# 字典(哈希表)

字典(哈希表)是redis最基本的数据库，每个DB就是一个字典。同时字典也是哈希对象的使用的数据结构之一。

## 关键数据结构

### dict

typedef struct dict {

dictType \*type;

void \*privdata;

dictht ht[2];

int rehashidx;

int iterators;

} dict;

**type**：该字段指定了哈希表存储的数据类型。type定义了各种哈希操作中所需要的方法，比如插入操作所需要的计算哈希索引，查找操作所需要的key比较等方法。redis中不同的数据类型都实现了自己的哈希表操作所需要的方法，比如dbDictType，setDictType等。

**privdata**：

**ht**：实际存储数据的哈希表数组。两个哈希表是因为rehash过程中需要使用。

**rehashidx**：rehash过程中当前的索引。

**iterators**：目前正在运行的迭代器数量。

### dictht

typedef struct dictht {

dictEntry \*\*table;

unsigned long size;

unsigned long sizemask;

unsigned long used;

} dictht;

**table**：指针数组，数组成员是dictEntry构成的链表。

**size**：哈希表槽位数。

**sizemask**：计算索引值用的，始终等于size – 1。

**used**：哈希表已有的节点数量。

### dictEntry

typedef struct dictEntry {

void \*key;

union {

void \*val;

uint64\_t u64;

int64\_t s64;

} v;

struct dictEntry \*next;

} dictEntry;

**key**：哈希表成员的键。

**v**：哈希表成员的值。

**next**：指向相同槽位下的下一个哈希表成员。

## 常用方法

### dictCreate

入参type：哈希表成员的类型。

入参privateDataPtr：私有数据类型的指针。

出参：创建出哈希表的指针。

函数很简单，就是分配一个dict数据结构，然后对其进行初始化。哈希表默认大小为4个槽位。

### dictAdd

入参d：执行插入操作的哈希表指针。

入参key：待插入的键。

入参val：待插入的值。

出参：DICT\_OK表示成功，DICT\_ERR表示失败。

1. 调用dictAddRaw，判断待插入的键是否在哈希表中存在。如果不存在则新建一个entry。如果已经存在则终止插入操作。
2. dictAddRaw函数中会判断是否要进行单步的rehash操作。一种惰性的rehash。
3. 调用\_dictKeyIndex判断键是否已经存在哈希表中。判断的过程就是根据key值计算出哈希索引，然后遍历索引对应的哈希槽位中的成员，看是否有一样的存在。需要注意的是，如果此时正在进行rehash过程，那么需要遍历ht[0]和ht[1]才能判定键是否存在。

### dictFind

入参d：指向待查找哈希表的指针。

入参key：用于查找的键。

出参：如果找到返回对应的dictEntry结构，找不到返回NULL。

函数过程和上面dictKeyIndex基本一致，不再赘述。

### dictDelete

入参ht：待操作的哈希表。

入参key：待删除的键。

出参：DICT\_OK表示成功删除，DICT\_ERR表示键不存在。

函数过程先通过上面的查找方式找到对应的键，然后将键从哈希槽位的链表删除。最后调用键自定义的释放函数，释放键内存。

## 哈希扩容与缩容

### 扩容条件：

负载因子的计算方式：factor = ht[0].used / ht[0].size

负载因子大于5的时候，开始执行哈希扩容。扩容之后大小是第一个大于ht[0].used\*2这个值的2的倍数。

在BGSAVE和BGREWRITEAOF执行期间不能执行rehash操作。

### 缩容条件：

负载因子小于0.1。

缩容后的大小是ht[0].used，但最小不能小于4个槽位。

### 渐进式rehash

1. 为ht[1]分配空间，将rehashidx设置为0，表示rehash开始。
2. rehash操作分为两种，一种是通过定时器，每隔一段时间就迁移一部分哈希表中的成员。另一种就是依靠哈希增删改查的操作，在rehash期间，每次操作也会触发当前槽位执行rehash操作。
3. 定时rehash的入口是在databasesCron，用1ms的时间尝试完成100个槽位的键迁移。100个槽位迁移完成或者时间耗尽都会结束。
4. 迁移的过程就是将ht[0]中，rehashidx所对应槽位下所有的成员重新计算哈希值，然后加入到ht[1]的表中。
5. 迁移完成之后，释放原来ht[0]表中的内容，然后将ht[1]中的内容给到ht[0]。

