Redis设计与实现

一、SDS的定义

```
struct sdshdr {
    // 记录 buf 数组中已 使用字节的数量
    // 等于SDS所保存字符串的长度
    int len;

    // 记录 buf 数组中未使用字节的数量
    int free;

    // 字节数组，用于保存字符串
    // 保留1字节存空字符'\0'不计入len
    //例如：'R','e','d','i','s','o','\0'
    char buf[];
}
```

与C字符串的区别

1、获取字符串长度为 O(1)

2、修改SDS时会检查空间是否满足，杜绝缓冲区溢出

3、减少修改字符串时内存重新分配次数，分配策略如下：

小于1M时，原len为n，也再分配n，实际长度变为n+n+1 = 2n+1。

大于1M时，会分配1M的未使用空间。实际长度变为 nM+1M+1byte

4、惰性空间释放，缩短字符串时，并没真正回收内存，而是用free记录。（也提供相应API真正释放未使用空间）

5、可以保存二进制数据。使用len判断结尾，而不是空字符串

具体API见17页。

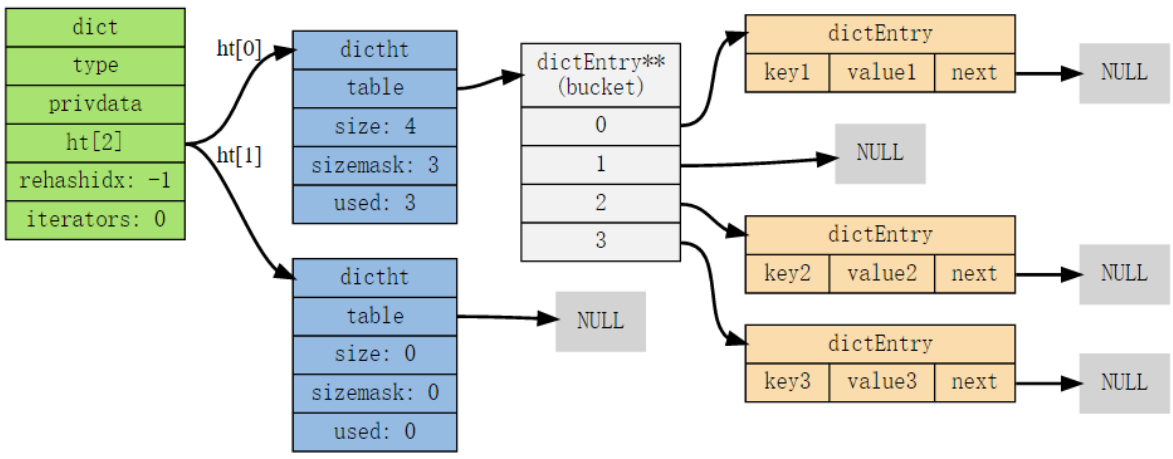
二、链表

```
typedef struct listNode {
    // 前置节点
    struct listNode *prev;

    // 后置节点
    struct listNode *next;

    // 节点的值
    void *value;
}listNode; // 双端 无环

typedef struct listNode {
    // 表头节点
    listNode *head;

    // 表尾节点
    listNode *tail;

    // 链表所包含的节点数量
    unsigned long len;

    // 节点值复制函数
    // 用于复制链表节点所保存的值
    void *(*dup) (void *ptr);

    // 节点值释放函数
    // 用于释放链表节点所保存的值
    void (*free) (void *ptr);

    // 节点值对比函数
    // 用于对比链表节点所保存的值和另一个输入的值是否相等
    void *(*match) (void *ptr, void *key);
} list
```

三、字典

```
// 哈希表
typedef struct dictht {
```

```
    // 哈希表节点指针数组（俗称桶，bucket）
    dictEntry **table;

    // 指针数组的大小
    unsigned long size;

    // 指针数组的长度掩码，用于计算索引值
    // 总是等于size-1
    unsigned long sizemask;

    // 哈希表已有的节点数量
    unsigned long used;

} dictht;

// 哈希表节点
typedef struct dictEntry {
    // 键
    void *key;

    // 值
    union {
        void *val;
        uint64_t u64;
        int64_t s64;
    } v;

    // 链往后继节点
    struct dictEntry *next;
} dictEntry;

// 字典
// 每个字典使用两个哈希表，用于实现渐进式 rehash
typedef struct dict {

    // 特定于类型的处理函数
    dictType *type;

    // 类型处理函数的私有数据
    void *privdata;

    // 哈希表（2 个）
    dictht ht[2];

    // 记录 rehash 进度的标志，值为 -1 表示 rehash 未进行
    int rehashidx;

    // 当前正在运作的安全迭代器数量
    int iterators;
} dict;
```



