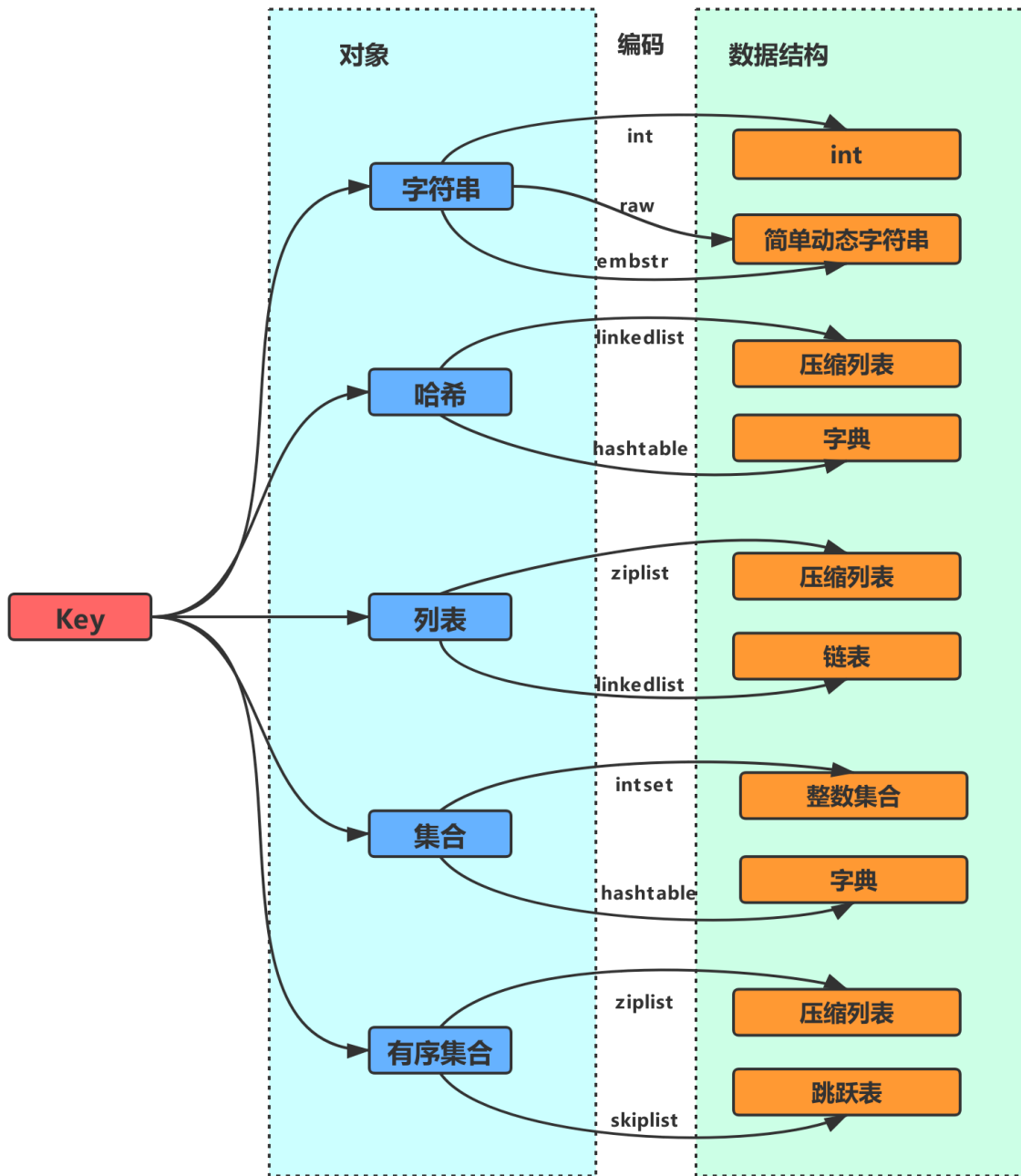


Redis

对象系统

Redis 没有直接自身的数据结构来直接实现键值对数据库，而是基于这些数据结构创建了一个对象系统。对象系统中每种对象通过 `redisObject` 来表示，每个 `redisObject` 可以使用不同的底层数据结构，通过 `redisObject::encoding` 可以自由改进内部编码而对外的数据结构和命令没有影响（**底层结构对上层应用透明**），在不同的场景下可以使用不同的内部编码方式发挥各自的优势，从而优化对象在不同场景下的使用效率。

```
1 typedef struct redisObject {
2     unsigned type:4; // 类型 (5种有效类型)
3     unsigned encoding:4; // 编码 (10种有效编码)
4     unsigned lru:REDIS_LRU_BITS; // 对象最后一次被访问的时间
5     int refcount; // 引用计数，用于内存回收与对象共享
6     void *ptr; // 指向实际值的指针
7 } robj;
```



	典型场景	string	list	set	zset	hash
quicklist		✗	✓	✗	✗	✗
linkedlist		✗	! 3.2 版前	✗	✗	✗
ziplist	元素个数和元素大小都小于各自的类型相关阈值	✗	! 3.2 版前	✗	✓	✓
hashtable		✗	✗	✓	✗	✓
intset	元素全整数且个数小于类型相关阈值	✗	✗	✓	✗	✗
skiplist		✗	✗	✗	✓	✗
int	long 以内的整数	✓	✗	✗	✗	✗
raw	字符串 >32B	✓	✗	✗	✗	✗
embstr	字符串 <=32B	✓	✗	✗	✗	✗

键key

非二进制安全的字符串类型，其值不能太长且可读性要强。

```

1 //查询
2 keys pattern //返回匹配的key
3 randomkey //从当前数据库随机返回一个key
4 type key //返回key数据类型
5 exists key //检查key是否存在
6
7
8 //删除，修改
9 flushdb //删除当前库中的key
10 flushall //删除所有库中的key
11 delkey [key...] //删除某个或者多个key-value
12 move key num //将当前数据库中的key移到数据库num中
13 rename key newkey //改名，newkey存在时则会覆盖
14 renamenx key newkey //当且仅当newkey不存在的时候才执行
15
16 //时间相关
17 expire key second //设置key的过期时间、秒
18 expireat key timestamp //设置key的过期时间、时间戳
19 pexpire key milliseconds //设定多少毫秒内过期
20 pexpireat key timestamp-milliseconds //设置为时间戳，毫秒级
21 ttl key //查看给定键距离过期时间还有多少秒
22 pttl key //查看给定键距离过期时间还有多少毫秒
23 persist key //移除过期时间
24
25 //返回查询key的属性
26 object refcount|encoding|idletime key //返回key，引用次数|内部存储编码|空转时间
27 #可编为多种方式编码：
28 #1、字符串可存为raw（一般字符串）、int（小数字）
29 #2、列表可存为ziplist、linkedlist
30 #3、集合可存为inset（数字小集合）、hashtable

```

```

31 #4、散列可存为zipmap（小散列）、hashtable
32 #5、有序集合可存为ziplist（小有序集合）、skiplist（任何大小）
33
34 //排序
35 sort source-key [by pattern] [limit offset count] [get pattern...] [asc|desc] [alpha]
   [store dest-key]
36 #1、source-key: 排序的key
37 #2、by pattern: 表示可以通过外部权重进行排序（即外部key，需要与排序key有关联）。例如：链表
   key为list-userid（1,2,3,4...），则外部key名为，goods_1、goods_2...，则by pattern为（by
   goods_*）
38 #3、limit offset count: 表示排序后返回的数据行
39 #4、get pattern...: 获取外部数据（参数与by pattern一样）
40 #5、asc|desc: 升序|降序
41 #6、alpha: 采用字符排序，默认是数字排序
42 #7、store dest-key: 表示将结果存入dest-key中

