目录

[一、 简单动态字符串(simple dynamic string,SDS) 2](#_Toc40127799)

[(一) 定义 2](#_Toc40127800)

[(二) SDS和C字符串的区别 2](#_Toc40127801)

[(三) SDS的主要操作API 3](#_Toc40127802)

[二、 链表 4](#_Toc40127803)

[(一) 链表节点的实现 4](#_Toc40127804)

[(二) 链表的实现 4](#_Toc40127805)

[(三) 链表API 5](#_Toc40127806)

[三、 字典 6](#_Toc40127807)

[(一) 字典的实现 6](#_Toc40127808)

[1) 哈希表 6](#_Toc40127809)

[2) 哈希表节点 6](#_Toc40127810)

[3) 字典 6](#_Toc40127811)

[(二) 哈希算法 8](#_Toc40127812)

[(三) 解决键冲突 8](#_Toc40127813)

[(四) Rehash 8](#_Toc40127814)

[(五) 渐进rehash 8](#_Toc40127815)

[(六) 字典API 9](#_Toc40127816)

[四、 跳表 10](#_Toc40127817)

[(一) 跳表的实现 10](#_Toc40127818)

[(二) 跳表节点的实现 10](#_Toc40127819)

[(三) 跳表节点的跨度 11](#_Toc40127820)

[(四) 跳表API 11](#_Toc40127821)

[五、 整数集合 12](#_Toc40127822)

[(一) 整数集合的实现 12](#_Toc40127823)

[1) Content数组 12](#_Toc40127824)

[(二) 升级 12](#_Toc40127825)

[(三) 为什么要升级 12](#_Toc40127826)

[(四) 整数集合不支持降级 12](#_Toc40127827)

[(五) 整数集合API 13](#_Toc40127828)

[六、 压缩列表 13](#_Toc40127829)

[(一) 压缩列表的构成 13](#_Toc40127830)

[(二) 压缩列表节点的构成 13](#_Toc40127831)

[1) Previous\_entry\_length 14](#_Toc40127832)

[2) Encoding 14](#_Toc40127833)

[3) Content 14](#_Toc40127834)

[(三) 连锁更新 14](#_Toc40127835)

[(四) 压缩列表API 15](#_Toc40127836)

