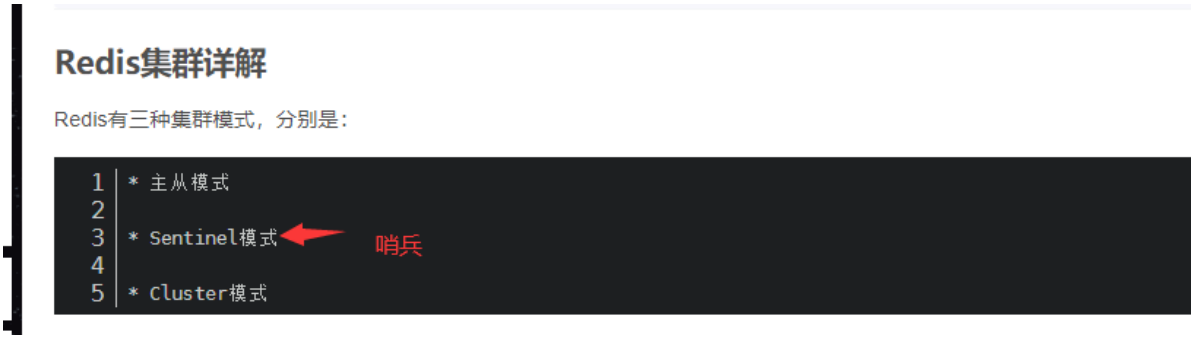
1、数据分区：数据分区(或称数据分片)是集群最核心的功能。

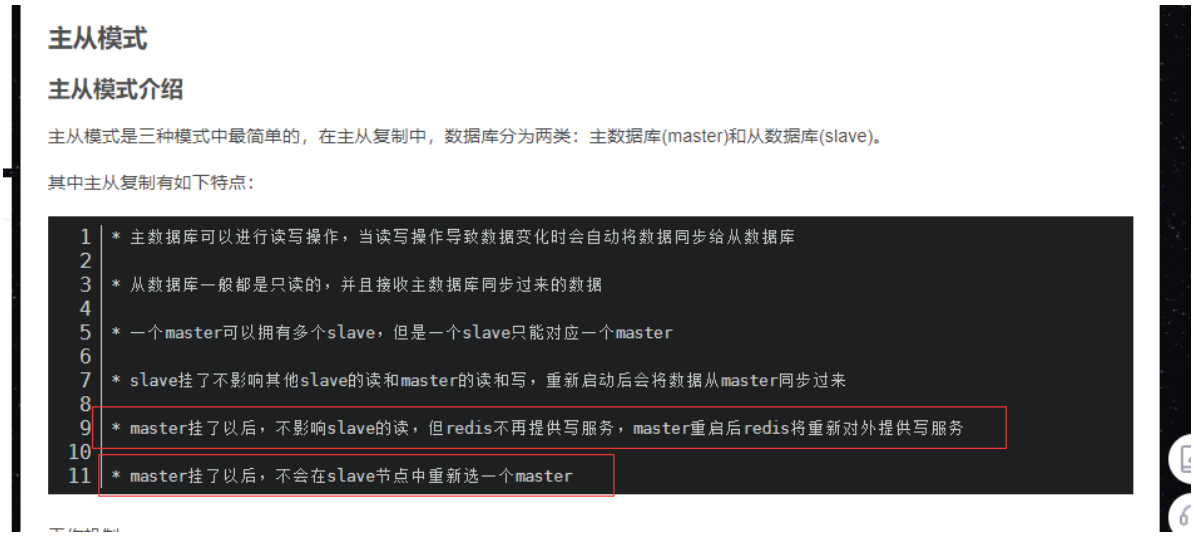
集群将数据分散到多个节点，一方面突破了Redis单机内存大小的限制，存储容量大大增加；另一方面每个主节点都可以对外提供读服务和写服务，大大提高了集群的响应能力。

2、高可用：集群支持主从复制和主节点的自动故障转移（与哨兵类似）；当任一节点发生故障时，集群仍然可以对外提供服务。

在Redis集群中，借助槽实现数据分区，集群有16384个槽，槽是数据管理和迁移的基本单位。当数据库中的16384个槽都分配了节点时，集群处于上线状态（ok）；如果有任意一个槽没有分配节点，则集群处于下线状态（fail）。根据 CRC16(key) mod 16384的值，决定将一个key放到哪个桶中。

redis集群使用一致性哈希算法来实现对槽的分区





工作机制：

当slave启动后，主动向master发送SYNC命令。master接收到SYNC命令后在后台保存快照（RDB持久化）和缓存保存快照这段时间的命令，然后将保存的快照文件和缓存的命令发送给slave。slave接收到快照文件和命令后加载快照文件和缓存的执行命令。