```

值value数据类型

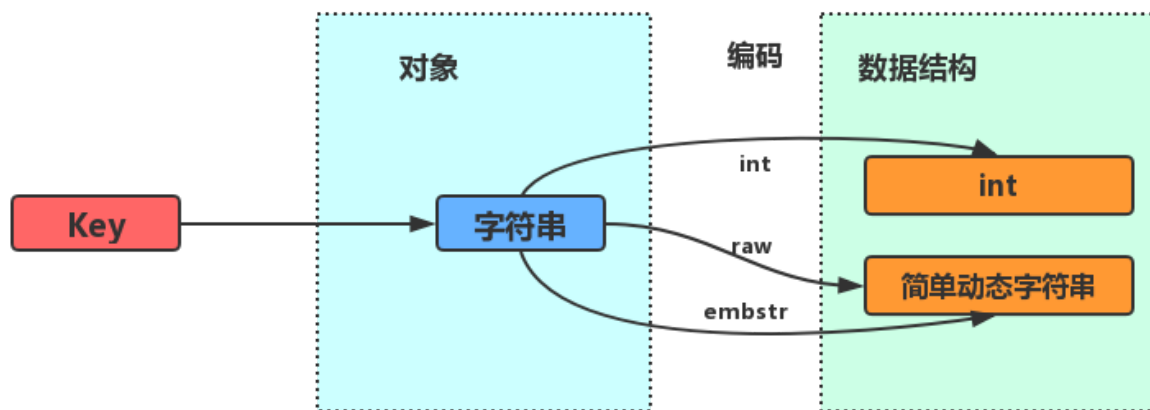
string 字符串

二进制安全的字符串类型，`string` 可以包含任何数据（图片或者序列化的对象），最大上限是 **1GB**。

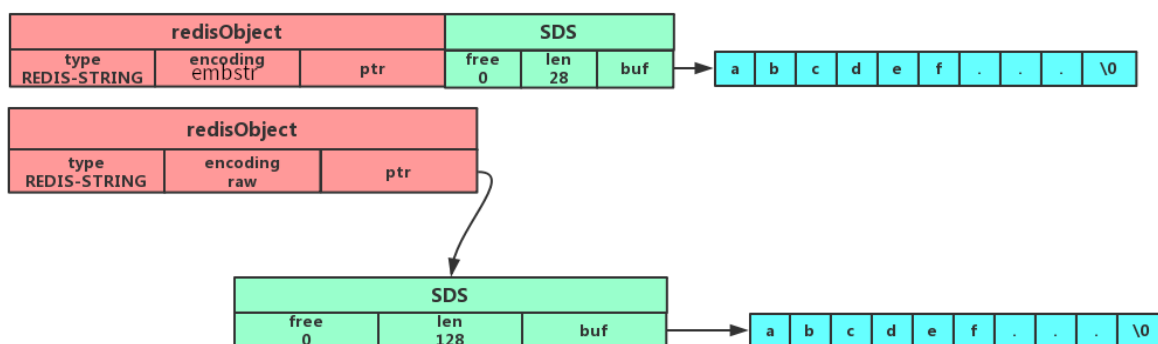
```

1  strlen key //字符串长度
2
3  //写
4  set key value //设置key-value对/
5  mset key1 value1 key2 value2...//同时写多个
6  setnx key value //当且仅当key不存在时才设置
7  setrange key-name offset value //将value代替从offset开始的字符串,超过的长度使用0代替
8
9  //读
10 get key //获取key值
11 getset key value //获取旧值设置新值
12 mget key1 key2...//同时读多个
13 setex key seconds value //设置k-v对时并且设置过期时间
14 getrange key-name start end //返回次字符串的start到end之间的字符
15
16 //修改
17 append key value //追加值
18 incr、decr、incrby、decrby、incrbyfloat //对值作原子运算
19
20 //比特位操作
21 getbit key-name offset //将字符串看做是二进制位串，并返回位串中的偏移量offset的二进制位
   的值
22 setbit key-name offset value //将字符串看做是二进制位串，并将位串中偏移量offset的二进制
   值设置为value
23 bitcount key-name [start end] //统计二进制位串里面值为1的数量
24 bitop and|or|xor|not dest-key key1 key2... //对多个key执行并或异或非，并将结果存入到
   dest-key

```



string支持int、raw和embstr三种编码方式。`embstr`（短字符串）和`raw`（长字符串）编码都会使用SDS来保存值，但不同之处在于`embstr`会通过一次内存分配函数来分配一块连续的内存空间（减少内存分配与释放的次数，利用连续内存存储最大化CPU缓存的作用）来保存`redisObject`和SDS。而`raw`编码会通过调用两次内存分配函数来分别分配两块空间来保存`redisObject`和SDS。



list 列表

按插入顺序排序的双端无环链表，每个列表最多可以存储 $2^{32} - 1$ 个元素。用户可以从序列两端推入或者弹出元素，在插入时，如果该键并不存在，Redis将为该键创建一个新的链表。如果链表中所有的元素均被移除，那么该键也将会被从数据库中删除。

```

1  llen key-name //返回key-name链的长度
2
3  //插入
4  lpush/rpush key element //将element插入到列表key的头部/尾部，不存在则创建
5  lpushx/rpushx key element //当key存在时才将element插入到列表key的头部/尾部
6  lset key index value //将key中下标为index更新值为value。index超过则报错
7  linsert key before|after val value //在key中位于val值前或者后，插入value值。key或者val不存在，则返回错误
8
9  //删除
10 lpop/rpop key // 将key-element从列表头部/尾部移除
11 lrem key count value count=0 // 删除全部一样的；count>0从左边检索删除count个；count<0从右边检索，删除count个
12 ltrim key-name start end //保持start到end所有元素，其他删除
13
14 //查找
15 lrange key start end //返回偏移量中的值，负数为倒数
16 lindex key offset //返回偏移量为offset中的值，负数为倒数
17
18 //超时操作
19 blpop/brpop key-name[key-name2...] timeout //多少秒内阻塞并等待可弹出元素出现

