2017/7/20 Redis设计与实现

底层数据结构：sds、list、dict、skiplist、intset、ziplist

1.String:

Redis构建了简单动态字符串SDS来作为默认字符串表示，属于可修改字符串的值。

当一些如打印日志等不需被修改的字符串则用C语言传统字符串表示。

sds用于存储字符串、AOF缓冲区、客户端状态中的输入缓冲区等。

sds实际是char型指针，即C语言的字符串表述形式

sdshdr是redis中的简单动态字符串结构，而实际上在使用字符串时，

依旧是使用char* 而不是sdshdr，在C中可根据地址偏移，

得到该char* （sds）所在的sdshdr的地址，借用指针进行操作。

sds定义：

```
struct sdshdr{
    int len;//记录buf数组中已使用字节长度，等于SDS保存的字符串长度
    int free;//记录buf数组中未使用的字节长度
    char buf[];字符数组，用于保存字符串
}
```

buf[]保存字符，最后一个字节保存空字符'\0'结尾，这1字节空间不计算在len属性中。

遵循空字符结尾惯例，可对C字符串函数库中进行一些重用。

```
/* 根据给定的初始化字符串 init 和字符串长度 initlen
 * 创建一个新的 sds
 * 参数
 *  init : 初始化字符串指针
 *  initlen : 初始化字符串的长度
 * 返回值
 *  sds : 创建成功返回 sdshdr 相对应的 sds
 *        创建失败返回 NULL
 * 复杂度
 *  T = O(N) */
sds sdsnewlen(const void *init, size_t initlen) {
    struct sdshdr *sh;
    // 根据是否有初始化内容，选择适当的内存分配方式
    // T = O(N)
    if (init) {
        // zmalloc 不初始化所分配的内存
        sh = zmalloc(sizeof(struct sdshdr)+initlen+1);
    } else {
        // zcalloc 将分配的内存全部初始化为 0
        sh = zcalloc(sizeof(struct sdshdr)+initlen+1);
    }
    // 内存分配失败，返回
    if (sh == NULL) return NULL;
    // 设置初始化长度
    sh->len = initlen;
    // 新 sds 不预留任何空间
    sh->free = 0;
    // 如果有指定初始化内容，将它们复制到 sdshdr 的 buf 中
    // T = O(N)
    if (initlen && init)
        memcpy(sh->buf, init, initlen);
```

```
            // 以 \0 结尾
            sh->buf[initlen] = '\0';
            // 返回 buf 部分，而不是整个 sdshdr
            return (char*)sh->buf;
        }
```

SDS与C字符串相比：

①SDS结构的len属性记录了字符串长度，当要获取时，复杂度仅为O(1)，无需进行O(n)的遍历；

②杜绝缓冲区溢出：在对SDS字符串进行修改时，会检查SDS剩余空间(free属性)是否充足，
若不足则先进行扩展。

③减少修改字符串时的内存重分配次数

④可保存二进制数据

⑤兼容部分C字符串函数

2.链表：

Redis链表结构(adlist.h/listNode)

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

对listNode进行一层包装(adlist/list)

```
typedef struct list{
    listNode *head//表头节点
    listNode *tail;//表尾结点
    unsigned long len;//链表包含的节点数量
    void *(*dup)(void *ptr);//节点值复制函数
    void (*free)(void *ptr);//节点值释放函数
    int (*match)(void *ptr,void *key);//节点值对比函数
}list;
```

```
|list      listNode   <-- listNode   <--listNode
|head  -->  value=.. -->  value=..  -->  value=..  -->null
|tail  ------------------------------------↑
|len=3
|dup    -->.....        list结构中的3个listNode
|free   -->....
|match  -->...
```

Redis链表特性：

双端：链表节点有prev和next指针

无环：表头节点的prev和表尾节点的next指向null 不循环

带头指针和尾指针：list结构的head指针和tail指针

计数器:list结构的len属性保存节点个数

多态：链表节点使用void*指针保存节点值，
可通过list结构的dup、free、match属性为节点值设置类型特定函数
因此链表可保存各种不同类型的值。

（①void指针可以指向任意类型的数据，亦即可用任意数据类型的指针对void指针赋值
②可以用void指针来作为函数形参，就可以接受任意数据类型的指针作为参数）

3.字典(map映射，用于保存键值对 key-value)

Redis哈希表结构(dict.h/dictht)

```
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(=size-1)
    unsigned long used;//已有节点数量
}dictht;
```

table数组中每个元素指向dict.h/dictEntry结构的指针。
每个dictEntry结构保存一个键值对。size属性记录哈希表大小(table数组大小)
哈希表节点dictEntry

```
typedef struct dictEntry{
    void *key;//键
    union{ //值
        void *val;
        uint64_tu64;
        int64_ts64;
    }v;
    struct dictEntry *next;//下个哈希表节点
}dictEntry;
```

key属性保存键值对中的键，v属性保存值，值可以是一个指针、uint64_t整数或int64_t整数。
next属性指向另一个哈希表节点指针，可以将多个哈希值相同的键值对连接起来，解决键冲突问题。
例：

```
|dictht                                  两个索引值相同的键k1 k0 通过dictEntry结构的next指针连接起来
|table  ---->   dictEntry*[4]
|size=4         |0          -->null
|sizemask=3     |1          -->null
|used=2         |2          -->null
                |3          --> dictEntry  --> dictEntry -->null
                                  |k1 |v1        |k0 |v0
```

dict.h/dict结构表示字典(在dictht上再包装一层)

```
typedef struct dict{
    dictType *type;//类型特定函数
    void *privdata;//私有数据
    dictht ht[2];//哈希表
    int trehashidx;//rehash索引，当rehash不在进行时，为-1
}dict;
```

type属性和privdata属性针对不同类型的键值对，
type指向dictType结构指针，每个dictType结构保存了一簇特定类型键值对的操作函数
privdata属性保存了需要传给特定函数的可选参数

```
typedef struct dictType{
    unsigned int (*hashFunction)(const void *key);
    void *(*keyDup) (void *privdata,const void *key);
    void *(*valDup) (void *privdata,const void *key);
    int (*keyCompare)(void *privdata,const void *key1,const void *key2);
    void *(*keyDestructor) (void *privdata, void *key);
```

```
            void *(*valDestructor) (void *privdata,void *obj);
        }dictType;
```

ht属性是包含两个项(dictht)的数组，一般只使用ht[0]哈希表,当对ht[0]进行rehash时才使用ht[1]
rehashidx属性记录rehash,若当前没有在进行，则值为-1。

例：普通状态下（没有rehash）的字典 （图4-3普通状态下的字典）

4.哈希算法：

当将一个新键值对加入到字典中时，先计算键的哈希值和索引值(哈希值对sizemask取模)，
再根据索引值将新键值节点放入到dictht的table数组中合适的dictEntry链表中。

```
hash=dict->type->hashFunction(key);
index=hash& dict->ht[x].sizemask;
```

例：添加一个新键值对的过程图（图4-5添加新键值对）

键冲突问题:有两个或以上数量的键分配到同一个索引上时
开放地址法(再散列，直到索引不冲突)：反复计算索引，并要求有足够的索引能用来存储。
链地址法：当索引冲突时，在该索引下以链表的方式存储
Redis中dictEntry节点组成的链表没有指向尾部的指针，因此采用头插法，将新节点添加到链表表头O(1)

5.rehash:当哈希表保存的键值对逐渐增多或减少时，为了维持合理的负载因子，对哈希表大小进行相应的扩展或收缩

步骤如下：

1)为字典ht[1]哈希表分配空间，
  若是扩展操作，则ht[1]的大小为第一个大于等于ht[0].used*2 的2^n (即size=2^n>=ht[0].used*2)
  若是收缩操作，则ht[1]的大小为第一个大于等于ht[0].used的2^n (即size=2^n>=ht[0].used)
2)将保存在ht[0]中的所有键值对重新计算散列到ht[1]中
3)完成上述rehash操作后，释放ht[0],更换ht[1]为ht[0],创建一个新的空ht[1]为下次rehash使用
    (即 free(ht[0]),*ht[0]=*ht[1],ht[1]=new dictht)

过程图（图4-8rehash扩展）

```
int dictRehash(dict *d, int n) {
    // 只可以在 rehash 进行中时执行
    if (!dictIsRehashing(d)) return 0;
    // 进行 N 步迁移
    // T = O(N)
    while(n--) {
        dictEntry *de, *nextde;
        /* Check if we already rehashed the whole table... */
        // 如果 0 号哈希表为空，那么表示 rehash 执行完毕
        // T = O(1)
        if (d->ht[0].used == 0) {
            // 释放 0 号哈希表
            zfree(d->ht[0].table);
            // 将原来的 1 号哈希表设置为新的 0 号哈希表
            d->ht[0] = d->ht[1];
            // 重置旧的 1 号哈希表
            _dictReset(&d->ht[1]);
            // 关闭 rehash 标识
```

```
                    d->rehashidx = -1;
                    // 返回 0 ，向调用者表示 rehash 已经完成
                    return 0;
                }
                /* Note that rehashidx can't overflow as we are sure there are more
                 * elements because ht[0].used != 0 */
                // 确保 rehashidx 没有越界
                assert(d->ht[0].size > (unsigned)d->rehashidx);

                // 略过数组中为空的索引，找到下一个非空索引
                while(d->ht[0].table[d->rehashidx] == NULL) d->rehashidx++;

                // 指向该索引的链表表头节点
                de = d->ht[0].table[d->rehashidx];
                /* Move all the keys in this bucket from the old to the new hash HT */
                // 将链表中的所有节点迁移到新哈希表
                // T = O(1)
                while(de) {
                    unsigned int h;
                    // 保存下个节点的指针
                    nextde = de->next;
                    /* Get the index in the new hash table */
                    // 计算新哈希表的哈希值，以及节点插入的索引位置
                    h = dictHashKey(d, de->key) & d->ht[1].sizemask;
                    // 插入节点到新哈希表
                    de->next = d->ht[1].table[h];
                    d->ht[1].table[h] = de;
                    // 更新计数器
                    d->ht[0].used--;
                    d->ht[1].used++;
                    // 继续处理下个节点
                    de = nextde;
                }
                // 将刚迁移完的哈希表索引的指针设为空
                d->ht[0].table[d->rehashidx] = NULL;
                // 更新 rehash 索引
                d->rehashidx++;
            }
            return 1;
        }
```

哈希表扩展与收缩条件：(以下条件满足一个即可)

负载因子=哈希表已保存节点数量/哈希表大小

load_factor=ht[0].used/ht[0].size;

扩展：

1)服务器目前没有在执行BGSAVE/BGREWRITEAOF命令，且哈希表负载因子>=1

2)服务器目前正在执行BGSAVE/BGREWRITEAOF命令，但哈希表负载因子>=5

收缩：

当负载因子<0.1时，执行收缩操作。

6.渐进式rehash：当哈希表中的键值对比较多时，如果采用集中式一次性完成rehash会造成一定的影响

为了避免对服务器性能造成影响，采用多次渐进式地将ht[0]里的键值对rehash到ht[1]

步骤如下：

1)为ht[1]分配空间，让字典同时持有ht[0]、ht[1]

2)维持索引计数器变量rehashidx,设置为0，表示rehash开始。

3)rehash期间，对字典进行正常操作的同时，会顺带将ht[0]上rehashidx索引上的键值对rehash到ht[1],完成后rehashidx+1

4)随着字典操作的不断进行，最终使ht[0]上所有键值对rehash到ht[1]上，修改rehashidx值为-1，过程结束。

```
// 在给定毫秒数内，以 100 步为单位，对字典进行 rehash 。
int dictRehashMilliseconds(dict *d, int ms) {
    // 记录开始时间
    long long start = timeInMilliseconds();
    int rehashes = 0;
    while(dictRehash(d,100)) {
        rehashes += 100;
        // 如果时间已过，跳出
        if (timeInMilliseconds()-start > ms) break;
    }
    return rehashes;
}
```

注意：rehashidx的值范围为[-1,ht[0].sizemask]

在渐进式rehash期间，对字典的操作会在ht[0]中先查找对应键，没有命中则在ht[1]中查找，

对于新增的键，会一律存在ht[1]中，使ht[0]逐渐变成空表。

7.跳跃表：有序数据结构，在每个节点中维持多个指向其他节点的指针，达到快速访问节点的目的。

有序链表中，节点具有多个指向，可加快搜索，复杂度O(logn)

```
level 3   -INF-----------21↓-----------------55↓
level 2   -INF--2↓-------21↓-------37↓-------55↓
level 1   -INF-->2-->17-->21-->33-->37-->46-->55
```

Redis跳跃表由redis.h/zskiplistNode和redis.h/zskiplist结构定义

zskiplistNode表示跳跃表节点， zskiplist表示关于节点的相关信息，如节点数量、头尾指针等

(图5-1跳跃表)

```
typedef struct zskiplistNode{
    struct zskiplistLevel{ //层
        struct zskiplistNode *forward; //前进指针
        unsigned int span; //跨度
    }level[];

    struct zskiplistNode *backward;//后退指针
    double score;//分值
    robj *obj;//成员对象
}zskiplistNode;
```

1)层：跳跃表节点的level数组可包含多个元素，每个元素包含指向其他节点的指针。

level[i].forward代表 本节点在第i层中的指向的下一个节点

每次创建一个新跳跃表节点时，根据幂次定律随机生成一个介于1和32之间的值作为level数组大小，即高度

2)跨度：level[i].span属性，记录两个节点之间的距离。跨度用于计算目标节点在跳跃表中的排位。(将沿途访问过的所有层跨度累加)

3)后退指针：用于从尾部逆向访问至表头，每次仅后退一个节点。

4)分值和成员：跳跃表中节点按分值从小到大排序，obj成员指向一个字符串对象，保存SDS值

(同一个跳跃表中各节点的成员对象是唯一的，但分值可以重复)

