课程主题

Redis内存模型与内存优化

课程目标

1. 熟悉Redis 内存模型
2. 掌握 Reids数据存储的细节
3. 掌握理解Redis对象类型及其内部编码（重点）
4. 估算Redis内存使用量及设计优化

一、 缓存通识

缓存：存储在计算机上的一个原始数据复制集，以便于访问。

缓存是介于数据访问者和数据源之间的一种高速存储，当数据需要多次读取的时候，用于加快读取的速度。

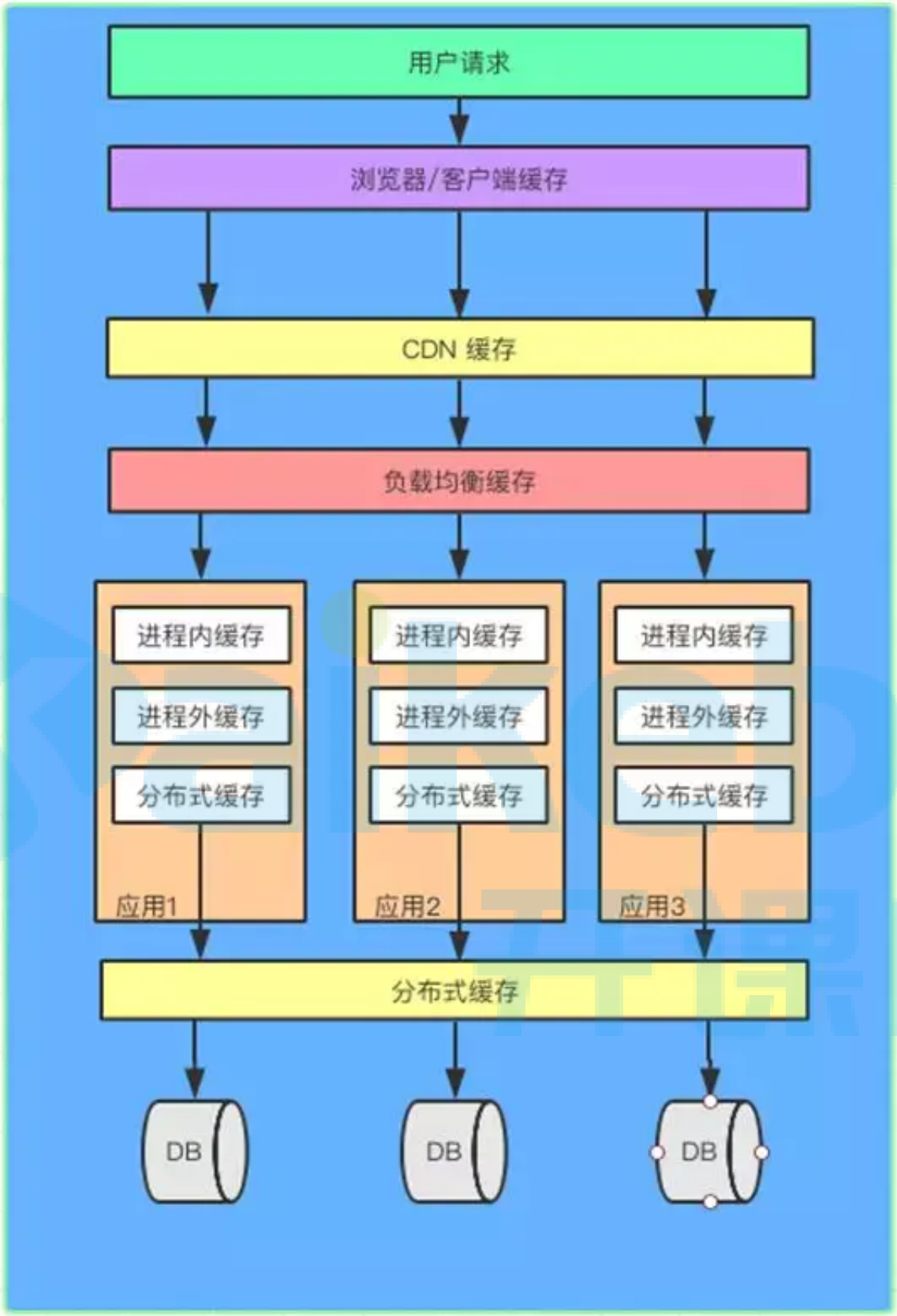
缓存(Cache) 和 缓冲(Buffer) 的分别?
缓存：一般是为了数据多次读取。

缓冲：比如CPU写到 把数据先硬盘，因为硬盘比较慢，先到缓冲设备Buffer，比如内存，Buffer读和写都需要。

1.1 无处不在的缓存

- CPU 缓存
- 操作系统缓存
- 数据库缓存
- JVM 编译缓存
- CDN 缓存
- 代理与反向代理缓存
- 前端缓存
- 应用程序缓存
- **分布式对象缓存**

1.2 多级缓存 （重点）



二、Redis简介

2.1 什么是Redis

Redis是用C语言开发的一个开源的高性能键值对（key-value）的NoSQL数据库。它通过提供多种键值数据类型来适应不同场景下的存储需求。

Redis作为一个单线程的应用，为什么处理请求性能如此NB？IO多路复用

NoSQL，泛指非关系型的数据库，NoSQL即Not-Only SQL，它可以作为关系型数据库的良好补充。

2.2 Redis的应用场景

- 缓存（数据查询、短连接、新闻内容、商品内容等等）。（最多使用）
- 分布式集群架构中的session分离。
- 聊天室的在线好友列表。
- 任务队列。（秒杀、抢购、12306等等）
- 应用排行榜。
- 网站访问统计。
- 数据过期处理（可以精确到毫秒）

三、Redis数据存储的细节

1、Redis数据类型

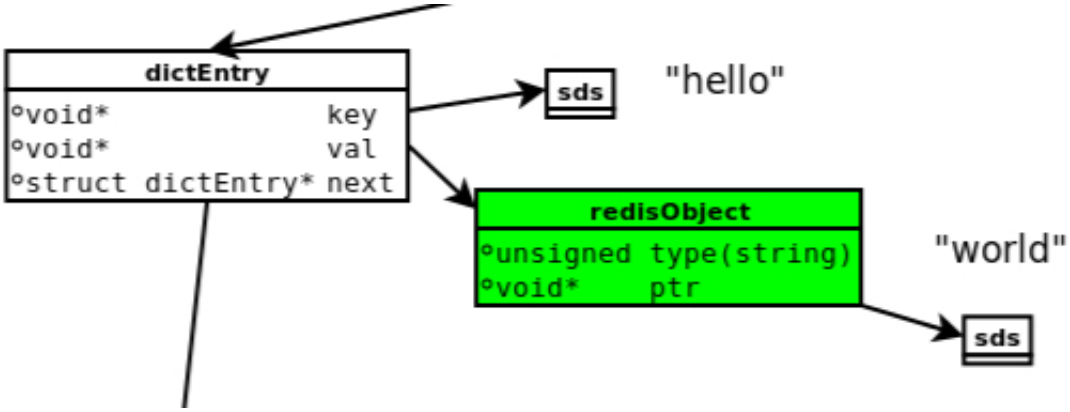
Redis整体上是一个KV结构，但是它的Value又可以分文以下五种数据类型。

目前为止Redis支持的键值数据类型如下：

- 字符串类型（string） set key value
- 散列类型(hash) hset key field value
- 列表类型(list) lpush key a b c d
- 集合类型(set) sadd key a b c d
- 有序集合类型(zset/sortedset) zadd key a score b score

2、内存结构

下图是执行set hello world时，所涉及到的数据模型。



1. **dictEntry**：Redis是Key-Value数据库，因此对每个键值对都会有一个dictEntry，里面存储了指向Key和Value的指针；next指向下一个dictEntry，与本Key-Value无关。
2. **Key**：图中右上角可见，Key（"hello"）并不是直接以字符串存储，而是存储在SDS结构中。
3. **redisObject**：Value("world")既不是直接以字符串存储，也不是像Key一样直接存储在SDS中，而是存储在redisObject中。实际上，不论Value是5种类型的哪一种，都是通过redisObject来存储的；而redisObject中的type字段指明了Value对象的类型，ptr字段则指向对象所在的地址。不过可以看出，字符串对象虽然经过了redisObject的包装，但仍然需要通过SDS存储。实际上，redisObject除了type和ptr字段以外，还有其他字段图中没有给出，如用于指定对象内部编码的字段；后面会详细介绍。
4. **jemalloc**：无论是DictEntry对象，还是redisObject、SDS对象，都需要内存分配器（如jemalloc）分配内存进行存储。