```

```

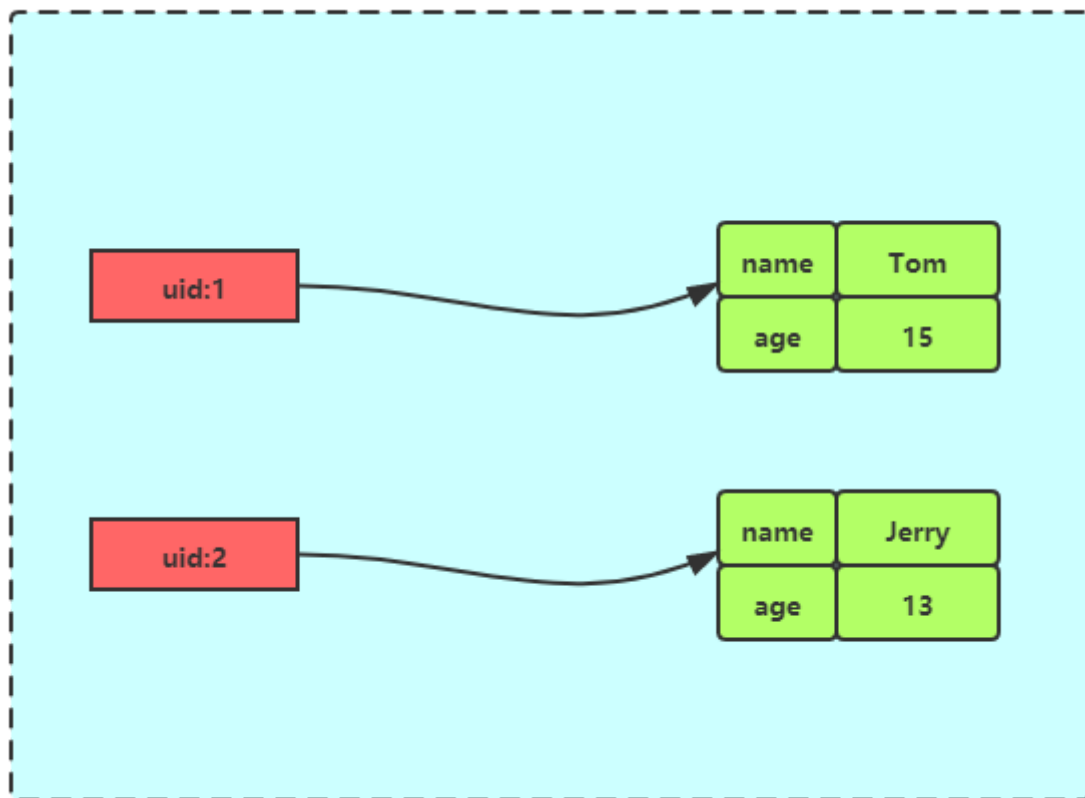
20
21 rpoplpush key-name1 key-name2 //从key-name1中弹出最右边的元素，推入key-name2最左边，并
    返回value元素
22 brpoplpush key-name value timeout //阻塞式

```

Redis3.2之前版本的list的底层编码支持linklist和ziplist，其中后者仅在元素个数小于 `list-max-ziplist-entries`（默认512）且每个元素长度都小于 `list-max-ziplist-value`（默认64B）时才会使用。Redis3.2后增加了quicklist的编码方式，list采用该编码方式。

hash 哈希

在redis中，哈希类型是指Redis键值对中的值本身又是一个键值对结构，形如 `value=[{field1, value1}, ...{fieldN, valueN}]`。



```

1  hlen key      //返回散列包含键值对的数量
2
3  //写
4  hset key field value      //设置散列值
5  hsetnx key field value    //当且仅当field不存在时设置
6  hmset key field value field2 value2... //设置多个
7
8  //读
9  hget key field //获取值
10 hmget key field field2... //获取多个
11 hgetall key //由于redis是单线程操作，假若hgetall返回的数据量大耗时大，将会导致其他客户端
    的请求得不到响应
12
13 //删
14 hdel key field field2... // 删除一个或者多个值
15
16 //查
17 hexists key-name field //检查field是否存key-name中

```

```

18 hkeys key/hvals key //返回key中的field、返回key中的value
19
20 //数值操作
21 hincrby key-name field num //给key-name中field的值（必须是数字）增加num
22 hincrbyfloat key-name key incre

```

set 集合

通过hash表实现string类型元素的不重复无序集合，

```

1 scard key //返回集合里包含的元素数量
2
3 //增
4 sadd key member[member2...] //添加一个或者多个member
5
6 //删
7 srem key member[member2...] //移除一个或者多个member
8 spop key //随机移除集合中的一个元素，并返回
9 smove key-name1 key-name2 value //将value元素从key-name1中移到key-name2中
10
11 //查
12 smembers key //返回key中所有的member
13 sismember key member //判断member是否在key中
14 srandmember key //随机返回一个member，不删除
15 srandmember key-name n //随机返回集合里的n个元素。n负数可重复，正数不出现重复
16
17 //集合操作
18 sdiff key-name1 [key-name2...] //差集
19 sdiffstore dest-key key-name1[key-name2...] //差集存入dest-key中
20 sinter key-name1[key-name2...] //返回交集
21 sinterstore dest-key key-name1[key-name2...] //交集存入dest-key中
22 sunion key-name1[key-name2...] //返回并集
23 sunionstore dest-key key-name1[key-name2...] //并集存入dest-key中

```

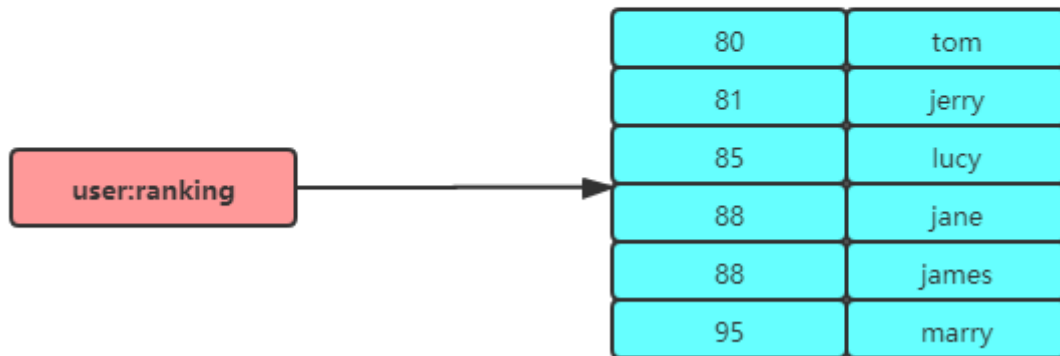
集合支持的底层编码方式包括intset和hashtable两种方式。只有在集合中的元素个数小于 `set-maxintset-entries`（默认为512个）且全是整数类型时才会使用intset的编码方式。