复制初始化后，master每次接收到的写命令都会同步发送给slave，保证主从数据一致性。

Redis rehash 渐进式

> Redis的rehash动作并不是一次性完成的，而是分多次、渐进式地完成的，原因在于当哈希表里保存的键值对数量很大时， 一次性将这些键值对全部rehash到ht[1]可能会导致服务器在一段时间内停止服务，这个是无法接受的。rehashidx：-1->0->不断增大->-1

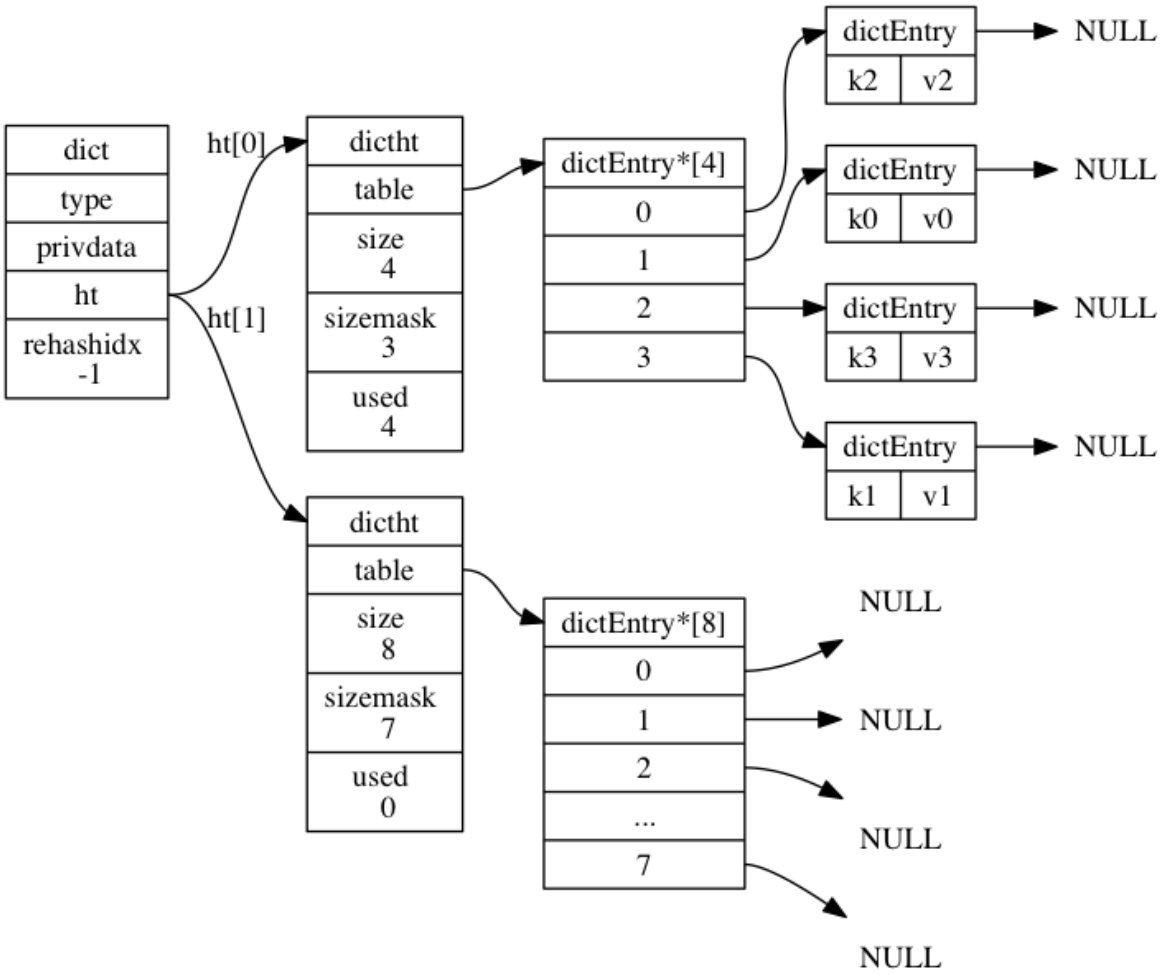
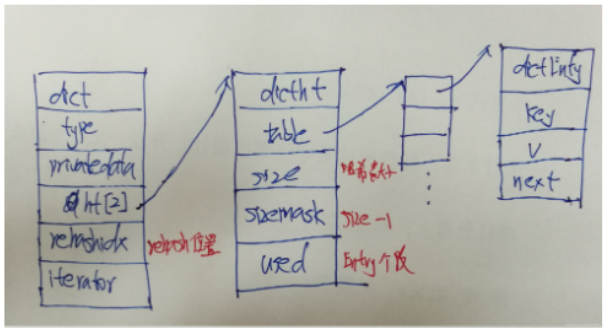