3、内存分配器

> Redis在编译时便会指定内存分配器；内存分配器可以是 libc 、jemalloc或者tcmalloc，默认是jemalloc。

jemalloc作为Redis的默认内存分配器，在减小内存碎片方面做的相对比较好。jemalloc在64位系统中，将内存空间划分为*小*、*大*、*巨大*三个范围；每个范围内又划分了许多小的内存块单位；当Redis存储数据时，会选择大小最合适的内存块进行存储。

> 在 jemalloc 类比过来的物流系统中，同城仓库相当于 tcache —— 线程独有的内存仓库；区域仓库相当于 arena —— 几个线程共享的内存仓库；全国仓库相当于全局变量指向的内存仓库，为所有线程可用。
>
> 在 jemalloc 中，整块批发内存，之后或拆开零售，或整块出售。整块批发的内存叫做 chunk，对于小件和大件订单，则进一步拆成 run。Chunk 的大小为 4MB（可调）或其倍数，且为 4MB 对齐；而 run 大小为页大小的整数倍。

在 jemalloc 中，小件订单叫做 small allocation，范围大概是 1-57344 字节。并将此区间分成 44 档，每次小分配请求归整到某档上。例如，小于8字节的，一律分配 8 字节空间；17-32分配请求，一律分配 32 字节空间。

对于上述 44 档，有对应的 44 种 runs。每种 run 专门提供此档分配的内存块（叫做 region）。

大件订单叫做 large allocation，范围大概是 57345-4MB不到一点的样子，所有大件分配归整到页大小。

jemalloc划分的内存单元如下图所示：

Category	Spacing	Size
Small	8	[8]
	16	[16, 32, 48, ..., 128]
	32	[160, 192, 224, 256]
	64	[320, 384, 448, 512]
	128	[640, 768, 896, 1024]
	256	[1280, 1536, 1792, 2048]
	512	[2560, 3072, 3584]
Large	4 KiB	[4 KiB, 8 KiB, 12 KiB, ..., 4072 KiB]
Huge	4 MiB	[4 MiB, 8 MiB, 12 MiB, ...]

> 例如，如果需要存储大小为130字节的对象，jemalloc会将其放入160字节的内存单元中。

4、redisObject

Redis对象有5种类型；无论是哪种类型，Redis都不会直接存储，而是通过redisObject对象进行存储。

redisObject对象非常重要，Redis对象的类型、内部编码、内存回收、共享对象等功能，都需要redisObject支持，下面将通过redisObject的结构来说明它是如何起作用的。

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）

```
{
  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:24;//记录最后一次被命令程序访问的时间
  //...
}robj;
```

1）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；如下图所示：

```
127.0.0.1:6379> set mystring helloredis
OK
127.0.0.1:6379> type mystring
string
127.0.0.1:6379> sadd myset member1 member2 member3
(integer) 3
127.0.0.1:6379> type myset
set
```

2）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，如下图所示：

```
127.0.0.1:6379> set key1 33
OK
127.0.0.1:6379> object encoding key1
"int"
127.0.0.1:6379> set key2 helloworld
OK
127.0.0.1:6379> object encoding key2
"embstr"
```

5种对象类型对应的编码方式以及使用条件，将在后面介绍。

3）ptr

> ptr指针指向具体的数据，如前面的例子中，set hello world，ptr指向包含字符串world的SDS。

4）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。

共享对象的具体实现

Redis的共享对象目前只支持整数值的字符串对象。之所以如此，实际上是对内存和CPU（时间）的平衡：共享对象虽然会降低内存消耗，但是判断两个对象是否相等却需要消耗额外的时间。对于整数值，判断操作复杂度为O(1)；对于普通字符串，判断复杂度为O(n)；而对于哈希、列表、集合和有序集合，判断的复杂度为O(n^2)。

虽然共享对象只能是整数值的字符串对象，但是5种类型都可能使用共享对象（如哈希、列表等的元素可以使用）。

共享对象池

共享对象池是指Redis内部维护[0-9999]的整数对象池。

创建大量的整数类型redisObject存在内存开销，每个redisObject内部结构至少占16字节，甚至超过了整数自身空间消耗。

所以Redis内存维护一个[0-9999]的整数对象池，用于节约内存。

除了整数值对象，其他类型如list、hash、set、zset内部元素也可以使用整数对象池。

因此开发中在满足需求的前提下，尽量使用整数对象以节省内存。

就目前的实现来说，Redis服务器在初始化时，会创建10000个字符串对象，值分别是0~9999的整数值；当Redis需要使用值为0~9999的字符串对象时，可以直接使用这些共享对象。10000这个数字定义在源码的`OBJ_SHARED_INTEGERS`常量中定义。

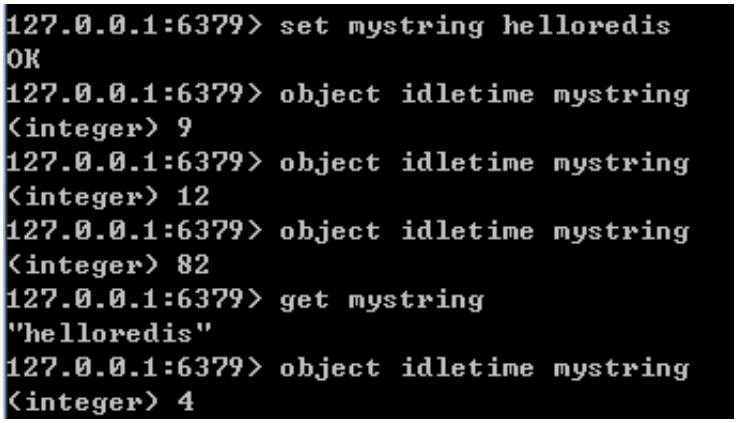
共享对象的引用次数可以通过**object refcount**命令查看，如下图所示。命令执行的结果页佐证了只有0~9999之间的整数会作为共享对象。

```
127.0.0.1:5379> set k1 9999
OK
127.0.0.1:5379> object refcount k1
(integer) 2147483647
127.0.0.1:5379> set k2 10000
OK
127.0.0.1:5379> object refcount k2
(integer) 1
127.0.0.1:5379>
```

5）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（2.6版本占22比特，4.0版本占24比特）。

通过对比lru时间与当前时间，可以计算某个对象的闲置时间；object idletime命令可以显示该闲置时间（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。

6）小结

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；一个redisObject对象的大小为16字节：

> 4bit（类型）+4bit（编码）+24bit（lru）+4Byte（refcount)+8Byte(指针)=16Byte

5、SDS

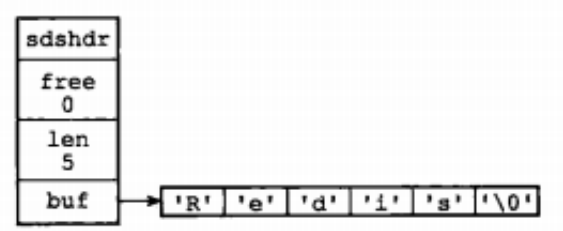
1）SDS内存结构

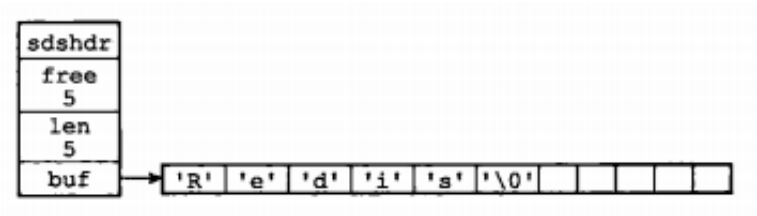
Redis没有直接使用C字符串(即以空字符'\0'结尾的字符数组)作为默认的字符串表示，而是使用了SDS。SDS是简单动态字符串(Simple Dynamic String)的缩写。