# 简单动态字符串(simple dynamic string,SDS)

## 定义

Struct{

Int len; //buf数组已使用字节的数量

Int free; //未使用字节的数量

Char buf[]; //字节数组

}

## SDS和C字符串的区别

1. **常数时间复杂度获得字符串长度**
2. **杜绝缓冲区的溢出（SDS会检查空间是否足够）**
3. **减少修改字符串带来的内存重分配次数**

C字符串：

每次增长或缩短一个C字符串都会堆这个保存C字符串的数组进行一次内存重分配操作。

SDS：

1. 空间预分配（优化增长操作）：减少了连续执行增长操作所需的内存重新分配次数

1）修改以后小于1mb则分配len大小的未使用空间

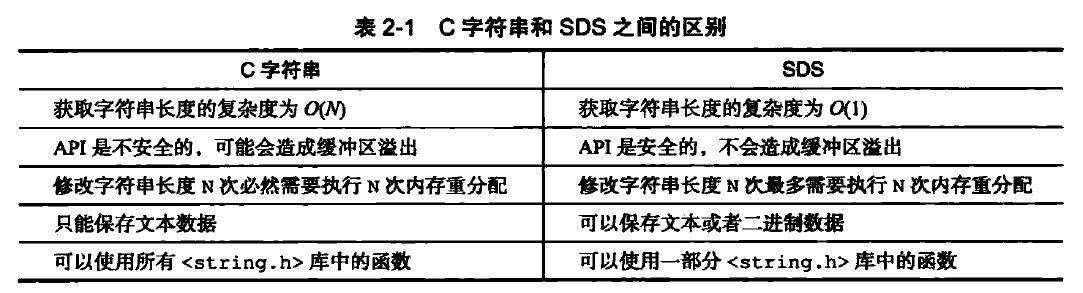
2）修改以后大于1mb则分配1mb的未使用空间

在扩展SDS空间之前，如果未使用空间足够，就不会内存重分配

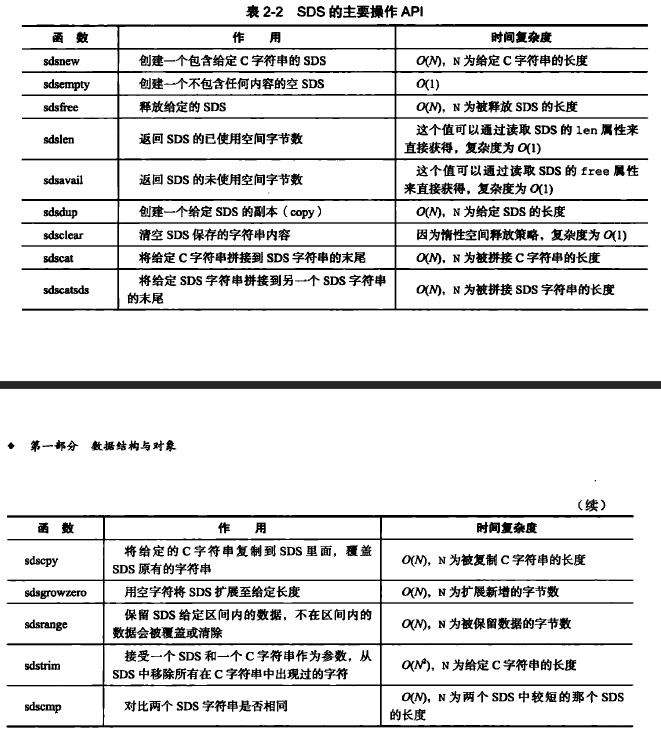
1. 惰性空间释放（优化缩短操作）：通过改变free属性模拟缩短
2. **二进制安全**

Buf被成为字节数组的原因——Redis不是用这个数组来保存字符而是用它来保存一系列二进制数据。（SDS API以处理二进制的方式来处理buf数组中的数据）

1. **兼容部分C字符串函数**
2. **总结：**



## SDS的主要操作API



# 链表

## 链表节点的实现

Typedef struct listNode{

struct listNode \*prev;

struct listNode \*next;

void \* value;

}

## 链表的实现

Typedef struct list{

listNode \*head;

listNode \*tail;

unsigned long len;

void \*(\*dup)(void \*ptr); //节点值复制函数

void \*(\*free)(void \*ptr); //节点值释放函数

void \*(\*match)(void \*ptr, void \*key); //节点值对比函数

}list

## 链表API



# 字典

## 字典的实现

### 哈希表

Typedef struct dictht{

dictEntry \*\*table; //哈希表数组

unsigned long size; //哈希表大小

unsigned long sizemask; //哈希表大小掩码，用于计算索引值，总是等于size-1

unsigned long used; //哈希表已有节点

}ditch;

Table属性是一个数组，数组的元素是指向dict.h/dictEntry结构的指针，每个dictEnrty结构保存着一个键值对。

### 哈希表节点

Typedef struct dictEntry{

void \*key; //键

union{ //值

void \*val;

uint64\_tu64;

int64\_ts64;

}v;

Struct dictEntry \*next;//指向下一个哈希表节点，形成链表

}

Next属性是指向另一个哈希表节点的指针，这个指针可以将多个相同的键值对连接在一起，以此来解决键冲突的问题。

### 字典

Typdef struct dict{

dictType \*type; //类型特定函数

void \*privdata; //私有数据

dictht ht[2]; //哈希表

//rehash索引

Int rehashidx; //当rehash不进行时，值为-1

}

**Type属性**是指向dictType结构的指针，每个dictTytpe结构保存了一簇用于操作特定类型键值对的函数，redis会为不用用途的字典设置不同类型的特定函数。

**Privdata属性**则保存了需要传给那些类型特定函数的可选参数。

Typedef struct dictType{

//计算哈希值的函数

Unsigned int (\*hashFunction)(const void \*key);

//复制键的函数

Void \*(\*keyDup) (void \*privdata, const void \*key);

//复制值的函数

Void \*(\*valDup) (void \*privdada, const void \*obj);

//对比键的函数

Int \*(keyCompare) (void \*privdata, const void \*key1, const void \*keys);

//销毁键的函数

Void (\*keyDestructor) (void \*privdata, void \*key);

