

# 数据结构和对象

键：字符串

值：字符串、列表、哈希、集合、有序集合

## 简单动态字符串

### 1.

创建了一种名为简单动态字符串（SDS），不同于C中的字符串。c字符只用于字符串字面量，用在一些无需对字符串进行修改的地方，比如打印日志。

```
redisLog(REDIS_WARNING, "Redis is now ready to exit, bye bye...");
```

### 2.

AOF缓冲区也是SDS实现的。

## 定义

### 3、SDS遵循C字符串以空字符结尾

```
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
    char buf[];
};
```

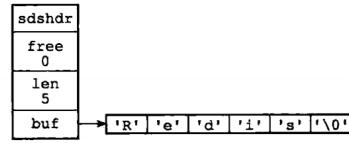


图 2-1 SDS 示例

### 4、sds与C的区别

4.1、**sds常数复杂度获取字符串长度**。获取字符串长度不会成为性能瓶颈 `strlen` 复杂度为  $O(1)$

4.2、**杜绝缓冲区溢出**。当SDS API需要对sds进行修改时，会先检查sds空间是否满足要求。**空间分配策略**。

4.3、**减少修改字符串时带来的内存重新分配次数**。c中增长字符串，必须重新分配，否则会产生缓冲区溢出、截取字符串，不分配会导致内存泄露。**sds通过free空间解除了字符串长度和底层数组长度的关联，实现了空间预分配和惰性空间释放两种优化策略**。

4.3.1、**空间预分配**：优化sds增长操作。小于1MB，分配两倍free。大于1MB，分配1MB free空间。

4.3.2、**惰性空间释放**：优化sds截短操作。不回收，使用free字段进行记录

4.4、**二进制安全**。通过len属性的值而不是空字符串来判断字符串是否结束，所以可以保存任意格式的二进制数据。

4.5、**兼容部分c字符串函数**。虽然时二进制安全的，但是还会分配一个字符存储'\0'空字符，为了可以使用`<string.h>`的函数

表 2-1 C 字符串和 SDS 之间的区别

C 字符串	SDS
获取字符串长度的复杂度为 $O(N)$	获取字符串长度的复杂度为 $O(1)$
API 是不安全的，可能会造成缓冲区溢出	API 是安全的，不会造成缓冲区溢出
修改字符串长度 N 次必然需要执行 N 次内存重分配	修改字符串长度 N 次最多需要执行 N 次内存重分配
只能保存文本数据	可以保存文本或者二进制数据
可以使用所有 <code>&lt;string.h&gt;</code> 库中的函数	可以使用一部分 <code>&lt;string.h&gt;</code> 库中的函数

## 5、SDS API

sdsnew、sdsfree、sdslen、

表 2-2 SDS 的主要操作 API

函 数	作 用	时间复杂度
sdsnew	创建一个包含给定 C 字符串的 SDS	$O(N)$ , N 为给定 C 字符串的长度
sdsempty	创建一个不包含任何内容的空 SDS	$O(1)$
sdsfree	释放给定的 SDS	$O(N)$ , N 为被释放 SDS 的长度
sdslen	返回 SDS 的已使用空间字节数	这个值可以通过读取 SDS 的 len 属性来直接获得, 复杂度为 $O(1)$
sdsavail	返回 SDS 的未使用空间字节数	这个值可以通过读取 SDS 的 free 属性来直接获得, 复杂度为 $O(1)$
sdsdup	创建一个给定 SDS 的副本 (copy)	$O(N)$ , N 为给定 SDS 的长度
sdsclear	清空 SDS 保存的字符串内容	因为惰性空间释放策略, 复杂度为 $O(1)$
sdscat	将给定 C 字符串拼接到 SDS 字符串的末尾	$O(N)$ , N 为被拼接 C 字符串的长度
sdscatsds	将给定 SDS 字符串拼接到另一个 SDS 字符串的末尾	$O(N)$ , N 为被拼接 SDS 字符串的长度
sdscopy	将给定的 C 字符串复制到 SDS 里面, 覆盖 SDS 原有的字符串	$O(N)$ , N 为被复制 C 字符串的长度
sdsgrowzero	用空字符将 SDS 扩展至给定长度	$O(N)$ , N 为扩展新增的字节数
sdsrange	保留 SDS 给定区间内的数据, 不在区间内的数据会被覆盖或清除	$O(N)$ , N 为被保留数据的字节数
sdstrim	接受一个 SDS 和一个 C 字符串作为参数, 从 SDS 中移除所有在 C 字符串中出现过的字符	$O(N^2)$ , N 为给定 C 字符串的长度
sdscmp	对比两个 SDS 字符串是否相同	$O(N)$ , N 为两个 SDS 中较短的那个 SDS 的长度

## 6、sds总结

- Redis 只会使用 C 字符串作为字面量, 在大多数情况下, Redis 使用 SDS ( Simple Dynamic String, 简单动态字符串 ) 作为字符串表示。
- 比起 C 字符串, SDS 具有以下优点:
  - 1 ) 常数复杂度获取字符串长度。
  - 2 ) 杜绝缓冲区溢出。
  - 3 ) 减少修改字符串长度时所需的内存重分配次数。
  - 4 ) 二进制安全。
  - 5 ) 兼容部分 C 字符串函数。

## 链表

### 7、链表和链表节点的实现

```

typedef struct list {
    // 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
    void *(*dup)(void *ptr);

    // 节点值释放函数
    void (*free)(void *ptr);

    // 节点值对比函数
    int (*match)(void *ptr, void *key);
}

双端列表、} list;

```

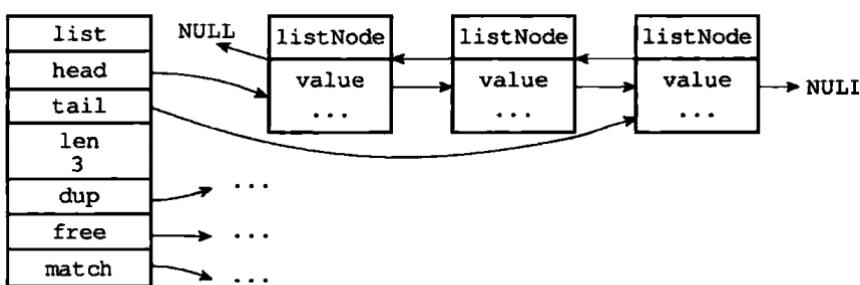


图 3-2 由 list 结构和 listNode 结构组成的链表

**特性总结：** 双端、无环、表尾指针和表头指针、链表长度计数器、多态：链表节点使用void\*指针保存节点值，并且可以通过dup、free、match、三个属性为节点设置类型特定的函数，所以链表可以用于保存各种不同的值。

## 8、链表节点API

表 3-1 链表和链表节点 API

函数	作用	时间复杂度
listSetDupMethod	将给定的函数设置为链表的节点值复制函数	复制函数可以通过链表的 dup 属性直接获得, $O(1)$
listGetDupMethod	返回链表当前正在使用的节点值复制函数	$O(1)$
listSetFreeMethod	将给定的函数设置为链表的节点值释放函数	释放函数可以通过链表的 free 属性直接获得, $O(1)$
listGetFree	返回链表当前正在使用的节点值释放函数	$O(1)$
listSetMatchMethod	将给定的函数设置为链表的节点值对比函数	对比函数可以通过链表的 match 属性直接获得, $O(1)$
listGetMatchMethod	返回链表当前正在使用的节点值对比函数	$O(1)$

<code>listLength</code>	返回链表的长度（包含了多少个节点）	链表长度可以通过链表的 <code>len</code> 属性直接获得， $O(1)$
<code>listFirst</code>	返回链表的表头节点	表头节点可以通过链表的 <code>head</code> 属性直接获得， $O(1)$
<code>listLast</code>	返回链表的表尾节点	表尾节点可以通过链表的 <code>tail</code> 属性直接获得， $O(1)$
<code>listPrevNode</code>	返回给定节点的前置节点	前置节点可以通过节点的 <code>prev</code> 属性直接获得， $O(1)$
<code>listNextNode</code>	返回给定节点的后置节点	后置节点可以通过节点的 <code>next</code> 属性直接获得， $O(1)$
<code>listNodeValue</code>	返回给定节点目前正在保存的值	节点值可以通过节点的 <code>value</code> 属性直接获得， $O(1)$
<code>listCreate</code>	创建一个不包含任何节点的新链表	$O(1)$
<code>listAddNodeHead</code>	将一个包含给定值的新节点添加到给定链表的表头	$O(1)$
<code>listAddNodeTail</code>	将一个包含给定值的新节点添加到给定链表的表尾	$O(1)$
<code>listInsertNode</code>	将一个包含给定值的新节点添加到给定节点之前或者之后	$O(1)$
<code>listSearchKey</code>	查找并返回链表中包含给定值的节点	$O(N)$ , N 为链表长度
<code>listIndex</code>	返回链表在给定索引上的节点	$O(N)$ , N 为链表长度
<code>listDelNode</code>	从链表中删除给定节点	$O(N)$ , N 为链表长度
<code>listRotate</code>	将链表的表尾节点弹出，然后将被弹出的节点插入到链表的表头，成为新的表头节点	$O(1)$
<code>listDup</code>	复制一个给定链表的副本	$O(N)$ , N 为链表长度

## 9、链表总结

- 链表被广泛用于实现 Redis 的各种功能，比如列表键、发布与订阅、慢查询、监视器等。
- 每个链表节点由一个 `listNode` 结构来表示，每个节点都有一个指向前置节点和后置节点的指针，所以 Redis 的链表实现是双端链表。
- 每个链表使用一个 `list` 结构来表示，这个结构带有表头节点指针、表尾节点指针，以及链表长度等信息。
- 因为链表表头节点的前置节点和表尾节点的后置节点都指向 `NULL`，所以 Redis 的链表实现是无环链表。
- 通过为链表设置不同的类型特定函数，Redis 的链表可以用于保存各种不同类型的值。

## 字典

C语言没有内置的自己的数据结构、redis实现了自己的字典。redis的数据库就是使用字典作为底层实现的，数据库的增删改查都是基于字典。字典还是哈希键的底层实现之一。

## 10、字典实现

### 10.1、哈希表

使用哈希表作为底层实现，哈希表里的哈希节点保存了字典中的一个键值对。

```
typedef struct dictht {
    // 哈希表数组
    dictEntry **table;
    // 哈希表大小
    unsigned long size;
    // 哈希表大小掩码，用于计算索引值
    // 总是等于 size-1
    unsigned long sizemask;
    // 该哈希表已有节点的数量
    unsigned long used;
}
```

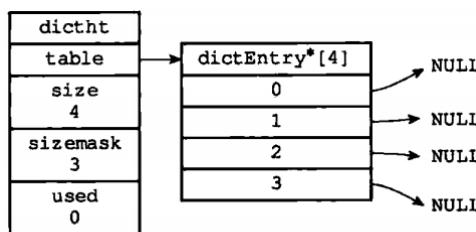


图 4-1 一个空的哈希表

table是一个数组，数组中每个元素都是一个指向dictEntry结构的指针，每个dictEntry的指针保存一个键值对。

## 10.2、哈希表节点

节点使用dictEntry，每个dictEntry的指针保存一个键值对。

```
typedef struct dictEntry {
```

```
    // 键  
    void *key;
```

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

```
    // 指向下个哈希表节点，形成链表  
    struct dictEntry *next;
```

```
} dictEntry;
```

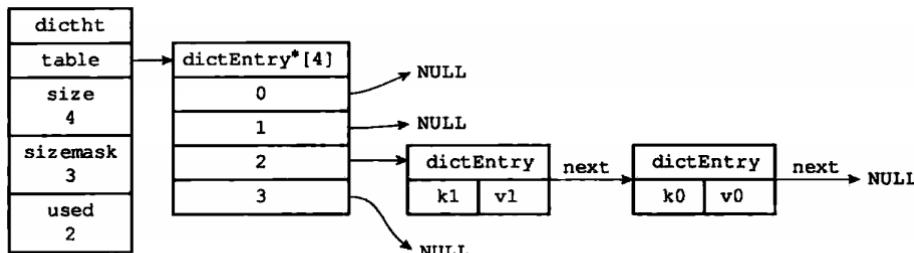


图 4-2 连接在一起的键 K1 和键 K0

next指针解决哈希冲突。

## 10.3、字典

```
typedef struct dict {  
    // 类型特定函数  
    dictType *type;  
  
    // 私有数据  
    void *privdata;  
  
    // 哈希表  
    dictht ht[2];  
  
    // rehash 索引  
    // 当 rehash 不在进行时，值为 -1  
    int trehashidx; /* rehashing not in progress if rehashidx == -1 */  
  
} dict;  
  
typedef struct dictType {  
    // 计算哈希值的函数  
    unsigned int (*hashFunction)(const void *key);  
  
    // 复制键的函数  
    void *(*keyDup)(void *privdata, const void *key);  
  
    // 复制值的函数  
    void *(*valDup)(void *privdata, const void *obj);  
  
    // 对比键的函数  
    int (*keyCompare)(void *privdata, const void *key1, const void *key2);  
  
    // 销毁键的函数  
    void (*keyDestructor)(void *privdata, void *key);  
  
    // 销毁值的函数  
    void (*valDestructor)(void *privdata, void *obj);  
}  
dictType;
```

**type** 和 **privdata** 针对不同类型的键值对，为创建**多态字典**设置的。

**dictType** 保存一族用于操作特定类型键值对的函数，**privdata** 就是该函数的可选参数。

ht属性包含两个人项的数组，每个项都是一个哈希表，一般指使用ht[0]，ht[1]只在rehash时使用。

**rehashidx** 记录目前rehash的进度，没有则为-1

图 4-3 展示了一个普通状态下（没有进行 rehash）的字典。

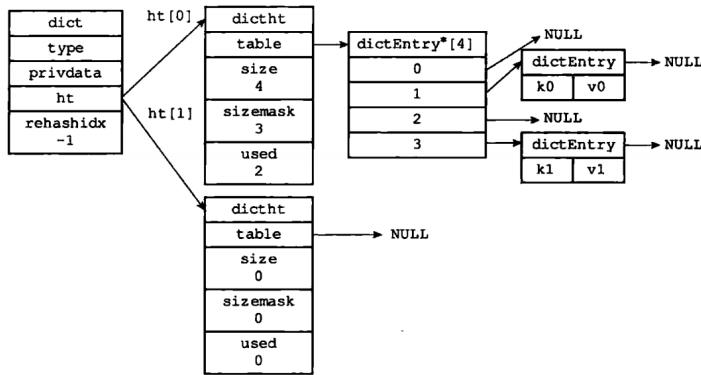


图 4-3 普通状态下的字典

## 11、哈希算法

如果将一个键值对 k0, v0 添加到字典里面：

```
hash = dict->type->hashFunction(k0);
```

计算键 k0 的哈希值。

假设计算得出的哈希值为 8，那么程序会继续使用语句：

```
index = hash&dict->ht[0].sizemask = 8 & 3 = 0;
```

计算出键 k0 的索引值 0，这表示包含键值对 k0 和 v0 的节点应该被放置到哈希表数组索引 0 位置上，如图 4-5 所示。

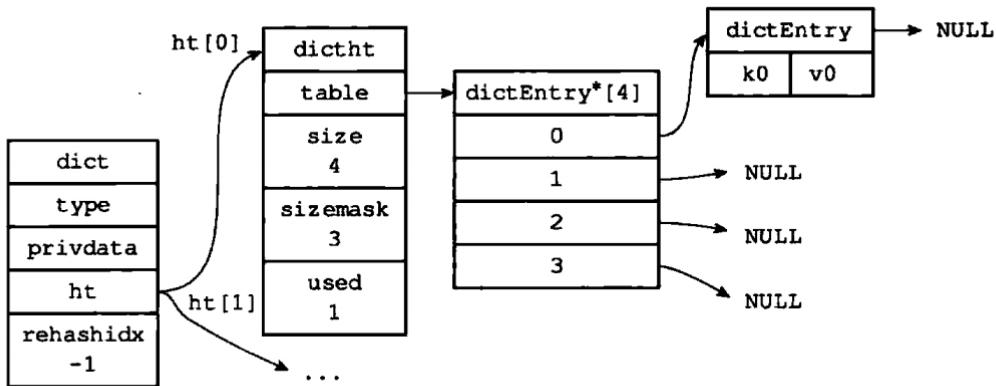


图 4-5 添加键值对 k0 和 v0 之后的字典

使用的哈希算法是 MurmurHash2.