3.2 之前

```
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[];
}
```

> 其中，buf表示字节数组，用来存储字符串；len表示buf已使用的长度，free表示buf未使用的长度。下面是两个例子。





通过SDS的结构可以看出，[buf数组的长度=free+len+1（其中1表示字符串结尾的空字符）；所以，一个SDS结构占据的空间为：free所占长度+len所占长度+ buf数组的长度+1=4+4+字符串长度+1=字符串长度+9。

3.2 之后

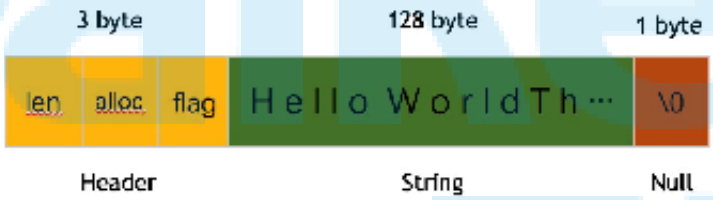
```
typedef char *sds;

struct __attribute__ ((__packed__)) sdshdr5 {      // 对应的字符串长度小于 1<<5 32字节
    unsigned char flags; /* 3 lsb of type, and 5 msb of string length   int embstr*/
    char buf[];
};
struct __attribute__ ((__packed__)) sdshdr8 {      // 对应的字符串长度小于 1<<8 256
    uint8_t len; /* used */                        //目前字符创的长度 用1字节存储
    uint8_t alloc;                                //已经分配的总长度 用1字节存储
    unsigned char flags;                          //flag用3bit来标明类型，类型后续解释，其余5bit目前没有使用 embstr raw
    char buf[];                                   //柔性数组，以'\0'结尾
};
struct __attribute__ ((__packed__)) sdshdr16 {     // 对应的字符串长度小于 1<<16
    uint16_t len; /*已使用长度，用2字节存储*/
    uint16_t alloc; /* 总长度，用2字节存储*/
    unsigned char flags; /* 3 lsb of type, 5 unused bits  */
    char buf[];
};
struct __attribute__ ((__packed__)) sdshdr32 {     // 对应的字符串长度小于 1<<32
   uint32_t len; /*已使用长度，用4字节存储*/
   uint32_t alloc; /* 总长度，用4字节存储*/
   unsigned char flags;/* 低3位存储类型，高5位预留 */
   char buf[];/*柔性数组，存放实际内容*/
};
struct __attribute__ ((__packed__)) sdshdr64 {     // 对应的字符串长度小于 1<<64
   uint64_t len; /*已使用长度，用8字节存储*/
```

```
    uint64_t alloc; /* 总长度，用8字节存储*/
    unsigned char flags; /* 低3位存储类型，高5位预留 */
    char buf[];/*柔性数组，存放实际内容*/
};
```

flag属性保存的是当前使用的SDS类型：

```
static inline char sdsReqType(size_t string_size) {
    if (string_size < 32)
        return SDS_TYPE_5;
    if (string_size < 0xff)                 //255(1个字节)
        return SDS_TYPE_8;
    if (string_size < 0xffff)               //65535(2个字节)
        return SDS_TYPE_16;
    if (string_size < 0xffffffff)           //4294967295(4个字节)
        return SDS_TYPE_32;
    return SDS_TYPE_64;                     //8个字节
}
```



2）SDS与C字符串的比较

- 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）。

五、Redis的对象类型与内存编码

Redis支持5种对象类型，而每种结构都有至少两种编码；

> **这样做的好处在于：**
>
> 一方面接口与实现分离，当需要增加或改变内部编码时，用户使用不受影响;
>
> 另一方面可以根据不同的应用场景切换内部编码，提高效率。

Redis各种对象类型支持的内部编码如下图所示(只列出重点的)：

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

1 字符串（SDS）

1）概况

字符串是最基础的类型，因为所有的键都是字符串类型，且字符串之外的其他几种复杂类型的元素也是字符串。

字符串长度不能超过512MB。

2）内部编码

字符串类型的内部编码有3种，它们的应用场景如下：

- int：8个字节的长整型。字符串值是整型时，这个值使用long整型表示。
- embstr：<=44字节的字符串。embstr与raw都使用redisObject和sds保存数据，区别在于，embstr的使用只分配一次内存空间（因此redisObject和sds是连续的），而raw需要分配两次内存空间（分别为redisObject和sds分配空间）。因此与raw相比，embstr的好处在于创建时少分配一次空间，删除时少释放一次空间，以及对象的所有数据连在一起，寻找方便。而embstr的坏处也很明显，如果字符串的长度增加需要重新分配内存时，整个redisObject和sds都需要重新分配空间，因此redis中的embstr实现为只读。
- raw：大于44个字节的字符串

> 3.2之后 embstr和raw进行区分的长度，是44；**是因为redisObject的长度是16字节**，sds的长度是4+字符串长度；因此当字符串长度是44时，embstr的长度正好是16+4+44 =64，jemalloc正好可以分配64字节的内存单元。
>
> 3.2 之前embstr和raw进行区分的长度，是39，因为redisObject的长度是16字节，sds的长度是9+字符串长度；因此当字符串长度是39时，embstr的长度正好是16+9+39 =64，jemalloc正好可以分配64字节的内存单元。

2 列表

1）概况

列表（list）用来存储多个有序的字符串，每个字符串称为元素；

一个列表可以存储2^32-1个元素。

Redis中的列表支持两端插入和弹出，并可以获得指定位置（或范围）的元素，可以充当数组、队列、栈等。

linkedList

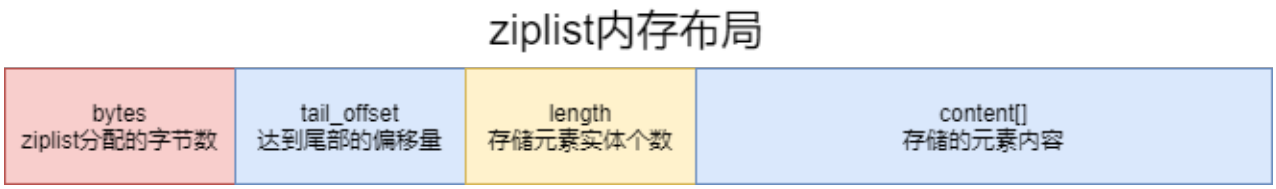
2）内部编码

Redis3.0之前列表的内部编码可以是压缩列表（ziplist）或双端链表（linkedlist）。选择的折中方案是两种数据类型的转换，但是在3.2版本之后 因为转换也是个费时且复杂的操作，引入了一种新的数据格式，结合了双向列表linkedlist和ziplist的特点，称之为quicklist。所有的节点都用quicklist存储，省去了到临界条件是的格式转换。

3）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，并且每个列表项是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。

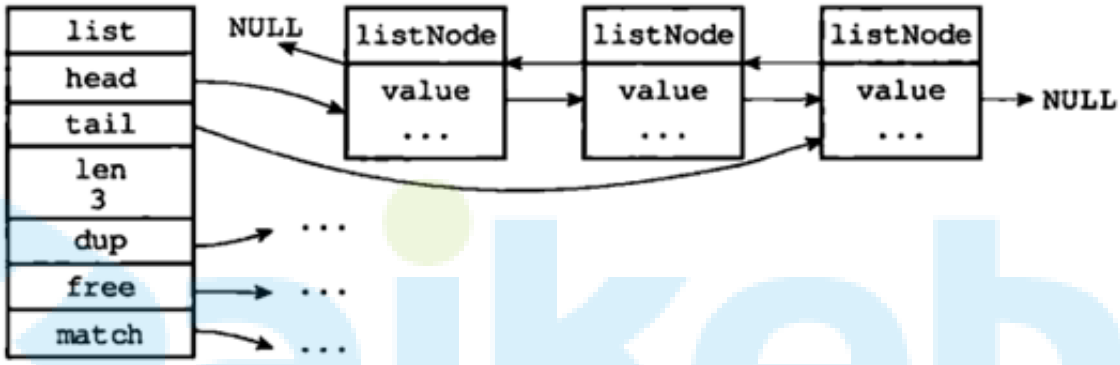
压缩列表（ziplist）是Redis为了节省内存而开发的，是由一系列特殊编码的连续内存块组成的顺序型数据结构，一个压缩列表可以包含任意多个节点（entry），每个节点可以保存一个字节数组或者一个整数值，放到一个连续内存区。