zset 有序集合

string类型的元素构成的不重复有序集合，为了有序需要将每个元素给定分数属性（分数可以重复），排序时先按照分数后按照字典序排序。

Key

Value



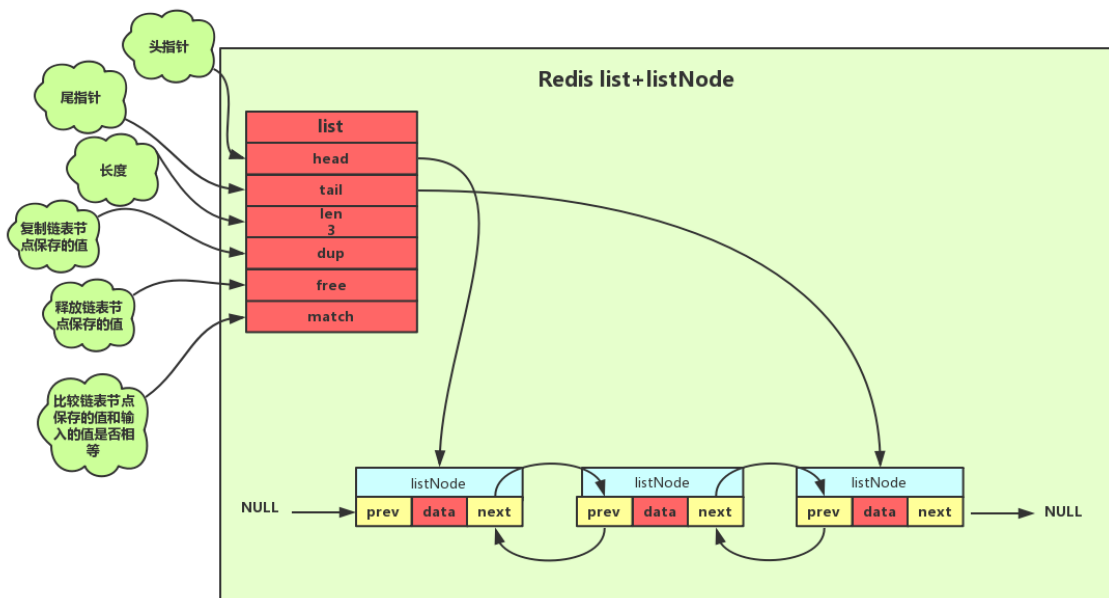
```
1  zcard key //返回有序集合里的成员数量
2
3  //增
4  zadd key score member //给有序集合key添加member
5
6  //删
7  zrem key member[member2...] //移出一个或者多个成员
8  zremrangebyrank key-name start stop //移出所有有序集合排名介于start和stop之间的元素
9  zremrangebyscore key-name min max //移出所有有序集合score介于min和max之间的元素
10
11 //查询
12 zcount key-name min max //返回分值介于min和max之间的成员数量
13 zscore key member //返回成员的分值
14
15 //分值操作
16 zincrby key increment member //将member成员分值加上increment
17 zrange key start stop [withscores] //成员按分值从小到大排列，返回有序集合给定排名范围
18 zrevrange key start stop [withscores] //成员按分值从大到小排列，返回有序集合给定排名范围
19
20 //基于分值的查询
21 zrank key member //分值从小到大排序，返回member有序集合的排名
22 zrevrank key member //分值从大到小排序，返回member有序集合的排名
23 zrangebyscore key-name min max [withscores][limit offset count] //返回有序集合中
    介于min和max之间的所有成员、从小到大
24 zrevrangebyscore key-name max min [withscores][limit offset count] //返回有序集合中
    介于max和min之间的所有成员、从大到小
25
26 zinterstore dest-key key-count key-name1[key-name2...] [weights weight1 weight2...]
    [aggregate sum|min|max] //先对应分值乘以weights，再取交集，分值对应aggregate，默认
    sum。结果存入dest-key中
27 zunionstore dest-key key-count key-name1[key-name2...] [weights weight1 weight2...]
    [aggregate sum|min|max]
```

有序集合可以由 [ziplist \(压缩列表\)](#) 或 [skiplist \(跳跃表\)](#) 组成。当有序集合较小（元素个数小于128且每个元素长度小于64B）时使用 `ziplist`，否则采用 `skiplist`。

编码方式

双向无环链表 **linkedlist**

```
1 typedef struct listNode{
2     struct listNode *prev; //前置节点
3     struct listNode *next; //后置节点
4     void *value; //节点的值, void*指针可以保存各种类型的值(多态)
5 }listNode; //单个链表节点
6
7 typedef struct list{
8     listNode *head; //表头节点
9     listNode *tail; //表尾节点
10    unsigned long len; //链表所包含的节点数量
11    void (*free) (void *ptr); //节点值复制函数
12    void (*free) (void *ptr); //节点值释放函数
13    int (*match) (void *ptr, void *key); //节点值对比函数
14 }list; //链表
```

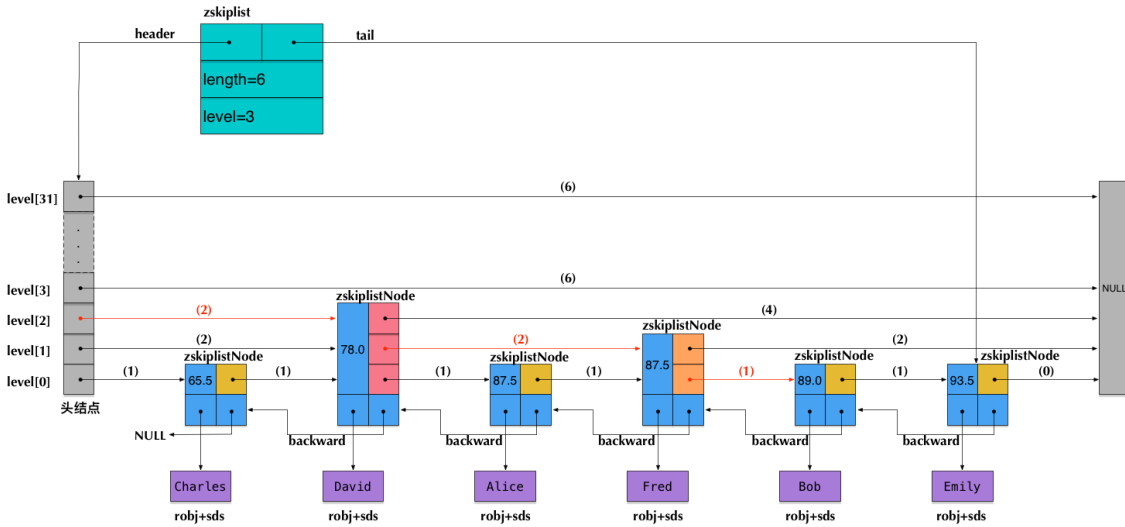


跳表 **skiplist**

经典的跳表要求每一层链表的元素的个数成倍数关系, 为了保持着这种关系, 在插入删除时需要调整整个跳表的每一层。为了避免这一问题 **skiplist** 不要求上下相邻两层链表之间的节点个数有严格的对应关系, 而是为每个节点随机出一个层数(level)。

```
1 typedef struct zskiplistNode {
2     struct zskiplistLevel{
3         struct zskiplistNode *forward; //前进指针
4         unsigned int span; //前进跨度
5     }level[]; //多层链
6     struct zskiplistNode *backward; //后退指针
7     double score; //分值
8     robj *obj; //成员对象
9 } zskiplistNode; //跳表单个节点
10
11 typedef struct zskiplist{
```

```
12 structz skiplistNode *header, *tail;//表头节点和表尾节点
13 unsigned long length;//表中节点的数量
14 int level;//表中层数最大的节点的层数
15 }zskiplist;
```



操作

查找：从高层开始向下层下阶梯式查找（如果待查节点大于当前层当前节点而小于当前层下一节点就到当前层的下一层中查找）。

插入：首先确定插入的顶层层数K（有其[概率计算公式](#)），然后将元素插入到所有小于等于K的所有层。

删除：在各个层中找到包含指定值的节点，然后将节点从链表中删除即可，如果删除以后只剩下头尾两个节点，则删除这一层。

对比

插入	内存占用	插入删除	范围查找	有序性	查找
平衡树	每节点2个指针	子树结构调整	支持、一般平衡树上实现复杂	有序	log(n)
哈希表	开链法指针	哈希冲突与标记	不支持	无序	1
跳表	节点平均指针可变（典型为1.33）	修改少	支持、实现简单	有序	log(n)

