Redis的设计与实现

1. SDS简单动态字符串：

Sds是Redis自己实现的，用来代替普通的C字符串的一种数据结构，它的定义如下：

Struct sdshdr{

int len;

int free;

char buf[];

}

相比于普通的C字符串，SDS有如下优点：

（1）因为记录了len属性，SDS可以在常数复杂度下获取到字符串的长度，而普通的C字符串因为要从头开始数，知道\0，所以需要O(n)的时间。这样就确保了获取字符串长度，不会成为Redis的性能瓶颈。

（2）因为记录了free属性，可以杜绝缓冲区的溢出，而C字符串会有溢出的问题，比如使用strcat函数时，如果没有分配足够的内存，不会报错，而是会覆盖掉原本的数据以外的东西，会有隐患。

（3）SDS的char buf[]数组，采用空间预分配与惰性释放的策略，预留一部分空余的空间，可以减少修改字符串时带来的内存重新分配，而修改字符串对于Redis这种数据库来说可以说是非常频繁地，这样整体性能就得到了优化。

（4）由于SDS的成员buf也是以\0结尾的C风格字符串，所以他也可以直接调用一部分C字符串的函数。

2. 链表

链表是Redis中的一种数据结构，他可以作为最上层的列表键、哈希键等的底层实现。List类的结构如下所示：

typedef struct list{

// 链表头节点

listNode \*head;

// 链表尾节点

listNode \*tail;

// 链表节点数目

unsigned long len;

// 节点值复制函数

void \*(\*dup)(void \*ptr);

// 节点值释放函数

void (\*free)(void \*ptr);

}

其中listNode是链表节点的类，含有双向指针，以及节点存储的值。

链表节点的value使用void\*进行保存，可以存储不同类型的数据，再赋予不同的复制函数与释放函数，起到多态的作用。

3. 字典

字典与链表相同，也是Redis底层的数据结构之一，Redis为用户提供的数据库与哈希键底层有时就是用字典来实现的。字典是使用哈希表来实现的。

哈希表的定义如下所示：

typedef struct dictht{

// 哈希表数组

dictEntry \*\*table;

// 哈希表大小

unsigned long size;

// 哈希表大小掩码，用于计算索引值

unsigned long sizemask;

// 哈希表已有节点数量

unsigned long used;

}

table作为具体的哈希表，是一个数组，数组的每个位置存放着一个entry的指针。

table使用链表来解决冲突问题，每个entry会存放指向下一个entry的指针。

字典类的定义如下所示：

typedef struct dict{

//记录一系列类型特定的函数，实现多态

dictType \*type;

// 私有数据

void \*privdata

// 哈希表

dictht ht[2];

// rehash进展情况

Int trehashidx;

} dict;

从定义上可以看到，dict类保存了两个哈希表，这是在rehash时需要用到的，当哈希表的负载因子太大或者太小时，就需要通过rehash的操作，使得哈希表更加平衡。

\*字典的rehash采用渐进式的策略，以防整个哈希表太大，rehash的执行导致整个redis停止服务。在rehash时，用户每进行一次put,get操作，都会把某个索引位上进行rehash，然后索引+1,直到做完整个哈希表的rehash为止。