4）双向链表

> 双向链表（linkedlist）：由一个list结构和多个listNode结构组成；

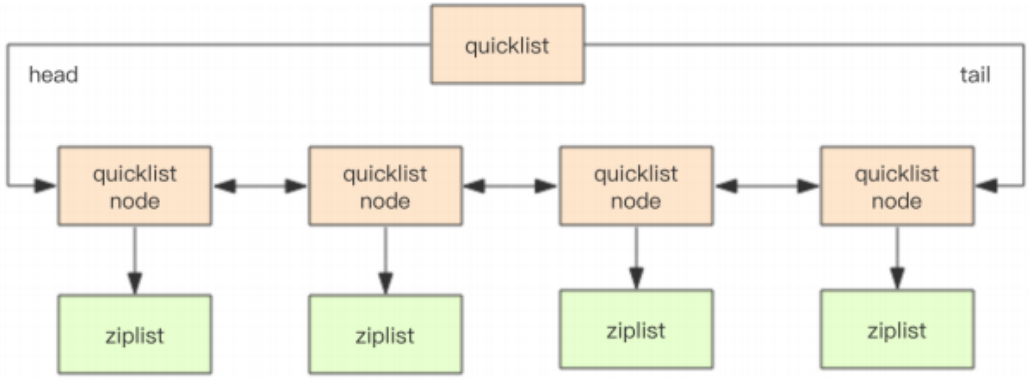
典型结构如下图所示：



通过图中可以看出，双端链表同时保存了表头指针和表尾指针，并且每个节点都有指向前和指向后的指针；链表中保存了列表的长度；dup、free和match为节点值设置类型特定函数，所以链表可以用于保存各种不同类型的值。而链表中每个节点指向的是type为字符串的redisObject。

5）快速列表

简单的说，我们仍旧可以将其看作一个双向列表，但是列表的每个节点都是一个ziplist，其实就是linkedlist和ziplist的结合。quicklist中的每个节点ziplist都能够存储多个数据元素。
Redis3.2开始，列表采用quicklist进行编码。




```
//32byte 的空间
typedef struct quicklist {
    // 指向quicklist的头部
```

```
    quicklistNode *head;
     // 指向quicklist的尾部
    quicklistNode *tail;
    // 列表中所有数据项的个数总和
    unsigned long count;
    // quicklist节点的个数，即ziplist的个数
    unsigned int len;
    // ziplist大小限定，由list-max-ziplist-size给定
    // 表示不用整个int存储fill，而是只用了其中的16位来存储
    int fill : 16;
    // 节点压缩深度设置，由list-compress-depth给定
    unsigned int compress : 16;
} quicklist;

typedef struct quicklistNode {
    struct quicklistNode *prev;  // 指向上一个ziplist节点
    struct quicklistNode *next;  // 指向下一个ziplist节点
    unsigned char *zl;           // 数据指针，如果没有被压缩，就指向ziplist结构，反之
指向quicklistLZF结构
    unsigned int sz;             // 表示指向ziplist结构的总长度(内存占用长度)
    unsigned int count : 16;     // 表示ziplist中的数据项个数
    unsigned int encoding : 2;   // 编码方式，1--ziplist，2--quicklistLZF
    unsigned int container : 2;  // 预留字段，存放数据的方式，1--NONE，2--ziplist
    unsigned int recompress : 1; // 解压标记，当查看一个被压缩的数据时，需要暂时解压，标
记此参数为1，之后再重新进行压缩
    unsigned int attempted_compress : 1; // 测试相关
    unsigned int extra : 10; // 扩展字段，暂时没用
} quicklistNode;
```

3 哈希（压缩列表和哈希表）

1）概况

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（与字符串、列表、集合、有序结合并列），也是Redis作为Key-Value数据库所使用的数据结构。为了说明的方便，后面当使用“内层的哈希”时，代表的是Redis对外提供的5种对象类型的一种；使用“外层的哈希”代指Redis作为Key-Value数据库所使用的数据结构。

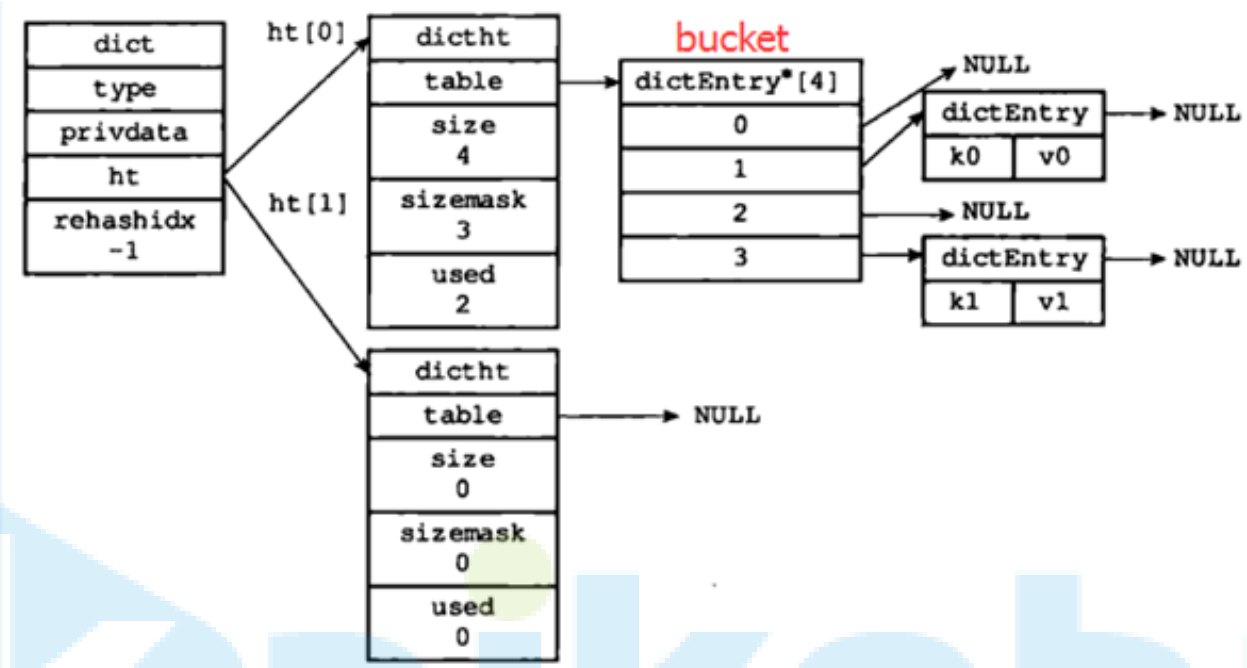
2）内部编码

内层的哈希使用的内部编码可以是压缩列表（ziplist）和哈希表（hashtable）两种；Redis的外层的哈希则只使用了hashtable。

> 压缩列表前面已介绍。与哈希表相比，压缩列表用于元素个数少、元素长度小的场景；其优势在于集中存储，节省空间；同时，虽然对于元素的操作复杂度也由O(1)变为了O(n)，但由于哈希中元素数量较少，因此操作的时间并没有明显劣势。

hashtable：一个hashtable由1个dict结构、2个dictht结构、1个dictEntry指针数组（称为bucket）和多个dictEntry结构组成。

正常情况下（即hashtable没有进行rehash时）各部分关系如下图所示：



1.dict

一般来说，通过使用dictht和dictEntry结构，便可以实现普通哈希表的功能；但是Redis的实现中，在dictht结构的上层，还有一个dict结构。下面说明dict结构的定义及作用。

dict结构如下：

```
typedef struct dict{
    dictType *type;  // type里面主要记录了一系列的函数,可以说是规定了一系列的接口
    void *privdata; // privdata保存了需要传递给那些类型特定函数的可选参数
   //两张哈希表
    dictht ht[2];//便于渐进式rehash
    int trehashidx; //rehash 索引，并没有rehash时，值为 -1
    //目前正在运行的安全迭代器的数量
    int iterators;
} dict;
```

