课前准备

- 准备redis安装包

课堂主题

Redis底层数据结构、缓存淘汰策略、Redis事务、Redis乐观锁

课堂目标

- 掌握Redis数据类型的底层数据结构
- 理解LRU
- 能够编写Redis事务处理，理解弱事务
- 理解Redis乐观锁及秒杀的实现

知识要点

Redis内存模型

Redis内存统计

```
127.0.0.1:6379>  info memory
# Memory
#Redis分配的内存总量,包括虚拟内存(字节)
used_memory:853464
#占操作系统的内存，不包括虚拟内存(字节)
used_memory_rss:12247040
#内存碎片比例 如果小于0说明使用了虚拟内存
mem_fragmentation_ratio:15.07
#Redis使用的内存分配器
mem_allocator:jemalloc-5.1.0
```

Redis内存分配

数据

作为数据库，数据是最主要的部分；这部分占用的内存会统计在 used_memory 中。

Redis 使用键值对存储数据，其中的值（对象）包括 5 种类型，即字符串、哈希、列表、集合、有序集合。

这 5 种类型是 Redis 对外提供的，实际上，在 Redis 内部，每种类型可能有 2 种或更多的内部编码实现。

进程

Redis 主进程本身运行肯定需要占用内存，如代码、常量池等等；这部分内存大约几M，在大多数生产环境中与 Redis 数据占用的内存相比可以忽略。

这部分内存不是由 jemalloc 分配，因此不会统计在 used_memory 中。

补充说明：除了主进程外，Redis 创建的子进程运行也会占用内存，如 Redis 执行 AOF、RDB 重写时创建的子进程。

当然，这部分内存不属于 Redis 进程，也不会统计在 used_memory 和 used_memory_rss 中。

缓冲内存

缓冲内存包括客户端缓冲区、复制积压缓冲区、AOF 缓冲区等；其中，客户端缓冲区存储客户端连接的输入输出缓冲；复制积压缓冲区用于部分复制功能；AOF 缓冲区用于在进行 AOF 重写时，保存最近的写入命令。

在了解相应功能之前，不需要知道这些缓冲的细节；这部分内存由 jemalloc 分配，因此会统计在 used_memory 中。

内存碎片

内存碎片是 Redis 在分配、回收物理内存过程中产生的。例如，如果对数据的更改频繁，而且数据之间的大小相差很大，可能导致 Redis 释放的空间在物理内存中并没有释放。

但 Redis 又无法有效利用，这就形成了内存碎片，内存碎片不会统计在 used_memory 中。

内存碎片的产生与对数据进行的操作、数据的特点等都有关；此外，与使用的内存分配器也有关系：如果内存分配器设计合理，可以尽可能的减少内存碎片的产生。如果 Redis 服务器中的内存碎片已经很大，可以通过安全重启的方式减小内存碎片：因为重启之后，Redis 重新从备份文件中读取数据，在内存中进行重排，为每个数据重新选择合适的内存单元，减小内存碎片。

Redis数据结构

Redis 没有直接使用 C 字符串(即以空字符'\0'结尾的字符数组)作为默认的字符串表示，而是使用了 SDS。SDS 是简单动态字符串(Simple Dynamic String)的缩写。

它是自己构建了一种名为 简单动态字符串（simple dynamic string,SDS）的抽象类型，并将 SDS 作为 Redis的默认字符串表示。

SDS 定义：

```
struct sdshdr{
    //记录buf数组中已使用字节的数量
    //等于 SDS 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[];
}
```

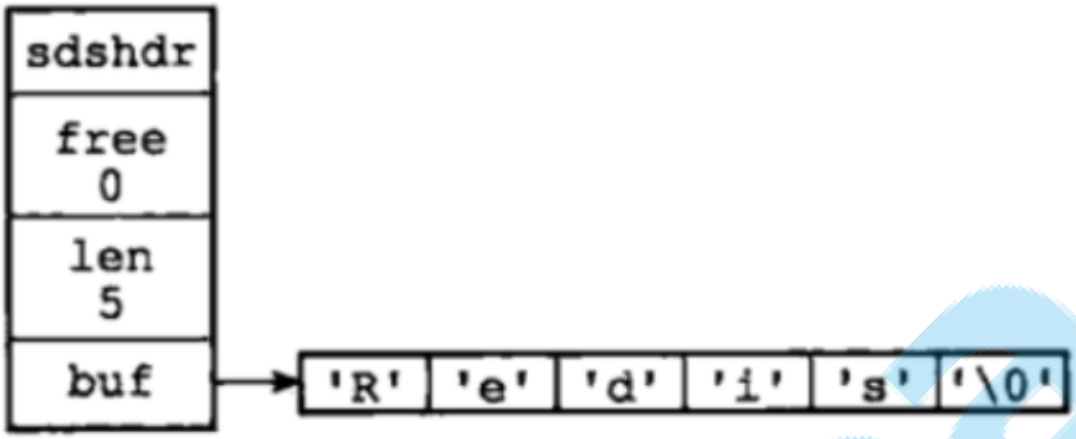
我们看上面对于 SDS 数据类型的定义：

1、len 保存了SDS保存字符串的长度

2、buf[] 数组用来保存字符串的每个元素

3、free j记录了 buf 数组中未使用的字节数量

简单动态字符串（SDS）



buf数组的长度=free+len+1

好处：

SDS 在 C 字符串的基础上加入了 free 和 len 字段，带来了很多好处：

获取字符串长度：SDS 是 O(1)，C 字符串是 O(n)。

缓冲区溢出：使用 C 字符串的 API 时，如果字符串长度增加（如 strcat 操作）而忘记重新分配内存，很容易造成缓冲区的溢出。

而 SDS 由于记录了长度，相应的 API 在可能造成缓冲区溢出时会自动重新分配内存，杜绝了缓冲区溢出。

修改字符串时内存的重分配：对于 C 字符串，如果要修改字符串，必须要重新分配内存（先释放再申请），因为如果没有重新分配，字符串长度增大时会造成内存缓冲区溢出，字符串长度减小时会造成内存泄露。