## 11、哈希冲突

使用链地址法。

## 12、rehash

使得负载因子合理范围，必须对哈希表大小进行扩展或者收缩，通过 rehash (重新散列) 实现。

步骤如下：

1、为 ht[1] 分配空间：如果是扩展，第一个  $\geq ht[0].used \times 2$  的 2 的 n 次幂；如果是收缩， $ht[1] \geq$  第一个  $ht[0].used \times 2$  的 n 次幂。

2、迁移 ht0->ht1

3、释放 ht0，将 ht1 设置为空白。

## 13、什么时候扩展或者收缩哈希表

- 1、没有执行BGSAVE或者BgREWTRITEAOF，且负载因子 $\geq 1$
- 2、正在执行BGSAVE或者BgREWTRITEAOF，且负载因子 $\geq 5$

# 负载因子 = 哈希表已保存节点数量 / 哈希表大小

负载因子计算：  
load\_factor = ht[0].used / ht[0].size

执行BGSAVE或者BgREWTRITEAOF时候会有子进程进行写时复制，提高负载因子阈值，避免不必要的内存写入内存。

负载因子 $<0.1$ 自动收缩

## 14、渐进式rehash

扩展或者收缩哈希表需要使用rehash将ht0->ht1，这个步骤是分多次，渐进式的完成。

详细步骤：

- 1) 为 ht[1] 分配空间，让字典同时持有 ht[0] 和 ht[1] 两个哈希表。
- 2) 在字典中维持一个索引计数器变量 rehashidx，并将它的值设置为 0，表示 rehash 工作正式开始。
- 3) 在 rehash 进行期间，每次对字典执行添加、删除、查找或者更新操作时，程序除了执行指定的操作以外，还会顺带将 ht[0] 哈希表在 rehashidx 索引上的所有键值对 rehash 到 ht[1]，当 rehash 工作完成之后，程序将 rehashidx 属性的值增一。
- 4) 随着字典操作的不断执行，最终在某个时间点上，ht[0] 的所有键值对都会被 rehash 至 ht[1]，这时程序将 rehashidx 属性的值设为 -1，表示 rehash 操作已完成。

渐进式 rehash 的好处在于它采取分而治之的方式，将 rehash 键值对所需的计算工作均摊到对字典的每个添加、删除、查找和更新操作上，从而避免了集中式 rehash 而带来的庞大计算量。

在进行渐进式rehash时=，字典会使用ht0和ht1，字典的增删改查都会在两个哈希表同时进行。只在ht1中保存，现在ht0中查询

## 15、字典 API

函 数	作 用	时间复杂度
dictCreate	创建一个新的字典	O(1)
dictAdd	将给定的键值对添加到字典里面	O(1)
dictReplace	将给定的键值对添加到字典里面，如果键已经存在于字典，那么用新值取代原有的值	O(1)
dictFetchValue	返回给定键的值	O(1)
dictGetRandomKey	从字典中随机返回一个键值对	O(1)
dictDelete	从字典中删除给定键所对应的键值对	O(1)
dictRelease	释放给定字典，以及字典中包含的所有键值对	O(N), N 为字典包含的键值对数量

## 16、字典总结

- 字典被广泛用于实现 Redis 的各种功能，其中包括数据库和哈希键。
- Redis 中的字典使用哈希表作为底层实现，每个字典带有两个哈希表，一个平时使用，另一个仅在进行 `rehash` 时使用。
- 当字典被用作数据库的底层实现，或者哈希键的底层实现时，Redis 使用 MurmurHash2 算法来计算键的哈希值。
- 哈希表使用链地址法来解决键冲突，被分配到同一个索引上的多个键值对会连接成一个单向链表。
- 在对哈希表进行扩展或者收缩操作时，程序需要将现有哈希表包含的所有键值对 `rehash` 到新哈希表里面，并且这个 `rehash` 过程并不是一次性地完成的，而是渐进式地完成的。

## 跳跃表（跳表）

有序数据结构，每个节点维护多个指向其他节点的指针，达到快速访问节点的目的。最好 $\log n$ ，最坏 $n$ 。大部分情况可以和平衡树媲美。

作为有序集合键的底层实现之一。

**用途：**有序集合键、集群节点中用作内部数据结构

### 17、跳跃表的实现

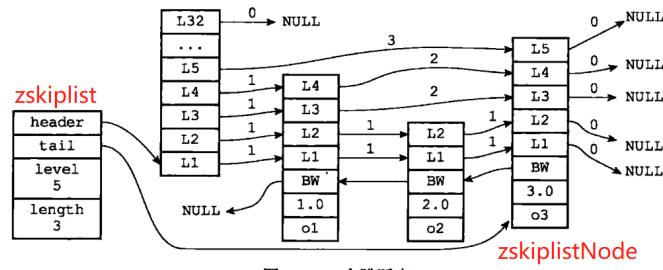


图 5-1 一个跳跃表

- **header:** 指向跳跃表的表头节点。
- **tail:** 指向跳跃表的表尾节点。
- **level:** 记录目前跳跃表内，层数最大的那个节点的层数（表头节点的层数不计算在内）。
- **length:** 记录跳跃表的长度，也即是，跳跃表目前包含节点的数量（表头节点不计算在内）。

位于 `zskiplist` 结构右方的是四个 `zskiplistNode` 结构，该结构包含以下属性：

- **层 (level):** 节点中用 `L1`、`L2`、`L3` 等字样标记节点的各个层，`L1` 代表第一层，`L2` 代表第二层，以此类推。每个层都带有两个属性：前进指针和跨度。前进指针用于访问位于表尾方向的其他节点，而跨度则记录了前进指针所指向节点和当前节点的距离。在上面的图片中，连线上带有数字的箭头就代表前进指针，而那个数字就是跨度。当程序从表头向表尾进行遍历时，访问会沿着层的前进指针进行。
- **后退 (backward) 指针:** 节点中用 `BW` 字样标记节点的后退指针，它指向位于当前节点的前一个节点。后退指针在程序从表尾向表头遍历时使用。

- **分值 (score):** 各个节点中的 `1.0`、`2.0` 和 `3.0` 是节点所保存的分值。在跳跃表中，节点按各自所保存的分值从小到大排列。

- **成员对象 (obj):** 各个节点中的 `o1`、`o2` 和 `o3` 是节点所保存的成员对象。

注意表头节点和其他节点的构造是一样的：表头节点也有后退指针、分值和成员对象，不过表头节点的这些属性都不会被用到，所以图中省略了这些部分，只显示了表头节点的各个层。

本节接下来的内容将对 `zskiplistNode` 和 `zskiplist` 两个结构进行更详细的介绍。

#### 17.1、跳跃表节点

跳跃表节点的实现由 `redis.h/zskiplistNode` 结构定义：

```
typedef struct zskiplistNode {
    // 层
    struct zskiplistLevel {
        // 前进指针
        struct zskiplistNode *forward;
        // 跨度
        unsigned int span;
    } level[];
    // 后退指针
    struct zskiplistNode *backward;
    // 分值
    double score;
    // 成员对象
    robj *obj;
} zskiplistNode;
```

跳跃表节点： `zskiplistNode`;

**层**：level数组可以包含多个元素，每个元素指向其他节点的指针，通过这些层提高访问速度，**层数越多，访问其他节点就越快**

创建新节点，根据幂次定律（数越大，出现概率越小）随机生成1-32的值作为level数组大小。

**前进指针**：从表头到表尾访问节点。可以一次跳过多个节点

**跨度**：记录两个节点之间的距离：跨度越大距离越远。跨度只用来计算排位

**后退节点**：从表尾到表头访问节点。每次只能后退一个节点。

**分值和成员**：分值：double类型，按照分值排序；成员对象：指针，指向字符串对象，保存一个sds对象

**同一跳跃表，成员对象唯一，分值可以相同**

## 18、跳跃表API

函数	作用	时间复杂度
<code>zslCreate</code>	创建一个新的跳跃表	$O(1)$
<code>zslFree</code>	释放给定跳跃表，以及表中包含的所有节点	$O(N)$ , $N$ 为跳跃表的长度
<code>zslInsert</code>	将包含给定成员和分值的新节点添加到跳跃表中	平均 $O(\log N)$ , 最坏 $O(N)$ , $N$ 为跳跃表长度
<code>zslDelete</code>	删除跳跃表中包含给定成员和分值的节点	平均 $O(\log N)$ , 最坏 $O(N)$ , $N$ 为跳跃表长度
<code>zslGetRank</code>	返回包含给定成员和分值的节点在跳跃表中的排位	平均 $O(\log N)$ , 最坏 $O(N)$ , $N$ 为跳跃表长度
<code>zslGetElementByRank</code>	返回跳跃表在给定排位上的节点	平均 $O(\log N)$ , 最坏 $O(N)$ , $N$ 为跳跃表长度
<code>zslIsInRange</code>	给定一个分值范围 ( <code>range</code> )，比如 0 到 15, 20 到 28，诸如此类，如果跳跃表中有至少一个节点的分值在这个范围之内，那么返回 1，否则返回 0	通过跳跃表的表头节点和表尾节点，这个检测可以用 $O(1)$ 复杂度完成
<code>zslFirstInRange</code>	给定一个分值范围，返回跳跃表中第一个符合这个范围的节点	平均 $O(\log N)$ , 最坏 $O(N)$ , $N$ 为跳跃表长度
<code>zslLastInRange</code>	给定一个分值范围，返回跳跃表中最后一个符合这个范围的节点	平均 $O(\log N)$ , 最坏 $O(N)$ , $N$ 为跳跃表长度
<code>zslDeleteRangeByScore</code>	给定一个分值范围，删除跳跃表中所有在这个范围之内的节点	$O(N)$ , $N$ 为被删除节点数量
<code>zslDeleteRangeByRank</code>	给定一个排位范围，删除跳跃表中所有在这个范围之内的节点	$O(N)$ , $N$ 为被删除节点数量

## 19、跳跃表总结

- 跳跃表是有序集合的底层实现之一。
- Redis 的跳跃表实现由 `zskiplist` 和 `zskiplistNode` 两个结构组成，其中 `zskiplist` 用于保存跳跃表信息（比如表头节点、表尾节点、长度），而 `zskiplistNode` 则用于表示跳跃表节点。
- 每个跳跃表节点的层高都是 1 至 32 之间的随机数。
- 在同一个跳跃表中，多个节点可以包含相同的分值，但每个节点的成员对象必须是唯一的。
- 跳跃表中的节点按照分值大小进行排序，当分值相同时，节点按照成员对象的大小进行排序。

## 整数集合

集合键的底层实现之一：集合只包含整数值，并且元素不多时，使用整数集合作为集合键的底层实现。

## 20、整数集合的实现

保证集合中不出现重复元素，int16\_t、int32\_t、int64\_t

```
typedef struct intset {
    // 编码方式
    uint32_t encoding;
    // 集合包含的元素数量
    uint32_t length;
    // 保存元素的数组
    int8_t contents[];
} intset;
```

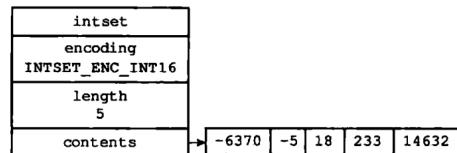


图 6-1 一个包含五个 int16\_t 类型整数值的整数集合

**content**数组是整数集合的底层实现：整数集合每个元素都是该数组的一个数组项，有序不重复。

该数组真正的类型取决于encoding，并不是int8\_t。

int16\_t的数组插入一个int64\_t类型的数，会导致**升级**

## 21、整数集合升级

- 1、根据新元素类型，扩展整数集合底层数组的空间大小，并为新元素分配空间
- 2、将底层数组现有的所有元素都转换成与新元素相同的类型，并将转换后的元素按照有序放在合适的位置
- 3、将新元素放入底层数组

时间复杂度为  $O(n)$

优点：1.提升灵活性：C语言是静态语言，为避免类型错误，一般不会将两种不同类型的值放在同一数据结构里面。但是升级可以放。

2.节约内存：只有用到64位时才会占用64位的空间

**注：不支持降级操作**

## 22、整数集合api

函数	作用	时间复杂度
intsetNew	创建一个新的压缩列表	$O(1)$
intsetAdd	将给定元素添加到整数集合里面	$O(N)$
intsetRemove	从整数集合中移除给定元素	$O(N)$
intsetFind	检查给定值是否存在于集合	因为底层数组有序，查找可以通过二分查找法来进行，所以复杂度为 $O(\log N)$
intsetRandom	从整数集合中随机返回一个元素	$O(1)$
intsetGet	取出底层数组在给定索引上的元素	$O(1)$
intsetLen	返回整数集合包含的元素个数	$O(1)$
intsetBlobLen	返回整数集合占用的内存字节数	$O(1)$

## 23、整数集合总结

- 整数集合是集合键的底层实现之一。
- 整数集合的底层实现为数组，这个数组以有序、无重复的方式保存集合元素，在有需要时，程序会根据新添加元素的类型，改变这个数组的类型。
- 升级操作为整数集合带来了操作上的灵活性，并且尽可能地节约了内存。
- 整数集合只支持升级操作，不支持降级操作。

## 压缩列表

列表键和哈希键的底层实现之一。当列表键只包含小整数或者短字符串。当哈希键只包含少量键值对，每个键值对的键和值是小整数或者短字符串

## 24、压缩列表的构成

特殊编码的连续内存块组成的顺序性数据结构。一个压缩列表可以包含多个节点，每个节点可以保存一个字节数组或者一个整数值

zbytes	ztail	zlen	entry1	entry2	...	entryN	zend
--------	-------	------	--------	--------	-----	--------	------

图 7-1 压缩列表的各个组成部分

属性	类型	长度	用途
zbytes	uint32_t	4 字节	记录整个压缩列表占用的内存字节数：在对压缩列表进行内存重分配，或者计算 zend 的位置时使用
ztail	uint32_t	4 字节	记录压缩列表表尾节点距离压缩列表的起始地址有多少字节：通过这个偏移量，程序无须遍历整个压缩列表就可以确定表尾节点的地址
zlen	uint16_t	2 字节	记录了压缩列表包含的节点数量：当这个属性的值小于 <code>UINT16_MAX</code> (65535) 时，这个属性的值就是压缩列表包含节点的数量；当这个值等于 <code>UINT16_MAX</code> 时，节点的真实数量需要遍历整个压缩列表才能计算得出
entryX	列表节点	不定	压缩列表包含的各个节点，节点的长度由节点保存的内容决定
zend	uint8_t	1 字节	特殊值 0xFF (十进制 255)，用于标记压缩列表的末端