简单动态字符串 sds

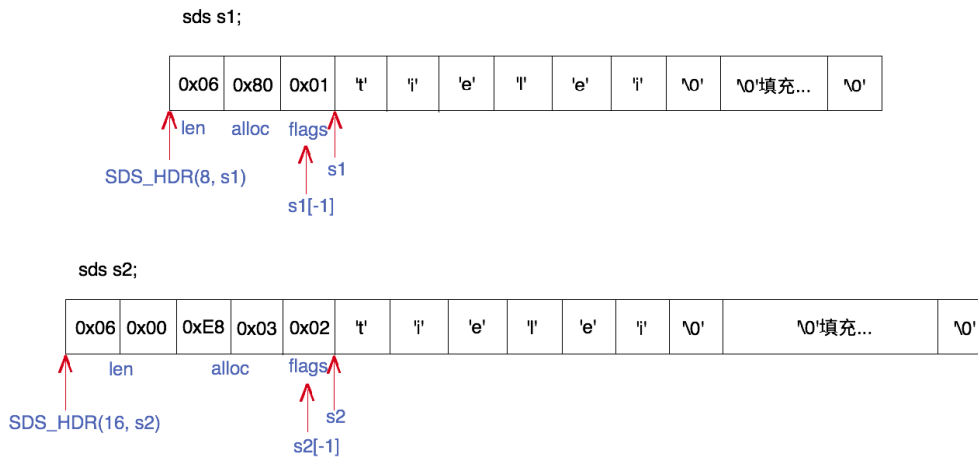
sds 只是redis提供的一个统一的接口，实际的数据存储在 sdshdr* 的结构中，根据要存储的字符串的长度会选用不同的 sdshdr，再通过宏去访问不同的数据。

```
1 //__attribute__((packed))，是为了让编译器以紧凑模式来分配内存
2 struct __attribute__((__packed__)) sdshdr5 {
3     unsigned char flags;
```

```

4   char buf[];
5   };
6
7   struct __attribute__((__packed__)) sdshdr8|16|32|64 {
8       uint8|16|32|64_t len; //字符串的真正长度（不包含NULL结束符）
9       uint8|16|32|64_t alloc; //字符串的最大容量（不包含NULL结束符）
10      unsigned char flags; //最低3个bit用来表示header的类型，其他bit无意义
11      char buf[]; //字节数组（总长度为alloc+1,多出来的一个字节为\0）
12  };
13
14  typedef char *sds; //所有的sdshdr（存储实际数据）对外都表现为sds

```



空间管理：空间预分配（每次多申请：以新申请的长度 $L=1M$ 为界，小于时多分配 L 长，大于时多分配 $SDS_MAX_PREALLOC=1M$ ）、惰性空间释放（先记录空闲而不释放）。

二进制安全：SDS的buf字节数组保存二进制数组而非字符，SDS API都会以二进制的方式来处理buf数组里的数据，使用len属性的值而不是空字符来判断字符串是否结束。

其他优点：无缓冲区溢出、减少了内存重分配次数、部分兼容C字符串。

哈希表Dict

```

1   typedef struct dict{
2       void *type; //类型特定函数,指向dictType结构体
3       void *privdata; //私有数据
4       dictht ht[2]; //哈希表, ht[0]保存当前的数据, ht[1]用于rehash
5       int trehashidx; //rehash 索引 当rehash不在进行时 值为-1
6   }dict;
7
8   typedef struct dictType {
9       unsigned int (*hashFunction)(const void *key); //对key进行哈希值计算的哈希算法
10      void *(*keyDup)(void *privdata, const void *key); //对key进行深拷贝
11      void *(*valDup)(void *privdata, const void *obj); //对value进行深拷贝
12      int (*keyCompare)(void *privdata, const void *key1, const void *key2); //定义两个key的比较操作
13      void (*keyDestructor)(void *privdata, void *key); //key的析构函数
14      void (*valDestructor)(void *privdata, void *obj); //value的析构函数
15  } dictType; //为不同类型的hash节点提供hash操作的支持函数
16

```

```

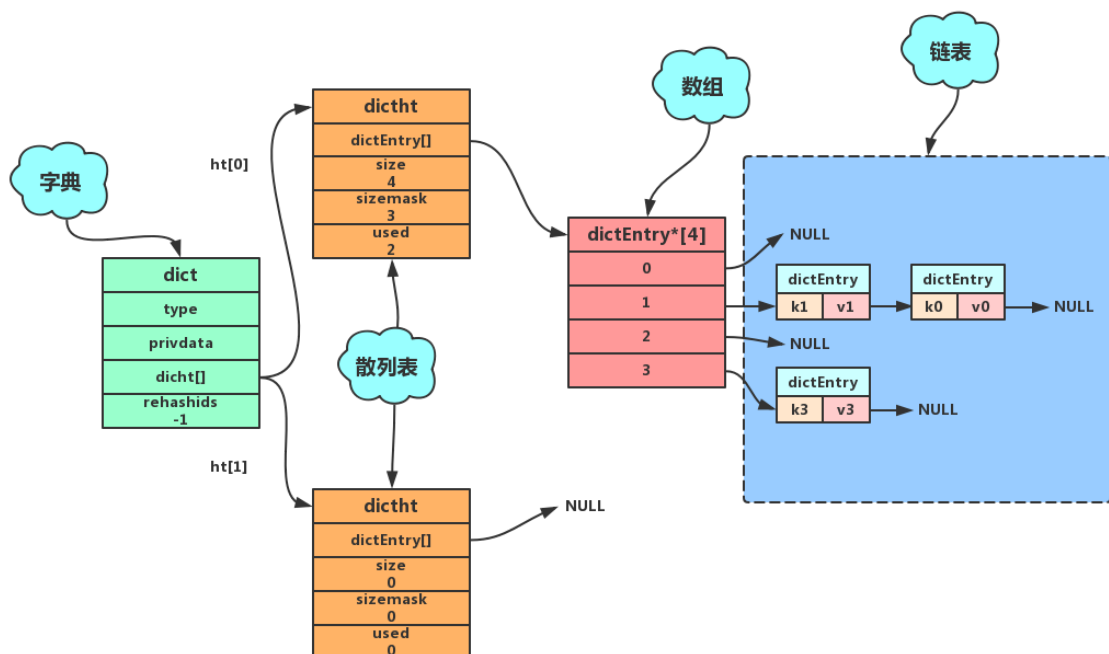
17 typedef struct dictht{
18     dictEntry **table;//哈希表数组/桶数组
19     unsigned long size;//哈希表桶大小
20     unsigned long sizemask;//哈希表桶大小掩码，用于计算索引值，总是等于 size-1
21     unsigned long used;//该哈希表已有总节点的数量
22 }dictht;//散列表
23
24 typedef struct dictEntry{
25     void *key;//键
26     union{
27         void *val;
28         uint64_tu64;
29         int64_tts64;
30     }v;//值
31     struct dictEntry *next;//指向下一个哈希表节点，使用链地址法解决哈希冲突形成链表
32 }dictEntry;//每一个桶/散列表节点

```

`dict::type`：指向 `dictType` 结构的指针，每个 `dictType` 提供了用于操作特定类型键值对的函数，Redis 会为用途不同的字典设置不同的类型特定函数。

`dict::privdata`：保存了需要传给那些类型特定函数的可选参数。

Redis 字典结构



rerehash 重新散列

当哈希表保存的键值对太多或者太少时需要对哈希表进行相应的扩展（依据负载因子的大小）或者收缩，每次扩容或者收缩会先创建原桶大小两倍或一半的新哈希表，然后对原有元素重哈希到新的哈希表上，最后释放原有空间（申请新->填充->释放旧）。

重哈希的操作发生在字典的内部，`dict[1]`用于 `rehash`、`dict[0]` 保存当前的数据，当 `rehash` 完成后，原有的散列表数据占用的内存空间将被释放。

