# 第一部分 数据结构与对象

## 简单动态字符串

1. Redis没有直接使用c语言传统的字符串（以空字符结尾的字符数组）而是自己构建了一种叫做简单动态字符串的类型（SDS）
2. SDS的结构在sds.h/sdshdr

struct sdshdr{

int len;//记录buf数组已使用的字节数目

int free;//未使用的字节数目

char buf[];//保存字符串

}；

由于记录了长度信息，所以可以在常数复杂度的情况下获取字符串长度；还能避免缓冲区的溢出；通过预留空间，减少修改字符串时带来的内存重分配次数

1. SDS增加空间的规则：
2. SDS修改后，SDS长度小于1MB；那么就使len和free值相等，即总空间为 2\*len+1
3. 大于等于1MB,则令free=1MB,即总空间为 len+1MB+1
4. 惰性空间释放：free来记录释放空间的值
5. SDS是二进制安全的，因为是用len来保证字符数组长度
6. 在buf数组最后自动补上’\0’，可以重用<string.h>中的函数

## 链表

1. 由于原来Redis使用的c语言没有内置的链表结构，所以实现了自己的链表结构
2. 链表结构

typedef struct listNode{

struct listNode\* prev;

struct listNode\* next;

void\* value;

}listNode;

typedef struct list{

listNode\* head;

listNode\* tail;

unsigned long len;

void\* (\* dup)(void\* ptr);//节点复制函数

void (\*free)(void\* ptr);//节点值释放函数

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

};

这样设计让list具有多态性，数据用void\*，而函数用的函数指针，可以根据自己需求定义函数

## 字典

1. 字典即使一个键值对，Redis的数据库就是用字典作为底层实现的；字典也是哈希键的底层实现之一
2. 字典是用哈希表作为底层实现的：

节点：

typedef struct dictEntry{

void\* key;

union{

void\* val;

uint64\_t u64;

int64\_t s64;

}v;

struct dictEntry\* next;//指向写个哈希表节点，hash值和掩码得到值一样的，形成一个链表

}dictEntry;

哈希表：

typedef struct dictht{  
 dictEntry\*\* table;//哈希数组

unsigned long size;//哈希表大小

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

unsigned long used;//已用节点数

}dictht;

字典：

typedef struct dict{  
 dictType\* type;//类型特定函数

void\* privdata;//私有数据

dictht ht[2];//一般只会用ht[0]哈希表，在rehash是才会用ht[1]

int rehashidx;//rehash索引，当不在rehash时，值为-1

};

typedef struct dictType{  
 unsigned int (\*hashFunction)(const void\* key);

void\* (\*keyDup)(void\* privatdata,const void\* key);

void\* (\*valDup)(void\* privatedata,const void\* obj);

void (\*keyCompare)(void\* privatedata,const void\* key1,const void\*key2);

void (\*keyDestructor)(void\* privatedata,void\* key);

void (\*valDestructor)(void\* privatedat,void\* obj);

}dictType;

1. rehash:为了让哈希表的负载因子维持在一个合理额范围内。在哈希表保存的键值对数量太多或者太少时，程序需要对哈希的大小进行相应的扩展或者收缩；即扩大或缩小哈希表的size,重新计算哈希值和掩码值，放入对应的哈希表的位置；rehash的过程是渐进式的，不是一下将所有的键值对移动到新的哈希表中，即从哈希表数组中按序号一组一组的将键值对转移出去，由rehashidx记录已经做完的数组序号

## 跳跃表

1. 大部分情况下，跳跃表的效率可以和平衡树差不多，而且跳跃表的实现更为简单
2. 跳跃表两个使用地方：a.Redis使用跳跃表作为有序集合键的底层实现之一;b.在集群节点中用作内部数据结构
3. 跳跃表实现

跳跃表节点：

typedef struct zskiplistNode{

struct zskiplistNode\* backward;

double score;//按照它排列的

robj\* obj;

struct zskiplistLevel{

struct zskiplistNode\* forward;

unsigned int span;

}level[];

}zskiplistNode;

跳跃表：

typedef struct zskiplist{

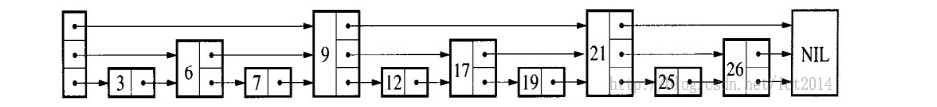
struct zskiplistNode\* header,tail;

unsigned long length;

int level;

}zskiplist;

1. 举例：



结合上图，如果想查找19是否存在，从最高层开始，首先和头结点的最高层的后继结点9进行比较，19大于9，因此接着和9在该层上的后继结点21进行比较，小于21，那这个值肯定在9结点和21结点之间。

        因此，下移一层，接着和9在该层上的后继结点17进行比较，19大于17，然后和21进行比较，小于21，此时肯定在17结点和21结点之间。

        接着下移一层，和17在该层上的后继结点19进行比较，这样就最终找到了。

        上面就是跳跃表的基本思想，跳跃表结点包含多少个指向后继元素的指针，是通过一个随机函数生成器得到的。这就是为什么论文“Skip Lists : A Probabilistic Alternative to Balanced Trees ”中有“概率”的原因了，就是通过随机生成一个结点中指向后续结点的指针数目。

