redis-数据结构与对象

*简单动态字符串

- SDS(简单动态字符串)
- SDS的定义

```
struct sdshdr{
    int len;//记录buf数组中已使用字节的数量=SDS中所保存的字符串的长度
    int free;//记录buf数组中未使用的字节的数量
    char buf[];//字符数组。用于保存字符串，最后的一个字节为'\0'
}
```

- **SDS与C的区别**
  - **常数复杂度获取字符串的长度**

    对于c语言的字符串数组来说，每次获取数组的长度是O(n)，然而对于SDS来讲，它的结构体中存储了len这个变量，设置和更新SDS长度的工作是由SDS的API在执行的时候自动完成更新的，不需要手工修改长度的工作，因此对于SDS来说，获取字符串的长度的时间复杂度就变成O(1).
  - **杜绝缓冲区溢出**
    - 对于c语言来说，如果给字符串S分配的空间资源小于字符串S最终拼接之后的空间大小的话，那么就会造成缓冲区的溢出。例子：两个字符串相邻排列，但对第一个字符串进行拼接字符串的时候，那么就会影响到第二个字符串。
    - 当SDSAPI需要对SDS进行修改的时候，API会先检查SDS的空间是否够用，如果不满足的话，API会自动将SDS的空间扩展至执行所需要的空间大小，然后才会去执行字符串的连接。
  - **减少修改字符串时带来的内存重分配次数**
    - 当c语言的字符串增加或者删减的时候，程序都要对更新过后的字符串进行内存的重新分配。
    - SDS操作
      - 空间预分配：用于优化SDS的字符串的增长操作，当SDS的API对一个SDS进行操作的时候，并且需要对SDS进行空间扩展的时候，程序不仅会为SDS分配修改所需要的空间，还会为SDS分配额外未使用的空间。通过这种空间预分配的策略，可以减少执行字符串增长操作所需要的的内存重新分配的次数。也就是说通过每次扩容时，多分配一些空间来减少扩容的次数。
      - 惰性空间释放：用于优化SDS的字符串缩短操作，当SDS的API要对SDS进行缩短的操作时，程序并不会立即收回缩短后多余出来的字节，而是使用free属性保存起来，并等待下次使用。通过这种策略主要是为了防止缩短过后的操作时连接操作，这样一来，就不需要申请空间了。
  - **二进制安全**
    - c语言在存储字符串的时候，遇见空格就会停止读取，所以只能保存文本文件，而不能保存图片，音频，视频，压缩文件这样的二进制数据。
    - SDS的API都是二进制安全的，所有的SDSAPI都会以处理二进制的方式来处理SDS存放的buf数组中的数据。
  - SDS兼容了C字符串的函数
    - SDS中buf字符串的末尾都会以'\0'结尾，这是为了可以重用一部分<string.h>库中定义的函数，比如：strcat等函数。

*链表

链表节点的结构

```
typedef struct listNode{
    //前置节点
    struct listNode *prev;
    //后置节点
    struct listNode *next;
    //节点的值
    void *value;
}
```

list用来持有链表，操作更加方便一些

```
typedef struct list{
    //表头结点
    listNode *head;
    //表尾节点
    listNode *tail;
    //链表所包含的节点数量
    unsinged long len;
    //节点值赋值函数
    void *(*dup)(void *ptr);
    //节点值释放函数
    void (*free)(void *ptr);
    //节点值对比函数
    int (*match) (void *ptr,void *key);
}
```

*字典

- Redis的字典使用哈希表作为底层实现，一个哈希表里面可以有多个哈希表节点，而每个哈希表节点就保存了字典中的一个键值对。
- 字典的实现

```
typedef struct dict{
    //类型特定函数
    dictType *type;
    //私有数据
    void *privdata;
    //哈希表
    dictht ht[2];
    //rehash索引，当rehash不再进行的时候，值为-1
    int trehashidx;
}dict;
```

ht属性是一个包含两个项的数组，数组中的每一项都是dictht哈希表一般情况下，字典只使用ht[0]哈希表，h[1]的作用是用来扩容时的临时的哈希表。

- 哈希表

```
typedef struct dictht{
    //哈希表数组
    dictEntry **table;
    //哈希表大小
    unsigned long size;
    //哈希表掩码，用于扩容的时候sizemask=len-1;
    unsigned long sizemask;
    //该哈希表已经使用的节点数量
    unsigned long used;
}dicht;
```