1、rehash
扩展空间：ht[1]的大小为第一个大于等于ht[0].used*2的2的n次方幂。若ht[0].used为6，则12<16（2的4次方）
收缩空间：ht[1]大小为第一个大于等于ht[0].used的2的n次方幂。若ht[0].used为6，则6<8（2的3次方）
将ht[0]中的所有数据rehash到ht[1]中，然后释放ht[0]
将ht[1]设置为ht[0]，并在ht[1]创建新表的空白的哈希表，为下次rehash准备

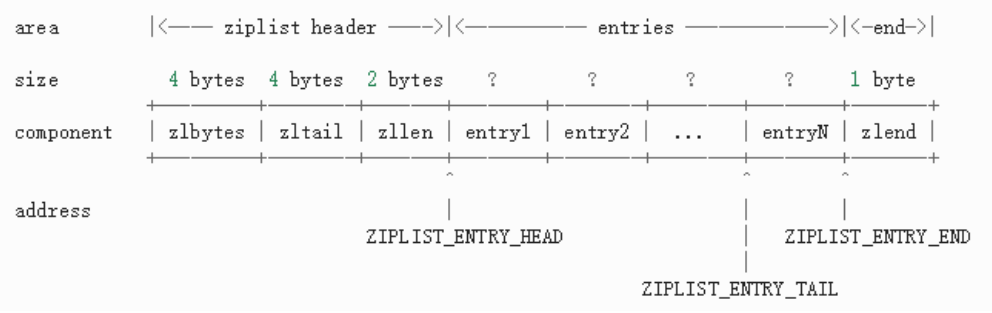
2、渐进式 rehash
为避免大量数据一次性rehash，rehashidx 索引标记rehash位置
每次CRUD操作时，两个ht都会被修改到

四、跳跃表。略。类似公交、brt、动车、飞机
五、整数集合

```
typedef struct intset {

    // 保存元素所使用的类型的长度
    // #define INTSET_ENC_INT16 (sizeof(int16_t))
    // #define INTSET_ENC_INT32 (sizeof(int32_t))
    // #define INTSET_ENC_INT64 (sizeof(int64_t))
    uint32_t encoding;

    // 元素个数
    uint32_t length;

    // 保存元素的数组
    // 元素不重复；
    // 元素在数组中由小到大排列
    int8_t contents[];

} intset;
```

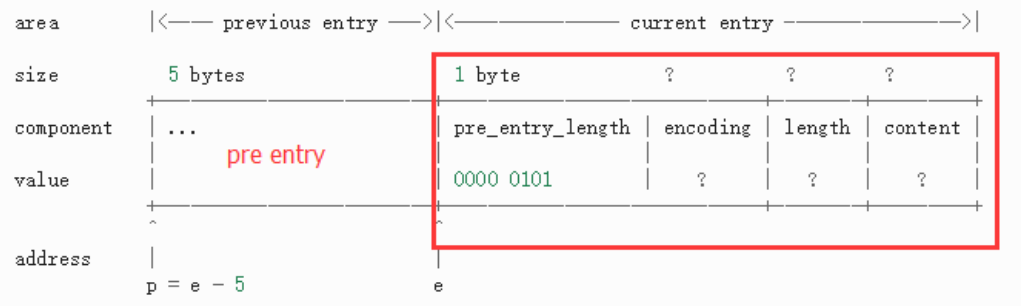
并不使用 int8_t 类型来保存任何元素，而是根据 encoding的值，对 contents 进行类型转换和指针运算，计算出元素在内存中的正确位置
升级：新元素的类型比整数集合现有类型更长时，要先进行升级，然后才能添加
1、 对新元素进行检测，看保存这个新元素需要什么类型的编码
2、 将集合 encoding 属性的值设置为新编码类型，并根据新编码类型，对整个 contents 数组进行内存重分配
3、 调整 contents 数组内原有元素在内存中的排列方式，从旧编码调整为新编码
4、 将新元素添加到集合中

五、压缩列表



zlbytes 整个 ziplist 占用的内存字节数 （4字节）
zltail 从头到尾节点的偏移量（4字节）
zllen ziplist 中节点的数量（？ 字节）
zlend 用于标记 ziplist 的末端（1字节）