使用zskiplist结构维持跳跃表，快速访问表头、表尾节点，获取节点数量等信息。

```
typedef struct zskiplist{
    struct zskiplistNode *header,*tail;//表头尾节点
    unsigned long length;//表中节点数量
    int level;//表中最高层数
}zskiplist;

/* 创建一个层数为 level 的跳跃表节点，
 * 并将节点的成员对象设置为 obj ，分值设置为 score 。
 * 返回值为新创建的跳跃表节点 */
zskiplistNode *zslCreateNode(int level, double score, robj *obj) {
    // 分配空间
    zskiplistNode *zn = zmalloc(sizeof(*zn)+level*sizeof(struct zskiplistLevel));
    // 设置属性
    zn->score = score;
    zn->obj = obj;
    return zn;
}

/* 创建并返回一个新的跳跃表,ZSKIPLIST_MAXLEVEL=32 */
zskiplist *zslCreate(void) {
    int j;
    zskiplist *zsl;
    // 分配空间
    zsl = zmalloc(sizeof(*zsl));
    // 设置高度和起始层数
    zsl->level = 1;
    zsl->length = 0;
    // 初始化表头节点
    // T = O(1)
    zsl->header = zslCreateNode(ZSKIPLIST_MAXLEVEL,0,NULL);
    for (j = 0; j < ZSKIPLIST_MAXLEVEL; j++) {
        zsl->header->level[j].forward = NULL;
        zsl->header->level[j].span = 0;
    }
    zsl->header->backward = NULL;
    // 设置表尾
    zsl->tail = NULL;
    return zsl;
}
```

由于跳跃表的第一层`level[0]`是简单顺序链表形式保存所有节点关系的。
因此在需要释放表时遍历`level[0]`依次释放即可。

```
    void zslFree(zskiplist *zsl) {
        zskiplistNode *node = zsl->header->level[0].forward, *next;
        // 释放表头
        zfree(zsl->header);
        // 释放表中所有节点
        // T = O(N)
        while(node) {
```

```
                next = node->level[0].forward;
                zslFreeNode(node);
                node = next;
            }
            // 释放跳跃表结构
            zfree(zsl);
        }

         e.HyperLogLog: hyperloglog.c 中的 hllhdr

        struct hllhdr {
            char magic[4];      /* "HYLL" */
            uint8_t encoding;   /* HLL_DENSE or HLL_SPARSE. */
            uint8_t notused[3]; /* Reserved for future use, must be zero. */
            uint8_t card[8];    /* Cached cardinality, little endian. */
            uint8_t registers[]; /* Data bytes. */
        };

    7.5 HyperLogLog: 可以接受多个元素作为输入，并给出输入元素的基数估算值：
          基数：集合中不同元素的数量。比如 {'apple', 'banana', 'cherry', 'banana', 'apple'} 的基数就是 3 。
          估算值：算法给出的基数并不是精确的，可能会比实际稍微多一些或者稍微少一些，但会控制在合
        理的范围之内。
        HyperLogLog 的优点是，即使输入元素的数量或者体积非常非常大，计算基数所需的空间总是固定
        的、并且是很小的。
        在 Redis 里面，每个 HyperLogLog 键只需要花费 12 KB 内存，就可以计算接近 2^64 个不同元素的基
        数。这和计算基数时，元素越多耗费内存就越多的集合形成鲜明对比。
        但是，因为 HyperLogLog 只会根据输入元素来计算基数，而不会储存输入元素本身，所以
        HyperLogLog 不能像集合那样，返回输入的各个元素。

        redis> PFADD str1 "apple" "banana" "cherry"
        (integer) 1
        redis> PFCOUNT str1
        (integer) 3
        redis> PFADD str2 "apple" "cherry" "durian" "mongo"
        (integer) 1
        redis> PFCOUNT str2
        (integer) 4
        redis> PFMERGE str1&2 str1 str2
        OK
        redis> PFCOUNT str1&2
        (integer) 5

    8.整数集合：用于保存整数值的集合抽象数据结构，保存int16_t、int32_t、int64_t，无重复元素。
        intset.h/intset结构：
            typedef struct intset {
                uint32_t encoding;//编码方式
                uint32_t length;//元素数量
                int8_t contents[];//元素数组
            }intset;
        contents数组中元素按从小到大排列，不含重复项。
```

```
        contents元素类型取决于encoding

        |intset
        |encoding=INT16
        |length=5
        |contents  --> |-5|18|89|252|14632|
        sizeof(int16_t)*5=80位空间大小

        当新添加的整数类型比原集合编码类型要大时，则对集合进行升级更新，将数组内元素都变为较大的类型并调整内存空间位
        如，当原集合类型为INT_16，新增一个INT_64时，则将原元素都更改为INT_64 调整集合空间大小.

        升级：更改编码并修改原底层数组中元素值的地址，改为新编码方式赋予
              详见 源码intset.c/intsetUpgradeAndAdd 函数
        例：  原先contents[0] 为INT_16编码存储的整数5 ，地址范围为 0X....a - 0X....b
            当contents编码升级为INT_32时，对于整数5的地址可用空间变大了0X...a -0X...c
            所以需要对整数5以INT_32编码形式重新赋予contents[0],覆盖整个可用空间地址。

            // 根据集合原来的编码方式，从底层数组中取出集合元素
            // 然后再将元素以新编码的方式添加到集合中
            // 当完成了这个步骤之后，集合中所有原有的元素就完成了从旧编码到新编码的转换
            // 因为新分配的空间都放在数组的后端，所以程序先从后端向前端移动元素
            // 举个例子，假设原来有 curenc 编码的三个元素，它们在数组中排列如下：
            // | x | y | z |
            // 当程序对数组进行重分配之后，数组就被扩容了（符号 ？ 表示未使用的内存）：
            // | x | y | z | ? |   ?   |   ?   |
            // 这时程序从数组后端开始，重新插入元素：
            // | x | y | z | ? |   z   |   ?   |
            // | x | y |   y   |   z   |   ?   |
            // |   x   |   y   |   z   |   ?   |
            // 最后，程序可以将新元素添加到最后 ？ 号标示的位置中：
            // |   x   |   y   |   z   |  new  |
            // 上面演示的是新元素比原来的所有元素都大的情况，也即是 prepend == 0
            // 当新元素比原来的所有元素都小时（prepend == 1），调整的过程如下：
            // | x | y | z | ? |   ?   |   ?   |
            // | x | y | z | ? |   ?   |   z   |
            // | x | y | z | ? |   y   |   z   |
            // | x | y |   x   |   y   |   z   |
            // 当添加新值时，原本的 | x | y | 的数据将被新值代替
            // |  new  |   x   |   y   |   z   |
            while(length--)
                _intsetSet(is,length+prepend,_intsetGetEncoded(is,length,curenc));
            //_intsetGetEncoded返回以旧编码获得的length位置上的整数值value，
            //_intsetSet将value以新编码放到contents数组的正确位置上。

    9.压缩列表：ziplist,列表键和哈希键的底层实现之一。
                压缩列表采取的方式犹如单向链表，比双端链表要节省空间，
                在节点方面，与双端链表相比，数据结构更加简单。
                但双端链表结构记录更详细信息，对于复杂情况更加快速。
        例：
            redis> RPUSH lst 1 3 5 12345 "hello" "good"
```

```
    (integer)6
    redis> OBJECT ENCODING lst
    "ziplist"
```

列表键lst中包含的是较小的整数值及短字符串。
当一个哈希键中只包含少量键值对，且键值是小整数值或短字符串，也会使用压缩列表
例：

```
    redis> HMSET profile "name" "Jack" "age" 28 "job" "Programmer"
    OK
    redis> OBJECT ENCODING profile
    "ziplist"
```

压缩列表由一系列特殊编码的连续内存块组成，顺序型数据结构。
一个压缩列表可包含任意多个结点entry,每个节点可保存一个字节数组或整数值。

压缩列表组成部分：

```
    |zlbytes|zltail|zllen|entry1|entry2|...|entryN|zlend|
```

```
zlbytes: uint32_t 记录整个压缩列表占用的内存字节数
zltail : uint32_t 记录压缩列表表尾节点距离起始地点的偏移量。
zllen  : uint16_t 记录压缩列表包含的节点数量(数量大于uint16_t_MAX时需要遍历计算)
entryX :          列表节点
zlend  : uint8_t  标记末端。
```

压缩列表节点组成部分：

```
    |previous_entry_length|encoding|content|
```

节点可保存一个字节数组或一个整数值

字节数组：

1)长度小于等于63（2^6 -1）字节的字节数组
2)长度小于等于16383（2^14 -1）字节的字节数组
3)长度小于等于4294967295（2^32 -1）字节的字节数组

整数值：

1)4位长，介于0-12的无符号整数
2)1字节长的有符号整数
3)3字节长的有符号整数
4)int16_t类型整数
5)int32_t类型整数
6)int64_t类型整数

|previous_entry_length|:1或5字节 记录压缩列表中前一个节点的长度
|encoding|:记录节点content属性所保存数据的类型及长度
|content|: 保存节点的值，值的类型和长度由encoding属性决定

```
    /* 保存 ziplist 节点信息的结构  */
        typedef struct zlentry {
            // prevrawlen : 前置节点的长度
            // prevrawlensize : 编码 prevrawlen 所需的字节大小 用来计算节点的编码
            unsigned int prevrawlensize, prevrawlen;
            // lensize : 编码 len 所需的字节大小
            unsigned int lensize, len;
```

```
                // 当前节点 header 的大小
                // 等于 prevrawlensize + lensize
                unsigned int headersize;
                // 当前节点值所使用的编码类型
                unsigned char encoding;
                // 指向当前节点的指针
                unsigned char *p;
            } zlentry;  // len : 当前节点值的长度

        压缩列表小结{
            是一种为节约内存开发的顺序型数据结构
            用作列表键和哈希键的底层实现之一
            可包含多个节点，每个节点保存一个字节数组或整数值
            添加新节点或删除节点，可能引发连锁更新操作，不过出现的几率不高
        }

10. 对象：Redis对象系统结构  string（字符串），hash（哈希），list（列表），set（集合）及zset（有序集合）。
      五种类型对象的构建用到了以上的主要数据结构：简单动态字符串、双端链表、字典、压缩列表、整数集合。

    Redis中的键值对，键总是一个字符串对象，值可以是五种对象中的一种。
        "字符串键"指key-value 中的value是字符串对象类型
        "列表键"指key-value 中的value是列表对象类型
        typedef struct redisObject {
            // 对象类型(五种对象类型)
            unsigned type:4;
            // 底层数据结构编码
            unsigned encoding:4;
            // 对象最后一次被访问的时间
            unsigned lru:REDIS_LRU_BITS; /* lru time (relative to server.lruclock) */
            // 引用计数
            int refcount;
            // 指向底层实现数据结构的指针
            void *ptr;
        } robj;
        对象：            对象type属性值：      TYPE命令输出：
        字符串对象          REDIS_STRING       "string"
        列表对象            REDIS_LIST         "list"
        哈希对象            REDIS_HASH         "hash"
        集合对象            REDIS_SET          "set"
        有序集合对象        REDIS_ZSET         "zset"
    对象的ptr指针指向对象的底层实现数据结构，由对象的encoding属性决定。
        编码常量：                        底层数据结构：
        REDIS_ENCODING_INT              long类型整数
        REDIS_ENCODING_EMBSTR           embstr编码的简单动态字符串   (长度<=32字节)
        REDIS_ENCODING_RAM              简单动态字符串
        REDIS_ENCODING_HT               字典
        REDIS_ENCODING_LINKEDLIST       双端链表
        REDIS_ENCODING_ZIPLIST          压缩列表
        REDIS_ENCODING_INTSET           整数集合
        REDIS_ENCODING_SKIPLIST         跳跃表和字典
```

```
        /*默认以RAW字符编码，若需要调整则在调用处显式更改 */
        robj *createObject(int type, void *ptr) {
            robj *o = zmalloc(sizeof(*o));
            o->type = type;
            o->encoding = REDIS_ENCODING_RAW;
            o->ptr = ptr;
            o->refcount = 1;
            /* Set the LRU to the current lruclock (minutes resolution). */
            o->lru = LRU_CLOCK();
            return o;
        }
        /* 创建一个 ZIPLIST 编码的列表对象  */
        robj *createZiplistObject(void) {
            unsigned char *zl = ziplistNew();
            robj *o = createObject(REDIS_LIST,zl);
            o->encoding = REDIS_ENCODING_ZIPLIST; //显式更改编码
            return o;
        }
```

11. 字符串对象：
    字符串对象的编码可以是int embstr raw
    根据不同的操作，会将值的编码进行转换。
    字符串命令和对应编码操作：(具体命令及方式见图8-7字符串命令及操作)
    命令：

SET	以对应编码保存值
GET	以字符串形式返回值
APPEND	以字符串形式追加值
INCRBYFLOAT	将值转换成浮点数进行计算保存
INCRBY	对整数进行加法计算保存
DECRBY	对整数进行减法计算保存
STRLEN	返回对应值的字符串长度
SETRANGE	
GETRANGE	获取对应值的字符串形式，及其索引字符

12.列表对象：
    列表对象的编码可以是ziplist linkedlist

```
redis>RPUSH members 1 "three" 5
(integer)3
```

    1)若为ziplist编码：

```
|redisObject
|type=REDIS_LIST
|encoding=REDIS_ENCODING_ZIPLIST
|ptr        ---->|zlbytes|zltail|zllen|1|"three"|5|zlend|
|...
```

    2)若为linkedlist编码：
    双端链表中每个节点是一个StringObject,每个StringObject的编码根据对应的值决定

```
|redisObject
|type=REDIS_LIST
|encoding=REDIS_ENCODING_LINKEDLIST
|ptr        ---->|StringObject:1|StringObject:"three"|StringObject:5|
```

```
        |...
    列表对象采用ziplist编码的条件：(两个都需要满足)
    ①列表对象保存的所有字符串元素长度都小于64字节
    ②列表对象保存的元素数量小于512个
    若不能满足以上两个条件，则采用linkedlist编码。

    列表命令及相应编码操作：(具体见图8-8列表命令及操作)
        LPUSH
        RPUSH
        LPOP
        RPOP
        LINDEX
        LLEN
        LINSERT
        LREM
        LTRIM
        LSET
13.哈希对象：
    哈希对象编码可以是ziplist hashtable
        redis> HSET profile name "tom"
        (integer)1
        redis> HSET profile age 25
        (integer)1
        redis> HSET profile career "programmer"
        (integer)1
    1)ziplist编码：当有新键值对要加入时，
                   先将保存键的压缩列表节点放置压缩列表表尾，
                   再将保存值的压缩列表节点放置压缩列表表尾。
      因此同一键值对的节点会紧挨在一起，且先添加入的键值对靠前，后添加的键值对靠后。
        |redisObject
        |type=REDIS_HASH
        |encoding=REDIS_ENCODING_ZIPLIST
        |ptr        ---->|zlbytes|zltail|zllen|"name"|"tom"|"age"|25|"career"|"programmer"|zlend|
        |...
    2)hashtable编码：  每个键值对使用一个字典键值对来保存
                       每个键和值都是一个字符串对象。
        |redisObject
        |type=REDIS_HASH
        |encoding=REDIS_ENCODING_HT
        |ptr        ----> |dict|
        |...               |StringObject:"age"    |-->|StringObject:25|
                           |StringObject:"career"|-->|StringObject:"programmer"|
                           |StringObject:"name"   |-->|StringObject:"tom"|

    哈希对象采用ziplist编码的条件：(两个都需要满足)
    ①哈希对象保存的所有键值对的键和值 字符串长度都小于64字节
    ②哈希对象保存的键值对数量小于512个
    若不能满足以上两个条件，则采用hashtable编码。

    哈希命令及相应编码操作：(具体见图8-9哈希命令及操作)
```