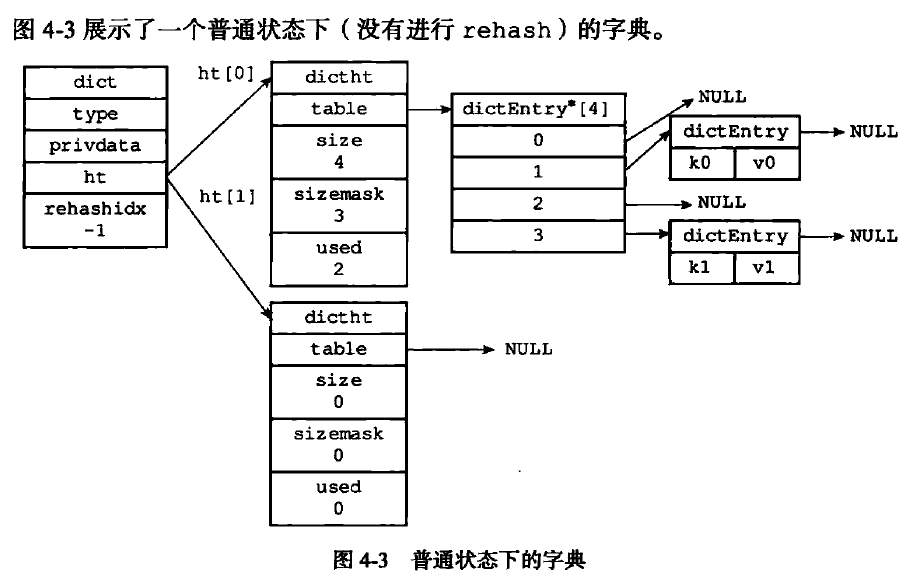
//销毁值的函数

Void (\*valDestructor) (void \*privdata, void \*obj);

}

**Ht属性**是一个包含两个哈希表项的数组。一般情况下字典只用ht[0],而ht[1]则在对ht[0]进行rehash时使用。

**Rehashidx属性**记录了rehash目前的进度，如果没有在进行则值为-1



## 哈希算法

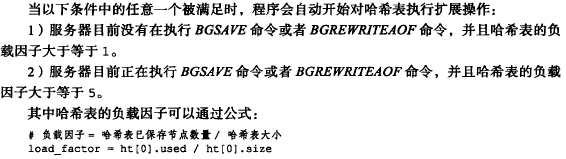
当字典被用作数据库的底层实现，或者哈希键的底层实现时，Redis使用 MurmurHash2算法来计算键的哈希值。（随机性好，速度快）

## 解决键冲突

使用链地址法，所有冲突的键组成一个链表。

## Rehash

1. 步骤：
2. 为字典ht[1]分配空间，空间大小取决于是扩展还是收缩：
   1. 扩展，第一个大于等于ht[0].used\*2的2^n
   2. 收缩，第一个大于等于ht[0].used的2^n
3. 将ht[0]上的键值对全部rehash到ht[1]上面，然后释放ht[0]，将ht[1]设置为ht[0]。同时ht[1]创建空哈希表
4. 哈希表的扩展与收缩：



1. 问题：
2. 为什么根据BGSAVE命令或者BAREWRITEAOF命令是否执行，所需负载因子不同

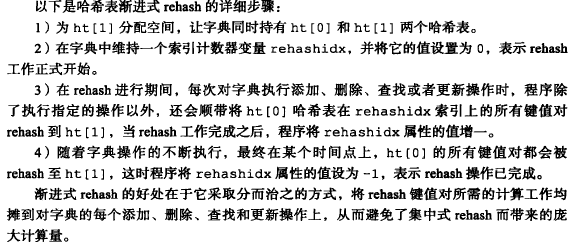
因为在命令执行过程中，Redis需要创建当前服务器进程的子进程，而大多数操作系统采用写时复制来优化子进程的使用效率，所以在子进程存在期间，服务器会提高扩展所需的负载因子，从而避免子进程存在期间进行哈希表扩展操作，避免不必要的内存写入操作，从而节约内存。

1. 什么时候会对哈希表进行收缩操作

负载因子小于0.1时。

## 渐进rehash

如果在ht[0]中有大量键值对，一次性地从ht[0]rehash到ht[1]会导致服务器在一段时间内停止服务。因此rehash的过程时分多次，渐进式地转移的。



注意：在rehash期间执行的哈希表操作会在两个表中进行。

## 字典API



# 跳表

## 跳表的实现

typedef struct zskiplist {

*// 头节点，尾节点*

struct zskiplistNode \*header, \*tail;

*// 节点数量*

unsigned long length;

*// 目前表内节点的最大层数*

int level;

} zskiplist;

## 跳表节点的实现

typedef struct zskiplistNode {

// member 对象

robj \*obj;

// 分值

double score;

// 后退指针

struct zskiplistNode \*backward;

// 层

struct zskiplistLevel {

// 前进指针

struct zskiplistNode \*forward;

// 这个层跨越的节点数量

unsigned int span;

} level[];

} zskiplistNode;