## 25、压缩列表节点的构成

- 4 位长，介于 0 至 12 之间的无符号整数；
- 1 字节长的有符号整数；
- 3 字节长的有符号整数；
- 长度小于等于  $63 (2^6 - 1)$  字节的字节数组；       `int16_t` 类型整数；
- 长度小于等于  $16\,383 (2^{14} - 1)$  字节的字节数组；       `int32_t` 类型整数；
- 长度小于等于  $4\,294\,967\,295 (2^{32} - 1)$  字节的字节数组；       `int64_t` 类型整数。

previous_entry_length	encoding	content
-----------------------	----------	---------

图 7-4 压缩列表节点的各个组成部分

`previous_entry_length`：字节为单位，记录了前一个节点的长度。可以通过当前指针地址减去 `previous_entry_length` 计算出前一个节点的地址。从表尾向表头遍历使用的就是这个原理

`encoding`：记录节点的 `content` 属性保存的数据类型和长度。最高位：00、01、10 表示是字节数组，11 表示是整数编码。长度为出去最高两位的其他位记录。

编 码		编 码 长 度	content 属性保存的值
00bbbbbb		1 字节	长度小于等于 63 字节的字节数组
01bbbbbb xxxxxxxx		2 字节	长度小于等于 16 383 字节的字节数组
10 _____ aaaaaaaaa bbbbbbbb ccccccccc dddddd		5 字节	长度小于等于 4 294 967 295 的字节数组
11000000	1 字节	<code>int16_t</code> 类型的整数	
11010000	1 字节	<code>int32_t</code> 类型的整数	
11100000	1 字节	<code>int64_t</code> 类型的整数	
11110000	1 字节	24 位有符号整数	
11111110	1 字节	8 位有符号整数	
1111xxxx	1 字节	使用这一编码的节点没有相应的 content 属性，因为编码本身的 xxxx 位已经保存了一个介于 0 和 12 之间的值，所以它无须 content 属性	

`content`：保存节点的值

## 26、连锁更新

zbytes	ztail	zlen	new	e1	e2	e3	...	eN	zend
--------	-------	------	-----	----	----	----	-----	----	------

↑  
扩展 e1  
并引发对 e2 的扩展

zbytes	ztail	zlen	new	e1	e2	e3	...	eN	zend
--------	-------	------	-----	----	----	----	-----	----	------

↑  
扩展 e2  
并引发对 e3 的扩展

zbytes	ztail	zlen	new	e1	e2	e3	...	eN	zend
--------	-------	------	-----	----	----	----	-----	----	------

↑  
扩展 e3  
并引发对 e4 的扩展

主要是由于插入或者删除时，编码长度 (`previous_entry_length`) 的不确定导致的每次空间重新分配最低时间复杂度是  $n$ ，所以连锁更新最低时间复杂度是  $n \times n$ 。实际操作中，连锁更新的操作很少。

## 27、压缩列表API

函数	作用	算法复杂度
ziplistNew	创建一个新的压缩列表	$O(1)$
ziplistPush	创建一个包含给定值的新节点，并将这个新节点添加到压缩列表的表头或者表尾	平均 $O(N)$ , 最坏 $O(N^2)$
ziplistInsert	将包含给定值的新节点插入到给定节点之后	平均 $O(N)$ , 最坏 $O(N^2)$
ziplistIndex	返回压缩列表给定索引上的节点	$O(N)$
ziplistFind	在压缩列表中查找并返回包含了给定值的节点	因为节点的值可能是一个字节数组，所以检查节点值和给定值是否相同的复杂度为 $O(N)$ , 而查找整个列表的复杂度则为 $O(N^2)$
ziplistNext	返回给定节点的下一个节点	$O(1)$
ziplistPrev	返回给定节点的前一个节点	$O(1)$
ziplistGet	获取给定节点所保存的值	$O(1)$
ziplistDelete	从压缩列表中删除给定的节点	平均 $O(N)$ , 最坏 $O(N^2)$
ziplistDeleteRange	删除压缩列表在给定索引上的连续多个节点	平均 $O(N)$ , 最坏 $O(N^2)$
ziplistBlobLen	返回压缩列表目前占用的内存字节数	$O(1)$
ziplistLen	返回压缩列表目前包含的节点数量	节点数量小于 65535 时为 $O(1)$ , 大于 65535 时为 $O(N)$

因为 `ziplistPush`、`ziplistInsert`、`ziplistDelete` 和 `ziplistDeleteRange` 四个函数都有可能会引发连锁更新，所以它们的最坏复杂度都是  $O(N^2)$ 。

## 28、压缩列表总结

- 压缩列表是一种为节约内存而开发的顺序型数据结构。
- 压缩列表被用作列表键和哈希键的底层实现之一。
- 压缩列表可以包含多个节点，每个节点可以保存一个字节数组或者整数值。
- 添加新节点到压缩列表，或者从压缩列表中删除节点，可能会引发连锁更新操作，但这种操作出现的几率并不高。

## 对象

redis没有直接使用前面的数据结构实现键值对数据库，而是基于这些数据库创建了一个**对象系统**，包含字符串对象、列表对象、哈希对象、集合对象、有序集合对象五种。

针对不同场景为对象设置不同的数据结构实现，优化效率。

实现了**基于引用计数的内存回收机制、对象共享机制**。

带有**访问时间记录信息**，可以用于计算数据库键的**空转时长**，其应用maxmemory的情况下，优先删除空转时长较大的键。

## 29、对象的类型和编码

键对象-值对象

```
typedef struct redisObject {
    // 类型
    unsigned type:4;
    // 编码
    unsigned encoding:4;
    // 指向底层实现数据结构的指针
    void *ptr;
    // ...
}
```

每个对象都是一个 `redisObject` 结构表示： } robj;

### 29.1、type

类型常量	对象的名称
REDIS_STRING	字符串对象
REDIS_LIST	列表对象
REDIS_HASH	哈希对象
REDIS_SET	集合对象
REDIS_ZSET	有序集合对象

五种类型，对于键来说，总是一个字符串对象，值可以是五种之一

`TYPE` 命令返回的是值对象类型。

## 29.2、编码和底层实现

类型	编码	对象
REDIS_STRING	REDIS_ENCODING_INT	使用整数值实现的字符串对象
REDIS_STRING	REDIS_ENCODING_EMBSTR	使用 embstr 编码的简单动态字符串实现的字符串对象
REDIS_STRING	REDIS_ENCODING_RAW	使用简单动态字符串实现的字符串对象
REDIS_LIST	REDIS_ENCODING_ZIPLIST	使用压缩列表实现的列表对象
REDIS_LIST	REDIS_ENCODING_LINKEDLIST	使用双端链表实现的列表对象
REDIS_HASH	REDIS_ENCODING_ZIPLIST	使用压缩列表实现的哈希对象
REDIS_HASH	REDIS_ENCODING_HT	使用字典实现的哈希对象
REDIS_SET	REDIS_ENCODING_INTSET	使用整数集合实现的集合对象
REDIS_SET	REDIS_ENCODING_HT	使用字典实现的集合对象
REDIS_ZSET	REDIS_ENCODING_ZIPLIST	使用压缩列表实现的有序集合对象
REDIS_ZSET	REDIS_ENCODING_SKIPLIST	使用跳跃表和字典实现的有序集合对象

根据不同的场景为一个对象设置不同的编码。比如：列表对象元素较少时使用压缩列表，较多时使用双端列表

## 30、字符串对象

字符串对象的编码可以是int、raw、embstr。

值	编码
可以用 long 类型保存的整数	int
可以用 long double 类型保存的浮点数	embstr 或者 raw
字符串值，或者因为长度太大而没办法用 long 类型表示的整数，又或者因为长度太大而没办法用 long double 类型表示的浮点数	embstr 或者 raw

长度大于32位使用raw。

raw会分配和回收两次内存，redisobject和sdshdr分别一次而embstr只分配回收一次，分配一块连续的内存。

### 30.1、编码转换

int和embstr会转换成raw

int->raw：通过append命令在int后追加字符串值，将其从int变成raw。

embstr：实际上是只读的，想要修改就必须转换成raw。

### 30.2、字符串命令的实现

SET	使用 int 编码保存值	使用 embstr 编码保存值	使用 raw 编码保存值
GET	拷贝对象所保存的整数值，将这个拷贝转换成字符串值，然后向客户端返回这个字符串值	直接向客户端返回字符串值	直接向客户端返回字符串值
APPEND	将对象转换成 raw 编码，然后按 raw 编码的方式执行此操作	将对象转换成 raw 编码，然后按 raw 编码的方式执行此操作	调用 sdscatlen 函数，将给定字符串追加到现有字符串的末尾
INCRBYFLOAT	取出整数值并将其转换成 long double 类型的浮点数，对这个浮点数进行加法计算，然后将得出的浮点数结果保存起来	取出字符串值并尝试将其转换成 long double 类型的浮点数，对这个浮点数进行加法计算，然后将得出的浮点数结果保存起来。如果字符串值不能被转换成浮点数，那么向客户端返回一个错误	取出字符串值并尝试将其转换成 long double 类型的浮点数，对这个浮点数进行加法计算，然后将得出的浮点数结果保存起来。如果字符串值不能被转换成浮点数，那么向客户端返回一个错误
INCRBY	对整数值进行加法计算，得出的计算结果会作为整数被保存起来	embstr 编码不能执行此命令，向客户端返回一个错误	raw 编码不能执行此命令，向客户端返回一个错误
DECRBY	对整数值进行减法计算，得出的计算结果会作为整数被保存起来	embstr 编码不能执行此命令，向客户端返回一个错误	raw 编码不能执行此命令，向客户端返回一个错误
STRLEN	拷贝对象所保存的整数值，将这个拷贝转换成字符串值，计算并返回这个字符串值的长度	调用 sdslen 函数，返回字符串的长度	调用 sdslen 函数，返回字符串的长度
SETRANGE	将对象转换成 raw 编码，然后按 raw 编码的方式执行此命令	将对象转换成 raw 编码，然后按 raw 编码的方式执行此命令	将字符串特定索引上的值设置为给定的字符串
GETRANGE	拷贝对象所保存的整数值，将这个拷贝转换成字符串值，然后取出并返回字符串指定索引上的字符	直接取出并返回字符串指定索引上的字符	直接取出并返回字符串指定索引上的字符

## 31、列表对象

编码可以是ziplist或者linkedlist

ziplist使用压缩列表作为底层

linkedlist使用双端队列作为底层实现

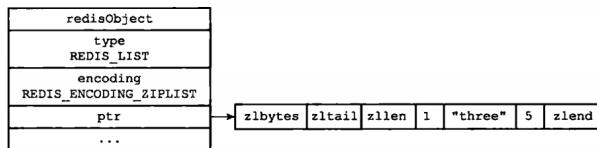


图 8-5 ziplist 编码的 numbers 列表对象

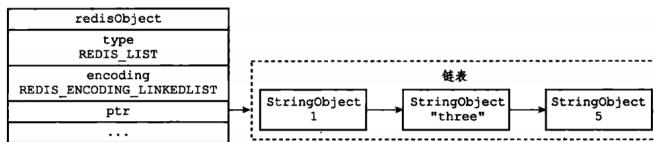
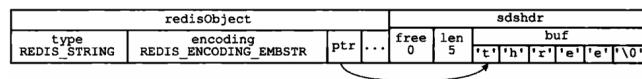


图 8-6 linkedlist 编码的 numbers 列表对象



### 31.1、编码转换

使用ziplist：同时满足1、列表对象的所有元素长度都小于64字节；2、列表对象保存的元素数量小于512  
当上述两个条件之一被破坏时，ziplist会被转移到linkedlist双端列表中。

### 31.2、命令实现

命令	ziplist 编码的实现方法	linkedlist 编码的实现方法
LPUSH	调用 ziplistPush 函数，将新元素推入到压缩列表的表头	调用 listAddNodeHead 函数，将新元素推入到双端链表的表头
RPUSH	调用 ziplistPush 函数，将新元素推入到压缩列表的表尾	调用 listAddNodeTail 函数，将新元素推入到双端链表的表尾
LPOP	调用 ziplistIndex 函数定位压缩列表的表头节点，在向用户返回节点所保存的元素之后，调用 ziplistDelete 函数删除表头节点	调用 listFirst 函数定位双端链表的表头节点，在向用户返回节点所保存的元素之后，调用 listDelNode 函数删除表头节点
RPOP	调用 ziplistIndex 函数定位压缩列表的表尾节点，在向用户返回节点所保存的元素之后，调用 ziplistDelete 函数删除表尾节点	调用 listLast 函数定位双端链表的表尾节点，在向用户返回节点所保存的元素之后，调用 listDelNode 函数删除表尾节点
LINDEX	调用 ziplistIndex 函数定位压缩列表中的指定节点，然后返回节点所保存的元素	调用 listIndex 函数定位双端链表中的指定节点，然后返回节点所保存的元素
LLEN	调用 ziplistLen 函数返回压缩列表的长度	调用 listLength 函数返回双端链表的长度
LINSERT	插入新节点到压缩列表的表头或者表尾时，使用 ziplistPush 函数；插入新节点到压缩列表的其他位置时，使用 ziplistInsert 函数	调用 listInsertNode 函数，将新节点插入到双端链表的指定位置
LREM	遍历压缩列表节点，并调用 ziplistDelete 函数删除包含了给定元素的节点	遍历双端链表节点，并调用 listDelNode 函数删除包含了给定元素的节点
LTRIM	调用 ziplistDeleteRange 函数，删除压缩列表中所有不在指定索引范围内的节点	遍历双端链表节点，并调用 listDelNode 函数删除链表中所有不在指定索引范围内的节点
LSET	调用 ziplistDelete 函数，先删除压缩列表指定索引上的现有节点，然后调用 ziplistInsert 函数，将一个包含给定元素的新节点插入到相同索引上面	调用 listIndex 函数，定位到双端链表指定索引上的节点，然后通过赋值操作更新节点的值

## 32、哈希对象

编码可以是ziplist或者hashtable

ziplist：先将保存键的接续你推入列表尾，然后将保存值的压缩列表推入列表尾。



图 8-10 profile 哈希对象的压缩列表底层实现

hashtable：使用字典作为底层实现，每个键值对使用一个字典键值对来保存。

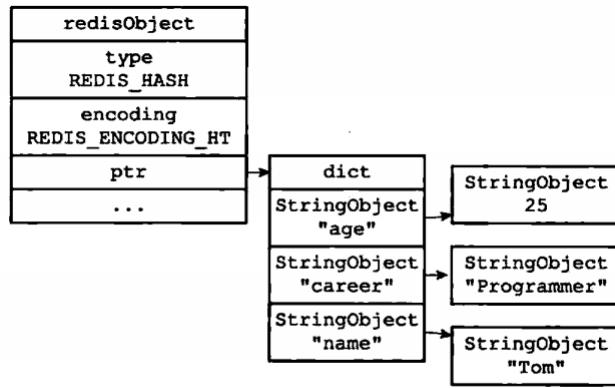


图 8-11 hashtable 编码的 profile 哈希对象

### 32.1、编码转换

当哈希对象可以同时满足以下两个条件时，哈希对象使用 ziplist 编码：

- 哈希对象保存的所有键值对的键和值的字符串长度都小于 64 字节；
- 哈希对象保存的键值对数量小于 512 个；不能满足这两个条件的哈希对象需要使用 hashtable 编码。