命令：
HSET
HGET
HEXISTS
HDEL
HLEN
HGETALL

2017/7/21

14.集合对象：

集合对象的编码可以是intset hashtable

1)intset编码：保存在整数集合中。

```
redis> SADD members 1 3 5
(integer)3

|redisObject                    |-> intset
|type=REDIS_SET                 |   encoding=INT16
|encoding=REDIS_ENCODING_INTSET |   length=3
|ptr        ---------------------   contents-->|1|3|5|
```

1)hashtable编码：字典的每个键都是字符串对象，
每个字符串对象包含一个集合元素
字典的值全部设置为null

```
redis> SADD members "apple" "banana" "cherry"
(integer)3

|redisObject
|type=REDIS_SET
|encoding=REDIS_ENCODING_HT
|ptr        ----> |dict|
|...              |StringObject:"apple" |-->NULL
                  |StringObject:"banana"|-->NULL
                  |StringObject:"cherry"|-->NULL
```

集合对象采用intset编码的条件：(两个都需要满足)

①集合对象保存的所有元素都是整数值

②集合对象保存的元素数量小于512个

若不能满足以上两个条件，则采用hashtable编码。

集合命令及相应编码操作：(具体见图8-10集合命令及操作)

命令：
SADD
SCARD
SISMEMBER
SMEMBERS
SRANDMEMBER
SPOP
SREM

15.有序集合：

有序集合对象的编码可以是ziplist skiplist

```
redis> SADD members  "apple" 8.0 "banana" 5.0 "cherry" 9.0
(integer)3
```

1)ziplist编码： 当有新键值对要加入时，会对分值低的键值对先插入
                 先将保存键的压缩列表节点放置压缩列表表尾，
                 再将保存值的压缩列表节点放置压缩列表表尾。
      因此同一键值对的节点会紧挨在一起，分值低的元素靠前，分值高的靠后。
        |redisObject
        |type=REDIS_ZSET
        |encoding=REDIS_ENCODING_ZIPLIST
        |ptr   ---->|zlbytes|zltail|zllen|"banana"|5.0|"apple"|8.0|"cherry"|9.0|zlend|
        |...

2)skiplist编码： 使用zset结构作为底层实现，zset结构包含一个字典和跳跃表
        typedef struct zset{
            zskiplist *zsl;
            dict *dict;
        }zset;
    zset结构中的zsl跳跃表按分值从小到大保存所有集合元素，每个跳跃表节点都保存了一个集合元素
    跳跃表节点的object属性保存元素的成员，score属性保存元素分值。
    zset结构中dict字典为有序集合创建了一个从成员到分值的映射，字典每个键值对都保存一个集合元素(成员对象和它的分值)
    有序集合每个元素的成员都是字符串对象，元素的分值都是double类型浮点数。
    （跳跃表和字典都通过指针共享相同元素的成员和分值，不会有重复元素副本）

为什么有序集合zset结构同时使用跳跃表和字典来实现？
    答：有序集合需要满足 有序、查找 等功能
        单独使用跳跃表实现，能够保证有序及范围型操作功能，但在根据成员查找分值操作时复杂度为O(logN)
        单独使用字典实现，能够保证成员和分值映射查找复杂度为O(1) 但不能以有序方式保存集合元素。
skiplist编码的有序集合实现方式，见图8-16skiplist编码有序集合对象

    有序集合对象采用ziplist编码的条件：(两个都需要满足)
    ①有序集合对象保存的所有元素成员长度小于64字节
    ②有序集合对象保存的元素数量小于128个
    若不能满足以上两个条件，则采用skiplist编码。
有序集合命令及相应编码操作：(具体见图8-11有序集合命令及操作)
    命令：
        ZADD
        ZCARD
        ZCOUNT
        ZRANGE
        ZREVRANGE
        ZRANK
        ZREVRANK
        ZREM
        ZSCORE
16.类型检查与命令多态
redis操作键的命令分两种
①可对任意类型的键执行：DEL EXPIRE RENAME TYPE OBJECT
②只对特定类型键执行：
字符串键：SET GET APPEND STRLEN
哈希键：HDEL HSET HGET HLEN
列表键：RPUSH LPOP LINSERT LLEN
集合键：SADD SPOP SINTER SCARD

有序集合键：ZADD ZCARD ZRANK ZSCORE

命令多态，会在执行时根据值对象的编码方式来选择正确的执行函数。
对列表对象执行LLEN时，若为ziplist编码，则使用ziplistLen函数，若为linkedlist编码，则用listLength函数

17.内存回收

Redis在对象系统中构建了一个引用计数实现内存回收机制，通过跟踪对象的引用计数信息，完成释放对象和内存回收
1)当新创建一个对象时，对象的引用计数值初始化为1
2)当对象被别处引用时，对象的引用计数值+1
3)当对象被引用解除时，对象的引用计数值-1
4)当对象的引用计数值=0，释放对象所占用的内存

18.对象空转时长

redisObject结构包含一个lru属性，记录对象最后一次被命令程序访问的时间
OBJECT IDLETIME命令可输出指定键的空转时长
例：

```
redis> SET msg "hello world"
OK
//等一会儿
redis> OBJECT IDLETIME msg
(integer)20
//再等一会儿
redis> OBJECT IDLETIME msg
(integer)120
redis> GET msg
"hello world"
redis> OBJECT IDLETIME msg
(integer)0
```

当redis服务器回收内存算法基于LRU时，当服务器占用内存超过某个界限，将优先释放空转时间较高的键

19.小结

Redis底层数据结构：简单动态字符串(SDS) 双端链表(linkedlist)
字典(hashtable) 跳跃表(zskiplist)
整数集合(intset) 压缩列表(ziplist)

对象系统结构：	编码方式：
string	int embstr raw
list	ziplist linkedlist
hash	ziplist hashtable
set	intset hashtable
zset	ziplist skiplist

20.服务器数据库

Redis服务器所有数据库都保存在服务器状态redis.h/redisServer结构的db数组中
db数组每个项都是redis.h/redisDb结构，每个redisDb结构代表一个数据库

```
struct redisServer{
    //....
    int dbnum;//服务器数据库数量 默认=16
    redisDb *db;//数组，保存所有数据库
    //....
}
```

Redis客户端默认为0号数据库，通过SELECT命令切换目标数据库。
redisClient结构的db属性记录客户端当前的目标数据库，指向redisDb的指针

```
typedef struct redisClient{
```

```
        //.....
        redisDb *db;
        //...
    }redisClient;
```

redisClient.db指向redisServer.db数组中其中一个元素，即客户端的目标数据库。

```
typedef struct redisDb {
    // 数据库键空间，保存着数据库中的所有键值对
    dict *dict;                 /* The keyspace for this DB */
    // 键的过期时间，字典的键为键，字典的值为过期事件 UNIX 时间戳
    dict *expires;              /* Timeout of keys with a timeout set */
    // 正处于阻塞状态的键
    dict *blocking_keys;        /* Keys with clients waiting for data (BLPOP) */
    // 可以解除阻塞的键
    dict *ready_keys;           /* Blocked keys that received a PUSH */
    // 正在被 WATCH 命令监视的键
    dict *watched_keys;         /* WATCHED keys for MULTI/EXEC CAS */
    struct evictionPoolEntry *eviction_pool;    /* Eviction pool of keys */
    // 数据库号码
    int id;                     /* Database ID */
    // 数据库的键的平均 TTL ，统计信息
    long long avg_ttl;          /* Average TTL, just for stats */
} redisDb;
```

数据库键空间：

服务器中的每个数据库都由一个redis.h/redisDb 结构表示，
其中redisDb中的dict字典保存了数据库中的所有键值对，
这个dict字典称为键空间。

对数据库键进行更新的具体操作：图9-8键空间HSET更新键

设置过期时间：

EXPIRE <key> <ttl> 将键key生存时间设置为ttl秒
PEXPIRE <key> <ttl>将键key生存时间设置为ttl毫秒
EXPIREAT <key> <timestamp>将键key生存时间设置为timestamp指定的秒时间戳
PEXPIREAT <key> <timestamp>将键key生存时间设置为timestamp指定的毫秒时间戳

在执行时都会先转换调用PEXPIREAT命令实现

redisDb结构中 expires字典保存数据库中所有键的过期时间，此为过期字典

```
typedef struct redisDb{
    //...
    dict *expires ;//过期字典
    //...
}redisDb;
```

过期字典的键是一个指针，指向键空间中的某个键对象
过期字典的值是一个long long 类型的整数，保存了对应的过期时间(毫秒精度的UNIX时间戳)

使用PERSIST命令移除过期时间，在过期字典中删去记录。
使用TTL命令以秒为单位返回键的剩余生存时间，PTTL返回以毫秒为单位
(通过在过期字典中搜索过期时间与当前时间相减获得返回)

过期键删除策略：
1)定时删除：
通过使用定时器使过期键尽可能快的被删除，释放内存，但对CPU时间不友好，
在过期键多的情况下，可能会占有比较久的CPU时间，而内存不紧张时应该注重
优先处理客户端请求
2)惰性删除：
在需要取出键时，才进行过期检查，但对内存不友好，占有内存累积不释放
若有大量过期键没有被访问到，则会一直占用内存
3)定期删除：
每隔一段时间执行一次过期键操作，通过限制删除操作执行的时长和频率来减少对CPU时间影响
难点在于确定时长和频率
Redis过期键策略实现：使用惰性删除和定期删除策略配合
1)惰性删除策略的实现：
所有读写数据库的Redis命令在执行前会调用db.c/expireIfNeeded函数对输入键进行检查
若输入键已过去，则删除
2)定期删除策略的实现：
每当Redis的服务器周期性操作redis.c/serverCron函数执行时，redis.c/activeExpireCycle函数会被调用
在规定的时间内分多次遍历服务器中的各个数据库，从expires字典(过期字典)中随机检查部分键的过期时间
伪代码过程如下：

```
{
    DEFAULT_DB_NUMBERS=16 //默认检查的数据库数量
    DEFAULT_KEY_NUMBERS=20//默认每个数据库检查的键数量
    current_db=0 //全局变量 记录检查进度
    def activeExpireCycle():
        /*初始化要检查的数据库数量
          优先以数据库实际数量和默认值中较小的为准
        */
        if server.dbnum < DEFAULT_DB_NUMBERS
            db_numbers = server.dbnum
        else
            db_numbers = DEFAULT_DB_NUMBERS

        for i in range(db_numbers)://遍历各个数据库
            /*
                若current_db的值等于服务器数据库数量
                将current_db重置为0，开始新一轮遍历
                由于函数执行是以限定时长为标准的，
                每次运行不一定都刚好遍历完所有数据库一次
                即每次全局变量current_db的值不一定从0开始
            */
            if current_db == server.dbnum
                current_db=0
            redisDb = server.db[current_db] //获取当前要处理的数据库
            current_db+=1 //数据库索引加一
            for j in range(DEFAULT_KEY_NUMBERS)//检查数据库键
                if redisDb.expires.size()==0 //若此数据库中无定时键
                    break
                key_with_ttl = redisDb.expires.get_random_key()//随机获取一个定时键
                if is_expired(key_with_ttl) //检查是否过期
```

```
                        delete_key(key_with_ttl)
                    if reach_time_limit()//达到函数运行时长了则退出
                        return
                }
    activeExpireCycle工作模式如下：
        每次函数运行时从一定量的数据库中随机取出一定量的定时键进行过期检查并删除
        current_db记录当前检查进度(到哪个数据库了)

    AOF、RDB和复制功能对过期键的处理
        1)生成RBD文件
            执行SAVE/BGSAVE命令创建一个新的RDB文件时，程序会对数据库中的键进行检查，已过期的键不会被保存进去
        2)载入RDB文件
            服务器以主服务器模式运行，在载入RDB文件时会对文件中的键进行检查，忽略已过期的键
            服务器以从服务器模式运行，在载入RDB文件时会全部载入，当主从服务器进行数据同步时，从服务器的数据库会被清空
        3)AOF文件写入
            当过期键被删除后，会向AOF文件追加一条DEL命令，显式地记录该键已删除
            当客户端访问过期的message键时，执行以下3个动作
            删除message键、追加DEL message到AOF文件、返回客户端空回复
        4)AOF重写
            在对AOF进行重写时，会对数据库中的键进行检查，已过期的键不会被写入文件
        5)复制
            服务器运行在复制模式下时，从服务器的过期键删除动作由主服务器控制
                主服务器删除一个过期键后，显式地向所有从服务器发送DEL命令告知
                从服务器只有接收到主服务器的DEL命令后才删除过期键，否则不会对过期键进行删除操作
                注意：客户端对从服务器进行读命令，即便键已过期，从服务器还是将其当未过期对待，返回给客户端
            过程图见图9-17主从服务器删除过期键
-------------------------------------------------------------------------------------
从服务器不会删除过期键，而是等待主服务器发现键过期后传来DEL命令，再删除
则此过程中可能会返回给客户端过期键。对于此现象，何解？
猜测：无其他操作，客户端获得过期键，不过由于只是读，不会造成太大影响，因为在写操作时，会在主服务器进行检查并且反馈信息。
Q2：客户端在从服务器获得过期键值100 未过期键值80，累加后赋予新键写入到主服务器 怎么办？ （本该是80，现在却是180）
猜测：与写有关的事务操作，全在主服务器进行，即在主服务器读，再赋值，再写入。即可解决过期键问题。
或是在主服务器操作的写事件执行前由业务执行，在主服务器再读一次。也可解决。
-------------------------------------------------------------------------------------
    小结回顾：见图9-20数据库相关小结

    21.RDB持久化
        通过保存数据库中的键值对来记录
        RDB持久化可以手动执行也可以根据服务器配置选项定期执行，将某时间点上的数据库状态保存到RDB文件中
        RDB文件是一个经过压缩的二进制文件

        RDB文件的创建与载入：
            生成RDB文件的两个命令-- SAVE BGSAVE
                SAVE命令会阻塞Redis服务器进程，直到RDB文件创建完毕为止
                BGSAVE命令派生一个子进程，由子进程负责创建RDB文件
                创建的工作实际由rdb.c/rdbSave函数完成
            在服务器启动时会自动检测RDB文件，进行载入，由rdb.c/rdbLoad函数完成
            (当AOF持久化开启时，会优先使用AOF文件还原数据库状态)

        可以设置自动间隔性执行BGSAVE
```