节点的构成：



pre_entry_length 域可能占用 1 字节或者 5 字节，如果前一节点的长度小于 254 字节，便使用一个字节保存它的值。

否则要5个字节
encoding 和 length 一起决定了 content 部分所保存的数据的类型及长度

六：对象

```
/*
 * Redis 对象
 */
typedef struct redisObject {

    // 类型
    unsigned type:4;

    // 对齐位
    unsigned notused:2;

    // 编码方式
    unsigned encoding:4;

    // LRU 时间（相对于 server.lruclock）
    unsigned lru:22;

    // 引用计数
    int refcount;

    // 指向对象的值
    void *ptr;

} robj;

/*
 * 对象类型 type
 */
#define REDIS_STRING 0  // 字符串
#define REDIS_LIST 1    // 列表
#define REDIS_SET 2     // 集合
#define REDIS_ZSET 3    // 有序集
#define REDIS_HASH 4    // 哈希表

/*
 * 对象编码 encoding
 */
#define REDIS_ENCODING_RAW 0           // 编码为字符串
#define REDIS_ENCODING_INT 1           // 编码为整数
#define REDIS_ENCODING_HT 2            // 编码为哈希表
#define REDIS_ENCODING_ZIPMAP 3        // 编码为 zipmap
#define REDIS_ENCODING_LINKEDLIST 4    // 编码为双端链表
#define REDIS_ENCODING_ZIPLIST 5       // 编码为压缩列表
#define REDIS_ENCODING_INTSET 6        // 编码为整数集合
#define REDIS_ENCODING_SKIPLIST 7      // 编码为跳跃表
```

ptr 是一个指针，指向实际保存值的数据结构，这个数据结构由 type 属性和 encoding 属性决定

1、字符串
int:存数字（会升级为raw），raw:长度>=40、embstr:长度<=39（不可修改，修改了会升级为raw类型）
2、列表对象
ziplist 或者 linkedlist
.... 详见书本

内存回收：Redis 在自己的对象系统中构建了一个引用计数技术实现内存回收机制
对象共享：1、将数据库键的值指向一个现在对象。2、被共享对象的引用计数加一

七、数据库
1、默认创建16个数据库
2、过期键删除策略
惰性删除：所有读写操作时调用expireIfNeed函数，检查是否过期
定期删除：周期性操作redis.c/serverCron函数

八、RDB持久化
RDB是经过压缩的二进制文件，SAVE/BGSAVE 创建RDB文件
RDB文件结构
REDIS|db_version|database|EOF|check_sum
db_version 4字节版本号
check_sum 一个8字节校验和
database, 任意多个非空数据库
SELECTDB|db_number|key_value_paires
key_value_paires 保存了一个或以上的数量键值对。TYPE|key|value

九、AOF持久化
AOF持久化是通过保存redis服务器所执行的写命令来记录数据库状态的
实现：命令追加、文件写入、文件同步
命令追加：追加到服务器状态的aof_buf缓冲区尾
文件写入：aof_buf缓冲区中内容写入AOF文件。定时
同步：同步到磁盘。

十、事件
1、文件事件：redis服务器通过套接字与客户端连接，通信会产生相应的文件事情，而服务器通过监听、处理这些事件来完成一系列网络通信操作。使用I/O多路复用程序来同时监听多个套接字，并根据套接字目前执行的任务来为套接字关联不同的事件处理器
2、时间事件：定时操作

十一、客户端
1、flags 属性标志客户端处于的状态：REDIS_MASTER 主服务器、REDIS_SLAVE 从服务器（复制数据时，服务器会变主从客户端）、REDIS_MULTI 客户端正在执行事务
2、输入缓冲：querybuf
3、对缓冲进行分析，命令、参数保存到argv数组里（StringObject类型数组）
4、输出缓冲：固定大小（16K），可变大小（链表）

十二、服务端
1、读取命令
2、查找命令实现：在命令表中查找台数所指定的命令。命令表是一个字典
3、执行预备操作：执行前检查。例如是否通过身份验证
4、调用函数实现
5、执行后续工作：例如：插入日志、写入AOF
6、回复发送给客户端

维护服务器时间缓存
检查持久化操作的运行状态
初始化服务器
还原数据库状态

十三、主从复制
旧版同步复制：
1、向主服务器发送SYNC命令
2、主BGSAVE生成RDB文件发送给从
3、期间的写操作记录在缓冲区发送给从
4、从将自己更新到主的状态
旧版命令传播复制：
主操作后，把命令发送给从，从执行后再次同步

效率太低

新版同步：
PSYNC命令代替SYNC：全量、增量复制

增量复制：
1、复制偏移量：主从都维护 offset
2、复制积压缓冲区：固定长度先进先出的队列，默认1M。（结构：每个字符记录相应的复制偏移量），若偏移太多，不在缓冲区，则全量复制
3、服务器运行ID：记录主ID。断线后，若主变，则重新全量复制

检测命令丢失：主offset 100 从offset 100 主更新到150，并向从传播操作命令。并询问，让从回复REPLCONF ACK
正常情况：从更新到150
传播丢失：从收到回复要求 REPLCONF ACK 100，主发现从还是100，则再次传播操作命令给从