以上两个条件之一被破坏，就会导致变为hashtable

### 32.2命令实现

命令	ziplist 编码实现方法	hashtable 编码的实现方法
HSET	首先调用 ziplistPush 函数，将键推入到压缩列表的表尾，然后再次调用 ziplistPush 函数，将值推入到压缩列表的表尾	调用 dictAdd 函数，将新节点添加到字典里面
HGET	首先调用 ziplistFind 函数，在压缩列表中查找指定键所对应的节点，然后调用 ziplistNext 函数，将指针移动到键节点旁边的值节点，最后返回值节点	调用 dictFind 函数，在字典中查找给定键，然后调用 dictGetVal 函数，返回该键所对应的值
HEXISTS	调用 ziplistFind 函数，在压缩列表中查找指定键所对应的节点，如果找到的话说明键值对存在，没找到的话就说明键值对不存在	调用 dictFind 函数，在字典中查找给定键，如果找到的话说明键值对存在，没找到的话就说明键值对不存在
HDEL	调用 ziplistFind 函数，在压缩列表中查找指定键所对应的节点，然后将相应的键节点、以及键节点旁边的值节点都删除掉	调用 dictDelete 函数，将指定键所对应的键值对从字典中删除掉
HLEN	调用 ziplistLen 函数，取得压缩列表包含节点的总数量，将这个数量除以 2，得出的结果就是压缩列表保存的键值对的数量	调用 dictSize 函数，返回字典包含的键值对数量，这个数量就是哈希对象包含的键值对数量
HGETALL	遍历整个压缩列表，用 ziplistGet 函数返回所有键和值（都是节点）	遍历整个字典，用 dictGetKey 函数返回字典的键，用 dictGetVal 函数返回字典的值

## 33、集合对象

集合对象可以是intset或者hashtable

intset：使用整数集合作为底层实现

hashtable：使用字典作为底层实现。字典每个键都是一个字符串对象，每个字符串对象包含一个集合元素。每个值设置为NULL。

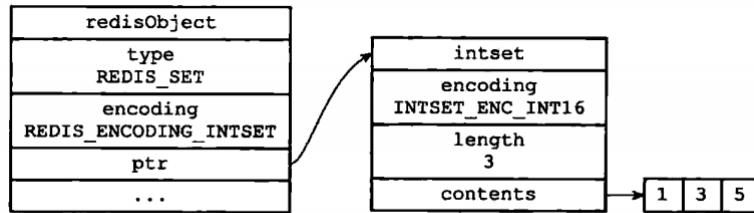


图 8-12 intset 编码的 numbers 集合对象

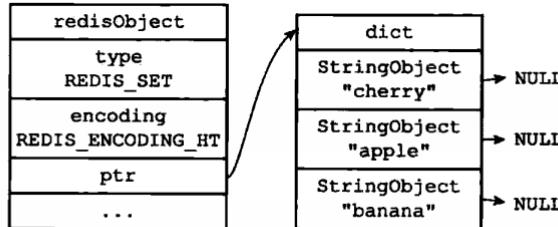


图 8-13 hashtable 编码的 fruits 集合对象

### 33.1、编码转换

当集合对象可以同时满足以下两个条件时，对象使用 intset 编码：

- 集合对象保存的所有元素都是整数值；
- 集合对象保存的元素数量不超过 512 个。

不能满足以上条件时，使用hashtable编码。

### 33.2、命令实现

命令	intset 编码的实现方法	hashtable 编码的实现方法
SADD	调用 intsetAdd 函数，将所有新元素添加到整数集合里面	调用 dictAdd，以新元素为键，NULL 为值，将键值对添加到字典里面
SCARD	调用 intsetLen 函数，返回整数集合所包含的元素数量，这个数量就是集合对象所包含的元素数量	调用 dictSize 函数，返回字典所包含的键值对数量，这个数量就是集合对象所包含的元素数量
SISMEMBER	调用 intsetFind 函数，在整数集合中查找给定的元素，如果找到了说明元素存在于集合，没找到则说明元素不存在于集合	调用 dictFind 函数，在字典的键中查找给定的元素，如果找到了说明元素存在于集合，没找到则说明元素不存在于集合
SMEMBERS	遍历整个整数集合，使用 intsetGet 函数返回集合元素	遍历整个字典，使用 dictGetKey 函数返回字典的键作为集合元素
SRANDMEMBER	调用 intsetRandom 函数，从整数集合中随机返回一个元素	调用 dictGetRandomKey 函数，从字典中随机返回一个字典键
SPOP	调用 intsetRandom 函数，从整数集合中随机取出一个元素，在将这个随机元素返回给客户端之后，调用 intsetRemove 函数，将随机元素从整数集合中删除掉	调用 dictGetRandomKey 函数，从字典中随机取出一个字典键，在将这个随机字典键的值返回给客户端之后，调用 dictDelete 函数，从字典中删除随机字典键所对应的键值对
SREM	调用 intsetRemove 函数，从整数集合中删除所有给定的元素	调用 dictDelete 函数，从字典中删除所有键为给定元素的键值对

## 34、有序集合对象

ziplist或者skipiplist

ziplist: 压缩列表底层。每个集合元素使用紧挨着的节点来保存，第一个保存元素成员，第二个保存元素分值。按照分值从小到大排序，



图 8-15 有序集合元素在压缩列表中按分值从小到大排列

skipiplist: 使用zset结构作为底层实现，一个zset包含一个字典和一个跳跃表

```
typedef struct zset{
    zskiplist *zsl;
    dict *dict;
}zset;
```

zsl跳跃表按照分值从小到大保存所有的集合元素，每个跳跃节点都保存了一个集合元素。通过这个跳跃表可以对有序集合进行范围性操作，ZRANK，ZRANGE。

dict字典为有序集合创建了一个从成员到分值的映射。通过dict可以在常数复杂度的情况下查找给定成员的分值。这两种数据结构通过指针共享相同的元素的成员和分值，不会额外浪费空间

#### 为什么有序集合需要同时使用跳跃表和字典来实现？

在理论上，有序集合可以单独使用字典或者跳跃表的其中一种数据结构来实现，但无论单独使用字典还是跳跃表，在性能上对比起同时使用字典和跳跃表都会有所降低。举个例子，如果我们只使用字典来实现有序集合，那么虽然以  $O(1)$  复杂度查找成员的分值这一特性会被保留，但是，因为字典以无序的方式保存集合元素，所以每次在执行范围型操作——比如ZRANK、ZRANGE等命令时，程序都需要对字典保存的所有元素进行排序，完成这种排序需要至少  $O(N \log N)$  时间复杂度，以及额外的  $O(N)$  内存空间（因为要创建一个数组来保存排序后的元素）。

另一方面，如果我们只使用跳跃表来实现有序集合，那么跳跃表执行范围型操作的所有优点都会被保留，但因为没有了字典，所以根据成员查找分值这一操作的复杂度将从  $O(1)$  上升为  $O(\log N)$ 。因为以上原因，为了让有序集合的查找和范围型操作都尽可能快地执行，Redis 选择了同时使用字典和跳跃表两种数据结构来实现有序集合。

总之，使用zsl来保证范围性操作的复杂度吗，使用dict保证查找的复杂度。

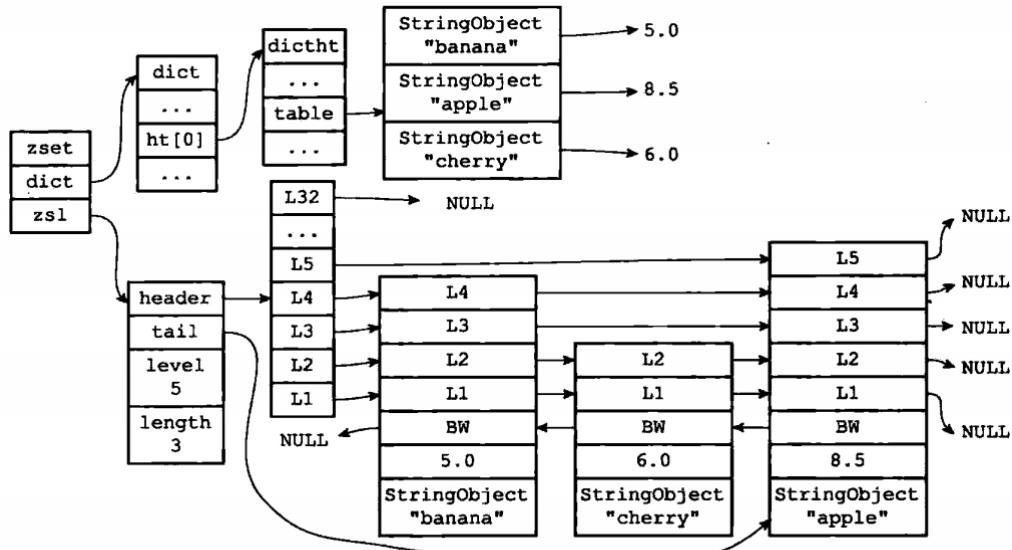


图 8-17 有序集合元素同时被保存在字典和跳跃表中

### 34.1、编码的转换

当有序集合对象可以同时满足以下两个条件时，对象使用 ziplist 编码：

- 有序集合保存的元素数量小于 128 个；
  - 有序集合保存的所有元素成员的长度都小于 64 字节；
- 不能满足以上两个条件的有序集合对象将使用 skipList 编码。

### 34.2、有序集合命令的实现

命令	ziplist 编码的实现方法	zset 编码的实现方法
ZADD	调用 ziplistInsert 函数，将成员和分值作为两个节点分别插入到压缩列表	先调用 zslInsert 函数，将新元素添加到跳跃表，然后调用 dictAdd 函数，将新元素关联到字典
ZCARD	调用 ziplistLen 函数，获得压缩列表包含节点的数量，将这个数量除以 2 得出集合元素的数量	访问跳跃表数据结构的 length 属性，直接返回集合元素的数量
ZCOUNT	遍历压缩列表，统计分值在给定范围内的节点的数量	遍历跳跃表，统计分值在给定范围内的节点的数量
ZRANGE	从表头向表尾遍历压缩列表，返回给定索引范围内的所有元素	从表头向表尾遍历跳跃表，返回给定索引范围内的所有元素
ZREVRANGE	从表尾向表头遍历压缩列表，返回给定索引范围内的所有元素	从表尾向表头遍历跳跃表，返回给定索引范围内的所有元素
ZRANK	从表头向表尾遍历压缩列表，查找给定的成员，沿途记录经过节点的数量，当找到给定成员之后，途经节点的数量就是该成员所对应元素的排名	从表尾向表头遍历跳跃表，查找给定的成员，沿途记录经过节点的数量，当找到给定成员之后，途经节点的数量就是该成员所对应元素的排名
ZREVRANK	从表尾向表头遍历压缩列表，查找给定的成员，沿途记录经过节点的数量，当找到给定成员之后，途经节点的数量就是该成员所对应元素的排名	从表尾向表头遍历跳跃表，查找给定的成员，沿途记录经过节点的数量，当找到给定成员之后，途经节点的数量就是该成员所对应元素的排名
ZREM	遍历压缩列表，删除所有包含给定成员的节点，以及被删除成员节点旁边的分值节点	遍历跳跃表，删除所有包含了给定成员的跳跃表节点，并在字典中解除被删除元素的成员和分值的关联
ZSCORE	遍历压缩列表，查找包含了给定成员的节点，然后取出成员节点旁边的分值节点保存的元组	直接从字典中取出给定成员的分值

## 35、类型检查和命令多态

命令分为两类：

其中一种命令可以对任何类型的键执行，比如说`DEL`命令、`EXPIRE`命令、`RENAME`

第一类：对任何类型的键执行：`命令`、`TYPE`命令、`OBJECT`命令等。

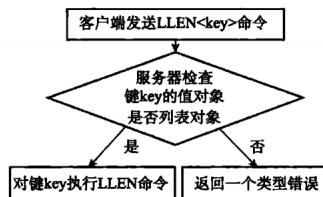
第二类：对指定类型的键执行：

而另一种命令只能对特定类型的键执行，比如说：

- `SET`、`GET`、`APPEND`、`STRLEN`等命令只能对字符串键执行；
- `HDEL`、`HSET`、`HGET`、`HLEN`等命令只能对哈希键执行；
- `R PUSH`、`LPOP`、`LINSERT`、`LLEN`等命令只能对列表键执行；
- `SADD`、`SPOP`、`SINTER`、`SCARD`等命令只能对集合键执行；
- `ZADD`、`ZCARD`、`ZRANK`、`ZSCORE`等命令只能对有序集合键执行；

### 35.1、类型检查的实现

通过`redisObject`中的`ttype`属性进行实现的：服务器检查输入数据库键的值对象是否时执行命令需要的类型。



### 35.2、多态命令的实现

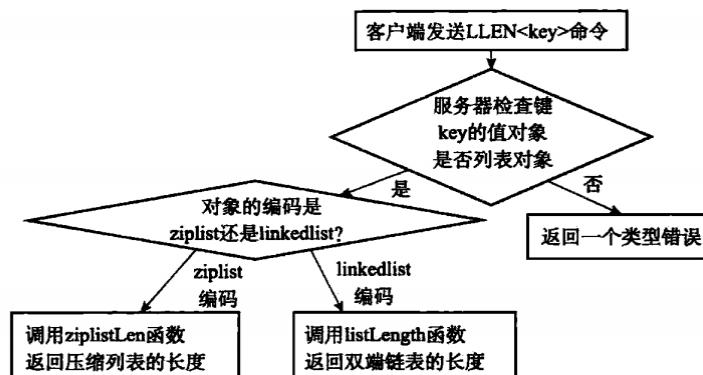


图 8-19 LLEN 命令的执行过程

## 36、内存回收

构建一个引用计数实现内存回收机制。

```
typedef struct redisObject {
    // ...
    // 引用计数
    int refcount;
    // ...
}
```

每个对象的引用信息有`redisObject`结构的`refcount`属性记录：`robj`；

对象的引用计数信息会随着对象的使用状态而不断变化：

- 1、创建时会置为1
- 2、被一个新程序使用时，+1
- 3、不在使用时-1
- 4、引用为0时，内存回收释放

表 8-12 修改对象引用计数的 API

函数	作用
<code>incrRefCount</code>	将对象的引用计数值增一
<code>decrRefCount</code>	将对象的引用计数值减一，当对象的引用计数值等于0时，释放对象
<code>resetRefCount</code>	将对象的引用计数值设置为0，但并不释放对象，这个函数通常在需要重新设置对象的引用计数值时使用

对象的三个生命周期：创建、操作、释放

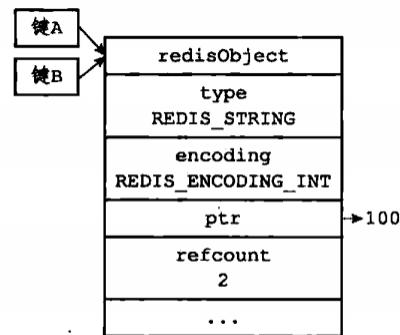
```
// 创建一个字符串对象 s，对象的引用计数为 1
robj *s = createStringObject(...)

// 对象 s 执行各种操作 ...

// 将对象 s 的引用计数减一，使得对象的引用计数变为 0
// 导致对象 s 被释放
decrRefCount(s)
```