图 4-12　准备开始 rehash

1，redis中hash表的数据结构

在redis中，与hash表存储有关的数据结构主要有3个，分别为dictEntry、dictht、dict。其中dictEntry存储键值对，dictht为包括dictEntry的hash表，dict为定义了两个dictht的字典（2个dictht是为了进行rehash），它们的关系如下：



在dictEntry中存储了key、v以及指向下一个entry的next指针（链地址法）；

在dictht中定义了二重指针table，指向dictEntry的指针个数为size，sizemask用于联合哈市函数计算哈西值，used用于记录哈西表中存储的键值对个数；

dict中与rehash有关的数据有两个，ht[2]表示两个哈西表，其中后者作为rehash过程中的临时哈西表，起过度作用。而rehashidx记录ht[0]中已经进行rehash的下标，如果值为-1表示没有进行rehash。

dict->dictht->dictEntry：分别存储rehash信息，hashTable信息，hashTable元素信息
在redis的具体实现中，使用了一种叫做渐进式哈希(rehashing)的机制来提高字典的缩放效率，避免 rehash 对服务器性能造成影响，渐进式 rehash 的好处在于它采取分而治之的方式， 将 rehash 键值对所需的计算工作均摊到对字典的每个添加、删除、查找和更新操作上， 从而避免了集中式 rehash 而带来的庞大计算量。
在redis中解决hash冲突的方式为采用链地址法。

rehash检查
随着操作的不断执行， 哈希表保存的键值对会逐渐地增多或者减少， 为了让哈希表的负载因子（load factor）维持在一个合理的范围之内， 当哈希表保存的键值对数量太多或者太少时， 程序需要对哈希表的大小进行相应的扩展或者收缩。

redis中，每次插入键值对时，都会检查是否需要扩容。如果满足扩容条件，则进行扩容。

在redis中，扩展或收缩哈希表需要将 ht[0] 里面的所有键值对 rehash 到 ht[1] 里面， 但是， 这个 rehash 动作并不是一次性、集中式地完成的， 而是分多次、渐进式地完成的。为了避免 rehash 对服务器性能造成影响， 服务器不是一次性将 ht[0] 里面的所有键值对全部 rehash 到 ht[1] ， 而是分多次、渐进式地将 ht[0] 里面的键值对慢慢地 rehash 到 ht[1] 。

以下是哈希表渐进式 rehash 的详细步骤：

（1）为 ht[1] 分配空间， 让字典同时持有 ht[0] 和 ht[1] 两个哈希表。

（2）在字典中维持一个索引计数器变量 rehashidx ， 并将它的值设置为 0 ， 表示 rehash 工作正式开始。

（3）在 rehash 进行期间， 每次对字典执行添加、删除、查找或者更新操作时， 程序除了执行指定的操作以外， 还会顺带将 ht[0] 哈希表在 rehashidx 索引上的所有键值对 rehash 到 ht[1] ， 当 rehash 工作完成之后， 程序将 rehashidx 属性的值增一。

（4）随着字典操作的不断执行， 最终在某个时间点上， ht[0] 的所有键值对都会被 rehash 至 ht[1] ， 这时程序将 rehashidx 属性的值设为 -1 ， 表示 rehash 操作已完成。