其中，type属性和privdata属性是为了适应不同类型的键值对，用于创建多态字典。

ht属性和trehashidx属性则用于rehash，即当哈希表需要扩展或收缩时使用。ht是一个包含两个项的数组，每项都指向一个dictht结构，这也是Redis的哈希会有1个dict、2个dictht结构的原因。通常情况下，所有的数据都是存在放dict的ht[0]中，ht[1]只在rehash的时候使用。dict进行rehash操作的时候，将ht[0]中的所有数据rehash到ht[1]中。然后将ht[1]赋值给ht[0]，并清空ht[1]。

> 因此，Redis中的哈希之所以在dictht和dictEntry结构之外还有一个dict结构，一方面是为了适应不同类型的键值对，另一方面是为了rehash。

2.dictht

dictht结构如下：

```
typedef struct dictht{
    //哈希表数组，每个元素都是一条链表
    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;
```

其中，各个属性的功能说明如下：

- table属性是一个指针，指向bucket；
- size属性记录了哈希表的大小，即bucket的大小；
- used记录了已使用的dictEntry的数量；
- sizemask属性的值总是为size-1，这个属性和哈希值一起决定一个键在table中存储的位置。

3.bucket

bucket是一个数组，数组的每个元素都是指向dictEntry结构的指针。Redis中bucket数组的大小计算规则如下：大于dictEntry的、最小的2^n；

> 例如，如果有1000个dictEntry，那么bucket大小为1024；如果有1500个dictEntry，则bucket大小为2048。
>
> n%32 = n&(32-1)

4.dictEntry

dictEntry结构用于保存键值对，结构定义如下：

```
// 键
typedef struct dictEntry{
    void *key;
    union{ //值v的类型可以是以下三种类型
        void *val;
        uint64_tu64;
        int64_ts64;
    }v;
     // 指向下个哈希表节点，形成链表
    struct dictEntry *next;
}dictEntry;
```

其中，各个属性的功能如下：

- key：键值对中的键
- val：键值对中的值，使用union(即共用体)实现，存储的内容既可能是一个指向值的指针，也可能是64位整型，或无符号64位整型；
- next：指向下一个dictEntry，用于解决哈希冲突问题

> 在64位系统中，一个dictEntry对象占24字节（key/val/next各占8字节）。

3）编码转换

如前所述，Redis中内层的哈希既可能使用哈希表，也可能使用压缩列表。

只有同时满足下面两个条件时，才会使用压缩列表：

- 哈希中元素数量小于512个；
- 哈希中所有键值对的键和值字符串长度都小于64字节。

下图展示了Redis内层的哈希编码转换的特点：

```
127.0.0.1:6379> hset myhash k1 v1
(integer) 1
127.0.0.1:6379> hset myhash k2 v2
(integer) 1
127.0.0.1:6379> hset myhash k3 v3
(integer) 1
127.0.0.1:6379> object encoding myhash
"ziplist"
127.0.0.1:6379> hset myhash k4 vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvvvvvvvvvvv
(integer) 1
127.0.0.1:6379> object encoding myhash
"hashtable"
127.0.0.1:6379> hdel myhash k4
(integer) 1
127.0.0.1:6379> object encoding myhash
"hashtable"
```

4 集合（整数集合和哈希表）

（1）概况

集合（set）与列表类似，都是用来保存多个字符串，但集合与列表有两点不同：集合中的元素是无序的，因此不能通过索引来操作元素；集合中的元素不能有重复。

> 一个集合中最多可以存储2^32-1个元素；除了支持常规的增删改查，Redis还支持多个集合取交集、并集、差集。

（2）内部编码

集合的内部编码可以是整数集合（intset）或哈希表（hashtable）。

哈希表前面已经讲过，这里略过不提；需要注意的是，集合在使用哈希表时，值全部被置为null。

整数集合的结构定义如下：

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

> 其中，encoding代表contents中存储内容的类型，虽然contents（存储集合中的元素）是int8_t类型，但实际上其存储的值是int16_t、int32_t或int64_t，具体的类型便是由encoding决定的；length表示元素个数。

> 整数集合适用于集合所有元素都是整数且集合元素数量较小的时候，与哈希表相比，整数集合的优势在于集中存储，节省空间；同时，虽然对于元素的操作复杂度也由O(1)变为了O(n)，但由于集合数量较少，因此操作的时间并没有明显劣势。

（3）编码转换

只有同时满足下面两个条件时，集合才会使用整数集合：

- 集合中元素数量小于512个；
- 集合中所有元素都是整数值。

如果有一个条件不满足，则使用哈希表；且编码只可能由整数集合转化为哈希表，反方向则不可能。

下图展示了集合编码转换的特点：

```
127.0.0.1:6379> sadd myset 111 222 333
(integer) 3
127.0.0.1:6379> object encoding myset
"intset"
127.0.0.1:6379> sadd myset helloworld
(integer) 1
127.0.0.1:6379> object encoding myset
"hashtable"
127.0.0.1:6379> srem myset helloworld
(integer) 1
127.0.0.1:6379> object encoding myset
"hashtable"
```

5 有序集合（压缩列表和跳跃表）

（1）概况

有序集合与集合一样，元素都不能重复；但与集合不同的是，有序集合中的元素是有顺序的。与列表使用索引下标作为排序依据不同，有序集合为每个元素设置一个分数（score）作为排序依据。

（2）内部编码

有序集合的内部编码可以是压缩列表（ziplist）或跳跃表（skiplist）。ziplist在列表和哈希中都有使用，前面已经讲过，这里略过不提。

跳跃表是一种有序数据结构，通过在每个节点中维持多个指向其他节点的指针，从而达到快速访问节点的目的。除了跳跃表，实现有序数据结构的另一种典型实现是平衡树；大多数情况下，跳跃表的效率可以和平衡树媲美，且跳跃表实现比平衡树简单很多，因此redis中选用跳跃表代替平衡树。跳跃表支持平均O(logN)、最坏O(N)的复杂点进行节点查找，并支持顺序操作。

[Redis的跳跃表实现由**zskiplist**和**zskiplistNode**两个结构组成：前者用于保存跳跃表信息（如头结点、尾节点、长度等），后者用于表示跳跃表节点。

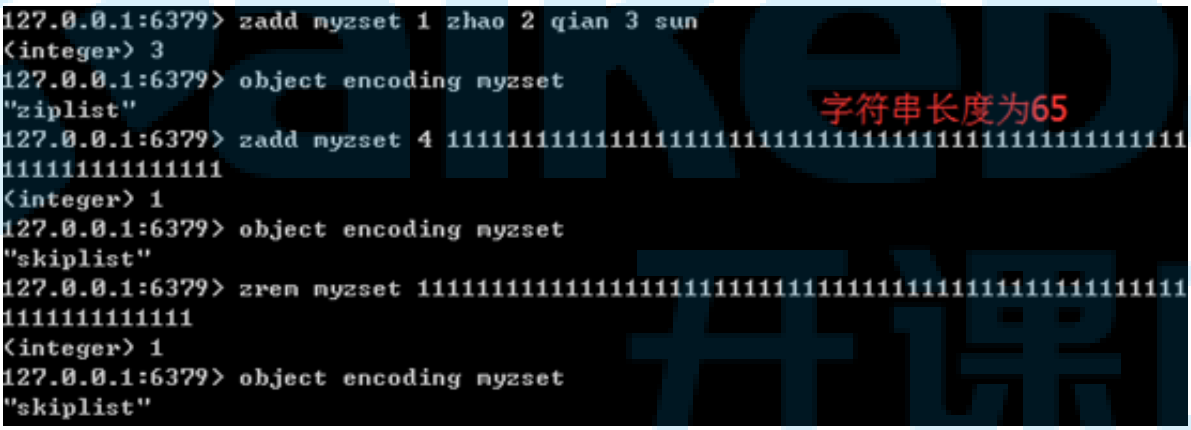
只有同时满足下面两个条件时，才会使用压缩列表：

1）有序集合中元素数量小于**128**个；

2）有序集合中**所有成员长度都不足64字节**。

如果有一个条件不满足，则使用跳跃表；且编码只可能由压缩列表转化为跳跃表，反方向则不可能。

下图展示了有序集合编码转换的特点：



（3）跳跃表

1.数据结构定义

有许多数据结构的定义其实是按照（结点+组织方式）来的，结点就是一个数据点，组织方式就是把结点组织起来形成数据结构，比如 双端链表 (ListNode+list)、字典（dictEntry+dictht+dict）等，今天所说的SkipList其实也一样，我们首先看下它的`结点`定义：