若在配置文件中设置
save 900 1 //900秒内数据库进行了至少1次修改
save 300 10//300秒内数据库进行了至少10次修改
save 60 10000//60秒内数据库进行了至少10000次修改
则只要满足其中一个就会执行BGSAVE命令。
服务器状态redisServer结构的saveparams属性记录设置的save保存条件
```
struct redisServer{
    //...
    struct saveparam *saveparams;//记录保存条件的数组
    //...
};
struct saveparam{
    //秒数
    time_t seconds;
    //修改数
    int changes;
};
```
例：
```
|redisServer|
|...|            saveparams[0] saveparams[1] saveparams[2]
|saveparams |--> |seconds=900| |seconds=300| |seconds=60|
|...|            |changes=1   | |changes=10 | |changes=10000|
```

服务器状态维持一个dirty计数器及lastsave属性
dirty计数器记录距离上一次成功执行SAVE/BGSAVE后，服务器对数据库状态进行了多少次修改
lastsave属性是一个UNIX时间戳，记录上次成功执行SAVE/BGSAVE命令的时间
```
struct redisServer{
    //...
    long long dirty;//修改计数器
    time_t lastsave ;//上次执行保存的时间
    //...
};
```
检查保存条件是否满足：
Redis服务器周期性函数serverCron默认每隔100毫秒执行一次
其中一项功能是检查save选项设置的条件是否满足。
RDB文件结构：
```
    |REDIS|db_version|databases|EOF|check_sum|
```
db_version长度为4字节，一个字符串表示的整数记录RDB文件的版本号
databases部分包含零或任意多个数据库，及其中的键值对数据
EOF常量长1字节，标志文件正文内容的结束
check_sum长8字节无符号整数，保存一个校验和，以供检查文件是否损坏

database结构：
```
    |SELECTDB|db_number|key_value_pairs|
```
SELECTDB长1字节，标志后面的是数据库号
db_number数据库号，服务器根据此号调用SELECT命令切换数据库，使后续键值对正确载入
key_value_pairs保存数据库中的所有键值对数据
例：两个非空的数据库 0号和3号
```
|REDIS|db_version|SELECTDB|0|key_value_pairs|SELECTDB|3|key_value_pairs|EOF|check_sum|
```

key_value_pairs结构：
|EXPIRETIME_MS|ms|TYPE|key|value|
EXPIRETIME_MS表示键值对是带过期时间的
ms表示该键值对的过期时间
根据TYPE类型保存value的结构类型
具体过程与对象类型根据编码保存键值对方式相似

22.AOF持久化

通过保存Redis服务器所执行的写命令来记录数据库状态
AOF持久化功能实现分三步骤：命令追加 文件写入 文件同步
1)命令追加
当服务器执行完一个写命令，会以协议格式将写命令追加到服务器状态的aof_buf缓冲区末尾

```
struct redisServer{
    //...
    sds aof_buf;//AOF缓冲区
    //..
};
```

2)AOF文件写入和同步
Redis服务器进程是一个事件循环loop，
循环中的文件事件负责接收客户端命令请求及回复
时间事件负责执行定时运行的函数

在服务器处理文件事件时可能会执行写命令，feedAppendOnlyFile函数使一些内容追加到aof_buf缓冲区
所以服务器每次结束一个事件循环之前会调用flushAppendOnlyFile函数
考虑是否需要将aof_buf缓冲区内容写入到AOF文件
伪代码过程如下：{

```
def eventLoop()
while true
    /*
        处理文件事件，接收命令请求以及发送命令回复
        处理时可能会有新内容追加到aof_buf缓冲区
    */
    processFileEvents()
    processTimeEvents()//处理时间事件
    flushAppendOnlyFile()//考虑是否需要将aof_buf缓冲区内容写入到AOF文件
```

}
flushAppendOnlyFile函数行为由服务器(redis.conf配置文件)配置的appendfsync选项决定

选项值	行为
always	将aof_buf缓冲区内容全写入同步到AOF文件
everysec(默认)	将aof_buf缓冲区内容全写入AOF文件，与上次同步时间超过1秒则进行AOF文件同步
no	将aof_buf缓冲区内容全写入到AOF文件，但此时不同步

三种选项值详细描述见图11-1AOF持久化效率和安全性

AOF文件载入与数据还原：
1)创建一个不带网络连接的伪客户端(fake client),用来执行载入AOF文件
2)从AOF文件中分析并读取出一条写命令
3)用伪客户端执行写命令
4)重复步骤2-3直到AOF文件中所有写命令被执行处理
AOF重写：
不断地向AOF文件中写入执行命令会导致文件体积不断加大，通过重写AOF文件以新文件代替旧文件不包含冗余命令

```
            如：
            redis> RPUSH mylist "A" "B" //{"A" "B"}
            (integer)2
            redis> RPUSH mylist "C"     //{"A" "B" "C"}
            (integer)3
            redis> RPUSH mylist "D" "E" //{"A" "B" "C" "D" "E"}
            (integer)5
            redis> LPOP mylist          //{"B" "C" "D" "E"}
            "A"
            redis> LPOP mylist          //{"C" "D" "E"}
            "B"
            redis> RPUSH mylist "F" "G" //{"C" "D" "E" "F" "G"}
            (integer)5
            需要向AOF文件写入六条命令（只记录写命令）
            可以通过直接从数据库中读取键mylist的值，用一条RPUSH mylist "C" "D" "E" "F" "G" 代替
                aof.c/rewriteAppendOnlyFile函数 重写过程伪代码：
                {
                    def aof_rewrite(new_aof_file_name)
                        f=creat_file(new_aof_file_name)//创建新AOF文件
                        for db in redisServer.db    //遍历数据库
                            if db.is_empty()        //忽略空数据库
                                break
                            f.write_command("SELECT"+db.id)
                            for key in db           //遍历数据库中所有键
                                if key.is_expired() //忽略已过期键
                                    continue
                                if      key.type==String    //根据键的类型进行重写
                                        rewrite_string(key)
                                elseif  key.type==List
                                        rewrite_list(key)
                                elseif  key.type==Hash
                                        rewrite_hash(key)
                                elseif  key.type==Set
                                        rewrite_set(key)
                                elseif  key.type==SortedSet
                                        rewrite_sorted_set(key)
                                if key.have_expire_time()   //对带有过期时间的定时键写入其过期时间
                                    rewrite_expire_time(key)
                        f.close()
                }
                {
                    def rewrite_list(key)
                        item1,item2....itemN=LRANGE(key,0,-1)//获取所有元素
                        f.write_command(RPUSH,key,item1,item2...itemN)//使用RPUSH命令重写
                } //其他相似操作过程伪代码见图11-2AOF重写过程函数伪代码
            AOF重写生成的新文件只包含当前数据库状态所必须的命令，不会造成硬盘空间浪费。
----MARK----------------
/*
AOF重写条件：
(在循环检查事件servCron函数中进行判断执行rewriteAppendOnlyFileBackground())
```

```
(1)没有bgsave命令在进行。
(2)没有bgrewriteaof在进行。
(3)当前aof文件大小大于server.aof_rewrite_min_size，注意它的默认值为1MB
*/
----MARK----------------
        注意：
            由于一份键值对的数据可能会很长，若重写时均采用以一条命令写入，可能会造成缓冲区溢出，
            因此在处理时会先检查键所包含的元素数量，若超过redis.h/REDIS_AOF_REWRITE_ITEM_PER_CMD常量
            则会使用多条命令记录键的值。
    AOF后台重写：
        aof_rewrite函数可以很好完成一个新AOF文件的任务，但会进行大量写入操作，调用此函数的线程会被长时间阻塞
        而Redis服务器是使用单个线程处理命令请求。因此，采用子进程调用执行的方式。

        aof.c/rewriteAppendOnlyFileBackground函数进行后台重写。
            1)子进程进行AOF重写期间，父进程可继续处理命令请求
            2)子进程带有父进程的数据副本，不使用线程可以避免使用锁。
        当子进程进行AOF重写期间，服务器可能继续对现有数据进行修改，因此设置一个AOF重写缓冲区
        当子进程开始使用后，Redis服务器的写命令会发送给"AOF缓冲区"和"AOF重写缓冲区"
        优点：
            1)AOF缓冲区会定时被写入和同步到AOF文件，对现有AOF文件处理工作正常进行
            2)创建子进程后，服务器执行的写命令记录到AOF重写缓冲区中，当子进程完成AOF重写后，
              向父进程发送信号，然后父进程将AOF重写缓冲区内容写入到新AOF文件并完成新旧AOF文件替换。
        后台重写完成后调用backgroundRewriteDoneHandler函数--> aofRewriteBufferWrite函数
        将aof重写缓冲区中的内容写入到aof重写文件中。

----------------------Mark----------------------
    /*
    子进程已经在重写AOF文件了
    服务器为什么还要把写命令发给AOF缓冲区
    (为什么不只发给AOF重写缓冲区)
    */
    /*
    个人理解，
    在进行AOF文件持久化时对于新的写命令，
    会记录到AOF缓冲区或AOF重写缓冲区
    (具体是哪个缓冲区应该根据当前是否是选择采用重写AOF文件)
    此理解的正确性，在之后看过源码再决定。此处先Mark
    */
    /*
    理解2：2017/7/26
    aof.c/feedAppendOnlyFile函数中，确实有对两个缓存都加入写命令，
    原因可能是aof写入是作为比较常规的方式，而重写aof在一定条件下进行，
    因此对常规aof_buf中保留一份写命令，也可以防止重写失败等。
    当重写aof进行完毕以后，即aof持久化成功，两者缓冲区都会清空的。
    */
----------------------Mark----------------------
2017/7/22
        23.事件:Redis服务器是一个事件驱动程序，处理两类事件，1.文件事件 2.时间事件
            网络部分属于Reactor模式，同步非阻塞模型！
            ae.c源代码文件有关事件的运行，ae*.c文件为其辅助。
```

```
1151            /* 事件处理器的状态*/
1152            typedef struct aeEventLoop {
1153                // 目前已注册的最大描述符
1154                int maxfd;   /* highest file descriptor currently registered */
1155                // 目前已追踪的最大描述符
1156                int setsize; /* max number of file descriptors tracked */
1157                // 用于生成时间事件 id
1158                long long timeEventNextId;
1159                // 最后一次执行时间事件的时间
1160                time_t lastTime;     /* Used to detect system clock skew */
1161                // 已注册的文件事件
1162                aeFileEvent *events; /* Registered events */
1163                // 已就绪的文件事件
1164                aeFiredEvent *fired; /* Fired events */
1165                // 时间事件
1166                aeTimeEvent *timeEventHead;
1167                // 事件处理器的开关
1168                int stop;
1169                // 多路复用库的私有数据
1170                void *apidata; /* This is used for polling API specific data */
1171                // 在处理事件前要执行的函数
1172                aeBeforeSleepProc *beforesleep;
1173            } aeEventLoop;
1174
1175            /*  事件处理器的主循环  */
1176            void aeMain(aeEventLoop *eventLoop) {
1177                eventLoop->stop = 0;
1178                while (!eventLoop->stop) {
1179                    // 如果有需要在事件处理前执行的函数，那么运行它
1180                    if (eventLoop->beforesleep != NULL)
1181                        eventLoop->beforesleep(eventLoop);
1182                    // 开始处理已触发的文件事件或已到的时间事件
1183                    //内部调用aeApiPoll查询epoll事件是否已到达(超时非阻塞)
1184                    aeProcessEvents(eventLoop, AE_ALL_EVENTS);
1185                }
1186            }
1187    ------------MARK----------------
1188    /*注意，redis 的 I/O 多路复用程序的所有功能都是
1189    通过包装常见的 select 、 epoll 、 evport 和 kqueue
1190    这些 I/O 多路复用函数库来实现的，
1191    每个 I/O 多路复用函数库在 Redis 源码中都对应一个单独的文件，
1192    比如 ae_select.c 、 ae_epoll.c 、 ae_kqueue.c ， 诸如此类。
1193
1194    因为 Redis 为每个 I/O 多路复用函数库都实现了相同的 API ， 所以 I/O 多路复用程序的底层实现是可以互换的
1195    程序会在编译时自动选择系统中性能最高的 I/O 多路复用函数库来作为 Redis 的 I/O 多路复用程序的底层实现。
1196    */
1197    #ifdef HAVE_EVPORT
1198    #include "ae_evport.c"
1199    #else
1200        #ifdef HAVE_EPOLL
```

```
    #include "ae_epoll.c"
    #else
        #ifdef HAVE_KQUEUE
        #include "ae_kqueue.c"
        #else
        #include "ae_select.c"
        #endif
    #endif
#endif

------------MARK----------------

            /* 事件状态  */
            typedef struct aeApiState {
                // epoll_event 实例描述符
                int epfd;
                // 事件槽
                struct epoll_event *events;
            } aeApiState;

            ae.c/aeCreateFileEvent函数创建新事件加入到eventloop

        1)文件事件：
            Redis服务器通过套接字与客户端进行连接，两者的通信会产生相应文件事件
            Redis基于Reactor模式开发网络事件处理器。
            ①文件事件处理器采用IO多路复用，监听多个套接字并分配关联不同的事件处理器
            ②当被监听的套接字准备好accpet、read、write、close等操作时，相对应的文件事件会产生
              由对应事件处理器来处理。
        文件事件处理器的构成：
            套接字、IO多路复用程序、文件事件分派器、事件处理器 4部分

        IO多路复用程序会将所有产生事件的套接字都放入一个队列中，以有序、同步的方式
        向文件事件分派器传送套接字。依次处理。

                    typedef struct aeFileEvent {
                        // 监听事件类型掩码，
                        // 值可以是 AE_READABLE 或 AE_WRITABLE ，
                        // 或者 AE_READABLE | AE_WRITABLE
                        int mask; /* one of AE_(READABLE|WRITABLE) */
                        // 读事件处理器
                        aeFileProc *rfileProc;
                        // 写事件处理器
                        aeFileProc *wfileProc;
                        // 多路复用库的私有数据
                        void *clientData;
                    } aeFileEvent;

        一次完整客户端与服务器连接事件示例：图12-7客户端与服务器通信过程

        2)时间事件：由三属性组成，时间事件id、事件到达时间when、处理函数timeProc
```