# 字符串

redis中使用SDS（简单动态字符串）来作为字符串存储的结构。相比传统C字符串，SDS有以下几点优势：

1. 常数时间获取字符串长度。
2. 避免缓冲区溢出。
3. 减少字符串修改时所带来的内存分配次数。
4. 二进制安全，因为sds是通过len字段来记录保存字符串的长度，而不是像C语言是通过’/0’来判断字符串结束。因此sds保存的字符串就可以包含’\0’字符。

SDS减少内存分配次数的方法就是空间预分配和空间惰性释放。

空间预分配：空间扩展的时候，如果SDS修改之后，长度小于1MB，则分配len 长度的未使用空间。如果修改之后，长度大于1MB，则预分配1MB的未使用空间。

空间惰性释放：当字符串剩余空间增加的时候，不会主动释放剩余空间，而是通过free字段来记录剩余空间的大小，以备后续使用。这种情况出现在字符串更新的时候，更新之后的字符串比原来的字符串短。

## 关键数据结构

### sdshdr

struct sdshdr {

int len;

int free;

char buf[];

};

**len**：已使用的字节数，即字符串的长度。

**free**：未使用的字节数。

**buf**：指向保存字符串的内存空间。

## 常用方法

### sdsnew

入参init：给定的字符串。

出参：返回新创建sds结构。

如果传入的字符串是空的，那么分配内存的时候会将内存全部初始化。如果传入的字符串非空，则不会初始化全部内存。新创建的sds没有任何的预留空间，所以free字段是0。

### sdsMakeRoomFor

入参s：需要扩展的sds结构。

入参addlen：需要扩展的长度。

出参：扩展之后的sds结构。

1. newlen = len + addlen，是扩展之后最少需要的空间大小。
2. 如果newlen < 1MB，那么newlen \*= 2。如果newlen > 1MB，那么newlen += 1MB。这和上面描述的一样。

### sdsRemoveFreeSpace

入参s：需要进行空间回收的sds结构。

出参：完成空间回收的sds结构。

空间回收之后，sds的空间与len字段相同，free字段为0。

# 压缩链表

redis中列表和哈希表都会用到压缩链表作为数据存储的结构。压缩链表一般用来存放一些小整数或者短字符串，使用一块完整连续的内存来实现链表。可以很好的减少存放小数据带来的内存碎片。

## 关键数据结构

### 链表头部

压缩链表头部没有定义具体的数据结构，而是使用一块连续内存，分隔出几个字段来记录链表的属性。头部包含以下几个字段：

zlbyte：第一个uint32字段，记录链表使用的内存大小。

zltail：第二个uint32字段，记录达到链表最后一个节点的偏移量。

zllen：第三个uint16字段，记录链表中节点数量。受限于字段大小，最大值为65535。因此当zllen小于65535时，表示的是真实的节点数量。当zllen等于65535时，真实的节点数量需要遍历链表才能得到。

zlend：第四个1字节的字段。赋值为255，表示链表结束。

### zlentry

typedef struct zlentry {

unsigned int prevrawlensize, prevrawlen;

unsigned int lensize, len;

unsigned int headersize;

unsigned char encoding;

unsigned char \*p;

} zlentry;

**prevrawlensize**：编码prevrawlen所需的字节大小。

**prevrawlen**：前置节点的长度。

**lensize**：编码len所需的字节大小。

**len**：当前节点的长度。

**headersize**：当前节点头部的大小。等于prevrawlensize + lensize。

**encoding**：当前节点所使用的编码类型。

**p**：指向当前节点的指针。

## 常用方法

### ziplistNew

入参：无。

出参：链表头部指针。

创建链表的头部，并初始化链表头部字段。链表头部格式上面已经说明。

### ziplistPush

入参zl：待加入数据的链表指针。

入参s：待加入的字符串。

入参slen：待加入的字符串长度。

入参where：加入的方向，是加入表头还是表尾。

出参：完成加入操作之后的链表头部指针。

### \_\_ziplistCascadeUpdate

## 连锁更新

当新的节点加入某个节点前面的时候，原节点header不够保存新加入节点的长度，就需要将原节点的header从1个字节扩展到5个字节。但由于压缩链表是一个连续内存，所以中间一个字段扩展之后，会引发后面所有字段都要做相应的调整。

# 跳跃表