## 整数集合

1. 整数集合是集合键的底层实现之一
2. 整数集合的实现

type struct intset{

uint32\_t encoding;//编码方式，决定contents是什么类型的数组

uint32\_t length;//集合包含元素数量

int8\_t contents[];//元素从小到大排列，虽然类型声明为int8\_t,但是具体类型是由encoding字段指定的

}intset;

1. 当添加到整数集合中的新元素和集合中现有元素类型要长的话，整数集合先需要升级（将集合中所有元素的大小变成新元素的大小，并放在对应的应该在的位置，最后将新元素插入）
2. 升级的好处：提升整数集合的灵活性（可以存储不同类型的整数，而不担心出错，通过升级来适配）；节约内存（在需要在会往大类型扩张）。整数集合不支持降级操作

## 压缩列表

1. 压缩列表是列表键和哈希键的底层实现之一，存一些小整数或者是长度比较短的字符串
2. 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

zlend

entryn

...

entry1

zllen

zlbytes

zltail

zlbytes（4字节）:整个压缩列表所占字节总数

zltail（4字节）:表尾节点所在位置（为了可以从后往前遍历）

zllen（2字节）:节点数

entry（不定）:节点

zlend（1字节）:标识列表末端（0xFF）

entry结构（节点内可以保存一个字节数组或者一个整数值）

previous\_entry\_length

content

encoding

previous\_entry\_length（1字节或5字节）:记录前一个节点的长度（为了从后往前遍历）

若前节点长度<254,使用一个字节

若 >=254,5字节，第一个字节为0xFE,后四个字节记录长度数

encoding:保存字符数组所用字节数和区分不同整数类型。字符数组（1,2,5字节），用第一个字节的前两位区分，分别为00,01,10

整数（1字节）11开头，其中整数中的int16\_t,int32\_t,int64\_t会进一步区分，分别为：

11000000,11010000,11100000

content:具体的数据

## 对象

1. 将前面提及的数据结构上再搭一层数据结构，即为对象
2. 对象的实现

typedef struct redisObject{

unsigned type:4;//类型

unsigned encoding:4;//编码

void\* ptr;//指向底层的数据结构

}robj;

type:如字符串，列表，哈希等

encoding:简单动态字符串，字典，双端链表，跳跃表等

每种type至少有两种不同的encoding

1. 字符串对象：编码方式，int,raw,embstr

整数值使用int

字符串长度小于等于39使用embstr

字符串长度大于39使用raw

1. 列表对象：编码方式，ziplist和linkedlist（节点是字符串对象，字符串对象是唯一种类型可被其他类型嵌套使用的）

ziplist的条件：a.保存的对象长度小于64字节；b.元素个数小于512个

其余情况用linkedlist

1. 哈希对象：编码方式：ziplist和hashtable

ziplist存储键值对方式：每来一个新的键值对，先存键到表的末尾，再存值到表的末尾

ziplist的条件：a.键和值长度小于64字节 b.键值对数量小于512字节

否则使用hashtable

1. 集合对象：编码方式：intset和hashtable

hashtable:字典中每个键包含一个集合元素，值设置为NULL

inset的条件：a.保存元素都是整数 b.元素数量不超过512个

其余用hashtable

1. 有序集合对象：编码方式：ziplist和skiplist

ziplist:每个集合元素使用两个紧挨在一起的压缩列表节点来保存，第一个节点保存元素成员，第二个节点保存分值

skiplist：对应的有序集合使用zset结构作为底层实现，而zset同时包含一个字典和一个跳跃表

为什么要同时使用字典和跳跃表：字典保证了能通过集合元素找到对应的分值，而跳跃表保证了范围查询。

ziplist使用条件：a.元素数量小于128个 b.元素长度小于64字节

## 数据库

1. Redis服务器实现

struct redisServer{

...

redisDb\* db;//保存服务器中所有的数据库，默认情况下工作在0号数据库，可以通过select切换数据库

int dbnum;//服务器的数据库数量，默认为16

...

};

1. Redis客户端实现

3.Redis数据库里面的映射由字典完成，它是Redis的起点，它的键为一个字符串对象，而它的值可以是字符串对象，列表对象，哈希表对象，集合对象和有序集合对象中的任意一种Redis对象

struct redisClient{

...

redisDb\* db;//记录当前正在使用的数据库

...

};

4.键空间的的管理，由于有些键设置了生存时间或过期时间，所以会更新更新键的操作，每次访问一个键时，都是更新它的LRU，这个会影响键的闲置时间，从而会影响键是否过期

typedef struct redisDb{

...

dict\* dict;//键空间字典

dict\* expires;//过期字典，存着带有过期时间的所有键的过期时间

...

};

1. 持久化：RDB,AOF

对于过期键的不同处理，RDB:过期的键不会存入RDB中

AOF:当键过期之后，程序会在AOF文件中追加一条DEL命令，来标识该键被删除

## RDB持久化

1. 因为Redis是内存数据库，所以需要把数据库的状态持久化，一个方式就是用RDB，两个方法，一种手动，一种服务器配置选项定期执行
2. 由于AOF文件的更新频率比RDB文件的频率高，所以服务器会优先使用AOF文件来还原数据库状态