定时事件
周期性事件

所有时间事件都放在一个无序链表(时间属性无序，但新事件靠近表头)中，每当时间事件执行器运行时，遍历整个链表查找已到达时间的事件并进行处理
目前redis中只使用serverCron一个时间事件。因此无序链表性能无影响

```
typedef struct aeTimeEvent {
    // 时间事件的唯一标识符
    long long id; /* time event identifier. */
    // 事件的到达时间
    long when_sec; /* seconds */
    long when_ms; /* milliseconds */
    // 事件处理函数
    aeTimeProc *timeProc;
    // 事件释放函数
    aeEventFinalizerProc *finalizerProc;
    // 多路复用库的私有数据
    void *clientData;
    // 指向下个时间事件结构，形成链表
    struct aeTimeEvent *next;
} aeTimeEvent;
```

时间事件实例：serverCron函数
工作如下。
更新服务器的各类统计信息，如时间、内存占用、数据库占用
清理数据库中过期键值对
关闭和清理连接失效的客户端
尝试进行AOF RDB持久化操作
定期对从服务器进行同步
Redis服务器以周期性事件方式运行serverCron函数，每隔一段时间执行一次
注意：对文件事件和时间事件的处理都是同步、有序、原子进行的，因此需要尽可能减少阻塞时间，
若某次执行的写入或读取字节数超过一个阈值，会先跳出本循环，以处理一些要紧的任务，

余下数据由下次轮转再继续处理。

24.客户端结构(位于服务器下的 redisServer中的redisClient结构)

```
typedef struct redisClient{
    //...
    sds querybuf;//输入缓冲区，用于保存客户端发送的命令请求，等取出解析后将命令参数及参数个数保存到argv argc
    robj **argv;//每个项是一个字符串对象，保存命令及参数
    int argc;//记录argv数组的长度
    struct redisCommand *cmd;//保存命令的信息，调用执行

    char buf[REDIS_REPLY_CHUNK_BYTES];//固定大小的输出缓冲区，保存长度较小的回复
    int bufpos;
    list *reply;//可变大小输出缓冲区
    int authenticated;//记录客户端是否通过身份验证
```

```
        time_t ctime;//创建客户端的时间
        time_t lastinteraction;//客户端与服务器最后一次互动时间
        time_t obuf_soft_limit_reached_time;//输出缓冲区第一次到达软性限制的时间
        //..
    }redisClient;

    newworking.c/processInputBuffer函数:
        处理客户端输入的命令内容，从querybuf中获取内容，解析成命令参数等放置到属性中
        调用processCommand函数执行命令，然后清理客户端状态循环处理下个命令
        void processInputBuffer(redisClient *c){
            //详见newworking.c/processInputBuffer
             while(sdslen(c->querybuf)) {
                 //....
                 // 将缓冲区中的内容转换成命令，以及命令参数
                 //..
                 // 执行命令，并重置客户端
                 if (processCommand(c) == REDIS_OK)
                    resetClient(c);
                 //...
             }
        }

        void resetClient(redisClient *c) {
            redisCommandProc *prevcmd = c->cmd ? c->cmd->proc : NULL;

            freeClientArgv(c);
            c->reqtype = 0;
            c->multibulklen = 0;
            c->bulklen = -1;
            /* We clear the ASKING flag as well if we are not inside a MULTI, and
             * if what we just executed is not the ASKING command itself. */
            if (!(c->flags & REDIS_MULTI) && prevcmd != askingCommand)
                c->flags &= (~REDIS_ASKING);
        }

        void sendReplyToClient(aeEventLoop *el, int fd, void *privdata, int mask)
        /*
         * 负责传送命令回复的写处理器,
         * while(c->bufpos > 0 || listLength(c->reply)){//.....}
         * 为了避免一个非常大的回复独占服务器,
         * 当写入的总数量大于 REDIS_MAX_WRITE_PER_EVENT ,
         * 临时中断写入，将处理时间让给其他客户端,
         * 剩余的内容等下次写入就绪再继续写入
         */

    处理LUA脚本的伪客户端会在服务器初始化时创建，一直存在，直到服务器关闭
    载入AOF文件的伪客户端会在载入工作开始时动态创建，载入完毕后关闭。
25.服务器
    a.命令请求执行过程:
        redis> SET key value
```

OK
客户端发送命令，获得OK回复，需以下步骤
1)客户端向服务器发送命令请求 SET key value
2)服务器接收，在数据库中设置操作，产生OK命令
3)服务器将OK命令发给客户端
4)客户端接收OK命令，打印显示

b.发送命令请求：
客户端键入一个命令请求时，将其转换成协议格式，通过套接字发给服务器
如：SET key value 转换成协议内容 *3\r\n$3\r\n\SET\r\n$3\r\nkey\r\n$5\r\nvalue\r\n

c.读取命令请求：
服务器调用命令请求处理器从套接字中读取协议格式的命令请求，保存到客户端状态的输入缓冲区
对输入缓冲区中的命令请求进行分析，提取命令参数及个数，保存到客户端状态的argv argc属性
调用命令执行函数，执行命令

例：

```
|redisClient
|...
|querybuf----> |sdshdr|
|...           |free=0
               |len=33
               |buf--> *3\r\n$3\r\n\SET\r\n$3\r\nkey\r\n$5\r\nvalue\r\n
```

对其解析，保存进argv argc属性

```
|redisClient
|...
|argv---->|argv[0]          |argv[1]          |argv[2]|
|argc=3   |StringObject:"SET"|StringObject:"key"|StringObject:"value"
|...
```

d.命令执行器：
查找命令实现
根据argv[0]参数在命令表(commandtable)中查找，保存到客户端状态的cmd属性
命令表是一个字典，键为命令名(如"set""get")值是redisCommand结构

```
struct redisCommand{
    char *name;//命令名
    redisCommandProc *proc;//命令的实现函数指针
    int arity;//命令参数的个数
    char *sflags;//命令属性(r、w、lua等)
    int flags;//
    long long calls;//服务器共执行了多少次该命令
    long long milliseconds;//服务器执行该命令耗费总时长
}
```

例：对于set命令请求

```
|commandtable
|....
|"set"--->  |redisCommand
|...        |name="set"
            |proc--> void setCommand(redisClient *c);
            |arity=-3 //意为三个或以上
            |sflags="wm"
            |....
```

找到"set"及它的键后，将该redisCommand引用给redisClient结构中的cmd
执行预备操作

检查cmd指针是否指向null
查看arity属性，即参数个数是否正确
检查客户端连接状态
检查服务器是否在执行持久化SAVE写入
检查是否在执行事务，需要先放入事务队列
检查..............
调用命令的实现函数
在一切准备就绪后，调用client->cmd->proc(client)
执行完后产生相应命令回复保存在客户端状态的输出缓冲区(buf reply属性)
将回复命令返回给客户端
将命令回复发送给客户端,客户端接收转换协议内容格式并打印

a.初始化服务器：
初始化服务器状态结构，创建struct redisServer实例变量，设置默认值
由initServerConfig函数完成初始化变量，设置服务器ID 默认运行频率 配置文件路径
运行架构 默认端口号 默认持久化条件 LRU时钟 创建命令表
b.载入配置选项
若启动时，有输入相关配置，则采取用户输入的，否则选择配置文件中的默认设置
c.初始化服务器数据结构
如server.clients链表，记录各客户端状态结构
server.db数组，包含服务器的所有数据库
server.lua
server.slowlog等........
主要由initServer函数完成：
为服务器设置进程信号处理器
创建共享对象(如"ok" 整数1-10000的字符串对象等)
打开服务器监听端口
为serverCron函数创建时间事件
若AOF持久化准备文件
初始化服务器后台IO模块
d.还原数据库状态
完成初始化后，载入RDB文件或AOF文件还原数据库状态
e.执行事件循环
进入服务器事件循环loop 开始接收客户端的连接请求和命令请求

26.服务器复制

通过执行SLAVEOF命令或设置slaveof选项让服务器A从服务器B处进行复制(B为主服务器)
a.同步
1)从服务器向主服务器发送SYNC命令
2)主服务器收到SYNC命令后执行BGSAVE，在后台生成RDB文件，并将后续的写命令记录在缓冲区
3)主服务器将RDB文件和缓冲区中的写命令 发给从服务器
4)从服务器先以RDB文件更新状态，再以接收到的写命令完善更新状态。
SYNC命令非常耗费资源
b.使用PSYNC代替SYNC
两种模式：完整重同步、部分重同步
1)完整重同步用于处理"初次复制"情况：执行步骤与SYNC命令基本相同 通过RDB文件及缓冲区的写命令同步
2)部分重同步处理"断线后重复制"情况：当从服务器在断线后重新连接主服务器时，若条件允许，
则将主从连接断开期间执行的写命令发给从服务器，只接收并更新这期间的写命令。
c.部分重同步的实现：
三部分构成：

①主服务器的复制偏移量、从服务器的复制偏移量
②主服务器的复制积压缓冲区
③服务器的运行ID

1)复制偏移量
主从服务器在执行同步复制的期间分别各自维护一个复制偏移量
用于检查二者是否处于一致状态

2)复制积压缓冲区
主服务器维护一个固定长度的先进先出FIFO队列，默认1MB(可在配置文件中修改)
主服务器进行命令传播时，在发送写命令给从服务器的同时，也记录进FIFO队列，
积压缓冲区中记录着写命令字节及其偏移量。
当断线从服务器连接上时，通过从服务器的复制偏移量offset来决定执行完整同步还是部分同步。
即，若从服务器需要的offset后数据 此刻还保存在积压缓存队列中，则采取部分重同步，否则采用完整重同步。

3)服务器运行ID
每个Redis服务器(无论主从)都有自己的运行ID
通过运行ID判断检查断开重连的服务器

d.PSYNC命令的实现
若从服务器此刻没有与任何主服务器搭建复制关系，则在开始一次新的复制时，向主服务器发送PSYNC ? -1 命令
主动请求进行"完整重同步"
若已有过复制关系，则在一次新复制时发送PSYNC <runid> <offset> //runid是上次复制的主服务器运行id
offset是从服务器当前复制偏移量

若主服务器返回 +FULLRESYNC <runid> <offset>
表示将进行完整重同步操作，runid为主服务器id,offset是主服务器当前复制偏移量
若主服务器返回 +CONTINUE 表示将进行部分重同步操作
若主服务器返回 -ERR 表示无法识别PSYNC，将开始SYNC命令完整重同步

e.复制的实现
1)设置主服务器的地址和端口
将主服务器的ip+port 保存进从服务器本身的 redisServer结构实例中

```
struct redisServer{
    //..
    char *masterhost;//主服务器地址
    int masterport;//主服务器端口
    // ...
}
```

2)建立套接字连接
从服务器A向主服务器B发起连接，成功建立后，主服务器B将A当做连接到自己的一个客户端对待
为A创建相应的客户端状态。

3)发送ping命令
从服务器发送ping命令检查套接字读写状态是否正常
检查主服务器能否正常处理命令请求

4)身份验证
若从服务器设置了masterauth选项则进行身份验证 发送AUTH xxxx给主服务器，若与主服务器设置的密码不同则返回错误

5)发送端口信息
从服务器执行命令 REPLCONF listening-port <port-number> 向主服务器发送从服务器的监听端口号

6)同步
从服务器向主服务器发送PSYNC，执行同步操作
(完成同步后还需要将缓冲区中写命令发给从服务器，要求主服务器成为从服务器的一个客户端状态)

7)命令传播
主服务器发送缓冲区写命令给从服务器

f.心跳检测

在命令传播阶段，从服务器以默认每秒一次的频率向主服务器发送命令
REPLCONF ACK <replication_offset> //从服务器当前的复制偏移量
作用：检查主从服务器连接状态、辅助实现min-slaves、检查命令丢失

27. Sentinel

Sentinel是Redis的高可用性解决方案：由一个或多个Sentinel实例组成的Sentinel系统可以监视任意多个主服务器及其下属的所有从服务器，当被监视的主服务器进入下线状态时，自动将其某个从服务器升级为新的主服务器。

启动并初始化Sentinel

```
$redis-sentinel /XXXX/path/sentinel.conf
```

需要进行以下步骤

1)初始化服务器
2)将普通Redis服务器使用的代码替换成Sentinel专用代码
3)初始化sentinel状态
4)根据配置文件，初始化sentinel监视主服务器列表
5)创建连向各主服务器的网络连接

a.初始化服务器

Sentinel是一个运行在特殊模式下的Redis服务器，但不使用数据库
在初始化过程中无需载入RDB或AOF文件

b.使用Sentinel专用代码

Sentinel使用sentinel.c/REDIS_SENTINEL_PORT(26379)作为服务器端口
使用sentinel.c/sentinelcmds作为服务器命令表 ， 命令执行函数也不同
（sentinel只能执行 PING SENTINEL INFO SUBSCRIBE UNSUBSCRIBE PSUBSCRIBE PUNSUBSCRIBE）

c.初始化sentinel状态

服务器初始化一个sentinel.c/sentinelState结构 保存服务器中所有与sentinel功能相关状态
一般的服务器状态仍使用redis.h/redisServer结构保存

```
struct sentinelState{
    uint64_t current_epoch;
    dict *masters;//监视的各主服务器 键为主服务器名，值为指向sentinelRedisInstance结构的指针
    int tilt;//是否进入tilt模式
    int running_scripts;//目前正在执行的脚本数量
    mstime_t tilt_start_time;//进入tilt模式的时间
    mstime_t previous_time;//最后一次执行时间处理器的时间
    list *scripts_queue;//FIFO队列，包含所有需要执行的用户脚本
}sentinel;
```

d.sentinelRedisInstance结构

```
typedef struct sentinelRedisInstance{
    int flags;
    char *name;
    char *runid;
    uint64_t config_epoch;
    sentinelAddr *addr;
    mstime_t down_after_period;//实例无响应x秒后判为下线
    int quorum;//判为下线
    int parallel_syncs;//从服务器数量
    mstime_t failover_timeout;//刷新故障迁移状态的最大时限
    //....
}sentinelRedisInstance;
```