## 37、对象共享

引用计数属性还提供对象共享功能。



在 Redis 中，让多个键共享同一个值对象需要执行以下两个步骤：

- 1) 将数据库键的值指针指向一个现有的值对象；
- 2) 将被共享的值对象的引用计数增一。

图 8-21 被共享的字符串对象

初始化redis服务器时，创建一万个字符串对象，这些对象包含了0-9999的所有整数值，当使用0-9999的字符串对象时就会使用到这些共享对象。

不仅仅在字符串键可以使用，嵌套了字符串对象的对象都可以使用。

因此，尽管共享更复杂的对象可以节约更多的内存，但受到 CPU 时间的限制，  
~~Redis 只对包含整数值的字符串对象进行共享。~~

## 38、对象的空转时长

```
typedef struct redisObject {  
    // ...  
    unsigned lru:22;  
    // ...  
} robj;
```

redisObject结构包含一个属性lru，记录了对象最后一次被程序访问的时间。  
OBJECT IDELTIME 打印空转时长，  
~~当前时间 - lru~~

如果服务器打开maxmemory选项并且回收算法为volatile-lru或者allkeys-lru，那么服务器内存超过阈值时，优先回收空转时间较高的那部分。

## 39、对象总结

- Redis 数据库中的每个键值对的键和值都是一个对象。
- Redis 共有字符串、列表、哈希、集合、有序集合五种类型的对象，每种类型的对象至少都有两种或以上的编码方式，不同的编码可以在不同的使用场景上优化对象的使用效率。
- 服务器在执行某些命令之前，会先检查给定键的类型能否执行指定的命令，而检查一个键的类型就是检查键的值对象的类型。
- Redis 的对象系统带有引用计数实现的内存回收机制，当一个对象不再被使用时，该对象所占用的内存就会被自动释放。
- Redis 会共享值为 0 到 9999 的字符串对象。
- 对象会记录自己的最后一次被访问的时间，这个时间可以用于计算对象的空转时间。

# 旨在单机数据库的实现

实现原理：保存键值对的方式、过期时间、自动删除过期键值对

持久化：AOF、RDB

事件：文件事件、时间事件、维护管理客户端、服务器运作机制

# 数据库

## 40、服务器中的数据库

保存在redisServer结构的DB数组里，db数组每一项都是一个redisDb结构、每个redisDB结构代表一个数据库

```
struct redisServer{
    redisDB* db;//保存所有的数据库
    int dbnum;//保存服务器数据库的适量。默认情况下为16, redis服务器会创建16个数据库
}
```

## 41、切换数据库

默认客户端目标数据库为0好数据库，通过执行 **SELECT** 来切换数据库。

redisClient结构的db属性记录了客户端当前的目标数据库

```
struct redisClient{
    redisDB* db;//指向redisDB的指针
}
```

## 42、数据库键空间

数据库由redisDb结构表示，每一个redisDb结构的dict字典保存了数据库中的所有键值对，这个字典称为键空间

```
struct redisDb{
    dict* dict;//数据库键空间，保存所有的键值对
}
```

键空间和用户所见的数据库是直接对应的：

- 键空间的键也就是数据库的键，每个键都是一个字符串对象。
- 键空间的值也就是数据库的值，每个值可以是字符串对象、列表对象、哈希表对象、集合对象和有序集合对象中的任意一种 Redis 对象。

所有针对数据库键空间的操作都是通过键空间字典进行操作的。

### 42.1、添加新键

就是将新键添加到键空间

```
set date "ssss"
```

### 42.2、删除键

```
redis> DEL book
(integer) 1
```

### 42.3、更新键

```
redis> HSET book page 320
(integer) 1
redis> SET message "blah blah"
OK
```

### 42.4、对键取值

```
redis> LRANGE alphabet 0 -1
1) "a"
2) "b"
3) "c"
redis> GET message
"hello world"
```

## 42.5、其他操作

除了上面列出的添加、删除、更新、取值操作之外，还有很多针对数据库本身的 Redis 命令，也是通过对键空间进行处理来完成的。

比如说，用于清空整个数据库的 **FLUSHDB** 命令，就是通过删除键空间中的所有键值对来实现的。又比如说，用于随机返回数据库中某个键的 **RANDOMKEY** 命令，就是通过在键空间中随机返回一个键来实现的。

另外，用于返回数据库键数量的 **DBSIZE** 命令，就是通过返回键空间中包含的键值对的数量来实现的。类似的命令还有 **EXISTS**、**RENAME**、**KEYS** 等，这些命令都是通过对键空间进行操作来实现的。

## 42.6、读写键空间时的维护操作

1. 根据是否存在更新服务器键空间命中/不命中次数。
2. 更新键的LRU
3. 删除过期键
4. watch监视键，如果修改则置为脏。
5. 服务器每次修改一个键后，会对脏键计数器的值+1.这个计数器会触发服务器的持久化以及复制操作

## 43、设置键的生存时间和过期时间

服务端 **EXPIRE** 和 **PEXPIRE** 设置秒或者毫秒级别的键的生存时间。

```
redis> EXPIRE key 5
(integer) 1

redis> GET key // 5 秒之内
"value"

redis> GET key // 5 秒之后
(nil)
```

客户端 **EXPIREAT** 和 **PEXPIREAT** 设置数据库的键的生存时间。

TTL和PTTL命令返回剩余的生存时间。

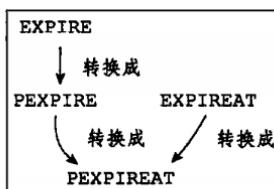


图 9-11 设置生存时间和设置过期时间的命令之间的转换

### 43.1、保存过期时间

redisDb结构的**expires**字典保存了数据库中所有键的过期时间，过期字典：

1. 过期字典的键是一个指针，指向键空间某个键对象
2. 过期字典的值是一个long long的整数，保存过期时间

### 43.2、移除过期时间

```
# 如果键不存在，或者键没有设置过期时间，那么直接返回
if key not in redisDb.expires:
    return0

# 移除过期字典中给定键的键值对关联
redisDb.expires.remove(key)

# 键的过期时间移除成功
PERSIST:  return 1
```

### 43.3、返回剩余的生存时间

TTL、PTTL，通过计算过期时间和当前时间的时间差

## 44、过期键删除策略

redis主要使用惰性删除和定期删除相结合

### 44.1、定时删除

设置过期键的时候创建一个定时器，让定时器在键的过期时间来临时，立即执行对键的删除操作。

缺点：

1. 对cpu时间不友好：过期键较多时，删除可能会占用cpu时间。
2. 服务器创建定时器，实现方式是无序列表，不能高效处理大量的时间事件

### 44.2、惰性删除

放任过期键不管，每次获取键时，检查是否过期，如果过期就删除。

对cpu友好：仅限于使用键才执行删除操作。

缺点：对内存不友好。比如日志，某个时间节点后，使用会大大减少，但是仍旧占用内存。

### 44.3、定期删除

每隔一段时间进行检查，删除过期键。

前两者中的一种折中：

- 通过限制删除操作执行的时长和频率来减少删除操作对cpu的影响。
- 定期删除，有效的节省内存。

难点：确定删除操作的时长和频率

redis通过expireIfNeeded实现惰性删除

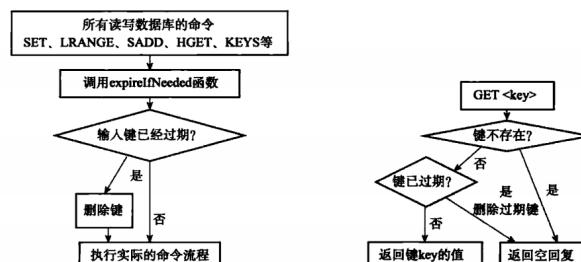


图 9-15 命令调用 expireIfNeeded 来删除过期键

图 9-16 GET 命令的执行过程

redis通过activateExpireCycle函数实现定期删除：

- 函数每次运行时，都从一定数量的数据库中取出一定数量的随机键进行检查，并删除其中的过期键。
- 全局变量 current\_db 会记录当前 activeExpireCycle 函数检查的进度，并在下一次 activeExpireCycle 函数调用时，接着上一次的进度进行处理。比如说，如果当前 activeExpireCycle 函数在遍历 10 号数据库时返回了，那么下次 activeExpireCycle 函数执行时，将从 11 号数据库开始查找并删除过期键。
- 随着 activeExpireCycle 函数的不断执行，服务器中的所有数据库都会被检查一遍，这时函数将 current\_db 变量重置为 0，然后再次开始新一轮的检查工作。

## 45、AOF、RDB和复制功能对过期键的处理

### 45.1、RDB

创建时：RDB文件时，会检查过期键，不会被保存到RDB文件中。

载入时：主服务器，只载入未过期的；从服务器，都载入。主从同步的时候从数据库就会被清空，所以过期键对从服务器没有印象

### 45.2、AOF

当过期键被删除后，回想AOF文件追加一个DEL命令，显示记录。

载入时，只会重写未过期的键

### 45.3、复制

复制模式下，过期键的删除操作由主服务器进行控制：

- 主服务器在删除一个过期键之后，会显式地向所有从服务器发送一个 *DEL* 命令，告知从服务器删除这个过期键。
- 从服务器在执行客户端发送的读命令时，即使碰到过期键也不会将过期键删除，而是继续像处理未过期的键一样来处理过期键。
- 从服务器只有在接到主服务器发来的 *DEL* 命令之后，才会删除过期键。

## 46、数据库通知

客户端通过订阅给定的频道或者模式来获知数据库中键的变化以及数据库中命令的执行情况。

获取键的变化：  
127.0.0.1:6379> SUBSCRIBE \_\_keyspace@0\_\_:\_message

```
127.0.0.1:6379> SUBSCRIBE __keyevent@0__:_del
Reading messages... (press Ctrl-C to quit)
```

获取命令执行情况：

## 47、数据库总结

- Redis 服务器的所有数据库都保存在 `redisServer.db` 数组中，而数据库的数量则由 `redisServer.dbnum` 属性保存。
- 客户端通过修改目标数据库指针，让它指向 `redisServer.db` 数组中的不同元素来切换不同的数据库。
- 数据库主要由 `dict` 和 `expires` 两个字典构成，其中 `dict` 字典负责保存键值对，而 `expires` 字典则负责保存键的过期时间。
- 因为数据库由字典构成，所以对数据库的操作都是建立在字典操作之上的。
- 数据库的键总是一个字符串对象，而值则可以是任意一种 Redis 对象类型，包括字符串对象、哈希表对象、集合对象、列表对象和有序集合对象，分别对应字符串键、哈希表键、集合键、列表键和有序集合键。
- `expires` 字典的键指向数据库中的某个键，而值则记录了数据库键的过期时间，过期时间是一个以毫秒为单位的 UNIX 时间戳。
- Redis 使用惰性删除和定期删除两种策略来删除过期的键：惰性删除策略只在碰到过期键时才进行删除操作，定期删除策略则每隔一段时间主动查找并删除过期键。
- 执行 `SAVE` 命令或者 `BGSAVE` 命令所产生的新 RDB 文件不会包含已经过期的键。
- 执行 `BGREWRITEAOF` 命令所产生的重写 AOF 文件不会包含已经过期的键。
- 当一个过期键被删除之后，服务器会追加一条 `DEL` 命令到现有 AOF 文件的末尾，显式地删除过期键。
- 当主服务器删除一个过期键之后，它会向所有从服务器发送一条 `DEL` 命令，显式地删除过期键。
- 从服务器即使发现过期键也不会自作主张地删除它，而是等待主节点发来 `DEL` 命令，这种统一、中心化的过期键删除策略可以保证主从服务器数据的一致性。
- 当 Redis 命令对数据库进行修改之后，服务器会根据配置向客户端发送数据库通知。

# RDB持久

## 48、RDB创建和载入

### SAVE 和 BGSAVE

save会阻塞服务器进程，bgsave会创建子进程，具体由 `rdbSave` 函数执行  
载入工作在服务器启动时自动执行，没有专门的命令，由函数 `rdbLoad` 函数完成  
如果开启了AOF，优先使用AOF，因为AOF更新频率更高

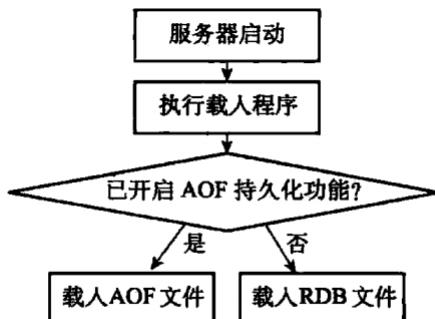


图 10-4 服务器载入文件时的判断流程

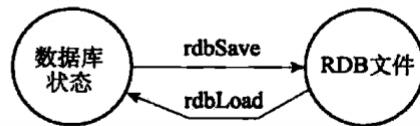


图 10-5 创建和载入 RDB 文件

### 48.1、Save执行时，服务器阻塞，客户端请求会被拒绝。bgsave可以继续处理客户端请求。

在执行bgsave时，save会被禁用，避免子进程竞争条件；bgsave也会被禁用，理由一样。  
RDB载入期间，一直阻塞

## 49、自动间隔性保存

通过设置保存条件，条件满足时执行bgsave

会有一个 `saveparams` 数组

redisServer	saveparams[0]	saveparams[1]	saveparams[2]
...	seconds 900	seconds 300	seconds 60
saveparams	changes 1	changes 10	changes 10000

图 10-6 服务器状态中的保存条件

### 服务器的dirty计数器和lastsave属性

dirty：上一次save之后，数据库进行了多少次修改

lastsave：记录上一次save的时间

周期性操作函数 `serverCron` 默认每隔100ms执行一次，其中一项工作就是检查save选项的保存条件是否满足。

## 50、RDB文件结构

REDIS	db_version	databases	EOF	check_sum
-------	------------	-----------	-----	-----------

图 10-10 RDB 文件结构

大写：常量，小写：变量和数据

REDIS：5字节，标志是一个RDB文件。RDB保存的是二进制文件。

db\_version：4字节，记录版本号。

databases：包含零个或者任意多个数据库，以及各个数据库的键值对数据：

如果服务器的数据库状态为空（所有数据库都是空的），那么这个部分也为空，长度为0字节。

如果服务器的数据库状态为非空（有至少一个数据库非空），那么这个部分也为非空，根据数据库所保存键值对的数量、类型和内容不同，这个部分的长度也会有所不同。

EOF：1字节，表示键值对载入完毕

check\_sum：8字节无符号整数，校验和。

REDIS	db_version	SELECTDB	0	pairs	SELECTDB	3	pairs	EOF	check_sum
-------	------------	----------	---	-------	----------	---	-------	-----	-----------

database0 database3

图 10-15 RDB 文件中的数据库结构示例

TYPE	key	value
------	-----	-------

EXPIRETIME_MS	ms	TYPE	key	value
---------------	----	------	-----	-------

图 10-16 不带过期时间的键值对  
pairs:

图 10-17 带有过期时间的键值对

## 51、RDB文件

调用od命令可以查看RDB文件

- 不含任何键值对的RDB文件

```
$ od -c dump.rdb
0000000  R E D I S 0 0 0 6 377 334 263 C 360 Z 334
0000020 362 V
0000022

$ od -c dump.rdb
0000000  R E D I S 0 0 0 6 376 \0 \0 003 M S G
0000020 005 H E L L O 377 207 z = 304 f T L 343
0000037
```