## 跳表节点的跨度

跳表节点的跨度的主要用途在于计算排名

## 跳表API



# 整数集合

整数集合是redis用于保存整数值的集合抽象数据结构，它可以保存类型为int16\_t,int32\_t,int64\_t的整数值，并保证集合内不存在重复元素。

## 整数集合的实现

Typedef struct intset{

Uint32\_t encoding; //编码方式

Uint32\_t length; //集合包含的元素数量

Int8\_t contents[]; //保存元素的数组

}

### Content数组

* 1. 数组元素按从小到大排序
  2. 虽然contents属性声明为int8\_t，实际上真正的类型取决于encoding属性

## 升级

升级分三步：

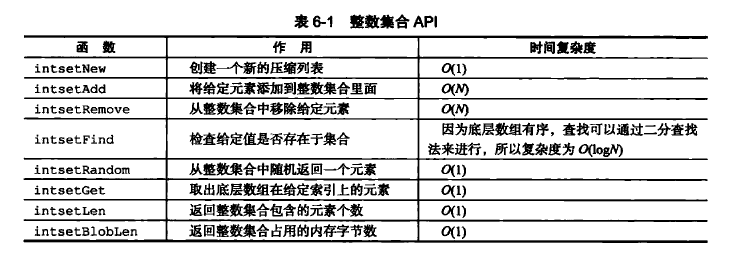
1. 根据新元素的类型，扩展底层数组的空间大小，并为新元素分配空间
2. 将底层数组现有的所有元素都转换成新元素相同的类型，然后放到正确的位置上，并在放置过程中维持有序性质（从后往前放）
3. 将新元素一俺家到底层数组里面

## 为什么要升级

1. 提升灵活性
2. 节约内存

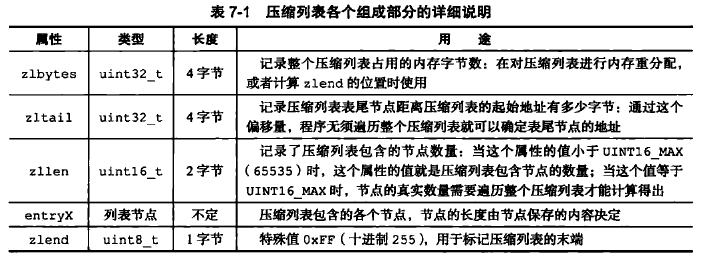
## 整数集合不支持降级

## 整数集合API



# 压缩列表

## 压缩列表的构成



## 压缩列表节点的构成

每个压缩列表节点可以保存一个字节数组或者一个整数值。其中：

字节数组可以是以下长度的一种：

长度小于等于(2^6-1)字节的字节数组

长度小于等于(2^14-1)字节的字节数组

长度小于等于(2^32-1)字节的字节数组

整数值可以是以下长度的一种：

4位长，介于0和12之间的无符号整数

1字节长的有符号整数

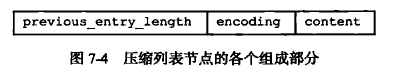
3字节长的有符号整数

Int16\_t

Int32\_t

Int64\_t

压缩列表节点的组成如下：

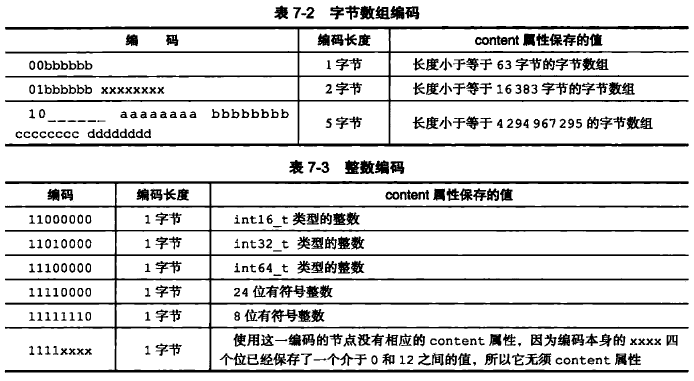


### Previous\_entry\_length

以字节为单位，记录压缩列表中前一个节点的长度（可以是1字节或5字节）。可以通过这个倒着遍历节点。

### Encoding

编码分为字符数组和整数



### Content

Content负责保存节点的值，节点值可以是一个字节数组或者整数，由encoding决定

## 连锁更新

体现为在某个位置加入节点后导致后面节点的previous\_entry\_length不足存的下新加入节点的长度，从而需要重新配置空间。重新配置后由导致后面的存不下…这样的反应。

要注意的是，尽管连锁更新的复杂度较高，当真正造成性能问题的概率很低：

1. 首先压缩列表要恰好多个连续的、长度介于250字节到253字节。在实际中，这种情况并不多见。
2. 其次只要被更新的节点数量不多就不会对性能造成任何影响。

## 压缩列表API