例：配置文件内容 创建实例 及字典内容 见图16-5sentinel状态及masters字典

```
sentinel monitor master1 127.0.0.1 6379 2
sentinel down_after_milliseconds master1 30000
sentinel parallel_syncs master1 1
```

```
                sentinel failover-timeout master1 900000
        e.创建连向各主服务器的网络连接
            对于每个被sentinel监视的主服务器来说，sentinel会创建两个连向主服务器的"异步网络连接"
            1)命令连接 向主服务器发送和接收命令  2)订阅连接 用于订阅主服务器的_sentinel_：hello 频道

            sentinel如何选择新的主服务器：见图16-21挑选新主服务器
2017/7/23
        28.集群
        集群通过分片(sharding)来进行数据共享,提供负责和故障转移功能
        节点：(运行在集群模式下的redis服务器)
            通过CLUSTER MEET命令连接节点
            CONTINUE MEET <ip> <port> 向一个节点发送命令，使node节点与后ip+port指定的节点握手
            node节点将其添加到自身所在集群中。
        a.集群数据结构
            clusterNode结构保存一个节点的当前状态 如节点创建时间 节点名字 当前配置及ip地址端口号等
                struct clusterNode{
                    mstime_t ctime;//创建节点的时间
                    char name[REDIS_CLUSTER_NAMELEN];//节点名十六进制
                    int flags;//节点状态标识
                    uint64_t configEpoch;//用于故障转移
                    char ip[REDIS_IP_STR_LEN];//节点ip号
                    int port;//端口号
                    clusterLink *link;//连接节点的有关信息
                    //....
                };
                clusterLink *link;//保存连接节点所需的有关信息
                typedef struct clusterLink{
                    mstime_t ctime;//连接创建时间
                    int fd;//TCP套接字描述符
                    sds sndbuf;//输出缓冲区
                    sds rcvbuf;//输入缓冲区
                    struct clusterNode *node;//与此连接相关的节点
                } clusterLink;
            注意：redisClient结构和clusterLink结构都有自己的套接字描述符和输入、输出缓冲区
            但redisClient结构是关于客户端的，clusterLink结构是关于连接节点的
                每个节点保存一个clusterState结构，记录集群状态
                    typedef struct clusterState{
                            clusterNode *myself;//当前节点指针
                            uint64_t currentEpoch;//集群当前配置纪元
                            int state;//集群状态(在线or下线)
                            int size;//节点数量
                            dict *nodes;//集群中的节点名单
                            clusterNode *slots[16384];//代表每个槽对应的节点信息
                            //.....
                    }clusterState;
        b.槽指派
            集群的整个数据库被分为16384个槽(slot) 数据库中的每个键都属于其中的一个，每个节点可最多处理16384个槽
            当数据库中的16384个槽都有节点在处理时，集群处于上线状态，否则为下线状态
            在由CLUSTER MEET 命令连接节点后，集群尚处于下线状态，还需CLUSTER ADDSLOTS命令分配槽指派给节点
            127.0.0.1:7000> CLUSTER ADDSLOTS 0 1 2 3 ... 5000
```

```
        OK
        将0至5000槽 指派给该节点
        当数据库中的16384个槽都指派完后，集群进入上线状态
            struct clusterNode{
                //...
                unsigned char slots[16384/8];
                int numslots;
                //...
            };
        clusterNode结构记录节点负责的槽
        slots字节数组以位向量方法记录槽，每个比特位代表一个槽
        1)传播节点的槽指派信息
            一个节点除了记录槽信息还会告知其他节点，自己目前负责哪些槽
            例：节点A记录下自己的slots数组信息，并发送告知给节点B
                节点B在clusterState.nodes字典中查找节点A对应的clusterNode结构，并更新其中的slots信息
            因此每个节点都能知道整体的各个槽由哪些节点负责
        2)CLUSTER ADDSLOTS命令实现
        {
            def CLUSTER_ADDSLOTS(*all_input_slots)
                //遍历所有槽，是否都未被指派
                for i in all_input_slots
                    if clusterState.slots[i]!=null //i槽点是否已有对应的节点
                        reply_error()  //若输入槽中，存在已被指派的槽，则报错返回
                        return
                //为未指派的槽分派给当前节点
                for i in all_input_slots
                    clusterState.slots[i]=clusterState.myself
                    setSlotBit(clusterState.myself.slots,i)
        }
    c.集群中执行命令
        客户端向节点发送与数据库键有关的命令时，节点会计算出该键是否属于自己负责的槽中
        若是，则执行命令，否则返回MOVED错误，指引客户端转向正确的节点
    例：
        127.0.0.1: 7000> SET date "2017-07-23"
        OK
            date键所在槽2022由节点7000负责，因此成功
        127.0.0.1: 7000> SET MSG "HAHA"
        ->Redirected to slot [6257] located at 127.0.0.1:7001
        OK
            MSG键由7001节点复制，因此返回MOVED错误，但会让客户端自动发送给7001节点
        127.0.0.1: 7001> GET MSG
        "HAHA"
    d.计算键属于哪个槽
        def slot_number(key)
            return CRC16(key)&16383
        CRC16(key)用于计算键key的CRC-16校验和

        CLUSTER KEYSLOT <KEY> 可查看一个指定键KEY属于哪个槽
    e.节点数据库的实现
        节点只能使用0号数据库，单机redis服务器无此限制
```

使用clusterState结构中的slots_to_keys跳跃表来保存槽和键的关系
跳跃表中的分值是槽号，成员是数据库键
当节点向数据库中添加一个新的键值对时，会关联到slots_to_keys跳跃表
当键值对被删除时，也会从跳跃表中删除。

f.重新分片

可以将任意数量已经被指派给节点A的槽 改为指派给节点B，并且相关键值对也会移到目标节点B
在重新分片过程中，集群不需要下线

重新分片实现原理：

由Redis集群管理软件redis-trib负责进行，步骤如下：

1)redis-trib对目标节点发送CLUSTER SETLOTS <slot> IMPORTING <source_id>，
让目标节点准备好从源节点导入槽slot的键值对

2)redis-trib对源节点发送CLUSTER SETLOTS <slot> MIGRATING <target_id>，
让源节点准备好将槽slot的键值对迁移到目标节点

3)redis-trib向源节点发送CLUSTER GETKEYSINSLOT <slot> <count>，
获得最多count个属于槽slot的键值对的键名

4)对于步骤3获得的每个键名，redis-trib向源节点发送MIGRATE <target_id> <target_port> <key-name> 0 <timeout>
将选中的键原子地从源节点迁移到目标节点

5)重复步骤3、4 直到槽slot的所有键值对都迁移完成

6)redis-trib向集群中的节点发送CLUSTER SETSLOT <slot> NODE <target_id>
将槽slot指派给目标节点，通知信息发送至整个集群。所有节点获得更新信息

g.ASK错误

当源节点向目标节点迁移槽时，客户端向源节点请求某个数据库键，而该键已被迁移(槽还没迁移完)，则返回ASK错误

ASK MOVED错误区别：

ASK错误即客户端请求访问的键key所在的槽正在迁移，尚未全部完成，而该键key已被迁移，
则会将让客户端发给目标节点去请求键key

MOVED错误是槽已经完成迁移，则接下来关于槽i的请求都发给目标节点

h.复制和故障转移

集群节点分为主节点和从节点，主节点用于处理槽，从节点用于复制某个主节点，当该主节点下线时，由复制的从节点继续处理命令请求

从节点相当于从服务器

当主节点下线时，其下属从节点基于Raft算法的领头选举方法实现产生新主节点

i.消息

集群中各个节点通过发送和接收消息来进行通信

节点发送的消息如下五种：

1)MEET消息，当发送者接到客户端发送的CLUSTER MEET命令时，就发MEET消息给接收者(目标节点)，将接收者加入发送者的所在集群

2)PING消息，用于检测节点间的状态

3)PONG消息，用于回复已收到MEET PING消息

4)FAIL消息，当主节点A判断主节点B已进入FAIL状态，A会向集群广播消息，收到的节点会将B标为下线

5)PUBLISH消息，节点收到PUBLISH命令时先执行并向集群广播消息。

一条消息由消息头、消息正文组成。

消息头：由cluster.h/clusterMsg结构表示

```
typedef struct{
    uint32_t totlen;//消息长度
    uint16_t type;//消息类型
    uint16_t count;//消息正文包含的节点信息数量
    uint64_t currentEpoch;//发送者所处的配置纪元
```

```
        uint64_t configEpoch;//
        char sender[REDIS_CLUSTER_NAMELEN];//发送者ID
        unsigned char myslots[REDIS_CLUSTER_SLOTS/8];//发送者目前槽指派信息
        char slaveof[REDIS_CLUSTER_NAMELEN];//发送者的主节点名字或NULL
        uint16_t port;//发送者端口号
        uint16_t flags;//发送者标识
        unsigned char state;//发送者所处集群状态
        union clusterMsgData data;//消息正文
    }clusterMsg;
     union clusterMsgData{
         struct{
             clusterMsgDataGossip gossip[1];
         }ping;
         struct{
             clusterMsgDataFail about;
         }fail;
         struct{
             clusterMsgDataPublish msg;
         }publish;
         //....其他消息正文
     };
```

clusterMsg结构的currentEpoch、sender、myslots等属性记录了发送者自身的节点信息，
接收者会根据这些信息在自己的clusterState.nodes字典里找到对应的clusterNode结构并更新

MEET\PING\PONG消息的实现：

各个节点通过Gossip协议来交换各自关于不同节点的状态信息，三种消息的正文都由两个cluster.h/clusterMsgDataGossip结构组成

```
    union clusterMsgData{
        //....
        //MEET\PING\PONG消息正文
        struct{
            clusterMsgDataGossip gossip[1];
        }ping;
        //....
    };
```

三种消息的正文使用相同，因此节点通过消息头的type属性进行判断辨别
每次发送此三种消息时，发送者从自己的已知节点列表中随机选出两个节点，将信息保存到两个clusterMsgDataGossip结构

```
    typedef struct{
        char nodename[REDIS_CLUSTER_NAMELEN];//节点名
        uint32_t ping_sent;//最有一次向该节点发ping消息时间戳
        uint32_t pong_received;//最后一次收到pong消息时间戳
        char ip[16];//节点ip地址
        uint16_t port;//节点端口号
        uint16_t flags;//节点标识值
    }clusterMsgDataGossip;
```

FAIL消息实现：

主节点A将主节点B标记为已下线FAIL时，A向集群广播一条关于B的FAIL消息
Gossip协议传播节点消息具有一定的延迟，FAIL消息正文由cluster.h/clusterMsgDataFail表示
只包含一个nodename属性，记录已下线的节点名字

```
    typedef struct{
```

```
                        char nodename[REDIS_CLUSTER_NAMELEN];
                    }clusterMsgDataFail;
            PUBLISH消息实现：
                若客户端向集群中一个节点A发送PUBLISH消息，节点A会向其他节点广播PUBLISH消息
                    typedef struct{
                        uint32_t channel_len;
                        uint32_t message_len;
                        unsigned char bulk_data[8];//记录客户端通过PUBLISH命令发送的channel参数和message参数
                    }clusterMsgDataPublish;
20177/24
        29.发布与订阅
            a.频道的订阅与退订 SUBSCRIBE UNSUBSCRIBE
                当一个客户端执行SUBSCRIBE命令订阅某个或某些频道时，建立起订阅关系
                Redis将所有频道的订阅关系都保存在服务器状态的pubsub_channels字典里，
                键是某个被订阅的频道，值是记录所有订阅的客户端链表
                    struct redisServer{
                        //..
                        dict *pubsub_channels;
                        //..
                    }
                例： client1 client2 client3订阅“news.it”频道，client4订阅“news.sport”频道
                    |pubsub_channels
                    |"news.it"  -->client1-->client2-->client3
                    |"news.sport" -->client4
            b.模式的订阅与退订 PSUBSCRIBE PUNSUBSCRIBE
                服务器将所有模式的订阅关系保存在服务器状态的pubsub_patterns属性中
                    struct redisServer{
                        //..
                        list *pubsub_patterns;
                        //..
                    }
                pubsub_patterns链表节点包含pubsubPattern结构，记录被订阅的模式和对应的客户端
                    /* 记录订阅模式的结构  */
                    typedef struct pubsubPattern {
                        // 订阅模式的客户端
                        redisClient *client;
                        // 被订阅的模式
                        robj *pattern;
                    } pubsubPattern;

                例：客户端client7订阅模式"music.*" client8订阅模式"book.*" client9订阅模式"news.*"
                    |redisServer
                    |...
                    |pubsub_patterns --> |pubsubPattern  -->|pubsubPattern  -->|pubsubPattern
                    |...                 |client=client7    |client=client8    |client=client9
                                         |pattern="music.*" |pattern="book.*"  |pattern="news.*"
----------MARK----------------
注意：
"频道"订阅关系是dict结构
键是某个被订阅的频道，值是记录所有订阅的客户端链表
```