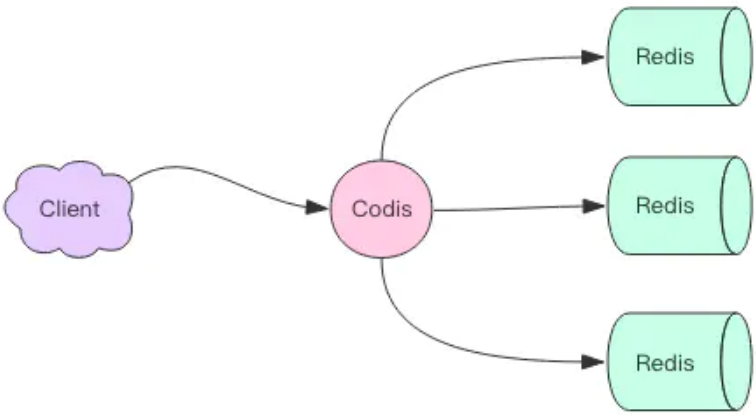
渐进式 rehash 的好处在于它采取分而治之的方式， 将 rehash 键值对所需的计算工作均滩到对字典的每个添加、删除、查找和更新操作上， 从而避免了集中式 rehash 而带来的庞大计算量。

因为在进行渐进式 rehash 的过程中， 字典会同时使用 ht[0] 和 ht[1] 两个哈希表， 所以在渐进式 rehash 进行期间， 字典的删除（delete）、查找（find）、更新（update）等操作会在两个哈希表上进行： 比如说， 要在字典里面查找一个键的话， 程序会先在 ht[0] 里面进行查找， 如果没找到的话， 就会继续到 ht[1] 里面进行查找， 诸如此类。

另外， 在渐进式 rehash 执行期间， 新添加到字典的键值对一律会被保存到 ht[1] 里面， 而 ht[0] 则不再进行任何添加操作： 这一措施保证了 ht[0] 包含的键值对数量会只减不增， 并随着 rehash 操作的执行而最终变成空表。

Codis集群

> Codis 是一个代理中间件，用的是 GO 语言开发的，如下图，Codis 在系统的位置是这样的。



`Codis`分为四个部分，分别是`Codis Proxy (codis-proxy)`、`Codis Dashboard (codis-config)`、`Codis Redis (codis-server)`和`ZooKeeper/Etcd`.

`Codis`就是起着一个中间代理的作用，能够把所有的`Redis`实例当成一个来使用，在客户端操作着`SDK`的时候和操作`Redis`的时候是一样的，没有差别。

`Codis`分片原理

在`Codis`中，`Codis`会把所有的`key`分成`1024`个槽，这`1024`个槽对应着的就是`Redis`的集群，这个在`Codis`中是会在内存中维护着这`1024`个槽与`Redis`实例的映射关系。这个槽是可以配置，可以设置成 2048 或者是4096个。看你的Redis的节点数量有多少，偏多的话，可以设置槽多一些。

`Codis`中`key`的分配算法，先是把`key`进行`CRC32`后，得到一个`32`位的数字，然后再`hash`%`1024`后得到一个余数，这个值就是这个`key`对应着的槽，这槽后面对应着的就是`redis`的实例。

`Codis`之间的槽位同步

`Codis`把这个工作交给了`ZooKeeper`来管理，当`Codis`的`Codis Dashbord`改变槽位的信息的时候，其他的`Codis`节点会监听到`ZooKeeper`的槽位变化，会及时同步过来。

`Codis`集群总结

- `Codis`是一个代理中间件，通过内存保存着槽位和实例节点之间的映射关系,槽位间的信息同步交给`ZooKeeper`来管理。
- 不支持事务和官方的某些命令，原因就是分布多个的`Redis`实例没有回滚机制和WAL,所以是不支持的.

Redis 内存回收

过期键删除策略(删除过期的键值对)：定期删除 惰性删除

内存淘汰机制(内存不足时触发)：

volatile-lru	**从已设置过期时间的数据集中挑选最近最少使用的数据淘汰(常用的)**
volatile-lfu	从已设置过期时间的数据集中挑选最不经常使用的数据淘汰
volatile-ttl	从已设置过期时间的数据集中挑选将要过期的数据淘汰
volatile-random	从已设置过期时间的数据集中挑选任意数据淘汰
allkeys-lru	当内存不足写入新数据时淘汰最近最少使用的Key
allkeys-random	当内存不足写入新数据时随机选择key淘汰
allkeys-lfu	当内存不足写入新数据时移除最不经常使用的Key
no-eviction	当内存不足写入新数据时，写入操作会报错，同时不删除数据

- volatile为前缀的策略都是从已过期的数据集中进行淘汰。
- allkeys为前缀的策略都是面向所有key进行淘汰。