而对于 SDS，由于可以记录 len 和 free，因此解除了字符串长度和空间数组长度之间的关联，可以在此基础上进行优化。

空间预分配策略（即分配内存时比实际需要的多）使得字符串长度增大时重新分配内存的概率大大减小；惰性空间释放策略使得字符串长度减小时重新分配内存的概率大大减小。

存取二进制数据：SDS 可以，C 字符串不可以。因为 C 字符串以空字符作为字符串结束的标识，而对于一些二进制文件（如图片等）。

内容可能包括空字符串，因此 C 字符串无法正确存取；而 SDS 以字符串长度 len 来作为字符串结束标识，因此没有这个问题。

此外，由于 SDS 中的 buf 仍然使用了 C 字符串（即以'\0'结尾），因此 SDS 可以使用 C 字符串库中的部分函数。

但是需要注意的是，只有当 SDS 用来存储文本数据时才可以这样使用，在存储二进制数据时则不行（'\0'不一定是结尾）。

使用：

所有的key

数据里的字符串

AOF缓冲区和用户输入缓冲

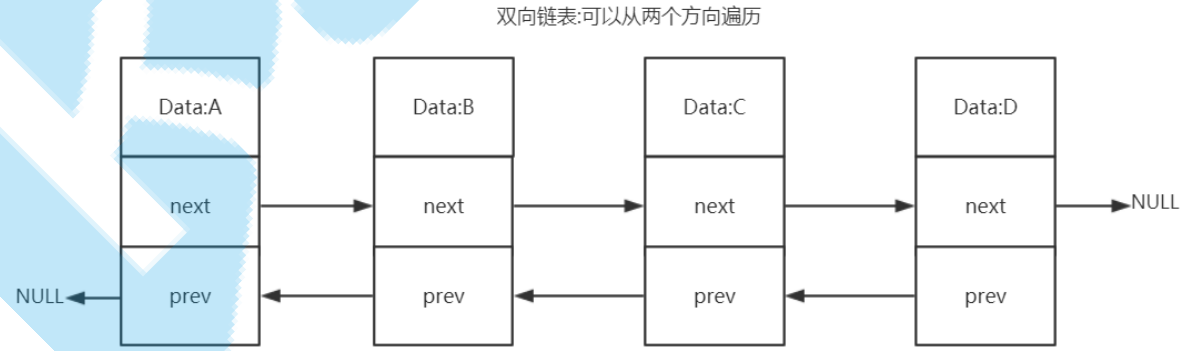
链表

链表和列表区别

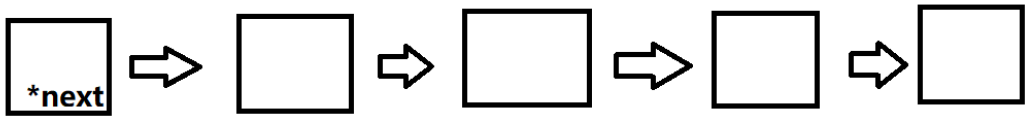
linkedlist和arraylist

链表在Redis中的应用非常广泛，列表(List)的底层实现之一就是双向链表。此外发布与订阅、慢查询、监视器等功能也用到了链表。

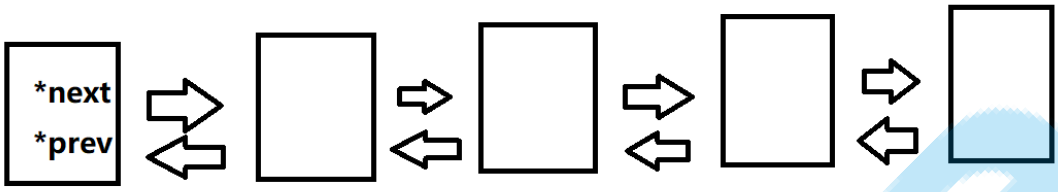
```
typedef struct listNode {
    //前置节点
    struct listNode *prev;
    //后置节点
    struct listNode *next;
    //节点的值
    void *value;
}listNode


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);
 } list;
```