"模式"订阅关系是list结构
节点是pubsubPattern结构{redisClient *client;robj *pattern;}
----------MARK----------------

c.发送消息
当客户端执行PUBLISH <channel> <message> 时服务器执行两个动作：
1)将消息message发给channel频道的所有订阅者
2)若有模式pattern与碰到channel匹配，则将消息message发给pattern模式的订阅者
pubsub.c

```
/* 将 message 发送到所有订阅频道 channel 的客户端，
 * 以及所有订阅了和 channel 频道匹配的模式的客户端. */
int pubsubPublishMessage(robj *channel, robj *message) {
    int receivers = 0;
    dictEntry *de;
    listNode *ln;
    listIter li;
    // 取出包含所有订阅频道 channel 的客户端的链表
    // 并将消息发送给它们
    de = dictFind(server.pubsub_channels,channel);
    if (de) {
        list *list = dictGetVal(de);
        listNode *ln;
        listIter li;
        // 遍历客户端链表，将 message 发送给它们
        listRewind(list,&li);
        while ((ln = listNext(&li)) != NULL) {
            redisClient *c = ln->value;
            // 回复客户端。
            // 示例：
            // 1) "message"
            // 2) "xxx"
            // 3) "hello"
            addReply(c,shared.mbulkhdr[3]);
            // "message" 字符串
            addReply(c,shared.messagebulk);
            // 消息的来源频道
            addReplyBulk(c,channel);
            // 消息内容
            addReplyBulk(c,message);
            // 接收客户端计数
            receivers++;
        }
    }
    // 将消息也发送给那些和频道匹配的模式
    if (listLength(server.pubsub_patterns)) {
        // 遍历模式链表
        listRewind(server.pubsub_patterns,&li);
        channel = getDecodedObject(channel);
        while ((ln = listNext(&li)) != NULL) {
            // 取出 pubsubPattern
```

```
                        pubsubPattern *pat = ln->value;
                        // 如果 channel 和 pattern 匹配
                        // 就给所有订阅该 pattern 的客户端发送消息
                        if (stringmatchlen((char*)pat->pattern->ptr,
                                           sdslen(pat->pattern->ptr),
                                           (char*)channel->ptr,
                                           sdslen(channel->ptr),0)) {
                            // 回复客户端
                            // 示例:
                            // 1) "pmessage"
                            // 2) "*"
                            // 3) "xxx"
                            // 4) "hello"
                            addReply(pat->client,shared.mbulkhdr[4]);
                            addReply(pat->client,shared.pmessagebulk);
                            addReplyBulk(pat->client,pat->pattern);
                            addReplyBulk(pat->client,channel);
                            addReplyBulk(pat->client,message);
                            // 对接收消息的客户端进行计数
                            receivers++;
                        }
                    }
                    decrRefCount(channel);
                }
                // 返回计数
                return receivers;
            }
```

30.事务

通过MULTI EXEC WATCH等命令实现事务功能，提供将多个命令请求打包，一次性按顺序执行的机制，在事务期间，不会去执行其他客户端命令请求。

例： 事务开始、命令入队、事务执行

```
redis> MULTI
OK
redis> SET "name" "Barry"
QUEUED
redis> GET "name"
QUEUED
redis> SET "career" "programmer"
QUEUED
redis> GET "career"
QUEUED
redis> EXEC
1)OK
2)"Barry"
3)OK
4)"programmer"
```

事务状态保存在客户端状态的mstate属性中

```
typedef struct redisClient{
    // ..
```

```
        multiState mstate;
        // ..
    }redisClient;
```

事务状态包含事务队列及计数器

```
    typedef struct multiState{
        multiCmd *commands;//事务FIFO队列(数组)
        int count;//已入队命令数
        int minreplicas;        /* MINREPLICAS for synchronous replication */
        time_t minreplicas_timeout; /* MINREPLICAS timeout as unixtime. */
    }multiState;
```

事务队列是multiCmd类型的数组

```
    typedef struct multiC{
        robj **argv;//参数
        int    argc;//参数数量
        struct redisCommand *cmd;//命令指针
    }multiCmd;
```

例：事务状态 见图19-2事务状态

每次新加命令入队时动态为 *commands扩展地址空间，并为其中的空闲地址赋予新multiCmd

```
    /* 将一个新命令添加到事务队列中 */
    void queueMultiCommand(redisClient *c) {
        multiCmd *mc;
        int j;
        // 为新数组元素分配空间
        c->mstate.commands = zrealloc(c->mstate.commands,
                sizeof(multiCmd)*(c->mstate.count+1));
        // 指向新元素
        mc = c->mstate.commands+c->mstate.count;
        // 设置事务的命令、命令参数数量，以及命令的参数
        mc->cmd = c->cmd;
        mc->argc = c->argc;
        mc->argv = zmalloc(sizeof(robj*)*c->argc);
        memcpy(mc->argv,c->argv,sizeof(robj*)*c->argc);
        for (j = 0; j < c->argc; j++)
            incrRefCount(mc->argv[j]);
        // 事务命令数量计数器增一
        c->mstate.count++;
    }
    /*结束并重置事务状态*/
    void discardTransaction(redisClient *c) {
        // 重置事务状态
        freeClientMultiState(c);
        initClientMultiState(c);
        // 屏蔽事务状态
        c->flags &= ~(REDIS_MULTI|REDIS_DIRTY_CAS|REDIS_DIRTY_EXEC);;
        // 取消对所有键的监视
        unwatchAllKeys(c);
    }
```

WATCH命令-乐观锁，在EXEC命令执行之前，监视任意数量的数据库键，并在EXEC执行时，检查被监视的键是否有被修改过，若是，则拒绝执行事务，返回失败回复

```
    typedef struct redisDb{
        // ..
        dict *watched_keys;//被watch命令监视的键
        // ..
    }
```

watched_keys字典由数据库保持，键为要监视的数据库键，值为在监视的客户端

例：

```
    |watched_keys
    |"name" --> client1-->client2
    |"age"  --> client3
```

所有对数据库进行修改的命令，如SET\LPUSH\SADD\ZREM.....执行后都会调用multi.c/touchWatchedKey函数

对watched_keys字典进行检查，是否有客户端在监视刚刚修改的键，若有则修改客户端的REDIS_DIRTY_CAS标识，

在执行execCommand函数中，会检查REDIS_DIRTY_CAS标识，判断是否键被修改，取消事务执行

伪代码如下：

```
    {
        def touchWatchedKey(db,key)
            if key in db.watched_keys
                for client in db.watched_keys[key]
                    client.flags|=REDIS_DIRTY_CAS
    }
```

服务器收到客户端的EXEC命令时，会根据此客户端是否打开了REDIS_DIRTY_CAS标识决定是否执行事务

```
/*执行事务，并传播给下属服务器节点或写入AOF文件*/
    void execCommand(redisClient *c) {
        int j;
        robj **orig_argv;
        int orig_argc;
        struct redisCommand *orig_cmd;
        int must_propagate = 0; /* Need to propagate MULTI/EXEC to AOF / slaves? */
        // 客户端没有执行事务
        if (!(c->flags & REDIS_MULTI)) {
            addReplyError(c,"EXEC without MULTI");
            return;
        }
        /* 检查是否需要阻止事务执行，因为：
         * 1) Some WATCHed key was touched.
         *    有被监视的键已经被修改了
         *  2) There was a previous error while queueing commands.
         *    命令在入队时发生错误，指语法错误，则事务处理时不执行此命令(因为没有入队)
                但，操作错误，如对list类型的key 进行set类型相关命令，会入队，
                而在EXCE时返回本命令错误，执行其他正常命令
         *    （注意这个行为是 2.6.4 以后才修改的，之前是静默处理入队出错命令）
         * 第一种情况返回多个批量回复的空对象
         * 而第二种情况则返回一个 EXECABORT 错误
         */
        if (c->flags & (REDIS_DIRTY_CAS|REDIS_DIRTY_EXEC)) {
            addReply(c, c->flags & REDIS_DIRTY_EXEC ? shared.execaborterr :
                                                      shared.nullmultibulk);
            // 取消事务
            discardTransaction(c);
```

```
1992                        goto handle_monitor;
1993                    }
1994                    // 已经可以保证安全性了，取消客户端对所有键的监视
1995                    unwatchAllKeys(c); /* Unwatch ASAP otherwise we'll waste CPU cycles */
1996                    // 因为事务中的命令在执行时可能会修改命令和命令的参数
1997                    // 所以为了正确地传播命令，需要现备份这些命令和参数
1998                    orig_argv = c->argv;
1999                    orig_argc = c->argc;
2000                    orig_cmd = c->cmd;
2001
2002                    addReplyMultiBulkLen(c,c->mstate.count);
2003                    // 执行事务中的命令
2004                    for (j = 0; j < c->mstate.count; j++) {
2005                        // 因为 Redis 的命令必须在客户端的上下文中执行
2006                        // 所以要将事务队列中的命令、命令参数等设置给客户端
2007                        c->argc = c->mstate.commands[j].argc;
2008                        c->argv = c->mstate.commands[j].argv;
2009                        c->cmd = c->mstate.commands[j].cmd;
2010                        /* 当遇上第一个写命令时，传播 MULTI 命令。
2011                         * 这可以确保服务器和 AOF 文件以及附属节点的数据一致性。
2012                         */
2013                        if (!must_propagate && !(c->cmd->flags & REDIS_CMD_READONLY)) {
2014                            // 传播 MULTI 命令
2015                            execCommandPropagateMulti(c);
2016                            // 计数器，只发送一次
2017                            must_propagate = 1;
2018                        }
2019                        // 执行命令
2020                        call(c,REDIS_CALL_FULL);
2021                        // 因为执行后命令、命令参数可能会被改变
2022                        // 比如 SPOP 会被改写为 SREM
2023                        // 所以这里需要更新事务队列中的命令和参数
2024                        // 确保附属节点和 AOF 的数据一致性
2025                        c->mstate.commands[j].argc = c->argc;
2026                        c->mstate.commands[j].argv = c->argv;
2027                        c->mstate.commands[j].cmd = c->cmd;
2028                    }
2029                    // 还原命令、命令参数
2030                    c->argv = orig_argv;
2031                    c->argc = orig_argc;
2032                    c->cmd = orig_cmd;
2033                    // 清理事务状态
2034                    discardTransaction(c);
2035                    // 将服务器设为脏(已变动)，确保 EXEC 命令也会被传播
2036                    if (must_propagate) server.dirty++;
2037                handle_monitor:
2038                    /* Send EXEC to clients waiting data from MONITOR. We do it here
2039                     * since the natural order of commands execution is actually:
2040                     * MUTLI, EXEC, ... commands inside transaction ...
2041                     * Instead EXEC is flagged as REDIS_CMD_SKIP_MONITOR in the command
```

```
                 * table, and we do it here with correct ordering. */
                if (listLength(server.monitors) && !server.loading)
                    replicationFeedMonitors(c,server.monitors,c->db->id,c->argv,c->argc);
            }


        事务的ACID性质：
            原子性Atomicity 一致性Consistency 隔离性Isolation 持久性Durability
        1)原子性：
            一系列动作要么都执行要么都不执行，Redis事务不支持回滚机制，
            对于语法错误的  redis会拒绝执行整个事务  因此不需要回滚
            对于操作错误的 redis会执行其中正确的操作，对失败的返回错误

        2)一致性：
            数据库在事务执行前后无论成功与否整体上都是一致的
        3)隔离性：
            即使数据库中有多个事务并发执行，各个事务间亦不会相互影响，各事务的产生结果与单事务串行结果是一样的
            Redis单线程方式执行事务，总是串行运行的
        4)持久性：
            事务执行完毕后结果进行保存，不会丢失(Redis中由数据库持久化方式决定)

    31.Lua脚本
            使用EVAL命令可以直接对输入的脚本进行求值
            redis> EVAL "return 'hello world'" 0
            "hello world"
            EVALSHA命令可以根据脚本的SHA1校验和对脚本进行求值(但此脚本必须至少被EVAL执行过一次)
            redis> EVAL "return 1+1" 0
            (integer)2
            redis> EVALSHA "a27e7e8a43702b7046d4f6a7ccf5b60cef6b9bd9" 0
            (integer)2
        a)创建并修改Lua环境
            1)创建一个基础Lua环境
                调用lua_open函数创建新的基础环境

            2)载入多个函数库到Lua环境，以便Lua脚本使用
                基础库(base library):包含Lua的核心函数
                表格库(table library):处理表格的通用函数，如拼接、插入、删除、排序
                字符串库(string library):处理字符串的通用函数，如查找、格式化、翻转...
                数学库(math library):标准C语言数学库接口，用于math计算
                调试库(debug library):对程序调试需要的函数
                Lua CJSON库:用于处理UTF-8的JSON格式,JSON格式的字符串与Lua值相互转换
                Struct库:在Lua值和C结构之间进行转换
                Lua cmsgpack库:用于处理MessagePack格式数据与Lua值的转换

            3)创建全局表格redis，包含对Redis进行操作的函数
                主要有用于执行Redis命令的redis.call redis.pcall ,计算SHA1校验和的redis.sha1hex等
                可在Lua脚本中执行Redis命令

            4)使用Redis自制随机函数替换Lua的随机函数，避免副作用
```

5)创建排序辅助函数，供Lua环境使用对结果排序
保证对于无序的数据结构的结果输出，是确定的，即每次输出都是相同的

6)创建redis.pcall函数的错误报告辅助函数

7)对Lua环境中全局环境进行保护，防止用户加入额外的全局变量

8)将完成修改的Lua环境保存到服务器状态的lua属性中，等待执行脚本

```
|redisServer
|...
|lua ---> Lua环境
|...
```

b.Lua环境协作组件

用于执行脚本中Redis命令的伪客户端、用于保存脚本的lua_scripts字典

1)伪客户端

脚本使用redis.call函数或redis.pcall函数执行一个Redis命令需以下步骤

①Lua环境将redis.call函数像执行的命令传给伪客户端

②伪客户端将脚本需执行的命令传给命令执行器

③命令执行器执行命令，将结果返回给伪客户端

④伪客户端接收命令执行器返回的结果，传给Lua环境

⑤Lua环境收到命令结果后返回给redis.call函数

⑥函数将接收到的结果返回给脚本的调用者

图20-2Lua脚本执行Redis命令的通信步骤

2)lua_scripts字典

键为某个Lua脚本的SHA1校验和，值为对应的Lua脚本

```
struct redisServe{
    // ..
    dict *lua_scripts;
    // ..
};
```

redis服务器将所有被EVAL命令执行过或被SCRIPT LOAD载入的脚本保存到lua_scripts字典里

3)EVAL命令的实现

分三步骤：

①根据给定的lua脚本在lua环境中定义一个lua函数

②将脚本保存到lua_scripts字典

③执行刚定义的函数

例：

```
EVAL "return 1+1" 0
```

服务器将定义函数，函数名为f_<SHA1校验和>

```
function f_a27e7e8a43702b7046d4f6a7ccf5b60cef6b9bd9()
    return 1+1
end
```