渐近式 **rehash**：

优化大哈希表的重哈希，在进行渐进式 **rehash** 期间，字典的删除查找更新等操作可能会在两个哈希表上进行，第一个哈希表没有找到，就会去第二个哈希表上进行查找。但是进行增加操作，一定是在新的哈希表上进行的。

整数集合 **intset**

保存类型为 `int16_t`、`int32_t`、`int64_t` 的不重复有序整数值构成的集合，他们占据的是一整块的连续的内存空间。

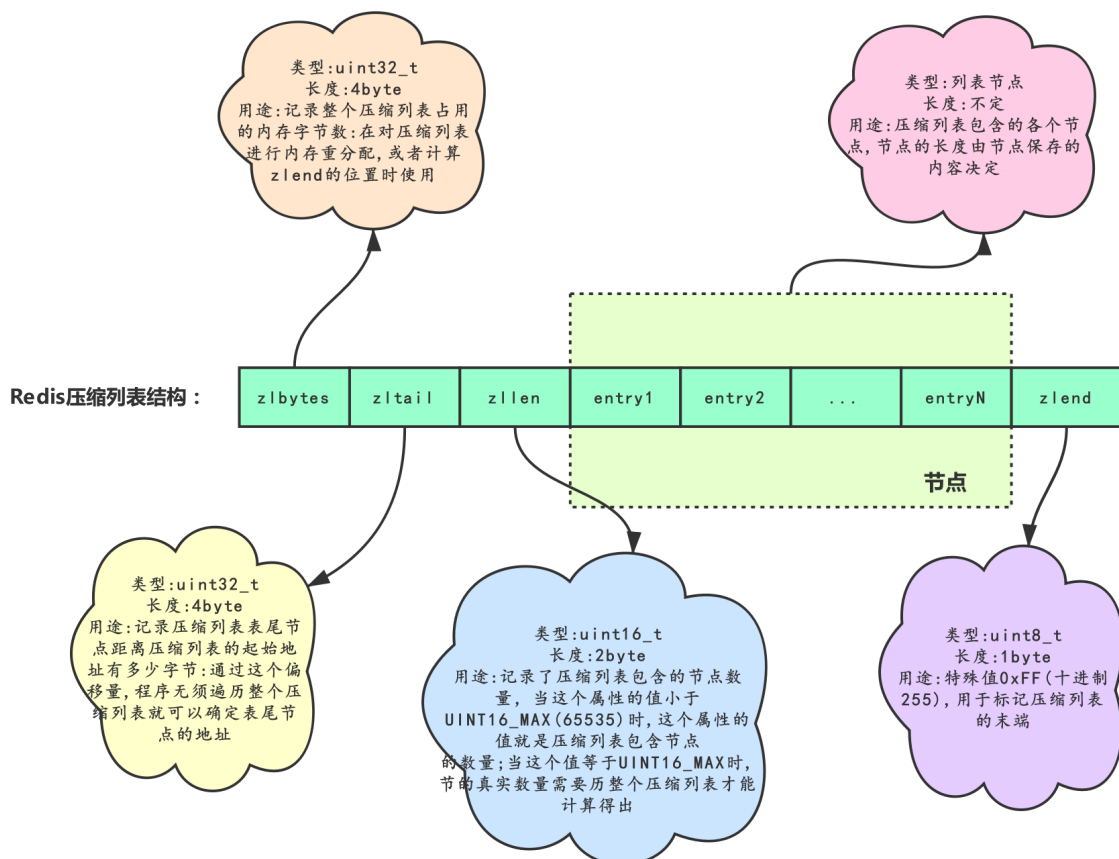
```
1 typedef struct intset{
2     uint32_t encoding; //contents元素的编码方式
3     uint32_t length; //集合包含的元素数量
4     int8_t contents[]; //保存由小到大元素的数组，保存的实际数据类型由encoding决定。
5 }intset;
```

升级

当新增的元素类型比保存的原有元素的类型要长时需要为保存元素的数组升级，一旦升级就不可降级。升级首先需要根据待插入数据确定新的编码格式，然后由新编码格式需要计算新增的内存空间，最后从后往前调整数据。

压缩列表 **ziplist**

将任意多个节点（**entry**）数据（二进制整数或者字节数组）按照特殊编码存储在某一个连续内存块（而非采用了压缩算法）构成的顺序型（**sequential**）数据结构，该结构并不擅长对内部做修改操作，但提供了对表两端的访问方法，可以快速地在头尾进行操作。



```
1  typedef struct ziplist{ //小端存储
2      uint32_t bytes; /*ziplist分配的内存大小*/
3      uint32_t tail_offset; /*达到尾部的偏移量，以字节为单位*/
4      uint16_t length; /*存储元素实体个数*/
5      unsigned char* content[]; /*存储内容实体元素，内部有其自己的结构*/
6      unsigned char end; /*尾部标识0xFF=255*/
7  }ziplist; //一个压缩列表
8
9  typedef struct zlentry { //为了能逆序遍历，每个entry需要保存前边元素的一些信息
10     unsigned int prevrawlen; //前一个元素的总长度（变长编码）
11     unsigned int len; //当前元素的有效数据长度以及数据类型（变长编码）
12     unsigned char *data; /*元素实际内容*/
13 }zlentry; //抽象描述
```

prevrawlen 字节数	场景	编码方式
1B	前驱元素总长 <=254B	直接存储长度
5B	前驱元素总长 >254B	第一字节为0xFE(254)，后四字节存储真正长度

len字节数	len值	data
1B	前两bit为00，6bit表示长度值	字符串
2B	前两bit为00，14bit表示长度值	字符串
5B	前两bit为00，后4B表示长度值	字符串
1B	0xC0	2个字节的int16_t
1B	0xD0	4个字节的int32_t
1B	0xE0	8个字节的int64_t
1B	0xF0	3个字节长的整数
1B	0xFE	1个字节的整数
1B	0xF1-0xFD	无data字段，实际数据存在len中，值为后4bit值减1(即0-12)

该编码方式只适用于少量小对象，当 `ziplist` 变大时，每次插入或修改引发的 `realloc` 操作会有更大的概率造成内存拷贝或者**连锁更新**，从而降低性能。一旦发生内存拷贝，**内存拷贝的成本**也相应增加，因为要拷贝更大的一块数据。当 `ziplist` 数据项过多的时候，在它上面查找指定的数据项就会性能变得很低，因为 `ziplist` 上的**查找需要进行遍历**。