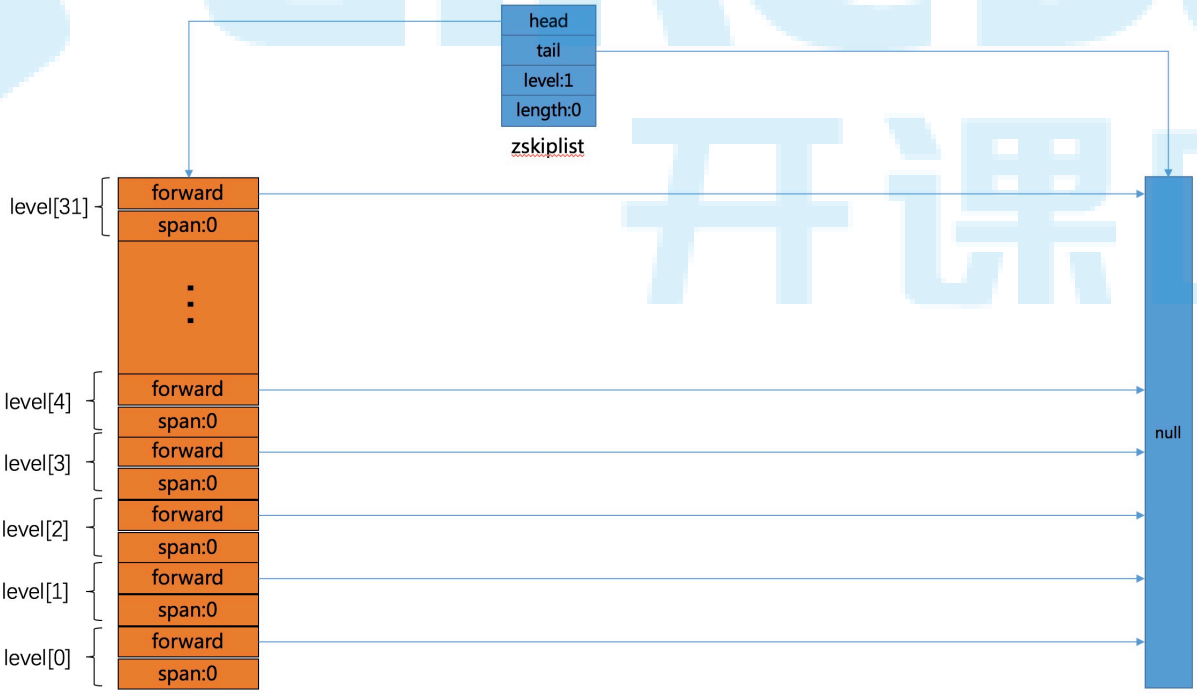
```
typedef struct zskiplistNode {
    sds ele;                                //数据域
    double score;                           //分值
    struct zskiplistNode *backward;         //后向指针，使得跳表第一层组织为双向链表
    struct zskiplistLevel {                 //每一个结点的层级
        struct zskiplistNode *forward;      //某一层的前向结点
        unsigned int span;                  //某一层距离下一个结点的跨度
    } level[];                              //level本身是一个柔性数组，最大值为32，由
ZSKIPLIST_MAXLEVEL 定义
} zskiplistNode;
```

接下来是组织方式，即使用上面的`zskiplistNode`组织起一个SkipList：

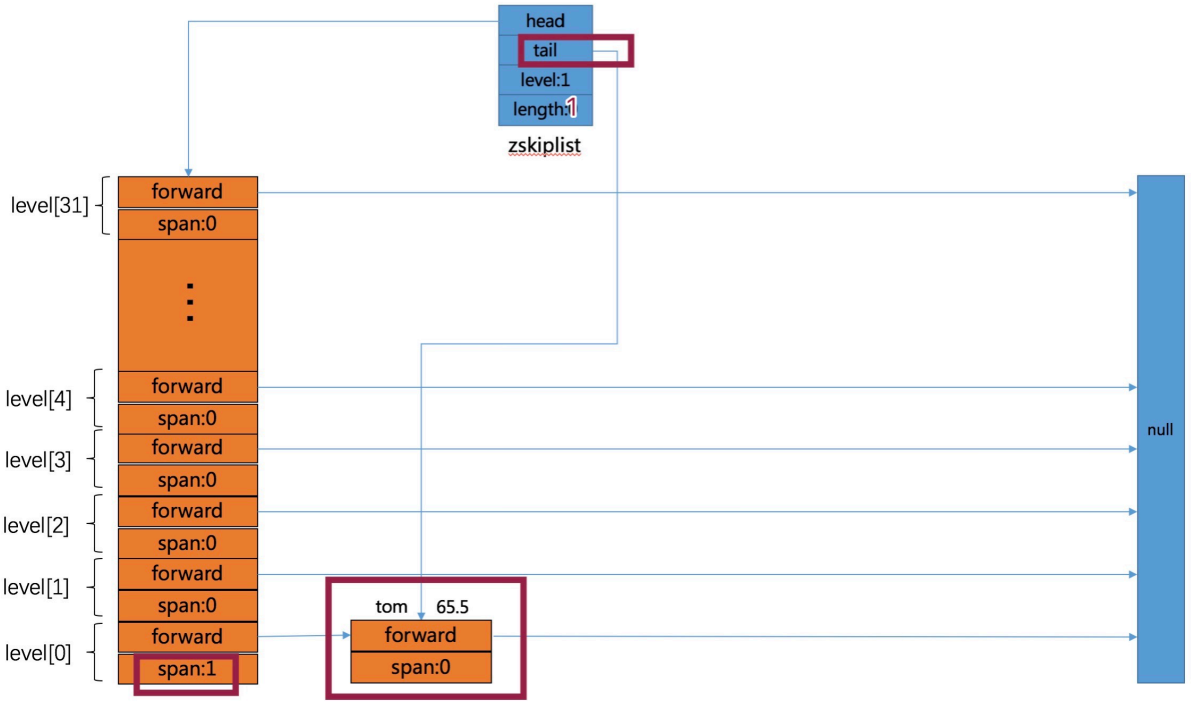
```
typedef struct zskiplist {
    struct zskiplistNode *header;      //头部
    struct zskiplistNode *tail;        //尾部
    unsigned long length;              //长度，即一共有多少个元素
    int level;                         //最大层级，即跳表目前的最大层级
} zskiplist;
```