- 包含键值对:

```
$ od -c dump.rdb
0000000  R E D I S 0 0 0 6 376 \0 374 \ 2 365 336
0000020 @ 001 \0 \0 \0 003 M S G 005 H E L L O 377
0000040 212 231 x 247 252 } 021 306
0000050
```

- 包含过期时间:

- 包含集合键:

```
$ od -c dump.rdb
0000000  R E D I S 0 0 0 6 376 \0 002 004 L A N
0000020 G 003 004 R U B Y 004 J A V A 001 C 377 202
0000040 312 r 352 346 305 * 023
0000047
```

以下是 RDB 文件各个部分的意义:

- ❑ REDIS0006 : RDB 文件标志和版本号。
- ❑ 376 \0: 切换到 0 号数据库。
- ❑ 002 004 L A N G : 002 是常量 REDIS\_RDB\_TYPE\_SET (这个常量的实际值为整数 2), 表示这是一个哈希表编码的集合键, 004 表示键的长度, LANG 是键的名字。
- ❑ 003 : 集合的大小, 说明这个集合包含三个元素。
- ❑ 004 R U B Y : 集合的第一个元素。
- ❑ 004 J A V A : 集合的第二个元素。
- ❑ 001 C : 集合的第三个元素。
- ❑ 377 : 代表常量 EOF。
- ❑ 202 312 r 352 346 305 \* 023 : 代表校验和。

## 52、总

- ❑ RDB 文件用于保存和还原 Redis 服务器所有数据库中的所有键值对数据。
- ❑ SAVE 命令由服务器进程直接执行保存操作, 所以该命令会阻塞服务器。
- ❑ BGSAVE 令由子进程执行保存操作, 所以该命令不会阻塞服务器。
- ❑ 服务器状态中会保存所有用 save 选项设置的保存条件, 当任意一个保存条件被满足时, 服务器会自动执行 BGSAVE 命令。
- ❑ RDB 文件是一个经过压缩的二进制文件, 由多个部分组成。
- ❑ 对于不同类型的键值对, RDB 文件会使用不同的方式来保存它们。

## AOF持久化



### 53、AOF实现

三步：

- 命令追加：

```
struct redisServer {  
    // ...  
    // AOF 缓冲区  
    sds aof_buf;  
    // ...  
    aof_buf缓冲区末尾); // ...
```

- 文件写入：

redis服务器进程就是一个事件循环，文件事件负责接受和返回客户端命令，时间事件负责执行类似serverCron函数这样的需要定时执行的函数。

每一次事件循环之前，会调用 **flushAppendOnlyFile** 函数判断是否进行同步 以及 是否将aof\_buf写入AOF文件里面。

- 文件同步：

### 54、文件载入和数据还原

- 创建一个不带网络链接的伪客户端
- 从AOF文件中分析并读取出一条写命令
- 使用伪客户端执行被读出的写命令
- 一直执行2-3，一直到处理完毕。

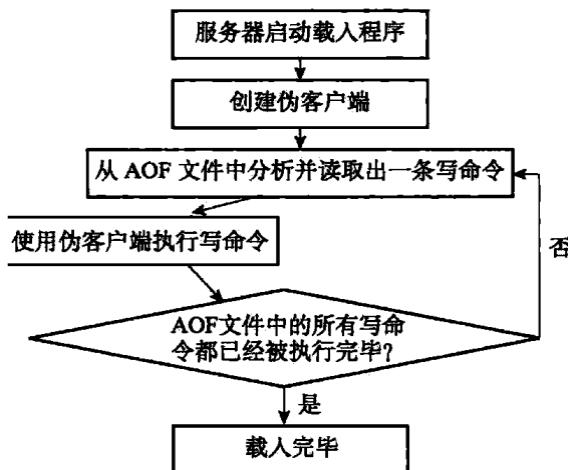


图 11-2 AOF 文件载入过程

### 55、AOF重写

为了解决AOF文件体积膨胀的问题。通过该功能，redis服务器可以创建一个新的AOF文件来代替现有的，但是新的AOF不会包含浪费空间的冗余命令。

**BGREWRITEAOF**

### 55.1、重写原理

不需要对对现有的AOF文件进行读取分析，是通过读取服务器当前的数据库状态来实现的。

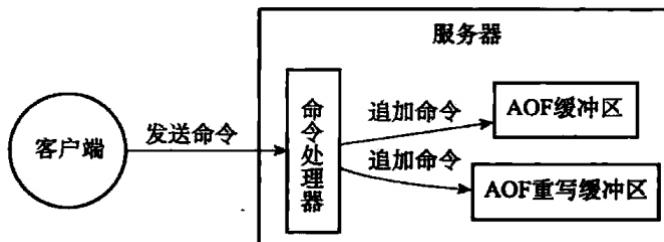
从数据库读取键现在的值，然后用一条命令去记录键值对，代替之前记录这个键值对的多条命令。

aof\_rewrite

### 55.2、后台重写

aof\_rewrite会进行大量的写操作，所以调用这个函数的线程将被长时间阻塞，所以会fork一个子进程进行重写，子进程带有服务器进程的数据副本，使用子进程而不是线程可以避免使用锁的情况下，保证数据的安全性。问题：子进程重写时，数据库会有新的命令。

解决：设置了一个AOF重写缓冲区。服务器会同时将写命令写入AOF缓冲区和AOF重写缓冲区。



## 56、总

- AOF 文件通过保存所有修改数据库的写命令请求来记录服务器的数据库状态。
- AOF 文件中的所有命令都以 Redis 命令请求协议的格式保存。
- 命令请求会先保存到 AOF 缓冲区里面，之后再定期写入并同步到 AOF 文件。
- appendfsync 选项的不同值对 AOF 持久化功能的安全性以及 Redis 服务器的性能有很大的影响。
- 服务器只要载入并重新执行保存在 AOF 文件中的命令，就可以还原数据库本来的状态。
- AOF 重写可以产生一个新的 AOF 文件，这个新的 AOF 文件和原有的 AOF 文件所保存的数据库状态一样，但体积更小。
- AOF 重写是一个有歧义的名字，该功能是通过读取数据库中的键值对来实现的，程序无须对现有 AOF 文件进行任何读入、分析或者写入操作。
- 在执行 BGREWRITEAOF 命令时，Redis 服务器会维护一个 AOF 重写缓冲区，该缓冲区会在子进程创建新 AOF 文件期间，记录服务器执行的所有写命令。当子进程完成创建新 AOF 文件的工作之后，服务器会将重写缓冲区中的所有内容追加到新 AOF 文件的末尾，使得新旧两个 AOF 文件所保存的数据库状态一致。最后，服务器用新的 AOF 文件替换旧的 AOF 文件，以此来完成 AOF 文件重写操作。

## 事件

redis服务器是一个事件驱动程序，处理两类事件：

- 文件事件
- 时间事件

### 57、文件事件

基于Reactor模式开发的网络事件处理器：

- 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
- 当监听到操作时，文件事件处理器就会调用套接字之前关联的事件处理器来处理事件。

虽然文件处理器是单线程，但是通过io多路服用，可以实现高性能得网络通信模型。

### 57.1、文件事件处理器的构成

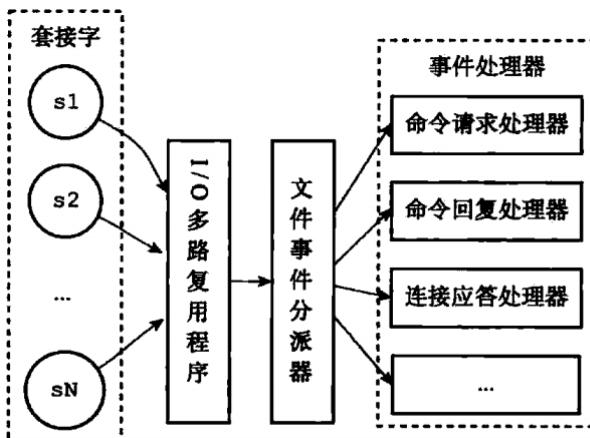


图 12-1 文件事件处理器的四个组成部分

### 57.2、IO多路复用程序的实现

通过封装常见的select、epoll、evport、kqueue这些io多路复用函数库类来实现的。

IO多路复用程序的底层实现是可以互换的，会自动选择性能最高的IO多路复用函数库来作为底层实现

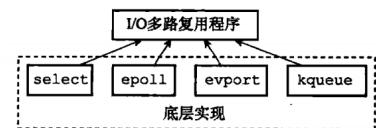


图 12-3 Redis 的 I/O 多路复用程序有多个 I/O

### 57.3、事件的类型

两类事件：AE\_READABLE、AE\_WRITEABLE

如果一个套接字同时监听套接字的上述两个事件，会先读套接字，然后写套接字

### 57.4、API

#### 57.5、文件事件的处理器

多个处理器：

1. 连接应答处理器：acceptTcpHandler函数，具体是对socket.h/accept函数的包装，当客户端待用connect函



数连接服务器监听套接字时，产生读事件，引发连接应答处理器

2. 命令请求处理器：networking.c/readQueryFromClient函数，是对unistd.h/read函数的包装。

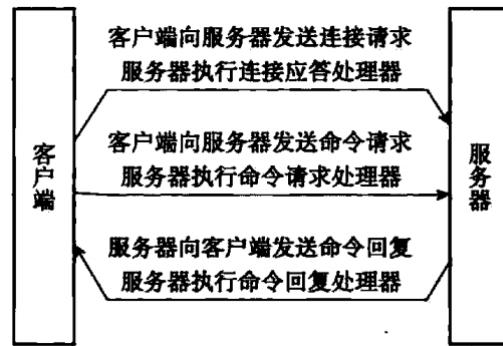


图 12-5 服务器接收客户端发来的命令请求



图 12-6 服务器向客户端发送命令回复

3. 命令回复处理器：是对unistd.h/write的封装



4. 完整的客户端和服务器连接事件示例

## 58、时间事件

两类时间事件：定时事件 (AE\_NOMORE) 、周期性事件 (非NO\_MORE) 。

时间事件的组成：

1. id
2. when：记录时间事件的到达时间
3. timeProc：事件处理函数

### 58.1、实现

所有的时间事件都放在一个无序列表中 (when属性无序、但是按照ID逆序)  
当时间事件达到时，遍历所有时间事件。

不会影响性能，正常模式下，只是用serverCron一个时间事件。

### 58.2、serverCron函数

负责定期对redis自身的资源状态进行检查和调整。

- 更新各类信息：事件，内存占用，数据库占用等
- 清理过期的键值对
- 关闭和清理连接失败的客户端
- 尝试进行RDB和AOF
- 对主服务器，定期进行同步
- 集群模式下，定期同步和连接测试

## 59、事件的调度和执行

由 `aeProcessEvents` 函数负责，将该函数置于一个循环里面，加上初始化和清理函数，就构成了一个redis服务器的主函数：

```

def main():

    # 初始化服务器
    init_server()

    # 一直处理事件，直到服务器关闭为止
    while server_is_not_shutdown():
        aeProcessEvents()

    # 服务器关闭，执行清理操作
    clean_server()

```

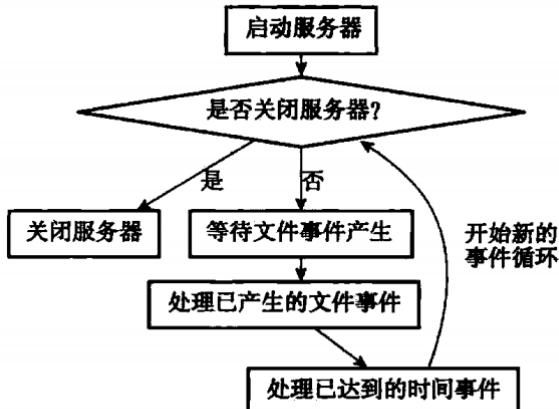


图 12-10 事件处理角度下的服务器运行流程

执行和调度规则：

- 文件事件和时间事件处理都是同步的，有序的，原子的，不会抢占，所以时间事件的实际处理时间，可能会比时间事件设定的到达时间稍晚一点。

## 60、事件总

- Redis 服务器是一个事件驱动程序，服务器处理的事件分为时间事件和文件事件两类。
- 文件事件处理器是基于 Reactor 模式实现的网络通信程序。
- 文件事件是对套接字操作的抽象：每次套接字变为可应答（acceptable）、可写（writable）或者可读（readable）时，相应的文件事件就会产生。
- 文件事件分为 AE\_READABLE 事件（读事件）和 AE\_WRITABLE 事件（写事件）两类。
- 时间事件分为定时事件和周期性事件：定时事件只在指定的时间到达一次，而周期性事件则每隔一段时间到达一次。
- 服务器在一般情况下只执行 serverCron 函数一个时间事件，并且这个事件是周期性事件。
- 文件事件和时间事件之间是合作关系，服务器会轮流处理这两种事件，并且处理事件的过程中也不会进行抢占。
- 时间事件的实际处理时间通常会比设定的到达时间晚一些。

## 客户端

一对多服务器、IO多路复用实现文件事件处理器、使用单线程单进程处理命令请求。  
服务器状态结构的clients属性是一个链表，保存了所有的客户端状态结构。

## 61、客户端属性

两类：

1. 通用属性
2. 特定功能相关的属性，db属性、dictid属性，mstate属性、watched\_keys属性

## 61.1、套接字描述符

1. 伪客户端:  $fd = -1$ 。该客户端来源于AOF文件或者Lua脚本, 而不是网络。
  2. 普通客户端:  $fd > -1$

```
redis> CLIENT list
addr=127.0.0.1:53428 fd=6 name= age=1242 idle=0 ...
addr=127.0.0.1:53469 fd=7 name= age=4 idle=4 ...
```

## 61.2、名字默认为空

### 61.3. 标志

记录客户端的角色以及目前的状态

- REDIS\_MASTER
  - REDIS\_SLAVE
  - REDIS\_PRE\_PSYNC

以下是一些 flags 属性的例子：

```
# 客户端是一个主服务器
REDIS_MASTER

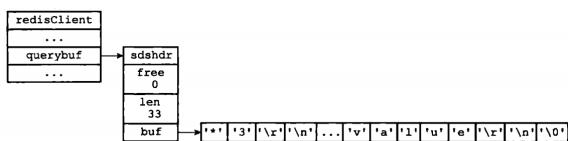
# 客户端正在被列表命令阻塞
REDIS_BLOCKED

# 客户端正在执行事务，但事务的安全性已被破坏
REDIS_MULTI | REDIS_DIRTY_CAS

# 客户端是一个从服务器，并且版本低于 Redis 2.8
REDIS_SLAVE | REDIS_PRE_PSYNC

# 这是专门用于执行 Lua 脚本包含的 Redis 命令的伪客户端
# 它强制服务器将当前执行的命令写入 AOF 文件，并复制给从服务器
REDIS LUA_CLIENT | REDIS_FORCE_AOF | REDIS_FORCE_REPL
```