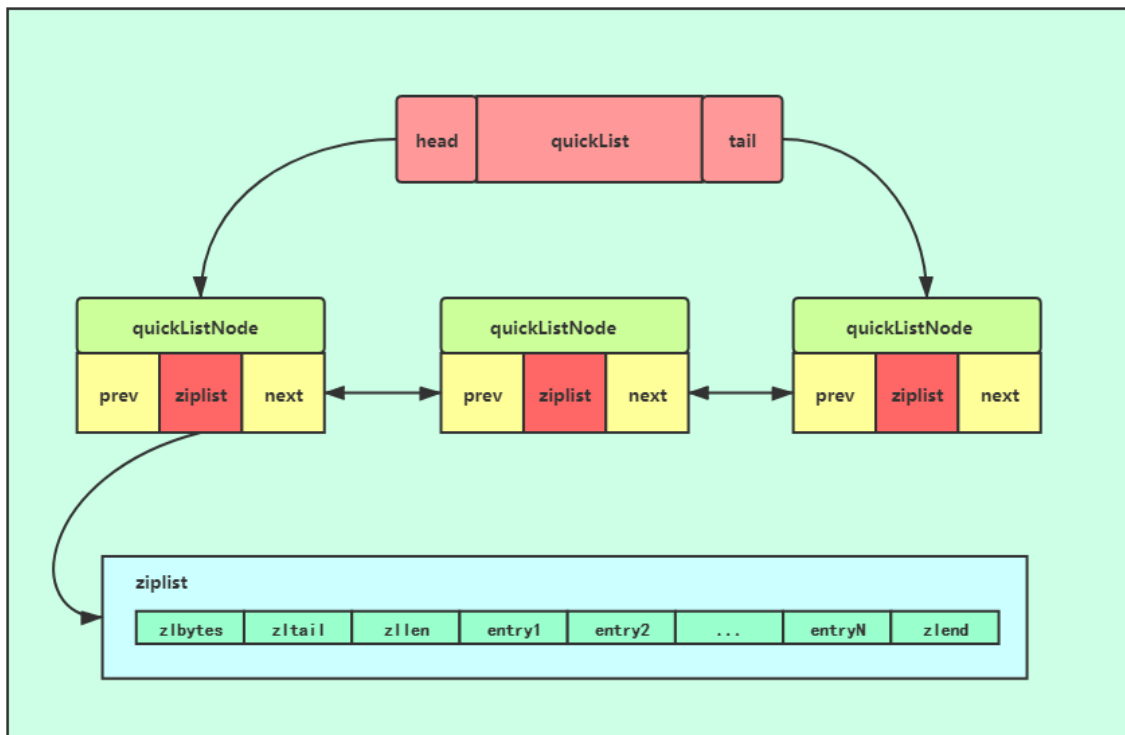
```

1 hash-max-ziplist-entries 512 //entry的总数不能大于512，否则就采用hashmap/链表/跳跃表结构
  存储
2 hash-max-ziplist-value 64 //每个entry值的长度不能大于64个字节，
3 list-max-ziplist-entries 512
4 list-max-ziplist-value 64
5 zset-max-ziplist-entries 128
6 zset-max-ziplist-value 64

```

快速列表quicklist

由于链表的附加空间太高，链表节点的内存分配容易造成碎片，Redis3.2版本开始对列表数据结构进行了改造，使用 `quicklist` 代替了 `ziplist` 和 `linkedlist`。`quicklist` 实际上是 `ziplist` 和 `LinkedList` 的混合体，它将 `LinkedList` 按段切分，每一段使用 `ziplist` 来紧凑存储，多个 `ziplist` 之间使用双向指针串接起来。



```

1 typedef struct quicklist {
2     quickListNode *head; //指向头节点（左侧第一个节点）的指针。
3     quickListNode *tail; //指向尾节点（右侧第一个节点）的指针。
4     unsigned long count; //所有ziplist数据项的个数总和。
5     unsigned long len; // quicklist节点的个数。
6     int fill : QL_FILL_BITS; //16bit, ziplist大小设置，存放list-max-ziplist-size参数的
    值。
7     unsigned int compress : QL_COMP_BITS; //16bit, 节点压缩深度设置，存放list-
    compress-depth参数的值。
8     unsigned int bookmark_count: QL_BM_BITS;
9     quicklistBookmark bookmarks[];
10 } quicklist;
11
12 typedef struct quickListNode {
13     struct quickListNode *prev; //上一个node节点
14     struct quickListNode *next; //下一个node
15     unsigned char *zl; //当前节点数据指针，指向ziplist或者quicklistLZF
16     unsigned int sz; //表示zl指向的ziplist的总大小，即使被压缩也还是压缩前的大小
17     unsigned int count : 16; //ziplist里面包含的数据项个数

```

```

18     unsigned int encoding : 2; //是否压缩RAW==1 or LZF==2
19     unsigned int container : 2; /* NONE==1 or ZIPLIST==2 */
20     unsigned int recompress : 1; /* was this node previous compressed? */
21     unsigned int attempted_compress : 1; /* node can't compress; too small */
22     unsigned int extra : 10; /* more bits to steal for future usage */
23 } quickListNode;
24
25 typedef struct quicklistLZF {
26     unsigned int sz; //压缩后的ziplist大小
27     char compressed[]; //存放压缩后的ziplist字节数组
28 } quicklistLZF

```

quickListNode 数量

过多的 quickListNode 容易造成许多无法使用的内存碎片，从而降低存储效率。过少的 quickListNode 又会造成 zipList 过长在调整时降低效率。因此在不同的场景下 zipList 应该有不同的长度，可以通过适当的配置参数 list-max-ziplist-size 进行更改。

参数值	x>0	-5	-4	-3	-2（默认）	-1
每节点ziplist长度	最多x个数 据项	总长 <64KB	总长 <32KB	总长 <16KB	总长 <8KB	总长 <4KB

LZF压缩

当列表中间部分很少被访问时、可以通过参数 list-compress-depth = x 配置启用LZF算法以 quickListNode 为单位进行无损压缩，无论何种配置列表两端的 quickListNode 都不会被压缩。数值x代表两端各有x个节点不压缩，数值0（默认值）代表都不压缩。

CLI

服务相关

```

1 select db_index //选择数据库，一共有16个数据库，默认在数据库0
2 dbsize //返回当前数据库key数量
3 quite //关闭与服务器连接退出客户端
4 ping //用于测试与服务器端连接是否生效，返回pong
5 echo message //打印消息，测试用
6 info //获取统计信息
7 monitor //监视
8 config get|set|resetstat|rewrite //获取/修改配置信

```

持久化

Snapshotting(快照RDB)

每隔一段时间将数据集以二进制压缩文件的形式转存到磁盘上，该动作既可以手动触发也可以自动触发。在Redis启动的时候，优先使用AOF恢复数据库，不开启AOF时只要检测到RDB文件的存在，就会自动加载RDB文件。


```
1  #自动触发
2  save x y#x秒内有超过y个key被更改就启动bgsave
3
4  #手动触发
5  save#阻塞当前Redis服务器，直到RDB过程完成为止
6  bgsave#创建一个子进程(而不是线程)进行RDB文件创建，父进程继续处理命令。
```

快照基于COW实现了数据的正确性（T时刻开始耗费t时间完成的快照数据是库在T时刻的状态，而非T+x时刻的状态）。

Append-only file(AOF)

通过保存 Redis 服务器所执行的写命令来记录数据库状态的。即每执行一个命令，Redis 在执行命令将数据写入内存之后再该命令写到日志文件里（避免写入错误的命令）。