核心的数据结构就是上面两个。

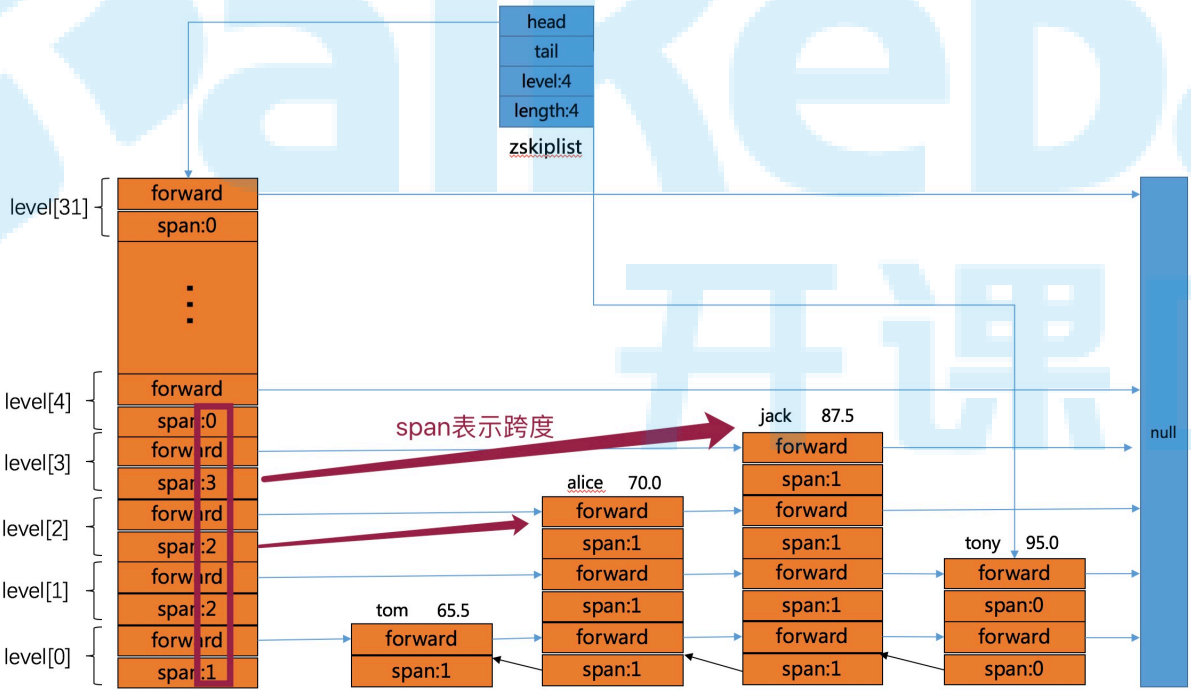
2.创建



需要注意的是span的含义，它表示`当前节点距离下一个节点的跨度`，之所以可以根据rank排名获取元素，就是根据span确定的。update[i]保存的就是第 i 层应该插入节点的前一个节点，在第三步更新指针的时候使用。插入了一个元素的zsl如下图所示(level=1)：



接着我们继续插入后面的三条数据，他们的level分别为`jack->4、alice->3、tony->2`,此时的zsl如下图所示，注意span的更新：



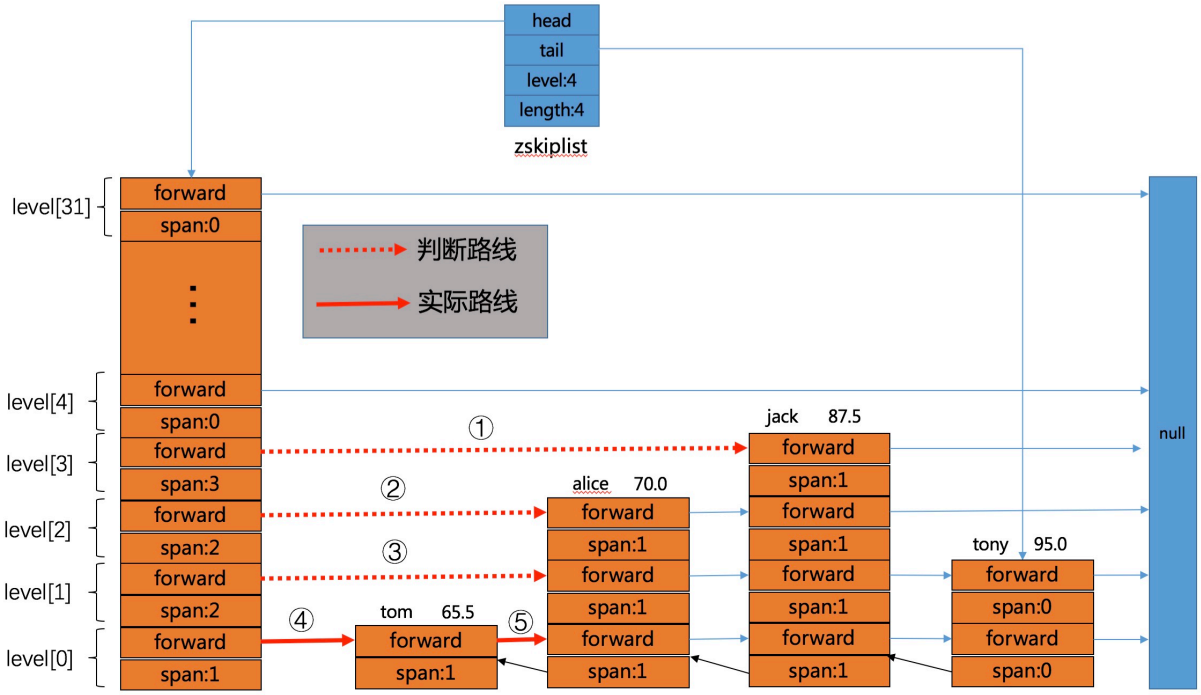
3.查找

遍历的核心思想是:

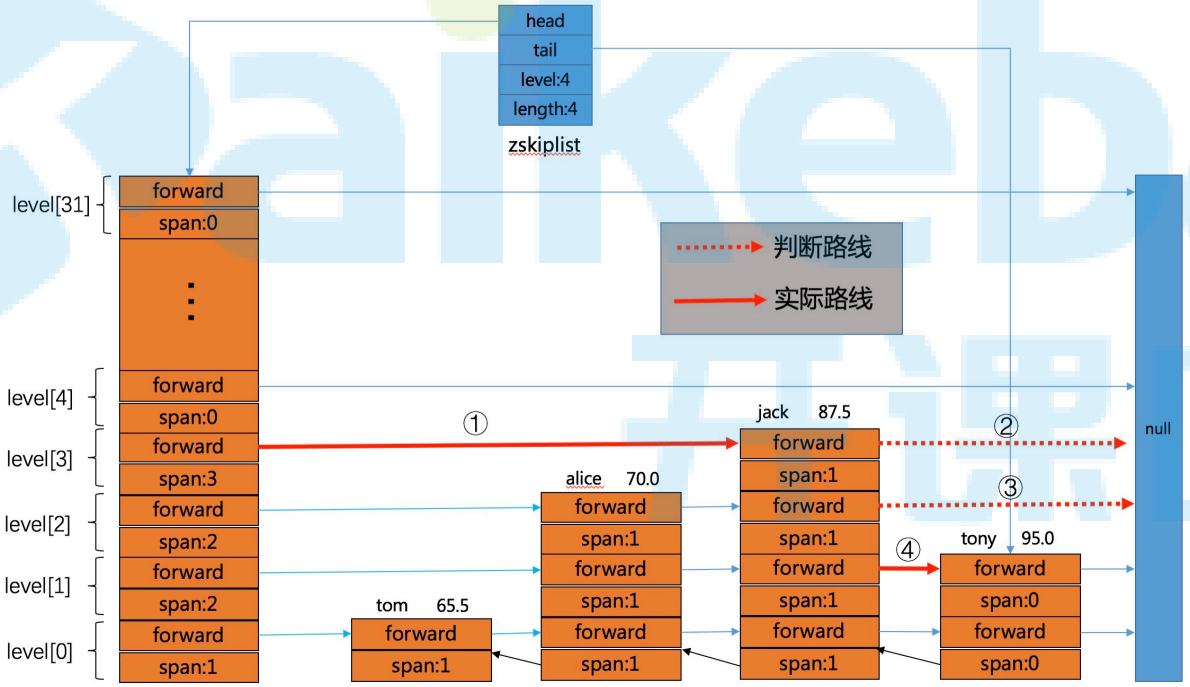
(1) 高Level -> 低Level

(2) 小score -> 大score

即在从高Level遍历比较过程中，如果此时的score小于了某个高level的值，就在这个节点前一个节点降低一层Level继续往前遍历，我们找`70.0`的路线如下图所示（图中红线）：