单向链表



双向链表



Redis链表优势：

①、双向：链表具有前置节点和后置节点的引用，获取这两个节点时间复杂度都为O(1)。

与传统链表（单链表）相比，Redis链表结构的优势有：

普通链表（单链表）：节点类保留下一节点的引用。链表类只保留头节点的引用，只能从头节点插入删除

②、无环：表头节点的 prev 指针和表尾节点的 next 指针都指向 NULL,对链表的访问都是以 NULL 结束。

③、带链表长度计数器：通过 len 属性获取链表长度的时间复杂度为 O(1)。

④、多态：链表节点使用 void* 指针来保存节点值，可以保存各种不同类型的值。

字典

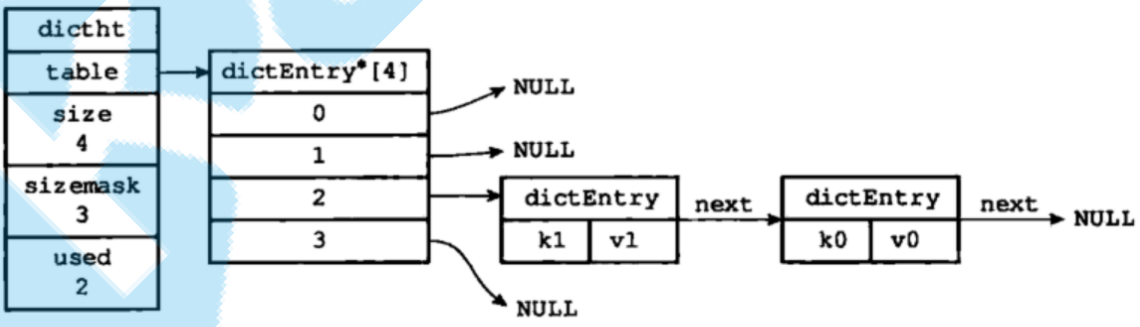


图 4-2　连接在一起的键 K1 和键 K0

字典又称为符号表或者关联数组、或映射（map），是一种用于保存键值对的抽象数据结构。

字典中的每一个键 key 都是唯一的，通过 key 可以对值来进行查找或修改。

Redis 的字典使用哈希表作为底层实现。

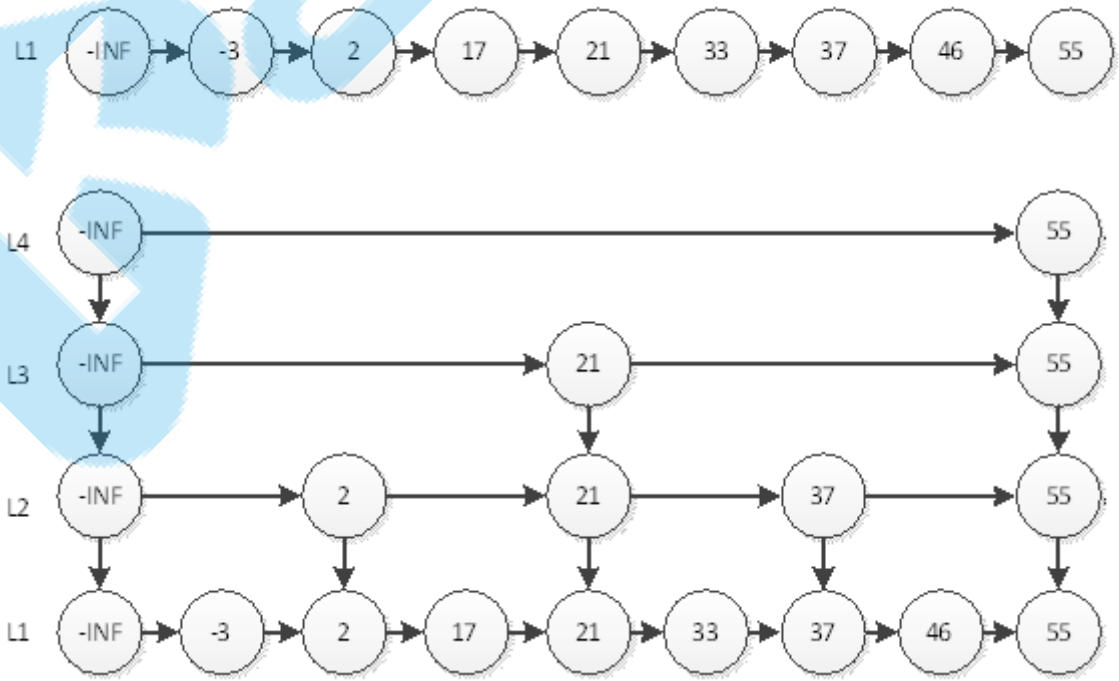
哈希（作为一种数据结构），不仅是 Redis 对外提供的 5 种对象类型的一种（hash），也是 Redis 作为 Key-Value 数据库所使用的数据结构。

```
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;

}dictht
/*哈希表是由数组 table 组成，table 中每个元素都是指向 dict.h/dictEntry 结构，
dictEntry 结构定义如下：
*/
typedef struct dictEntry{
    //键
    void *key;
    //值
    union{
        void *val;
        uint64_tu64;
        int64_ts64;
    }v;

    //指向下一个哈希表节点，形成链表
    struct dictEntry *next;
}dictEntry
```

跳跃表

普通单链表查询一个元素的时间复杂度为O(n)，即使该单链表是有序的。



查询

查找一个节点时，我们只需从高层到低层，一个个链表查找，每次找到该层链表中小于等于目标节点的最大节点，直到找到为止。由于高层的链表迭代时会“跳过”低层的部分节点，所以跳跃表会比正常的链表查找少查部分节点，这也是skiplist名字的由来。

例如：

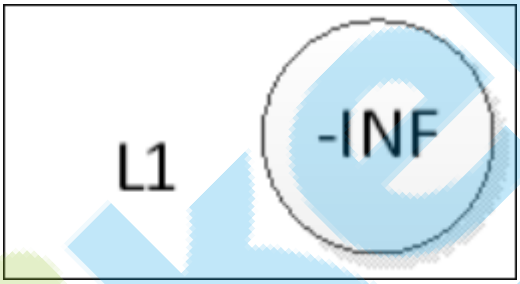
查找46： 55---21---55--37--55--46

插入

L1 层

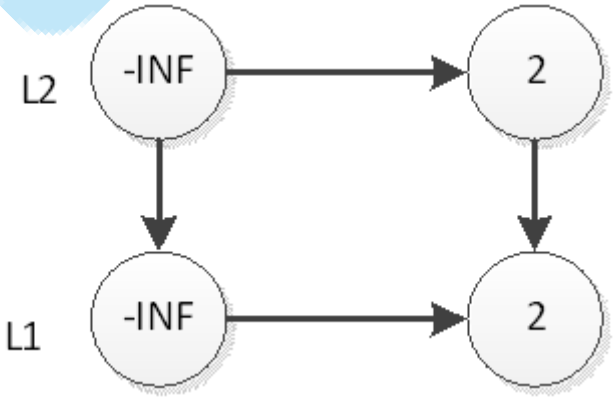
概率算法

在此还是以上图为例：跳跃表的初试状态如下图，表中没有一个元素：

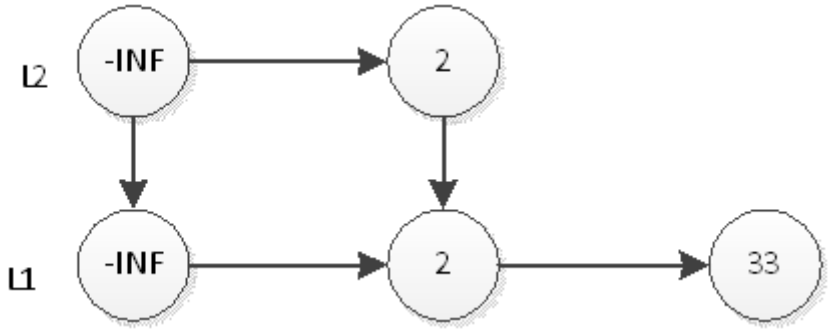


如果我们要插入元素2，首先是在底部插入元素2，如下图：

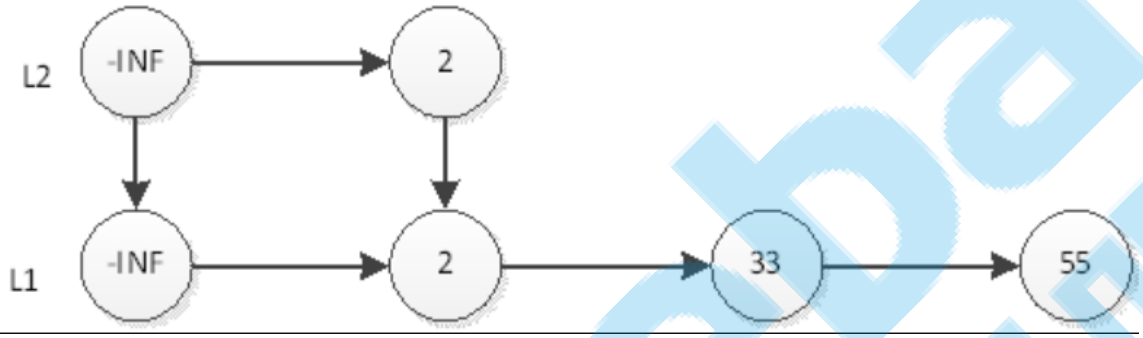
然后我们抛硬币，结果是正面，那么我们要将2插入到L2层，如下图



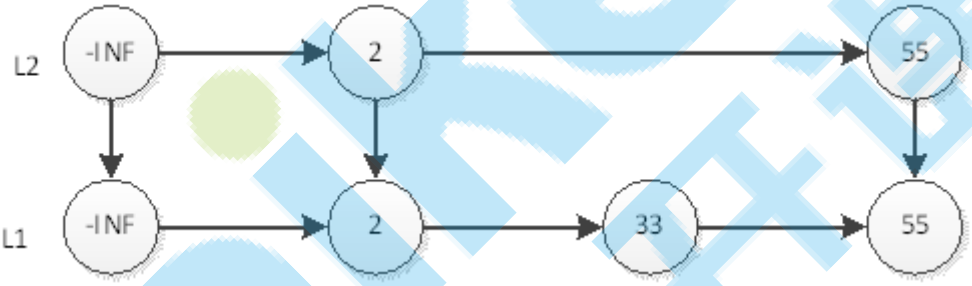
继续抛硬币，结果是反面，那么元素2的插入操作就停止了，插入后的表结构就是上图所示。接下来，我们插入元素33，跟元素2的插入一样，现在L1层插入33，如下图：