- 哈希表节点

```
typedef struct dictEntry{
    //键
    void *key;
    //值
    union{
        void *val;
        uint64_t u64;
        int64_t s64;
    } v;
    //指向下个哈希表节点，形成链表，用来解决冲突
    struct dictEntry *next;
}dictEntry;
```

解决哈希冲突的方法是链地址法,即头插法。

- rehash操作
  - 为字典的ht[1]哈希表分配空间，这个哈希表的空间大小要取决于所执行的操作
    - 如果执行的是扩展操作，那么ht[1]的大小为第一个大于等于ht[0].used*2的2^n。
    - 如果执行的是收缩操作，那么ht[1]的大小为第一个大于等于ht[0].used的2^n。
  - 将保存在ht[0]中的所有键值对rehash到ht[1]上面。
  - 当ht[0]中的所有键值对都rehash到ht[1]上之后，释放ht[0],将ht[1]设置成ht[0],并在ht[1]处创建一个空白哈希表，为下一次rehash做准备。
- rehash的情况
  - 服务器目前没有执行BGSAVE命令或者BGREWRITEAOF命令，并且哈希表的负载因子大于等于1
  - 服务器目前正在执行BGSAVE命令或者BGREWRITEAOF命令，并且哈希表的负载因子大于等于5

    负载因子：哈希表已保存的节点数量/哈希表的大小(ht[0].size)
- 渐进式rehash
  - 上面方式的缺点：如果hash表中的数据过多的时候，那么庞大的数据量将可能会导致服务器在一段的时间内停止服务。
  - 详细步骤：

        - 为ht[1]分配空间，让字典同时持有ht[0]和ht[1]两个哈希表。
        - 在字典中维持一个索引计数器rehashidx，并将它的值设置为0，表示rehash工作正式开始。
        - 在rehash期间，每次对字典执行添加，删除，查找或者更新的时候，程序除了执行指定的操作外，还会顺带将ht[0]哈希表在rehashidx索引上的所有简直对hash到ht[1]中，当rehash工作完成之后，程序rehashidx属性值增1。
        - 随着字典操作的不断执行，最终在某个 时间点上，ht[0]中所有的键值对都rehash到了ht[1]中，知识将rehashidx的值设置成-1，表示rehash操作已经完成。
    - 哈希表操作
        - 由于在渐进式rahash中，字典会同时使用ht[0]和ht[1]两个hash表，所以字典的删除，查找，更新等操作都会在两个hash表上进行。比如查找，先到h[0]中查找，然后再到ht[1]中查找。对于增加键值对来说，只会在ht[1]中进行。

*整数集合(intset)

- 整数集合是集合键的底层实现之一，当一个集合只包含证书值元素并且元素的个数不多的时候，那么此时redis就会使用整数集合作为集合键的底层实现。
- **底层实现**

```
typedef struct intset{
    //编码方式
    uint32_t encoding;//encoding的值可能有
INTSET_ENC_INT16,INTSET_ENC_INT32,INTSET_ENC_INT64
    //集合包含的元素数量
    uint32_t length;
    //保存元素的数组
    int8_t contents[];//从小到大排列，不包含重复项
}
```

- **升级操作**

  当一个新的数添加到intset中时，如果该数的位数大于当前intset的encoding时，那么此时集合就要进行升级操作。升级操作有三步：
    - 根据新元素的类型，扩展整数集合底层数组的空间大小，并为新元素分配空间。
    - 将底层数组现有的所有元素都转换成与新元素相同的类型，并将类型转换后的元素放置到正确的位置，并且保证原来元素的有序性。
    - 将新元素添加到新的底层数组中。
    - eg：原来数据类型时16的，并且有三个，编号为1，2，3.当来了一个32位的数值时，需要从原来的48位扩增到32*4=128位，并且将96-127的位置为新元素留出空间，然后将64-95位这些位置分配给原来编号为3的数字，同理编号为2的数字占据着32-63位，编号为1的数字占据着0-31位，然后将新添加的数字放置到96-127位，并且将encoding的值设置成INTSET_ENC_INT32，length++，至此升级完成。
    - 好处：提升集合的灵活性，另一个就是节约内存。
        - 提升灵活性：我们可以随意的将一个数字添加到集合当中，因为它会自动升级。
        - 节约内存：当数字不超过一定的范围时，这些数字的类型都是一致的。只有添加了一个超过当前encoding的类型时，才会去选择升级。

*压缩列表(ziplist)