## 61.4、输入缓冲区



保存客户端发送的命令请求

图 13-4 querybuf 属性示例

## 61.5、命令和命令参数

`argv` 属性是一个数组，数组中的每个项都是一个字符串对象，其中 `argv[0]` 是要执行的命令，而之后的其他项则是传给命令的参数。

`argc` 属性则负责记录 `argv` 数组的长度。

举个例子，对于图 13-4 所示的 querybuf 属性来说，服务器将分析并创建图 13-5 所示的 argv 属性和 argc 属性。

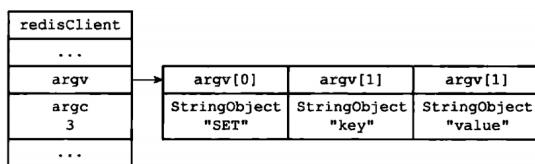


图 13-5 argv 属性和 argc 属性示例

## 61.6、命令实现函数

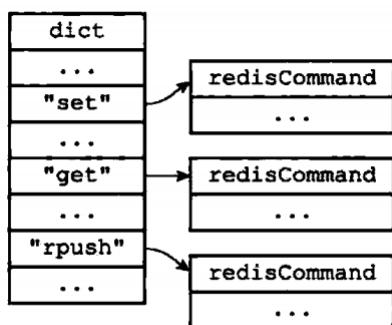


图 13-6 命令表

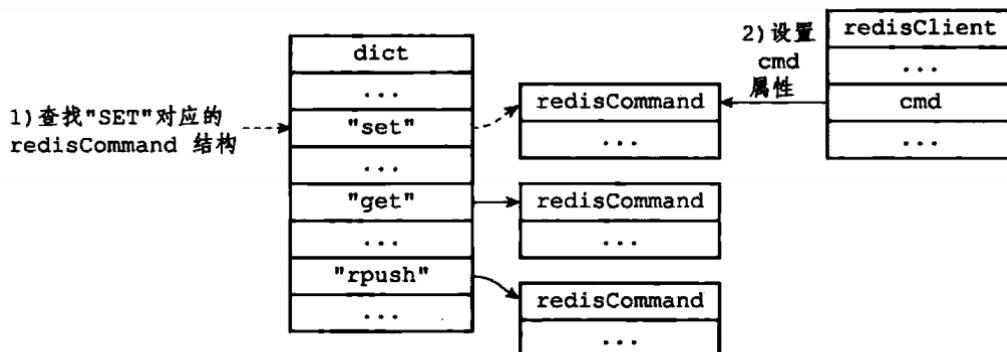


图 13-7 查找命令并设置 cmd 属性

## 61.7、输出缓冲区

### 61.8、身份验证

authenticated 属性  
=0：未通过验证

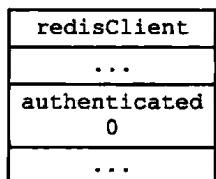


图 13-10 未验证身份时的客户端状态

authenticated 属性  
向服务器发送命令请

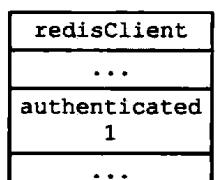


图 13-11 已经通过身份验证的客户端状态  
=1：通过

## 61.9、时间

ctime：创建客户端时间

lastinteraction：最后一次互动时间。计算空转时间。

## 62、客户端的创建和关闭

### 62.1、普通客户端

创建：通过网络连接的客户端。使用 `connect` 连接时，服务器调用连接事件处理器，为客户端创建相应的客户端状态。

关闭：

原因：

- 主动关闭
- 客户端或者发送给客户端的命令请求或者回复超过输出缓冲区

服务器使用两种模式限制客户端输出缓冲区的大小：

- 硬性限制：缓冲区超过规定大小就立即关闭客户端
- 软性限制：如果超出软性限制，但没有超过硬性限制，会监控该客户端，如果一直超过软性限制一定的时间，会关闭客户端。

### 62.2、Lua脚本的伪客户端

服务器初始化时会创建一个伪客户端，专门用来执行Lua脚本。该客户端会一直存在服务器运行的整个生命周期。

### 62.3、AOF文件的伪客户端

载入AOF文件时创建该伪客户端，载入完成后关闭。

## 63、总

- 服务器状态结构使用 `clients` 链表连接起多个客户端状态，新添加的客户端状态会被放到链表的末尾。
  - 客户端状态的 `flags` 属性使用不同标志来表示客户端的角色，以及客户端当前所处的状态。
  - 输入缓冲区记录了客户端发送的命令请求，这个缓冲区的大小不能超过 1 GB。
  - 命令的参数和参数个数会被记录在客户端状态的 `argv` 和 `argc` 属性里面，而 `cmd` 属性则记录了客户端要执行命令的实现函数。
  - 客户端有固定大小缓冲区和可变大小缓冲区两种缓冲区可用，其中固定大小缓冲区的最大大小为 16 KB，而可变大小缓冲区的最大大小不能超过服务器设置的硬性限制值。
  - 输出缓冲区限制值有两种，如果输出缓冲区的大小超过了服务器设置的硬性限制，那么客户端会被立即关闭；除此之外，如果客户端在一定时间内，一直超过服务器设置的软性限制，那么客户端也会被关闭。
  - 当一个客户端通过网络连接连上服务器时，服务器会为这个客户端创建相应的客户端状态。网络连接关闭、发送了不合协议格式的命令请求、成为 `CLIENT KILL` 命令的目标、空转时间超时、输出缓冲区的大小超出限制，以上这些原因都会造成客户端被关闭。
  - 处理 Lua 脚本的伪客户端在服务器初始化时创建，这个客户端会一直存在，直到服务器关闭。
  - 载入 AOF 文件时使用的伪客户端在载入工作开始时动态创建，载入工作完毕之后关闭。

## 服务端

### 64、命令请求执行过程

那么从客户端发送 SET KEY VALUE 命令到获得回复 OK 期间，客户端和服务器共需要执行以下操作：

- 1) 客户端向服务器发送命令请求 SET KEY VALUE。
- 2) 服务器接收并处理客户端发来的命令请求 SET KEY VALUE，在数据库中进行设置操作，并产生命令回复 OK。
- 3) 服务器将命令回复 OK 发送给客户端。
- 4) 客户端接收服务器返回的命令回复 OK，并将这个回复打印给用户观看。

#### 64.1、发送命令请求

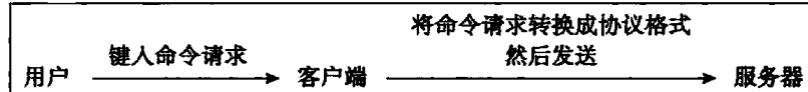


图 14-1 客户端接收并发送命令请求的过程

举个例子，假设用户在客户端键入了命令：

SET KEY VALUE

那么客户端会将这个命令转换成协议：

\*3\r\n\$3\r\nSET\r\n\$3\r\nKEY\r\n\$5\r\nVALUE\r\n\r\n

然后将这段协议内容发送给服务器。

#### 64.2、读取命令请求

当客户端与服务器之间的连接套接字因为客户端的写入而变得可读时，服务器将调用命令请求处理器来执行以下操作：

- 1) 读取套接字中协议格式的命令请求，并将其保存到客户端状态的输入缓冲区里面。
- 2) 对输入缓冲区中的命令请求进行分析，提取出命令请求中包含的命令参数，以及命令参数的个数，然后分别将参数和参数个数保存到客户端状态的 argv 属性和 argc 属性里面。
- 3) 调用命令执行器，执行客户端指定的命令。

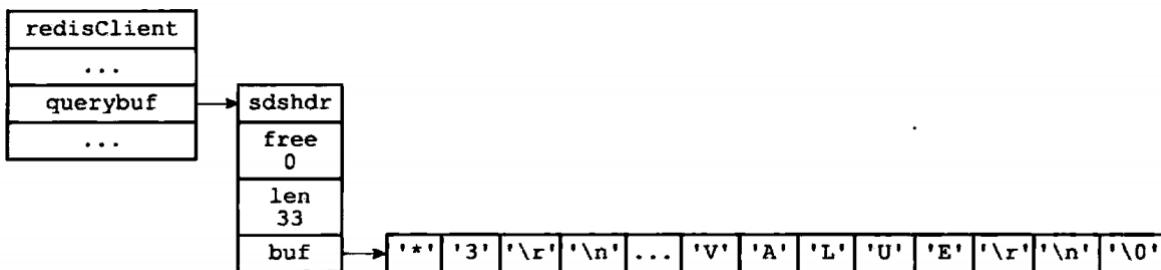


图 14-2 客户端状态中的命令请求

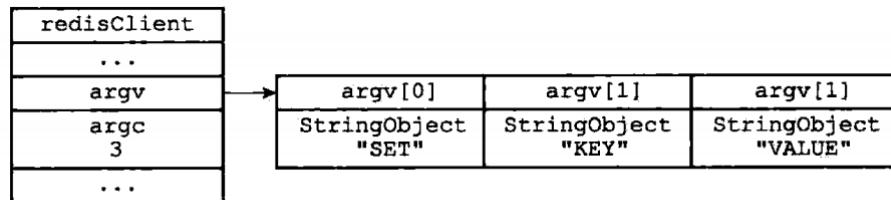


图 14-3 客户端状态的 argv 属性和 argc 属性

### 64.3、命令执行器

1. 在命令表中查找命令实现，对应的函数
2. 执行预备操作：检查命令名字是否正确、参数个数是否正确、是否通过身份验证....
3. 调用命令实现函数
4. 执行后续工作：慢查询日志、AOF写入...

### 64.4、回复给客户端

套接字变为可写状态，服务器回复，清空输出缓冲区

### 64.5、客户端接受并打印

## 65、serverCron函数

每个100ms执行一次，负责管理redis资源，保持良好运转。

- 更新服务器时间缓存：为了减少获取系统当前时间的系统调用，使用unixtime和mstime当作当前时间缓存。并不准确
- 更新LRU时钟：每10s更新一次lru属性的值。
- 更新服务器每秒执行的命令次数：每100ms执行一次。`serverCron`函数中的`trackOperationsPerSecond`函数
- 更新服务器内存峰值记录：`stat_peak_memory`属性记录了服务器的内存峰值大小
- 处理sigterm信号：对服务器状态的`shutdown_asap`属性进行检查，根据其值决定是否关闭服务器，值为1，关闭服务器。
- 管理客户端资源：调用`clientsCron`函数进行检查：1、连接是否超时；2、输入缓冲区是否溢出
- 管理数据库资源：调用`databasesCron`函数，检查数据库，删除过期键等等。
- 检查持久化操作的运行状态：`rdb_child_pid`属性和`aof_child_pid`属性记录bgsave和bgrewriteaof，命令的子进程id，用于检查是否正在执行。
  - 如果有信号到达，那么表示新的RDB文件已经生成完毕（对于`BGSAVE`命令来说），或者AOF文件已经重写完毕（对于`BGWRITEAOF`命令来说），服务器需要进行相应命令的后续操作，比如用新的RDB文件替换现有的RDB文件，或者用重写后的AOF文件替换现有的AOF文件。
  - 如果没有信号到达，那么表示持久化操作未完成，程序不做动作。

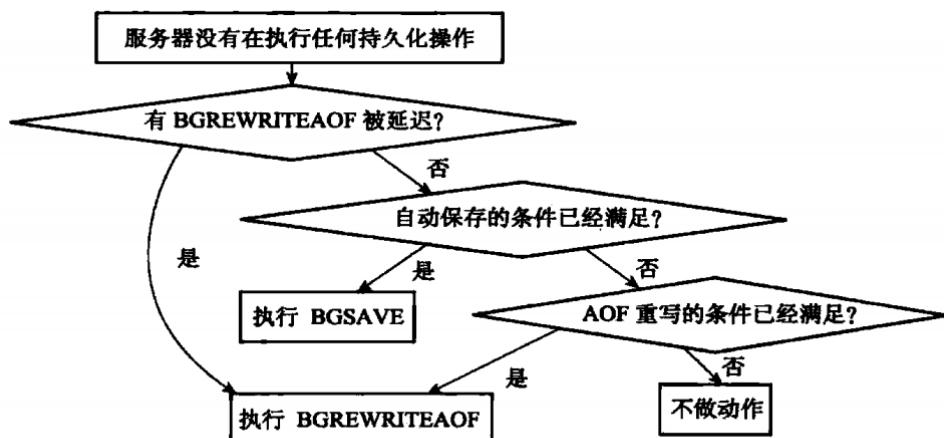


图 14-9 判断是否需要执行持久化操作

- 将AOF缓冲区的内容写入AOF文件
- 关闭异步客户端：关闭输出缓冲区大小超出限制的客户端
- 增加cronloops计数器的值：记录了serverCron函数执行的次数

## 66、初始化服务器

1. 初始化服务器状态结构：创建一个struct redisServer类型的示例变量server作为服务器的状态

- 设置服务器的运行 ID。
- 设置服务器的默认运行频率。
- 设置服务器的默认配置文件路径。
- 设置服务器的运行架构。
- 设置服务器的默认端口号。
- 设置服务器的默认 RDB 持久化条件和 AOF 持久化条件。
- 初始化服务器的 LRU 时钟。
- 创建命令表。

2. 载入配置选项：

3. 初始化服务器数据结构：

- server.clients 链表，这个链表记录了所有与服务器相连的客户端的状态结构，链表的每个节点都包含了一个redisClient结构实例。
- server.db 数组，数组中包含了服务器的所有数据库。
- 用于保存频道订阅信息的server.pubsub\_channels字典，以及用于保存模式订阅信息的server.pubsub\_patterns链表。
- 用于执行Lua脚本的Lua环境server.lua。
- 用于保存慢查询日志的server.slowlog属性。

性，而initServer函数主要负责初始化数据结构。

除了初始化数据结构之外，initServer还进行了一些非常重要的设置操作，其中包括：

- 为服务器设置进程信号处理器。
- 创建共享对象：这些对象包含Redis服务器经常用到的一些值，比如包含"OK"回复的字符串对象，包含"ERR"回复的字符串对象，包含整数1到10000的字符串对象等等，服务器通过重用这些共享对象来避免反复创建相同的对象。
- 打开服务器的监听端口，并为监听套接字关联连接应答事件处理器，等待服务器正式运行时接受客户端的连接。
- 为serverCron函数创建时间事件，等待服务器正式运行时执行serverCron函数。
- 如果AOF持久化功能已经打开，那么打开现有的AOF文件，如果AOF文件不存在，那么创建并打开一个新的AOF文件，为AOF写入做好准备。
- 初始化服务器的后台I/O模块(bio)，为将来的I/O操作做好准备。

4. 还原数据库状态：载入RDB文件或者AOF文件。

在初始化的最后一步，服务器将打印出以下日志：

[5244] 21 Nov 22:43:49.084 \* The server is now ready to accept connections on port 6379

5. 执行事件循环：并开始执行服务器的事件循环(loop)。

## 67、总

- 一个命令请求从发送到完成主要包括以下步骤：1) 客户端将命令请求发送给服务器；  
2) 服务器读取命令请求，并分析出命令参数；3) 命令执行器根据参数查找命令的实现函数，然后执行实现函数并得出命令回复；4) 服务器将命令回复返回给客户端。
- `serverCron` 函数默认每隔 100 毫秒执行一次，它的工作主要包括更新服务器状态信息，处理服务器接收的 `SIGTERM` 信号，管理客户端资源和数据库状态，检查并执行持久化操作等等。
- 服务器从启动到能够处理客户端的命令请求需要执行以下步骤：1) 初始化服务器状态；2) 载入服务器配置；3) 初始化服务器数据结构；4) 还原数据库状态；5) 执行事件循环。