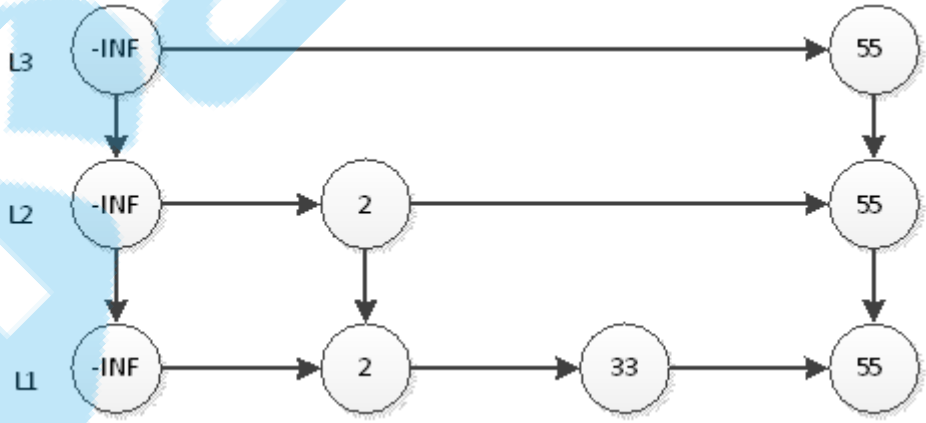
然后抛硬币，结果是反面，那么元素33的插入操作就结束了，插入后的表结构就是上图所示。接下来，我们插入元素55，首先在L1插入55，插入后如下图：



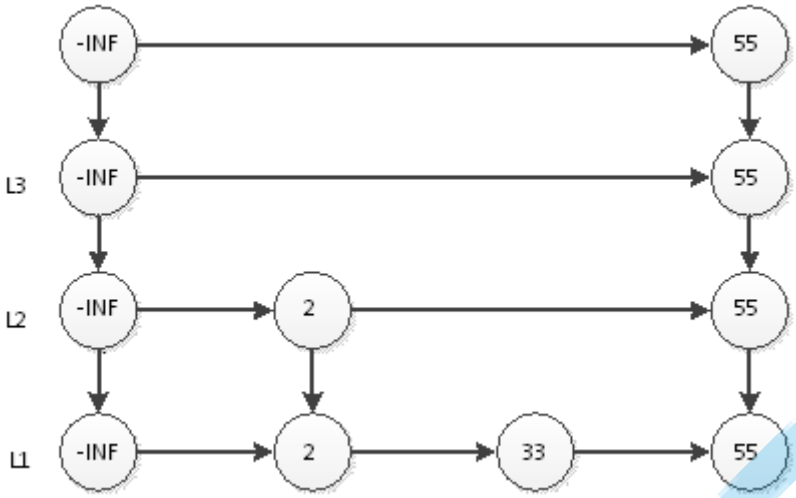
然后抛硬币，结果是正面，那么L2层需要插入55，如下图：



继续抛硬币，结果又是正面，那么L3层需要插入55，如下图：



继续抛硬币，结果又是正面，那么要在L4插入55，结果如下图：



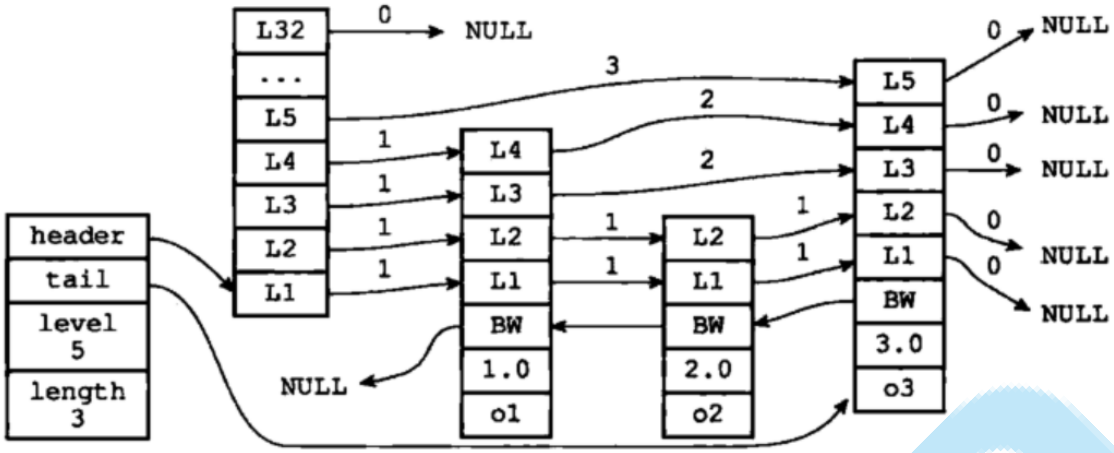
继续抛硬币，结果是反面，那么55的插入结束，表结构就如上图所示。

以此类推，我们插入剩余的元素。当然因为规模小，结果很可能不是一个理想的跳跃表。但是如果元素个数n的规模很大，学过概率论的同学都知道，最终的表结构肯定非常接近于理想跳跃表（隔一个一跳）。

删除

直接删除元素，然后调整一下删除元素后的指针即可。跟普通的链表删除操作完全一样。

```
typedef struct zskiplistNode {
      //层
      struct zskiplistLevel{
            //前进指针 后边的节点
            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

--链表
typedef struct zskiplist{
      //表头节点和表尾节点
      structz skiplistNode *header, *tail;
      //表中节点的数量
      unsigned long length;
      //表中层数最大的节点的层数
      int level;

}zskiplist;
```



①、搜索：从最高层的链表节点开始，如果比当前节点要大和比当前层的下一个节点要小，那么则往下找，也就是和当前层的下一层的节点的下一个节点进行比较，以此类推，一直找到最底层的最后一个节点，如果找到则返回，反之则返回空。

②、插入：首先确定插入的层数，有一种方法是假设抛一枚硬币，如果是正面就累加，直到遇见反面为止，最后记录正面的次数作为插入的层数。当确定插入的层数k后，则需要将新元素插入到从底层到k层。