十四、Sentinel

监视任意多个主服务器，以及这些主服务器下的所有从服务器。升级从服务器为主服务器。
获取主服务器信息：主服务器基本信息、所有从服务器的信息
获取从服务器信息：从服务器基本信息，状态，offset 等
Sentinel 通过其它 Sentinel 发送的__sentinel__:hello 频道消息互相感知，互相连接形成网络状
检测主观下线：多次发送PING命令，得到无效回复或无回复
检查客观下线：询问多个Sentinel 足够数量的下线判断后，标记客观下线
Leader选举：
A机器向B发送请求被设置为Leader的消息
假设B设置A为它的局部Leader后，其它C/D等的请求被拒绝，并回复A
A收到回复后，检查回复中选举的轮数（leader_epoch, 第几轮）是否和自己所处的轮数相同，是，则再把收到的回复再转发出去（也就是再把自己被设为Leader的消息转出去，请求成为更多机器的Leader）
超过半数机器的leader都是A，则A被选举成功，否则从选
故障转移：
1、选出新的主服务器
2、让已下线的主服务器下的所有从服务器复制新的主服务器
3、将下线的主服务器设置为新主服务器的从服务器

十五、集群
集群中每个节点保存一个clusterState 结构 记录了当前节点视角下，集群目前状态

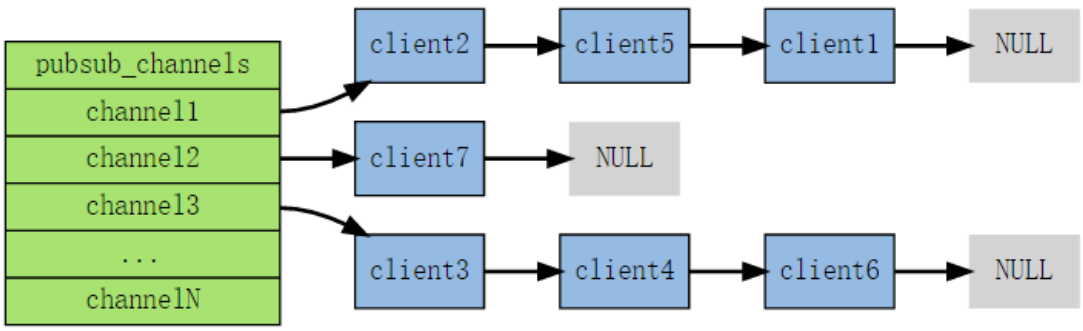
```
typedef struct clusterState {
    //指向当前节点的指针
    //保存节点的基本信息
    clusterNode *myself

    //集群当前的配置纪元，用于实现故障转移
    uint64_t currentEpoch;

    //集群当前的状态：在线不在线
    int state

    //集群中至少处理一个槽的节点数量

    int size

    //集群节点名单
    //字典key 为节点名字，value为clusterNode结构
    dict *nodes
}
```

CLUSTER MEET 命令：客户端可以让接收命令的A将另一个节点B加入到A所在的集群
CLUSTER MEET <ip> <port>
client --(CLUSTER MEET B_ip B_port)-->A 1 A发送MEET 给B 2 B返回PONG消息给A 3 A返回PING消息给B 添加成功

槽指派
集群的整个数据库被分为16384个槽。每个节点可以处理1到16384个槽
每个槽都有节点处理时，集群处于上线状态，否则下线
CLUSTER ADDSLOTS 指派槽给节点

重新分片
一直检查原节点A是否保存了新节点B的槽数据，是则迁移。否则将槽指向B

十六、发布与订阅
Redis 将所有频道的订阅关系都保存在pubsub_channels字典里面：（2、5、1订阅了channel1）



通过channel 拿到各个 client 发送消息

十七、事务

三个阶段：

1、事务开始。MULTI 事务开始的标识

2、命令入队。命令为EXEC/DISCARD/WATCH/MULTI 四个命令其中一个，则立即执行。否则不立即执行，放入事务队列。

3、事务执行：遍历事务队列，并执行命令，将返回值追加到回复队列，移除事务状态，将结果返回给客户端

感觉像是批处理，并没有回滚

WATCH命令，是一个乐观锁，EXEC命令执行前监视数据库键。SET/LPUSH/SADD/ZREM/DEL等操作时，会调用touchWatchKey函数，查看是否操作的对象正在被监视，是的话，标识该对象安全性已经被破坏，将拒绝执行事务

事务的ACID：原子性 Atomicity 一致性 Consistency 隔离性 Isolation 耐久性 Durability

隔离性：事务使用单线程方式、执行期间不会中断事务