将a27e7e8a43702b7046d4f6a7ccf5b60cef6b9bd9作为键，"return 1+1"为值保存到字典

4)EVALSHA命令的实现

检查输入的SHA1校验和是否存在于Lua环境中，若存在，则执行f_<SHA1校验和>函数

返回执行结果给客户端

5)脚本管理命令的实现

SCRIPT FLUSH 、SCRIPT EXISTS 、SCRIPT LOAD 、SCRIPT KILL

①SCRIPT FLUSH用于清除lua相关的信息，释放并重建lua_scripts字典，重建新环境

②SCRIPT EXISTS检查输入的SHA1校验和对应的脚本是否存在(遍历lua_scripts字典)

③SCRIPT LOAD导入脚本，创建对应函数，保存到lua_scripts字典

```
redis> SCRIPT LOAD "return 1+1"
"a27e7e8a43702b7046d4f6a7ccf5b60cef6b9bd9"
```

④SCRIPT KILL会检查脚本运行时间，若超过服务器设置的lua-time-limit则会终止脚本返回错误

6)脚本复制

当服务器运行在复制模式下时，具有写性质的脚本命令也会复制到从服务器，包括EVAL\EVALSHA\SCRIPT FLUSH\SCRIPT LOAD

①除EVALSHA命令外的其他命令，主服务器执行后会发送给从服务器，也执行一次即可。

②复制EVALSHA命令：

主服务器维持一个repl_scriptcache_dict字典，记录已传给从服务器的脚本

每当有一个新的从服务器加入时，主服务器就清空repl_scriptcache_dict字典，重新维持

若某个需要传播给从服务器的脚本记录在lua_scripts字典中，却不在repl_scriptcache_dict字典，

则说明从服务器未有过此脚本的校验值，应采用EVAL命令方法传播。

即，通过查找lua_scripts字典，获得sha1对应的lua脚本，然后将EVALSHA命令改为EVAL命令发送给从服务器

32.排序

底层采用快速排序，pqsort.c

a.SORT <KEY> 对一个包含数字值的键key进行排序

```
redis> RPUSH numbers 3 1 2
(integer)3
redis> SORT numbers
1)"1"
2)"2"
3)"3"
```

步骤如下：

①创建一个和number列表长度相同的数组

②遍历数组，将各个数组项的obj指针分别指向列表的各个项

```
|array              |array[0]  |array[1]
|redisSortObject -->|obj    -->|obj
                    |u         |u
```

③遍历数组，将各个obj指向的列表项值以double型存在数组的u.score里

④根据数组项u.score属性值对数组进行排序，按从小到大

⑤遍历数组返回结果

```
typedef struct redisSortObject{
    robj *obj;
    union {
        double score;
        robj *cmpobj;
    }u;

    }redisSortObject;
```

b.ALPHA选项的实现

SORT <KEY> ALPHA 通过ALPHA，SORT命令可对字符串值的键进行排序

```
redis> SADD fruits apple banana cherry
(integer)3
redis> SMEMBERS fruits
1)"apple"
2)"cherry"
3)"banana"
redis> SORT fruits ALPHA //排序输出
1)"apple"
```

```
2191                    2)"banana"
2192                    3)"cherry"
2193                步骤如下：
2194                    ①创建一个和fruits集合大小相同的redisSortObject数组
2195                    ②遍历数组，将各个数组项的obj指针分别指向集合的各个元素项
2196                    ③根据obj指针所指向的集合元素对数组进行字符串排序，从小到大
2197                    ④比较数值，依次输出
2198            c.ASC、DESC选项
2199                SORT <KEY> ASC 升序排列
2200                SORT <KEY> DESC 降序排列
2201            d.BY选项的实现
2202                通过BY选项，可以让SORT命令指定某些字符串键或某哈希键的某些域来作为元素权重进行排序
2203                    redis> SADD fruits apple banana cherry
2204                    (integer)3
2205                    redis> SORT fruits ALPHA
2206                    1)"apple"
2207                    2)"banana"
2208                    3)"cherry"
2209                    redis> MSET apple-price 8 banana-price 5.5 cherry-price 7
2210                    OK
2211                    redis> SORT fruits BY *-price
2212                    1)"banana"
2213                    2)"cherry"
2214                    3)"apple"
2215                步骤如下：
2216                    ①创建一个和fruits集合大小相同的redisSortObject数组
2217                    ②遍历数组，将各个数组项的obj指针分别指向集合的各个元素项
2218                    ③遍历数组，根据各obj指针指向的集合元素及BY选项给定的模式*-price 查找相应权重键
2219                    ④将各权重键的值转为double，保存在数组项的u.score
2220                    ⑤按u.score进行数组排序
2221            e.ALPHA+BY选项
2222                BY选项默认以权重键的值转为数字值，若需要以字符串类型当权重则需配合ALPHA
2223                    redis> SORT fruits BY *-price ALPHA
2224            f.LIMIT选项
2225                限制结果输出的长度，返回部分
2226                SORT <KEY> [ALPHA] LIMIT <num1> <num2>
2227                则对KEY进行排序，然后略过num1个有序元素，返回输出num2个有序元素
2228            g.STORE选项
2229                默认SORT命令只返回排序结果而不保存
2230                使用STORE命令将有序结果保存到另一个指定键中（指定键是List类型）
2231                SORT <KEY> [ALPHA] STORE <KEY2>
2232                redis> SADD fruits banana apple cherry
2233                (integer)3
2234                redis> SORT fruits ALPHA STORE sorted_fruits
2235                (integer)3
2236                redis> LRANGE sorted_fruits 0 -1
2237                1)"apple"
2238                2)"banana"
2239                3)"cherry"
2240
```

h.多个选项的执行顺序

调用SORT命令时，除GET选项外，其他选项的摆放顺序不影响执行

```
SORT <key> ALPHA DESC BY <by-pattern> LIMIT <offset> <count> GET <XX1> GET<XX2> STORE <store-key>
SORT <key> LIMIT <offset> <count> BY <by-pattern> ALPHA GET <XX1> GET<XX2> STORE <store-key> DESC
```

结果相同，但是GET<XX1> GET<XX2>的相对前后顺序不能改变

即GET<XX1> GET<XX2> 与 GET<XX2> GET<XX1>结果不同

33.二进制位数组

Redis提供SETBIT GETBIT BITCOUNT BITOP 四个命令用于处理二进制位数组(bit array)

1)SETBIT 指定键偏移量上的二进制位设置0或1

```
redis> SETBIT bit 0 1 //0000 0001
(integer)0
redis> SETBIT bit 3 1 //0000 1001
(integer)0
redis> SETBIT bit 0 0 //0000 1000
(integer)1
```

2)GETBIT获取位数组指定偏移量上的二进制位值

```
redis> GETBIT bit 0   //0000 1000
(integer)0
redis> GETBIT bit 3   //0000 1000
(integer)1
```

3)BITCOUNT统计位数组中值为1的二进制位数量

```
redis> BITCOUNT bit   //0000 1000
(integer)1
redis> BITCOUNT bit2   //0000 1011
(integer)3
```

4)BITOP对多个位数组进行按位与(AND)、或(OR)、异或(XOR)、取反(NOT)

```
redis> SETBIT X 3 1   //X=0000 1000
(integer)0
redis> SETBIT Y 2 1   //Y=0000 0100
(integer)0
redis> SETBIT Z 0 1   //Z=0000 0001
(integer)0

redis> BITOP OR or-result x y z //0000 1101
```

a.位数组的表示

使用字符串对象来表示位数组，SDS数据结构

例： 一字节长的位数组

```
|redisObject        --> |sdshdr
|...                |   |free=0
|type=REDIS_STRING  |   |len=1
|...                |   |buf -->buf[0]//1011 0010 注意，保存的位数组顺序是逆序 真实为0100 1101，因为二进制是从右往左，数组是从左往右
|ptr          -----       buf[1]//空字符
```

sdshdr.len=1表示SDS保存一字节长的位数组 (空字符不计入)

buf数组中buf[0]保存位数组，buf[1]保存自动添加的末尾空字符'\0'

注意:位数组以8位为单位保存在sdshdr的buf数组中，

在计算具体某一位时会先确定在buf数组的哪个项，

再计算在该项中的哪一位。

若现有位数组为8位即一字节大小，而命令请求设置大于8位，则需进行扩展

```
redis> SETBIT X 12 1
```

SETBIT执行示例见 图22-7SETBIT命令执行过程
SETBIT扩展示例见 图22-8SETBIT命令执行扩展数组过程

b.二进制位统计算法：variable-precision SWAR算法

BITCOUNT命令返回位数组中1的数量，顺序遍历的方式太过低效
而buf数组以每8位为一个单位，8位的可排列是有限的，因此可以组建
一个以8位二进制排列为键，值为其中1的数量的字典表
通过比对表可快速查找得到结果。表越大对于搜索节省的时间越多，但越消耗内存。

因此引用variable-precision SWAR算法

统计一个位数组中非0二进制位的数量，在数学上称为"计算汉明重量"
variable-precision SWAR算法通过一系列位移和位运算操作，可在常熟时间内计算多个字节的汉明重量，无需借助额外内存
计算32位的汉明重量

```
uint32_t swar(uint32_t i){
    i=(i&0x55555555)+((i>>1)&0x55555555);//step1
    i=(i&0x33333333)+((i>>2)&0x33333333);//step2
    i=(i&0x0F0F0F0F)+((i>>4)&0x0F0F0F0F);//step3
    i=(i*(0x01010101)>>24);//step4
    return i;
}
```

step1:计算出的值i的二进制表示可以按每两个二进制位为一组进行分组，各组的十进制表示即该组的汉明重量
step2:计算出的值i的二进制表示可以按每四个二进制位为一组进行分组，各组的十进制表示即该组的汉明重量
step3:计算出的值i的二进制表示可以按每八个二进制位为一组进行分组，各组的十进制表示即该组的汉明重量
step4: i*(0x01010101) 计算出bit array的汉明重量并记录在二进制位的高八位，>>24右移运算，将其移到低八位
则返回的就是bit array的汉明重量。

c.Redis实现二进制位统计算法

BITCOUNT命令的实现用到了查表和variable-precision SWAR算法

查表算法：

使用键长为8位的表，记录从0000 0000 到 1111 1111 的二进制位的汉明重量

variable-precision SWAR算法：

BITCOUNT命令每次循环中载入128个二进制位，调用4次 32位的SWAR函数来计算汉明重量

伪代码实现：(长度小于128位采用查表算法，否则采用SWAR算法)

```
{
    def BITCOUNT(bits)
        count=count_bit(bits)//计算位数组含多少个二进制位长度
        weight=0//初始化汉明重量
        while count >=128
            weight+=swar(bits[0:32))
            weight+=swar(bits[32:64))
            weight+=swar(bits[64:96))
            weight+=swar(bits[96:128))

            bits=bits[128:]
            count-=128
        while count
            index=bits_to_unsigned_int(bits[0:8])//将8位二进制转为无符号整数
```

```
                    weight+=weight_in_byte[index]//根据表查找这8位对应的汉明重量
                    bits=bits[8:]//指针后移，略过已处理的位
                    count-=8
                return weight    //返回汉明重量
        }
```

34.慢查询日志

用于记录执行时间超过给定时长的命令请求。可通过日志监视查询速度

配置选项：

slowlog-log-slower-than选项指定执行时间超过多少微秒的请求会被记录到日志中

slowlog-max-len选项指定服务器最多保存多少条慢查询日志

使用FIFO方式保存日志。

```
struct redisServer{
    // ..
    long long slowlog_entry_id;//下一条慢查询日志的id
    list *slowlog;//保存慢查询日志的链表
    long long slowlog_log_slower_than;//慢的阈值
    unsigned long slowlog_max_len;//最多保存多少条日志
    // ..
};
```

slowlog链表保存所有慢查询日志，每个节点为一个slowlogEntry结构

```
typedef struct slowlogEntry{
    long long id;
    time_t time;//命令执行时的时间戳
    long long duration;//命令消耗的时间，微秒
    robj **argv;//命令与参数
    int argc;//参数个数
}slowlogEntry;
```

例：

```
|redisServer
|...
|slowlog_entry_id=6              |slowlogEntry
|slowlog--------------------->   |id=5
|slowlog_log_slower_than=0       |time=1378781521
|slowlog_max_len=5               |duration=61
|..                              |argv --------> |argv[0]            |argv[1]                |argv[2]
                                 |argc=3          |StringObject:"SET" |StringObject:"number"
                                 |StringObject:"10086"
```

35.监视器

通过MONITOR命令，客户端可以使自己成为监视器，实时接收打印服务器当前处理的命令请求信息

每当一个客户端向服务器发送一个命令请求时，服务器会将关于请求的信息发送给所有监视器

伪代码如下：

```
{
    def MONITOR()
        client.flags|=REDIS_MONITOR
        server.monitors.append(client)
        send_reply("OK")
}
```

客户端C3向服务器发送MONITOR命令，则更新该客户端的REDIS_MONITOR标志，

将客户端添加到monitors链表表尾。

如：

```
                    |redisServer
                    |..
                    |monitors-->C1-->C2-->C3
                    |...
                服务器在每次处理命令请求前会调用replicationFeedMonitors函数，将相关信息发给监视器
                (回顾20服务器数据库：服务器在每次处理键命令还会调db.c/expireIfNeeded函数检查键是否存活)
                    伪代码如下：
                    {
                        def replicationFeedMonitors(client,monitors,dbid,argv,argc)
                            msg=create_message(client,dbid,argv,argc)
                            for monitor in monitors
                                send_message(monitor,msg)
                    }
    ————————————————————————————————
    Mon Jul 24 17:55:45 CST 2017(Edit)
    Thu Jul 27 14:09:05 CST 2017(Edit)
    ————————————————————————————————
    redis设计与实现(第二版) 学习日志。
    ————————————————————————————————
```