- 压缩列表时列表键和哈希键的底层实现之一。当一个列表键只包含少量的列表项，并且每个列表项都是小的整数值，要么就是较短的字符串；当哈希键包含的键值对少的时候，并且每个键值对要么是小的整数，要么时较短的字符串。此时redis就会采用压缩列表来实现列表键和哈希键。
- 作用：为了压缩内存而开发的，是由一系列特殊编码的连续内存块组成的顺序数据结构。
- 构成：
  - zlbytes(uint32_t)：记录整个压缩列表占用的内存字节数，在压缩列表进行内存重分配的时候，或者计算zlend的时候。
  - zltail(uint32_t)：记录压缩列表尾节点距离列表的起始节点有多少个字节，通过这个偏移量，程序无须遍历整个压缩列表就可以确定表尾节点的地址
  - zllen(uint16_t)：记录了压缩列表的节点的数量，当数量小于65535时，该变量就是当前节点数，当等于65535时，节点的真实数量就必须遍历整个压缩列表才可以知道。
  - entryX：代表的时压缩列表的节点信息，节点的长度由节点保存的内容决定。
  - zlend：特殊值，用来标记压缩列表的末端。是一个标志位。
- 压缩节点的构成
  - previous_entry_length

    以字节为单位，记录了压缩列表中前一个节点的长度。
    - 当前一个结点的长度小于254个字节时，previous_entry_length的值为1字节，前一节点的长度就保存到当前字节中。
    - 当前一个节点的长度大于254个字节的时候，previous_entry_length的值为5字节，首先属性额第一个字节会被设置成0xFE(254),之后的四个字节用来保存前一节点的长度。
    - 程序可以通过指针运算，根据当前节点的地址来计算出前一个节点的地址。
  - encoding

    节点ecoding属性记录了conent中所保存的数据的类型以及长度
  - content

    保存节点的值，节点值可以是一个字节数组或者整数，值的类型和长度由encoding决定。
- 连锁更新
  - 当一个节点的长度为与250-253的时候，比如说节点a，当a的前面节点的长度1的时候，并不会出现问题，但是当a的前面插入了一个节点，并且该节点的长度大于254，那么此时a的previous_entry_length的值就由1变成了5，那么此时a节点的长度也就大于了254，因此在a后面的节点的previous_entry_length也要做相应的变化，假如a节点的后面节点在previous_entry_length的值变化之后，其自身的字节长度大于254的话，那么就会出现和上面一样的情况，有可能一直往后更新，具有连锁反应。同理，删除压缩列表的extryX的时候也有可能初心这种情况。
  - 但是对于上面这种情况出现的可能性很小所以ziplistpush等命令的平均复杂度仅为O(N)，而并非O(N^2)

*对象

前面的数据结构并不是直接被用在redis中，而是基于这些数据结构创建了一个对象系统：字符串对象，列表对象，哈希对象，集合对象，有序集合对象。

- **TYPE命令**

```
SET msg "hello"
TYPE msg  ->  string
```

从上面可以看出TYPE命令返回的结果是数据库键对应的值对象的类型。

- **OBJECTENCODING**：查看数据库键的值对象的编码。

```
SET msg "hello"
OBJECTENCODING msg   输出：embstr（embatr编码的简单动态字符串）
SADD num 1 3 5
OBJECT ENCODING msg  输出：intset（整数类型）
```

- **字符串对象**
  - 当长度小于39时，采用的是embstr编码，大于39采用的是raw(简单动态字符串)。
  - raw和embstr的区别：
    - raw需要调用两次内存分配函数来分配redisObject(对象结构的底层实现：类型，编码，指向底层实现的指针)结构和sdshdr(SDS结构)结构，embstr只需要一次即可，并且这两个结构的时紧挨着
    - 回收时，raw需要调用两次内存释放函数，embstr只需要一次。
- **列表对象**

  列表对象的编码可以是ziplist(压缩列表)，linkedlist(链表)

  当列表对象保存的所偶字符串元素长度都小于64字节，并且列表对象保存的元素数量小于512个----》ziplist
- **哈希对象**

  哈希对象的编码可以是ziplist或者hashtable
- **集合对象**

  集合对象的编码可以时intset或者hashtable

  当集合对象保存的所有元素都是整数值，且集合对象保存的元素数量不超过512个，此时使用的时ziplist数据结构
- **有序集合对象**

  有序集合对象可以是ziplist或者skiplist(跳跃表和字典)