## 多机数据库的实现

复制、Sentinel（哨兵）、集群、

### 复制

#### 68、旧版功能复制实现

分为两个操作：同步和命令传播

1. 同步：将从服务器的数据库状态更新为主服务器的数据库状态。
2. 命令传播：主服务器状态被修改，主从不一致时，让主从服务器数据库重新回到一致的状态。

##### 68.1、同步

从服务器向主服务器发送SYNC命令，以下是SYNC命令的执行步骤：

1. 从服务器发送SYNS命令
2. 主服务器执行BGSAVE命令，生成RDB文件，并使用缓冲区记录现在开始执行的写命令
3. BGSAVE写完，将RDB发给从
4. 将缓冲区发送给从



5. 图 15-2 主从服务器在执行 SYNC 命令期间的通信过程

##### 68.2、命令传播

主服务器会将自己执行的写命令，发送给从服务器执行

##### 68.3、缺陷

复制的两种情况：1、初次复制 2、断线后重复制，处于命令传播阶段的主从服务器中断了复制，重新连接继续复制

**断线后重复制**效率很低，因为会重新发送SYNC命令，从头开始重新复制。SYNC时一个非常耗费资源的操作

## 69、新版复制功能

**PSYNC**代替**SYNC**

**PSYNC**具有完整重同步和部分重同步两种模式：

- 完整同步用于处理初次复制，与**SYNC**步骤一样
- 部分重同步用于处理断线后复制情况：将断连期间执行的写命令发送给服务器。

### 69.1、部分重同步的实现

三部分：

- 主从服务器复制偏移量：如果主从服务器偏移量一致，那么服务器处于一致状态
- 主从服务器积压缓冲区：该缓冲区保存最近一部分传播的写命令，并且维护相应的复制偏移量。如果偏移量超出积压缓冲区，会导致完整重同步，否则执行部分重同步。
- 服务器运行ID：定向到断连前的服务器

## 70、**PSYNC**命令的实现

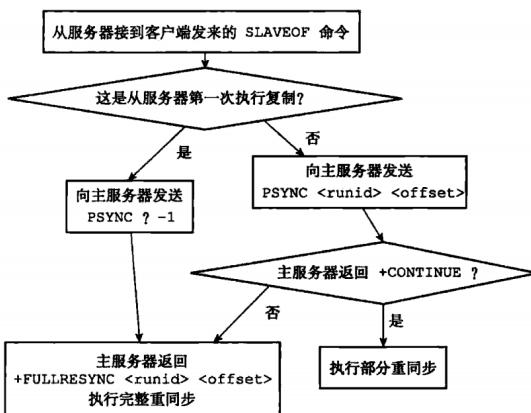


图 15-12 PSYNC 执行完整重同步和部分重同步时可能遇上的情况

## 80、复制的实现

1. 设置主服务器的地址和端口
2. 建立套接字连接
3. 发送ping命令：检查CS之间读写状态是否正常。返回超时或者错误就断开并重连主服务器
4. 身份验证
5. 发送端口信息：向主服务器发送从服务器的监听端口号
6. 同步：向主发送PSYNC，执行同步命令
7. 命令传播

## 81、心跳检测

命令传播阶段，每1s从服务器发送命令：

`REPLCONF ACK <replication_offset>`

其中 `replication_offset` 是从服务器当前的复制偏移量。

三个作用：

1. 检查主从服务器网络连接状态：
2. 辅助实现 `min_slaves` 选项：防止主服务器在不安全的情况下执行写命令，延迟太高或者从服务器数量太少
3. 检查命令丢失：命令半路丢失，心跳检测检测到偏移量不一致，会从积压缓冲区重新发送给服务器。

## 82、总

- Redis 2.8 以前的复制功能不能高效地处理断线后重复制情况，但 Redis 2.8 新添加的部分重同步功能可以解决这个问题。
- 部分重同步通过复制偏移量、复制积压缓冲区、服务器运行 ID 三个部分来实现。
- 在复制操作刚开始的时候，从服务器会成为主服务器的客户端，并通过向主服务器发送命令请求来执行复制步骤，而在复制操作的后期，主从服务器会互相成为对方的客户端。
- 主服务器通过向从服务器传播命令来更新从服务器的状态，保持主从服务器一致，而从服务器则通过向主服务器发送命令来进行心跳检测，以及命令丢失检测。

## 哨兵Sentinel

高可用解决方案：一个或多个哨兵实例租场哨兵系统。

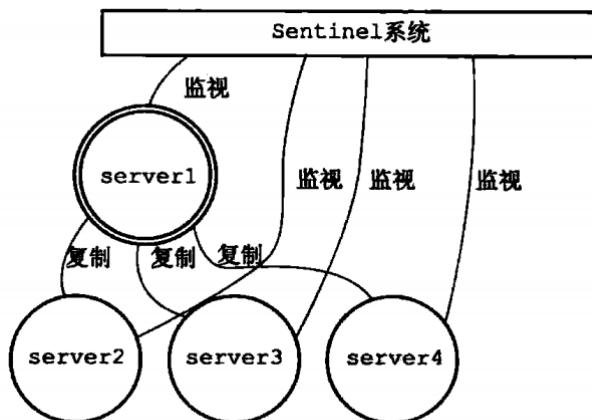


图 16-1 服务器与 Sentinel 系统

## 独立功能

发布与订阅、事务、Lua脚本、排序、二进制位数组、慢日志查询、监视器

## 发布与订阅

客户端可以订阅一个/多个频道或者模式  
PUBLISH、SUBSCRIBE、PUSHSCRIBE

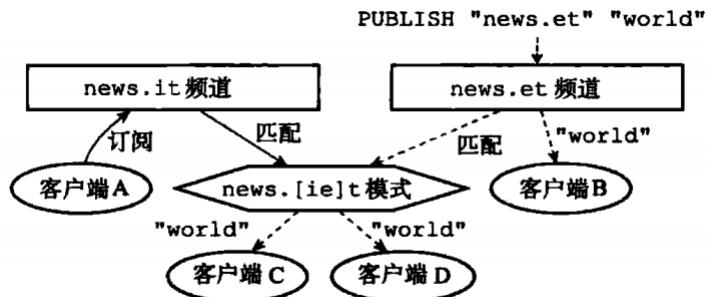


图 18-5 将消息发送给频道的订阅者和匹配模式的订阅者（2）

### 83、频道的订阅与退订

SUBSCRIBE订阅某个频道，订阅关系保存在 **pubsub\_channels** 字典

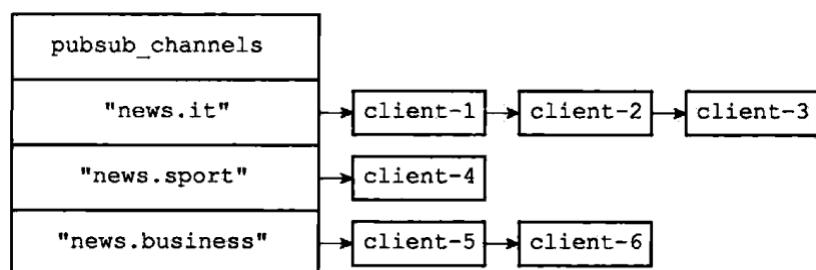


图 18-6 一个 pubsub\_channels 字典示例

退订：UNSUBSCRIBE UNSUBSCRIBE "news.sport" "news.movie"

### 84、模式的订阅和退订

保存在 **pubsub\_patterns** 里面，是一个链表，每个节点是一个pubsubPattern结构

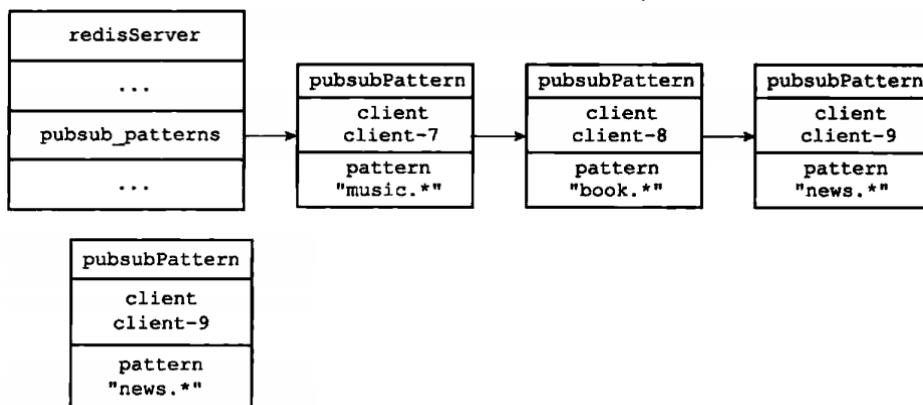


图 18-10 pubsubPattern

结构示例

PSUBSCRIBE订阅模式

PUNSUBSCRIBE退订模式

### 85、发送消息

客户端发送消息给频道，PUBLIST 发送消息 两步：

1. 将消息发送给频道所有的订阅者：查找字典、遍历链表。
2. 如果有一个或者多个模式和频道匹配，将消息发送给模式订阅者

## 86、查看订阅信息

PUBSUB CHANNELS：返回服务器当前被订阅的频道。  
PUBSUB NUMSUB：接受多个频道作为输入，返回这些频道的订阅者数量  
PUBSUB NUMPAT：返回服务器当前被订阅模式的数量。

## 87、总结

- 服务器状态在 `pubsub_channels` 字典保存了所有频道的订阅关系：`SUBSCRIBE` 命令负责将客户端和被订阅的频道关联到这个字典里面，而 `UNSUBSCRIBE` 命令则负责解除客户端和被退订频道之间的关联。
- 服务器状态在 `pubsub_patterns` 链表保存了所有模式的订阅关系：`PSUBSCRIBE` 命令负责将客户端和被订阅的模式记录到这个链表中，而 `PUNSUBSCRIBE` 命令则负责移除客户端和被退订模式在链表中的记录。
- `PUBLISH` 命令通过访问 `pubsub_channels` 字典来向频道的所有订阅者发送消息，通过访问 `pubsub_patterns` 链表来向所有匹配频道的模式的订阅者发送消息。
- `PUBSUB` 命令的三个子命令都是通过读取 `pubsub_channels` 字典和 `pubsub_patterns` 链表中的信息来实现的。

## 二进制位数组

```
redis> SETBIT bit 0 1      # 0000 0001
SETBIT: (integer) 0

redis> GETBIT bit 3 # 0000 1000
GETBIT: (integer) 1

redis> BITCOUNT bit    # 0000 1000
BITCOUNT: (integer) 1

BITOP: 对多个位数组进行按位与、或、异或运算
redis> BITOP AND and-result x y z      # 0000 0000
(integer) 1

redis> BITOP OR or-result x y z        # 0000 1111
(integer) 1

redis> BITOP XOR xor-result x y z      # 0000 1000
(integer) 1
```

## 88、位数组表示

因为SDS是二进制安全的，所以使用SDS保存。

- `redisObject.type` 的值为 `REDIS_STRING`，表示这是一个字符串对象。
- `sds hdr.len` 的值为 1，表示这个 SDS 保存了一个一字节长的位数组。
- `buf` 数组中的 `buf[0]` 字节保存了一字节长的位数组。
- `buf` 数组中的 `buf[1]` 字节保存了 SDS 程序自动追加到值的末尾的空字符 '\0'。

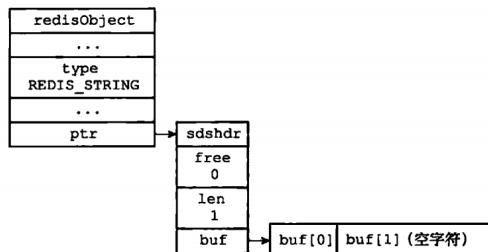


图 22-1 SDS 表示的位数组

注：保存数组的顺序和平时书写数组的顺序相反，10110010表示为数组为01001101，**使用逆序保存数组可以简化SETBIT命令的实现。**

## 89、GETBIT

1. offset/8计算指定二进制位保存在数组的哪个字节
2. offset mod 8 + 1, 计算指定的二进制位是字节的第几个二进制位
3. 定位这个位的值

## 90、SETBIT

返回旧值

1. len = offset/8 + 1 计算保存offset指定的二进制位至少需要检查多少字节。
2. 检查bitarray键保存的位数组长度是否小于len, 小于则扩展为len字节, 并将新扩展的二进制位设置为0.
3. offset/8, 计算指定二进制位保存在数组的哪个字节
4. offset mod 8 + 1, 计算指定的二进制位是字节的第几个二进制位
5. 定位该位的值、并修改
6. 返回旧值

时间复杂度  $O(1)$

## 91、BITCOUNT

### 91.1、遍历

效率低

### 91.2、查表算法

对于一个有限长度的位数组来说，它可以表示的二进制位排列也是有限的

创建一种表，表的键是某种排列的位数组，值为二进制位的数量，根据输入的位数组进行查表。

键 (位数组)	值 (值为 1 的位数量)
0000 0000	0
0000 0001	1
0000 0010	1
0000 0011	2
0000 0100	1
0000 0101	2
0000 0110	2
0000 0111	3
...	...
1111 1101	7
1111 1110	7
1111 1111	8

典型的空间换时间

限制：内存限制、cpu缓存，查表缓存的不命中概率和表格大小成正比

### 91.3、计算汉明重量，variable-precision SWAP算法

通过一系列位移和位运算操作，在常数时间内计算多个字节的汉明重量，并且不需要任何额外的内存。

```
uint32_t swar(uint32_t i) {
    // 步骤 1
    i = (i & 0x55555555) + ((i >> 1) & 0x55555555);

    // 步骤 2
    i = (i & 0x33333333) + ((i >> 2) & 0x33333333);

    // 步骤 3
    i = (i & 0x0F0F0F0F) + ((i >> 4) & 0x0F0F0F0F);

    // 步骤 4
    i = (i*(0x01010101) >> 24);

    return i;
}
```

步骤：

1. 计算出的二进制位表示可以按照每**两个二进制位为一组进行分组**，各组的十进制表示就是该组的汉明重量。
2. 计算出的值的二进制位按每**四个二进制位为一组进行分组**，各组的十进制表示就是该组的汉明重量。
3. 计算出的值的二进制位按每**八个二进制位为一组进行分组**，各组的十进制表示就是该组的汉明重量。
4. `i*0x01010101`计算出二进制数组的汉明重量并记录在二进制位的最高八位，而`>>24`语句则通过右移运算，将二进制位的汉明重量移动到最低八位，得出的结果就是二进制位数组的汉明重量。

每次执行可以计算32个二进制位的汉明重量，，比遍历块32倍，而且因为是单纯的计算，不会浪费空间

#### 91.4、reids的实现

##### 使用查表和variable-precision SWAP算法

步骤：

1. 使用键长8位的表
2. 每次循环载入128个二进制位，调用四次32位的variable-precision SWAP算法来计算128个二进制位的汉明重量。

未处理的二进制位大于128，使用variable-precision SWAP算法，否则使用查表

## 92、BITOP

直接基于C语言的基于字节的逻辑运算

## 93、bitarray总

- Redis 使用 SDS 来保存位数组。
- SDS 使用逆序来保存位数组，这种保存顺序简化了 `SETPBIT` 命令的实现，使得 `SETPBIT` 命令可以在不移动现有二进制位的情况下，对位数组进行空间扩展。
- `BITCOUNT` 命令使用了查表算法和 variable-precision SWAR 算法来优化命令的执行效率。
- `BITOP` 命令的所有操作都使用 C 语言内置的位操作来实现。