AOF缓冲区

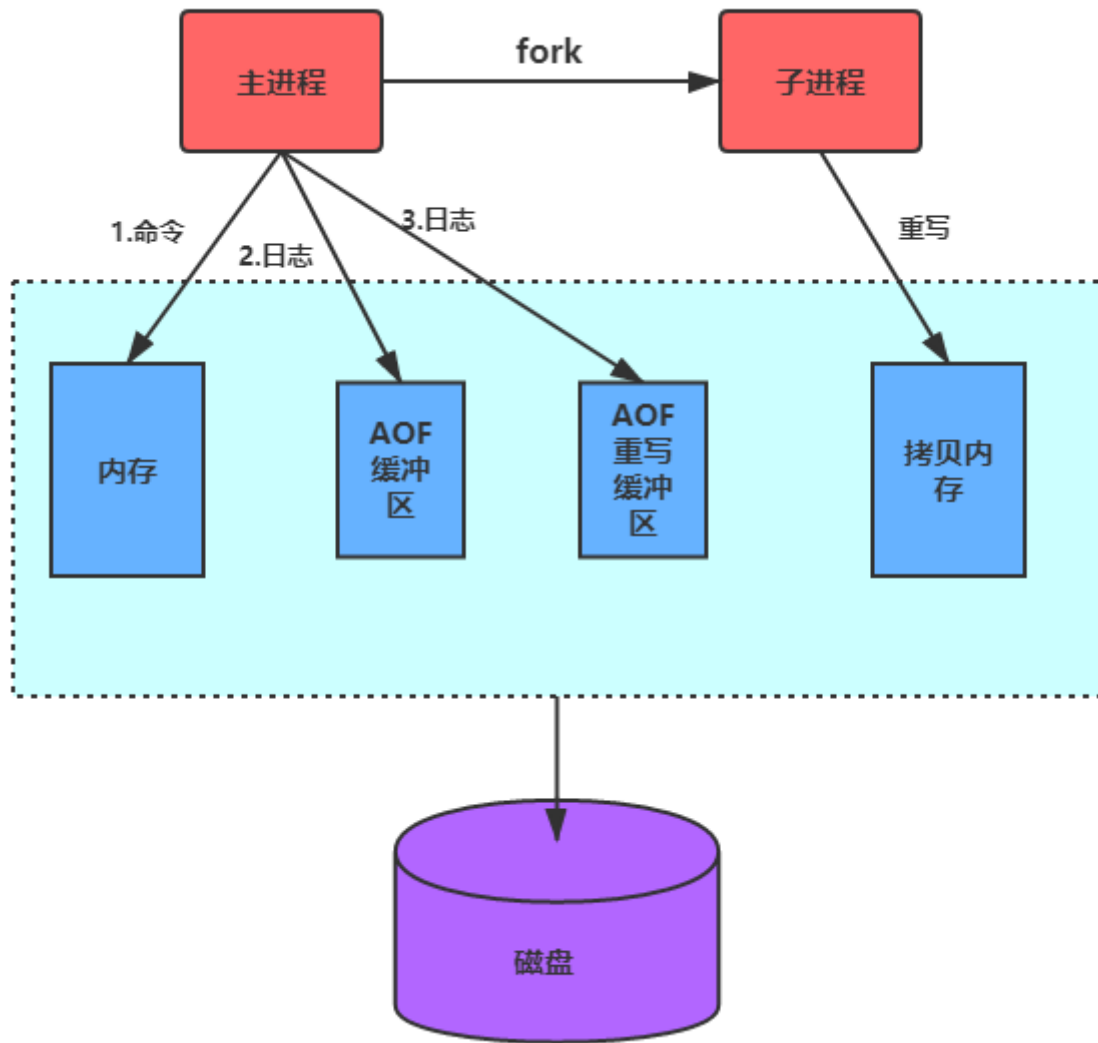
为了避免单进程下写日志阻塞后续的执行和减少数据丢失的风险，Redis 在执行完命令进行持久化的时候，并非直接写入磁盘日志文件，而是先写入 AOF 缓冲区内，之后再通过某种策略写到磁盘。

回写策略	动作
Always(同步写回)	命令写入 AOF 缓冲区后立即调用系统 fsync 同步到 AOF 文件，fsync 完成后线程返回
Everysec(每秒写回)	命令写入 AOF 缓冲区后调用系统 write 操作，write 完成后线程返回。fsync 同步文件操作由专门线程每秒调用一次
No(操作系统自动写回)	命令写入 AOF 缓冲区后调用系统 write 操作，不对 AOF 文件做 fsync 同步，同步硬盘操作由操作系统负责，通常同步周期最长30秒

AOF重写

AOF 文件是以追加的方式，逐一记录接收到的写命令的。当一个键值对被多条写命令反复修改时，前面命令可能对最终结果没有任何影响，因此不必记录所有的命令。

```
1  auto-aof-rewrite-min-size 64MB    // 当文件小于64M时不进行重写
2  auto-aof-rewrite-min-percentage 100 // 当文件比上次重写后的文件大100%时进行重写
```



同内存快照一样，AOF 重写的过程也是由父进程 fork 出 `bgrewriteaof` 子进程来完成的，在子进程执行 AOF 重写期间，服务器进程需要执行三个动作（执行客户端命令、执行命令后追加到 AOF 缓冲区、执行后追加到 AOF 重写缓冲区）。子进程完成 AOF 重写后，它向父进程发送一个信号，父进程收到信号后会调用一个信号处理函数（重写中唯一阻塞服务器进程），该函数把 AOF 重写缓冲区的命令追加到新 AOF 文件中然后替换掉现有 AOF 文件。父进程处理完毕后可以继续接受客户端命令调用。

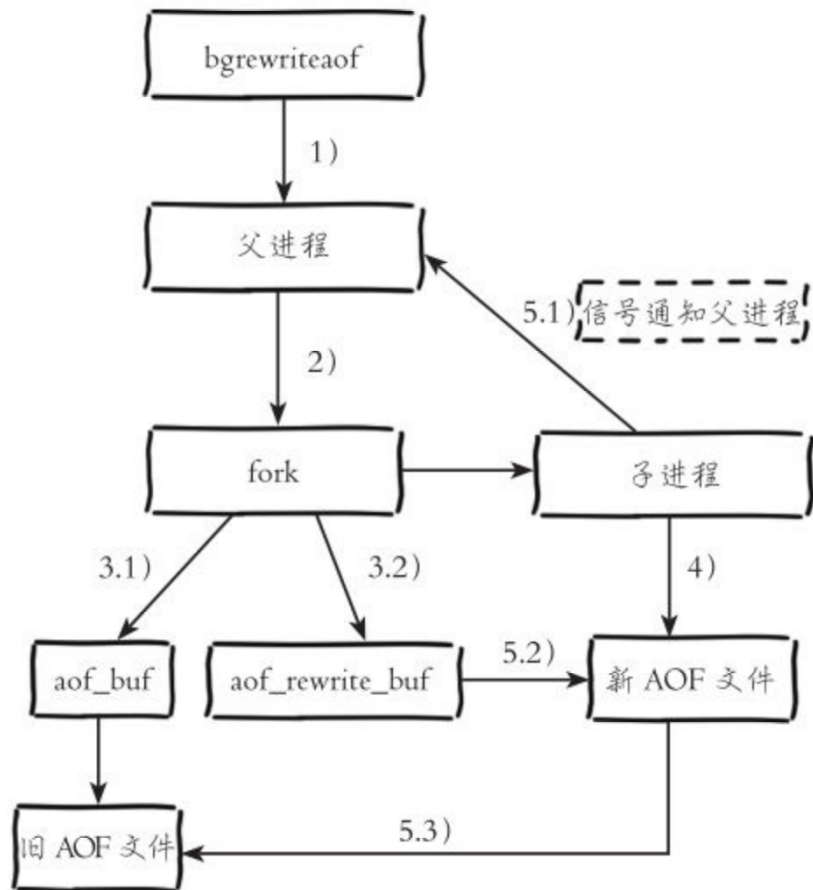


图 5-3 AOF 重写运作流程

<https://blog.csdn.net/Andy66669>

<https://blog.csdn.net/a147bb/article/details/106813604>

<https://www.cnblogs.com/yuhangwang/p/5817930.html>