③、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。

整数集合

整数集合（intset）是集合（set）的底层实现之一，当一个集合（set）只包含整数值元素，并且这个集合的元素不多时，Redis就会使用整数集合(intset)作为该集合的底层实现。整数集合（intset）是Redis用于保存整数值的集合抽象数据类型，它可以保存类型为int16_t、int32_t 或者int64_t 的整数值，并且保证集合中不会出现重复元素。

```
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;
```

压缩列表

压缩列表（ziplist）是列表键和哈希键的底层实现之一。当一个列表只包含少量列表项时，并且每个列表项时小整数值或短字符串，那么Redis会使用压缩列表来做该列表的底层实现。

压缩列表（ziplist）是Redis为了节省内存而开发的，是由一系列特殊编码的连续内存块组成的顺序型数据结构，一个压缩列表可以包含任意多个节点（entry），每个节点可以保存一个字节数组或者一个整数值。

放到一个连续内存区

压缩列表的每个节点构成如下：

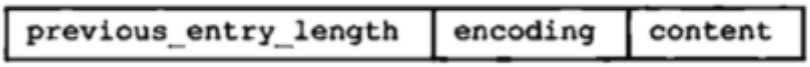


图 7-4　压缩列表节点的各个组成部分

previous_entry_ength：记录压缩列表前一个字节的长度。

encoding：节点的encoding保存的是节点的content的内容类型

content：content区域用于保存节点的内容，节点内容类型和长度由encoding决定。

对象

前面我们讲了Redis的数据结构，Redis不是用这些数据结构直接实现Redis的键值对数据库，而是基于这些数据结构创建了一个对象系统。包含字符串对象，列表对象，哈希对象，集合对象和有序集合对象。根据对象的类型可以判断一个对象是否可以执行给定的命令，也可针对不同的使用场景，对象设置有多种不同的数据结构实现，从而优化对象在不同场景下的使用效率。

Redis中的每个对象都是由如下结构表示（列出了与保存数据有关的三个属性）

```
typedef struct redisObject {
    unsigned type:4;//类型 五种对象类型
    unsigned encoding:4;//编码
    void *ptr;//指向底层实现数据结构的指针
    //...
    int refcount;//引用计数
    //...
    unsigned lru:22;//记录最后一次被命令程序访问的时间
    //...
}robj;
```

type

type 字段表示对象的类型，占 4 个比特；目前包括 REDIS_STRING(字符串)、REDIS_LIST (列表)、REDIS_HASH(哈希)、REDIS_SET(集合)、REDIS_ZSET(有序集合)。

当我们执行 type 命令时，便是通过读取 RedisObject 的 type 字段获得对象的类型，如下所示：

127.0.0.1:6379> type a1 string

encoding

encoding 表示对象的内部编码，占 4 个比特。对于 Redis 支持的每种类型，都有至少两种内部编码，例如对于字符串，有 int、embstr、raw 三种编码。

通过 encoding 属性，Redis 可以根据不同的使用场景来为对象设置不同的编码，大大提高了 Redis 的灵活性和效率。

以列表对象为例，有压缩列表和双端链表两种编码方式；如果列表中的元素较少，Redis 倾向于使用压缩列表进行存储，因为压缩列表占用内存更少，而且比双端链表可以更快载入。

当列表对象元素较多时，压缩列表就会转化为更适合存储大量元素的双端链表。

通过 object encoding 命令，可以查看对象采用的编码方式，如下所示：

127.0.0.1:6379> object encoding a1 "int"

lru

lru 记录的是对象最后一次被命令程序访问的时间，占据的比特数不同的版本有所不同（如 4.0 版本占 24 比特，2.6 版本占 22 比特）。

通过对比 lru 时间与当前时间，可以计算某个对象的空转时间；object idletime 命令可以显示该空转时间（单位是秒）。object idletime 命令的一个特殊之处在于它不改变对象的 lru 值。

lru 值除了通过 object idletime 命令打印之外，还与 Redis 的内存回收有关系。

如果 Redis 打开了 maxmemory 选项，且内存回收算法选择的是 volatile-lru 或 allkeys—lru，那么当 Redis 内存占用超过 maxmemory 指定的值时，Redis 会优先选择空转时间最长的对象进行释放。

refcount

refcount 与共享对象：refcount 记录的是该对象被引用的次数，类型为整型。refcount 的作用，主要在于对象的引用计数和内存回收。

当创建新对象时，refcount 初始化为 1；当有新程序使用该对象时，refcount 加 1；当对象不再被一个新程序使用时，refcount 减 1；当 refcount 变为 0 时，对象占用的内存会被释放。

Redis 中被多次使用的对象(refcount>1)，称为共享对象。Redis 为了节省内存，当有一些对象重复出现时，新的程序不会创建新的对象，而是仍然使用原来的对象。

这个被重复使用的对象，就是共享对象。目前共享对象仅支持整数值的字符串对象。

共享对象的引用次数可以通过 object refcount 命令查看，如下所示。命令执行的结果页佐证了只有 0~9999 之间的整数会作为共享对象。

127.0.0.1:6379> object refcount a1 (integer) 2147483647

ptr

ptr 指针指向具体的数据，比如：set hello world，ptr 指向包含字符串 world 的 SDS。

综上所述，RedisObject 的结构与对象类型、编码、内存回收、共享对象都有关系。

类型	编码	OBJECT ENCODING命令输出	对象
REDIS_STRING	REDIS_ENCODING_INT	“int”	使用整数值实现的字符串对象
REDIS_STRING	REDIS_ENCODING_EMBSTR	“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	“raw”	使用简单动态字符串实现的字符串对象
REDIS_LIST	REDIS_ENCODING_ZIPLIST	“ziplist”	使用压缩列表实现的列表对象
REDIS_LIST	REDIS_ENCODING_LINKEDLIST	“linkedlist”	使用双端链表实现的列表对象
REDIS_HASH	REDIS_ENCODING_ZIPLIST	“ziplist”	使用压缩列表实现的哈希对象
REDIS_HASH	REDIS_ENCODING_HT	“hashtable”	使用字典实现的哈希对象
REDIS_SET	REDIS_ENCODING_INTSET	“intset”	使用整数集合实现的集合对象
REDIS_SET	REDIS_ENCODING_HT	“hashtable”	使用字典实现的集合对象
REDIS_ZSET	REDIS_ENCODING_ZIPLIST	“ziplist”	使用压缩列表实现的有序集合对象
REDIS_ZSET	REDIS_ENCODING_SKIPLIST	“skiplist”	使用跳跃表和字典实现的有序集合对象

缓存淘汰策略

最大缓存

- 在 redis 中，允许用户设置最大使用内存大小**maxmemory**，默认为0，没有指定最大缓存，如果有新的数据添加，超过最大内存，则会使redis崩溃，所以一定要设置。
- redis 内存数据集大小上升到一定大小的时候，就会实行**数据淘汰策略。**

