第一部分 数据结构与对象

第 2 章 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
3.SDS 增加空间的规则：
a.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
b.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

第三章 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

第四章 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;
```

3.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

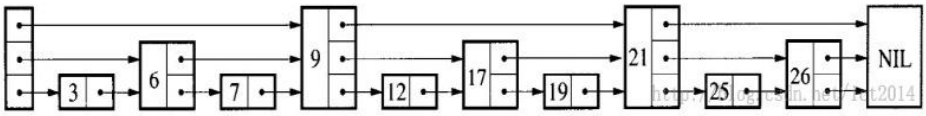
第五章 跳跃表

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;
```

4.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。

第六章 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;
```

3.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）
4.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

第七章 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

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（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

第 8 章 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
3.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
4.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
5.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
6.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
7.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

第九章 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
    ...
};
```

2.Redis 客户端实现

```
struct redisClient{
    ...
    redisDb* db;//记录当前正在使用的数据库
    ...
};
```

```
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
    ...
```

第十章 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
3.RDB 文件结构
所有的数据库 value 的对象，如字符串对象，列表对象，哈希表对象，集合对象等在实际持久化存储到 RDB 文件前都先转为字符串对象，统一的格式存储，基本上都是 长度+转换后的字符串对象

整体 RDB 文件

REDIS（5 字节）：即保存着“REDIS”五个字符，可以帮助程序在载入文件时，判断该文件是否是 RDB 文件
db_version（4 字节）:字符串表示的整数，标识版本号
databases:包含零个或任意个数据库，以及数据库中的键值对数据
EOF（1 字节）：特殊常量，标识 RDB 正文部分的结束
check_sum（8 字节）:校验和

databases 部分：

SELECTDB（1 字节）：标识后面的数据时一个数据库号码
db_number（1 字节，2 字节，5 字节）:保存一个数据库号码，由于数据库号码的大小不一样，可能导致这个字段长度不一样
key_value_pairs:保存了当前数据库的所有键值对

key_value_pairs 部分：
不带过期键：

TYPE（1 字节）:标识 value 的类型，如 REDIS_RDB_TYPE_STRING,REDIS_RDB_TYPE_LIST 等
key:字符串对象
value:不同类型的对象

带过期键

EXPIRETIME_MS（1 字节）：常量，标识后面的数据是一个毫秒为单位的过期时间
ms:过期时间
后面三个字段的含义同上

value 的不同编码

a.字符串对象

（1）字符串对象编码为 REDIS_ENCODING_INT

ENCODING:可以为 REDIS_RDB_ENC_INT8,REDIS_RDB_ENC_INT16,REDIS_RDB_ENC_INT32

（2）字符串对象编码为 REDIS_ENCODING_RAW

若长度小于 20 字节，原样保存

若长度大于 20 字节，压缩字符串

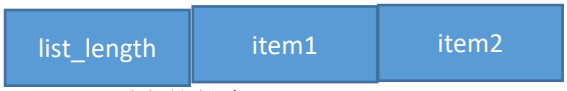
REDIS_RDB_ENC_LZF：标识字符串已经被 LZF 算法压缩了

compressed_len：压缩后长度

origin_len：原始长度

compressed_string:压缩后的字符串

b.列表对象



list_length:列表的长度

item1:字符串对象方式存储

c.集合对象

set_size:集合大小

elem1:字符串对象方式存储

d.哈希表对象

hash_size:键值对个数

key,value:字符串对象方式存储

e.有序集合对象

sorted_set_size:元素格式

member,score:字符串对象，本来不是字符串的话需要进行转化

第十一章 AOF 持久化

1.与 RDB 持久化保存数据库中的键值对来记录数据库状态不同，AOF 通过保存 Redis 服务器执行的写命令来记录数据库状态
2.AOF 持久化功能分为：命令追加，文件写入，文件同步
命令追加：这时候 Redis 执行过的写命令写入到 aof_buf（内存中的 AOF 缓冲区中）
文件写入和文件同步：由于调用 write 函数时，操作系统不会及时的写入到磁盘中，会做一部分的缓存，文件同步即是 flush 函数
3.AOF 重写：由于上述的 AOF 持久化过程会产生大量多余的命令，而 AOF 重写就是解决这个问题，它是通过分析当时 Redis 内存的键值对的状态，然后生成对应的命令。如当时一个键值对为：animals ---->{“Dog”,”Cat”,”Panda”,”Tiger”}，它会生成如下命令写入到 AOF 文件中：SADD animals “Dog”,”Cat”,”Panda”,”Tiger”;这样就忽略了这个键值对导致中间是由哪些命令生成的。ps:如果这个键值对太大，就会分成多个命令来写入

第十二章 事件

1.两类事件：文件事件（套接字），时间事件（定时操作）
2.对于文件事件，Redis 实现了一个文件事件处理机，主要分为两个部分，a.I/o 多路复用程序（调用系统的多路复用的库，如 epool,select 等）；b.文件事件分派器（将不同类型的请求发往不同的处理函数，如连接，写入，读取，关闭等）
3.服务器将所有的时间事件都放在一个无序链表中，每当时间事件执行时，它就遍历整个链表，查找所有已到达的时间事件，并调用相应的事件处理器

第十三章 客户端

1.Redis 服务器用一个 client 链表记录了所有和 Redis 服务器连接的客户端

```
struct redisServer{
    //...
    list* clients;
    //...
};
```

2.客户端属性

```
typedef struct redisClient{
    int fd;//套接字描述符
    robi* name;//客户端名字
    int flags;//标志，记录客户端的角色和状态
    sds querybuf;//输入缓冲区
    robj** argv;//命令参数
    int argc;//命令中参数个数
    struct redisCommand* cmd;//命令的实现函数（通过查找保存所有命令对应的函数的字典可得）

    //输出缓冲区固定大小,保存长度较小的回复，确认回复，错误回复
    char buf[REDIS_REPLY_CHUNK_BYTES];
    int bufpos;

    //输出缓冲区变长
    list* reply;

    int authenticated;//身份认证
}redisClient;
```

第三部分 多机数据库的实现

第 15 章 复制

1.旧版复制功能的缺陷：在主从服务器断线后重连，主服务器会将 Redis 的 RDB 文件发给从服务器，发送了一些不必要的信息
所以新版在这种情况下主服务器只会把断线后主服务器执行的命令发给从服务器，使得主从服务器达到一致

第 16 章 Sentinel

第 17 章 集群

1.Redis 集群的核心在于：总共集群中有 16384 个槽，没有中心节点，每个“小集群”中都保存了这 16384 个槽的分配信息，然后集群中每个“小集群”都可以申请分配一部分的槽，只有 16384 个槽都分配完之后，集群才算 health_ok，“小集群”的目的是用来提供容错功能（副本），一般为三个节点构成，有一个主节点，“小集群”中每个节点的数据都是一样的。
往 Redis 集群中存入一个键值对，它是通过计算 key 的 CRC-16 的校验和 & 16383 即（CRC16()key & 16383）计算该键值对应该存在哪个槽中，之后再通过查找找到这个槽是由哪个“小集群”管理的，最后写入“小集群”。