淘汰策略

redis淘汰策略配置：**maxmemory-policy** voltile-lru，支持热配置

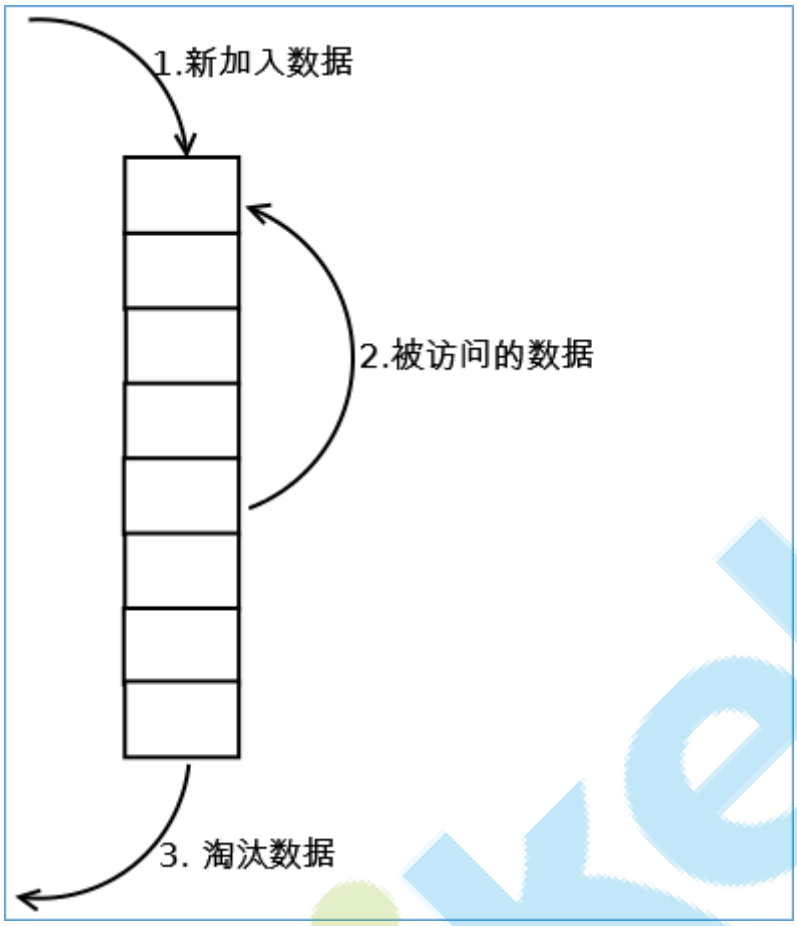
redis 提供 6种数据淘汰策略：

1. volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰
2. volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰
3. volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰
4. allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰
5. allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰
6. no-enviction（驱逐）：禁止驱逐数据

LRU原理

LRU（**Least recently used**，最近最少使用）算法根据数据的历史访问记录来进行淘汰数据，其核心思想是“如果数据最近被访问过，那么将来被访问的几率也更高”。

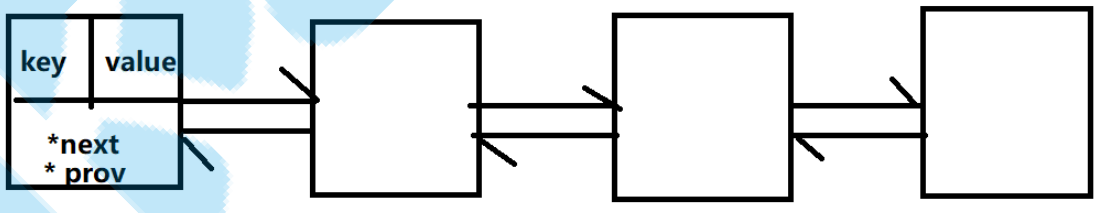
最常见的实现是使用一个链表保存缓存数据，详细算法实现如下：



1. 新数据插入到链表头部；
2. 每当缓存命中（即缓存数据被访问），则将数据移到链表头部；
3. 当链表满的时候，将链表尾部的数据丢弃。

在Java中可以使用**LinkHashMap**去实现LRU

利用哈希链表实现

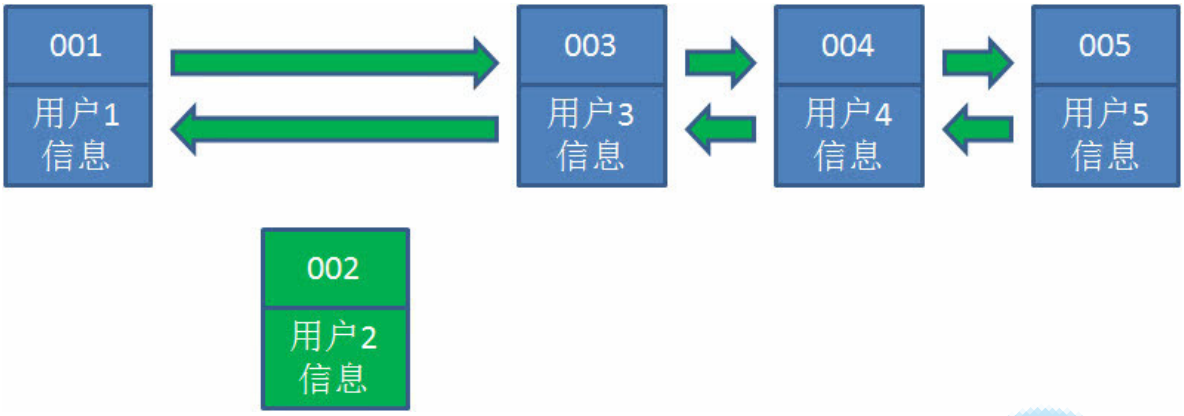


让我们以用户信息的需求为例，来演示一下LRU算法的基本思路：

1.假设我们使用哈希链表来缓存用户信息，目前缓存了4个用户，这4个用户是按照时间顺序依次从链表右端插入的。

2.此时，业务方访问用户5，由于哈希链表中没有用户5的数据，我们从数据库中读取出来，插入到缓存当中。这时候，链表中最右端是最新访问到的用户5，最左端是最近最少访问的用户1。

3.接下来，业务方访问用户2，哈希链表中存在用户2的数据，我们怎么做呢？我们把用户2从它的前驱节点和后继节点之间移除，重新插入到链表最右端。这时候，链表中最右端变成了最新访问到的用户2，最左端仍然是最近最少访问的用户1。



4.接下来，业务方请求修改用户4的信息。同样道理，我们把用户4从原来的位置移动到链表最右侧，并把用户信息的值更新。这时候，链表中最右端是最新访问到的用户4，最左端仍然是最近最少访问的用户1。

5.后来业务方换口味了，访问用户6，用户6在缓存里没有，需要插入到哈希链表。假设这时候缓存容量已经达到上限，必须先删除最近最少访问的数据，那么位于哈希链表最左端的用户1就会被删除掉，然后再把用户6插入到最右端。

以上，就是LRU算法的基本思路。

https://www.itcodemonkey.com/article/11153.html

Redis事务典型应用—Redis乐观锁

在生产环境里，利用redis乐观锁来实现秒杀，Redis乐观锁是Redis事务的经典应用。

秒杀场景描述:

秒杀活动对稀缺或者特价的商品进行定时，定量售卖，吸引成大量的消费者进行抢购，但又只有少部分消费者可以下单成功。因此，秒杀活动将在较短时间内产生比平时大数十倍，上百倍的页面访问流量和下单请求流量。

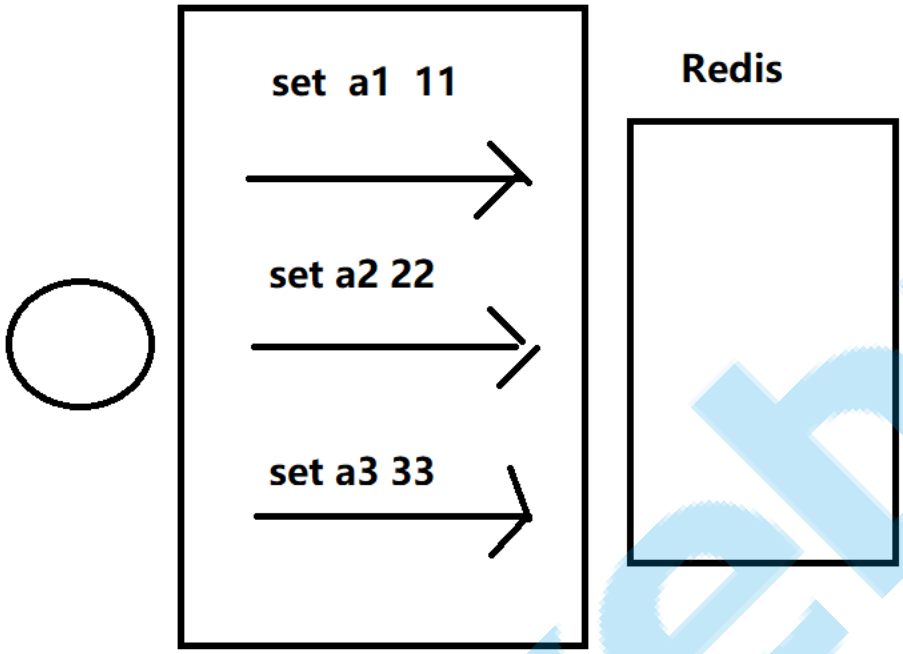
由于秒杀只有少部分请求能够成功，而大量的请求是并发产生的，所以如何确定哪个请求成功了，就是由redis乐观锁来实现。具体思路如下:

监控 锁定量，如果该值被修改成功则表示该请求被通过，反之表示该请求未通过。

从监控到修改到执行都需要在redis里操作，这样就需要用到Redis事务。

Redis事务介绍

- Redis的事务是通过MULTI、EXEC、DISCARD和WATCH这四个命令来完成的。
- Redis的单个命令都是原子性的，所以这里需要确保事务性的对象是**命令集合**。
- Redis将命令集合序列化并确保处于同一事务的**命令集合连续且不被打断**的执行
- Redis不支持回滚操作。



在一个事务中 处理命令集不会被干扰

Redis是单进程单线程的，所以不会出现线程并发。

事务命令

MULTI

用于标记事务块的开始。
Redis会将后续的命令逐个放入队列中，然后使用EXEC命令原子化地执行这个命令序列。

语法：

```
multi
```

EXEC

在一个事务中执行所有先前放入队列的命令，然后恢复正常的连接状态

语法：

```
exec
```

DISCARD

清除所有先前在一个事务中放入队列的命令，然后恢复正常的连接状态。

语法：

```
discard
```

WATCH

当某个[事务需要按条件执行]时，就要使用这个命令将给定的[键设置为受监控]的状态。

语法：

```
watch key [key…]
```

注意事项：使用该命令可以实现 `Redis` 的**乐观锁。**（后面实现）

UNWATCH

清除所有先前为一个事务监控的键。

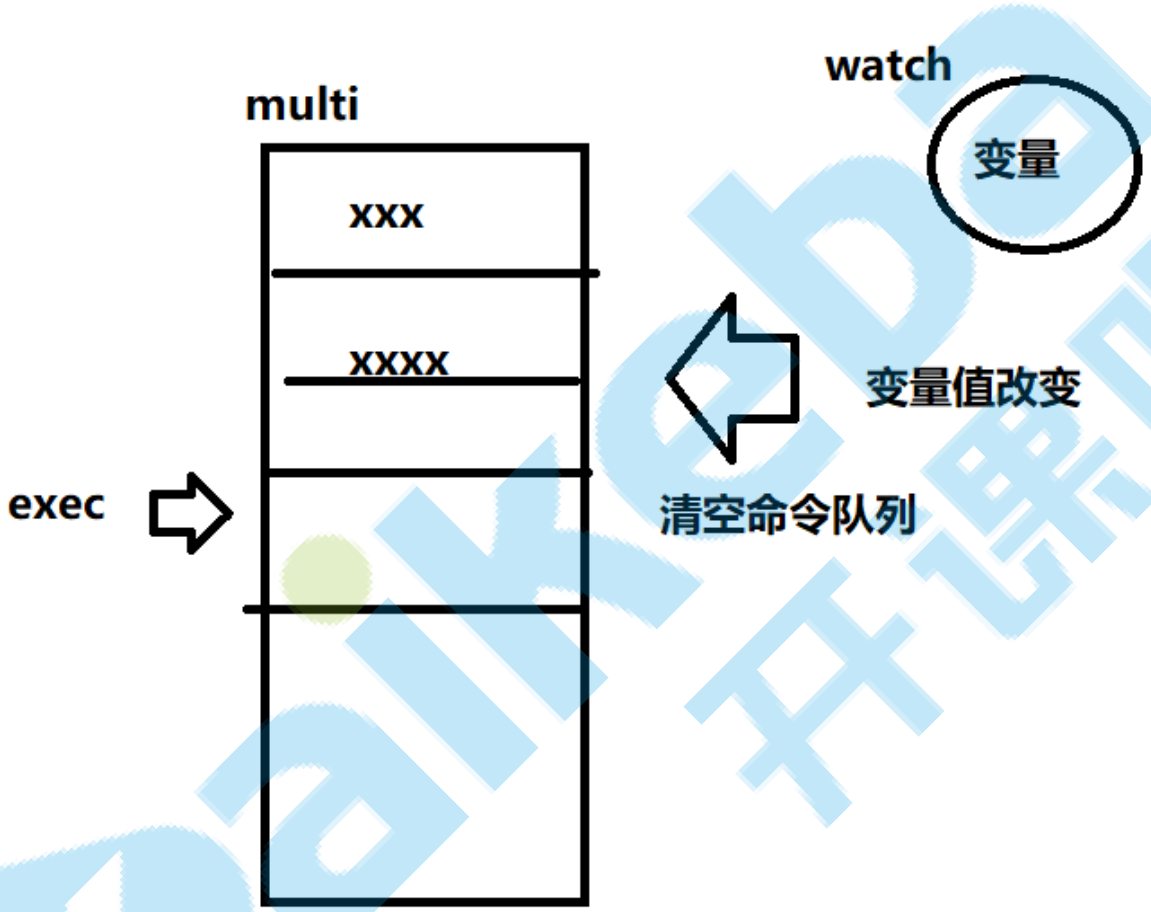
语法：

```
unwatch
```

事务演示

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set s1 111
QUEUED
127.0.0.1:6379> hset set1 name zhangsan
QUEUED
127.0.0.1:6379> exec
1) OK
2) (integer) 1
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set s2 222
QUEUED
127.0.0.1:6379> hset set2 age 20
QUEUED
127.0.0.1:6379> discard
OK
127.0.0.1:6379> exec
(error) ERR EXEC without MULTI

127.0.0.1:6379> watch s1
```

```
OK
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set s1 555
QUEUED
127.0.0.1:6379> exec          # 此时在没有exec之前，通过另一个命令窗口对监控的s1字段进行修改
(nil)
127.0.0.1:6379> get s1
111
```



事务失败处理

- `Redis`语法错误

整个事务的命令在队列里都清除

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379> sets s1 111
(error) ERR unknown command 'sets'
127.0.0.1:6379> set s1
(error) ERR wrong number of arguments for 'set' command
127.0.0.1:6379> set s4 444
QUEUED
127.0.0.1:6379> exec
(error) EXECABORT Transaction discarded because of previous errors.
127.0.0.1:6379> get s4
(nil)
```

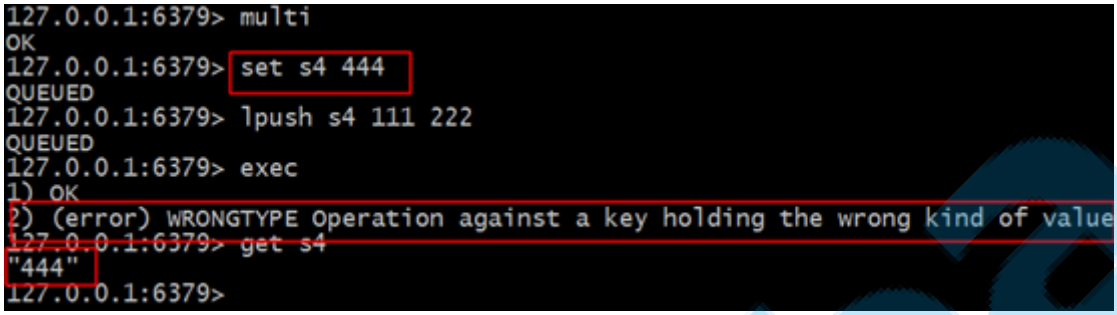
- `Redis`运行错误

在队列里正确的命令可以执行 (弱事务性)

弱事务性：

1、在队列里正确的命令可以执行 （非原子操作）

2、不支持回滚



- `Redis`不支持事务回滚（为什么呢）

  1、大多数事务失败是因为**语法错误或者类型错误**，这两种错误，在开发阶段都是可以预见的

  2、`Redis`为了**性能方面**就忽略了事务回滚。 （回滚记录历史版本）

Redis事务使用场景----Redis乐观锁

乐观锁基于CAS（Compare And Swap）思想（比较并替换），是不具有互斥性，不会产生锁等待而消耗资源，但是需要反复的重试，但也是因为重试的机制，能比较快的响应。因此我们可以利用redis来实现乐观锁。具体思路如下：

1、利用redis的watch功能，监控这个redisKey的状态值 2、获取redisKey的值 3、创建redis事务 4、给这个key的值+1 5、然后去执行这个事务，如果key的值被修改过则回滚，key不加1

```
public void watch() {
        try {
            String watchKeys = "watchKeys";
            //初始值 value=1
            jedis.set(watchKeys, 1);
            //监听key为watchKeys的值
            jedis.watch(watchkeys);

            //开启事务
            Transaction tx = jedis.multi();

            //watchKeys自增加一
            tx.incr(watchKeys);

            //执行事务，如果其他线程对watchKeys中的value进行修改，则该事务将不会执行
            //通过redis事务以及watch命令实现乐观锁
            List<Object> exec = tx.exec();
            if (exec == null) {
                System.out.println("事务未执行");
            } else {
                System.out.println("事务成功执行，watchKeys的value成功修改");
            }
        } catch (Exception e) {
            e.printStackTrace();
```

```
        } finally {
            jedis.close();
        }
}
```

Redis乐观锁实现秒杀

```
public class Second {
    public static void main(String[] arg) {
        String redisKey = "second";

        ExecutorService executorService = Executors.newFixedThreadPool(20);
        try {
            Jedis jedis = new Jedis("127.0.0.1", 6378);
            // 初始值
            jedis.set(redisKey, "0");
            jedis.close();
        } catch (Exception e) {
            e.printStackTrace();
        }

        for (int i = 0; i < 1000; i++) {

            executorService.execute(() -> {

                Jedis jedis1 = new Jedis("127.0.0.1", 6378);
                try {
                    jedis1.watch(redisKey);
                    String redisValue = jedis1.get(redisKey);
                    int valInteger = Integer.valueOf(redisValue);
                    String userInfo = UUID.randomUUID().toString();

                    // 没有秒完
                    if (valInteger < 20) {
                        Transaction tx = jedis1.multi();
                        tx.incr(redisKey);
                        List list = tx.exec();
                        // 秒成功    失败返回空list而不是空
                        if (list != null && list.size() > 0) {

                            System.out.println("用户：" + userInfo + "，秒杀成功!
当前成功人数：" + (valInteger + 1));

                        }
                        // 版本变化，被别人抢了。
                        else {
                            System.out.println("用户：" + userInfo + "，秒杀失败");
                        }
                    }
                    // 秒完了
                    else {
                        System.out.println("已经有20人秒杀成功，秒杀结束");

                    }
                } catch (Exception e) {
```

```
                    e.printStackTrace();
                } finally {
                    jedis1.close();
                }

            });
        }
        executorService.shutdown();

    }

}
```