查找score为`95.0`的元素：



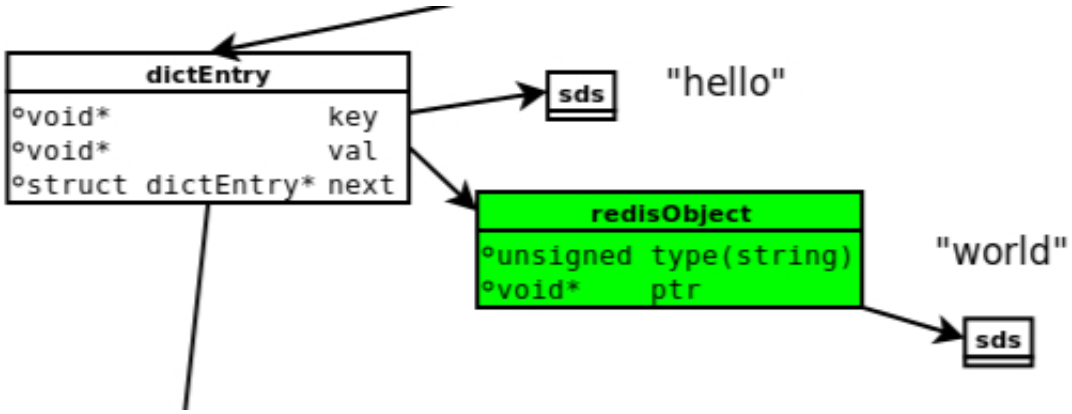
六、Redis 设计优化

1 估算Redis内存使用量

要估算redis中的数据占据的内存大小，需要对redis的内存模型有比较全面的了解，包括第一节课介绍的hashtable、sds、redisobject、各种对象类型的编码方式等。

下面以最简单的字符串类型来进行说明。

> 假设有90000个键值对，每个key的长度是12个字节，每个value的长度也是12个字节（且key和value都不是整数）；



下面来估算这90000个键值对所占用的空间。在估算占据空间之前，首先可以判定字符串类型使用的编码方式：embstr。

90000个键值对占据的内存空间主要可以分为两部分：一部分是90000个dictEntry占据的空间；一部分是键值对所需要的bucket空间。

每个dictEntry占据的空间包括：

1) 一个dictEntry结构，24字节，jemalloc会分配32字节的内存块(64位操作系统下，一个指针8字节，一个dictEntry由三个指针组成)

2) 一个key，12字节，所以SDS(key)需要12+4=16 个字节（[SDS的长度=4+字符串长度），jemalloc会分配16字节的内存块

3) 一个redisObject，16字节，jemalloc会分配16字节的内存块(4bit+4bit+24bit+4Byte+8Byte=16Byte)

4) 一个value，12字节，所以SDS(value)需要12+4=16个字节（[SDS的长度=4+字符串长度），jemalloc会分配16字节的内存块

5) 综上，一个dictEntry所占据的空间需要32+16+16+16=80个字节。

bucket空间：

bucket数组的大小为大于90000的最小的2^n，是131072；每个bucket元素（bucket中存储的都是指针元素）为8字节（因为64位系统中指针大小为8字节）。

因此，可以估算出这90000个键值对占据的内存大小为：[90000*80 + 131072*8 = 8248576

> 作为对比将key和value的长度由12字节增加到13字节，则对应的SDS变为17个字节，jemalloc会分配32个字节，因此每个dictEntry占用的字节数也由80字节变为112字节。此时估算这90000个键值对占据内存大小为：90000*112 + 131072*8 = 11128576。

2 优化内存占用

了解redis的内存模型，对优化redis内存占用有很大帮助。下面介绍几种优化场景和方式

1）利用jemalloc特性进行优化

上一小节所讲述的90000个键值便是一个例子。由于jemalloc分配内存时数值是不连续的，因此key/value字符串变化一个字节，可能会引起占用内存很大的变动；在设计时可以利用这一点。

> 例如，如果key的长度如果是13个字节，则SDS为17字节，jemalloc分配32字节；此时将key长度缩减为12个字节，则SDS为16字节，jemalloc分配16字节；则每个key所占用的空间都可以缩小一半。

2）使用整型/长整型

如果是整型/长整型，Redis会使用int类型（8字节）存储来代替字符串，可以节省更多空间。因此在可以使用长整型/整型代替字符串的场景下，尽量使用长整型/整型。

3）共享对象

利用共享对象，可以减少对象的创建（同时减少了redisObject的创建），节省内存空间。目前redis中的共享对象只包括10000个整数（0-9999）；可以通过调整`OBJ_SHARED_INTEGERS`参数提高共享对象的个数；

> 例如将`OBJ_SHARED_INTEGERS`调整到20000，则0-19999之间的对象都可以共享。论坛网站在redis中存储了每个帖子的浏览数，而这些浏览数绝大多数分布在0-20000之间，这时候通过适当增大`OBJ_SHARED_INTEGERS`参数，便可以利用共享对象节省内存空间。

4）缩短键值对的存储长度

键值对的长度是和性能成反比的，比如我们来做一组写入数据的性能测试，执行结果如下：

数据量	key 大小	value 大小	string:set 平均耗时	hash:hset 平均耗时
100w	20byte	512byte	1.13微秒	10.28微秒
100w	20byte	200byte	0.74微秒	8.08微秒
100w	20byte	100byte	0.65微秒	7.92微秒
100w	20byte	50byte	0.59微秒	6.74微秒
100w	20byte	20byte	0.55微秒	6.60微秒
100w	20byte	5byte	0.53微秒	6.53微秒

从以上数据可以看出，在 key 不变的情况下，value 值越大操作效率越慢，因为 Redis 对于同一种数据类型会使用不同的内部编码进行存储，比如字符串的内部编码就有三种：int（整数编码）、raw（优化内存分配的字符串编码）、embstr（动态字符串编码），这是因为 Redis 的作者是想通过不同编码实现效率和空间的平衡，然而数据量越大使用的内部编码就越复杂，而越是复杂的内部编码存储的性能就越低。

这还只是写入时的速度，当键值对内容较大时，还会带来另外几个问题：

- 内容越大需要的持久化时间就越长，需要挂起的时间越长，Redis 的性能就会越低；
- 内容越大在网络上传输的内容就越多，需要的时间就越长，整体的运行速度就越低；
- 内容越大占用的内存就越多，就会更频繁的触发内存淘汰机制，从而给 Redis 带来了更多的运行负担。

因此在保证完整语义的同时，我们要尽量的缩短键值对的存储长度，必要时要对数据进行序列化和压缩再存储，以 Java 为例，序列化我们可以使用 protostuff 或 kryo，压缩我们可以使用 snappy。

七、Reids 内存用量统计

查看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1说明使用了虚拟内存
mem_fragmentation_ratio:15.07
#内存碎片字节数
mem_fragmentation_bytes
#Redis使用的内存分配器
mem_allocator:jemlloc-5.1.0
```

used_memory

由Redis**内存分配器**分配的**数据内存**和**缓冲内存**的内存总量（单位是字节），包括使用的虚拟内存（即swap）used_memory_human只是显示更加人性化。

used_memory_rss

记录的是由**操作系统分配**的**Redis进程内存**和Redis内存中无法再被jemalloc分配的**内存碎片**（单位是字节）。

> **used_memory和used_memory_rss的区别：**
>
> 前者是从Redis角度得到的量，后者是从操作系统角度得到的量。二者之所以有所不同，一方面是因为内存碎片和Redis进程运行需要占用内存，使得前者可能比后者小，另一方面虚拟内存的存在，使得前者可能比后者大。

由于在实际应用中，Redis的数据量会比较大，此时进程运行占用的内存与Redis数据量和内存碎片相比，都会小得多；因此used_memory_rss和used_memory的比例，便成了衡量Redis内存碎片率的参数；这个参数就是mem_fragmentation_ratio。

mem_fragmentation_ratio

内存碎片比率，该值是used_memory_rss / used_memory的比值。

mem_fragmentation_ratio一般大于1，且该值越大，内存碎片比例越大。

mem_fragmentation_ratio<1，说明Redis使用了虚拟内存，由于虚拟内存的媒介是磁盘，比内存速度要慢很多，当这种情况出现时，应该及时排查，如果内存不足应该及时处理，如增加Redis节点、增加Redis服务器的内存、优化应用等。

一般来说，mem_fragmentation_ratio在1.03左右是比较健康的状态（对于jemalloc来说）；刚开始的mem_fragmentation_ratio值很大，是因为还没有向Redis中存入数据，Redis进程本身运行的内存使得used_memory_rss 比used_memory大得多。

mem_allocator

Redis使用的内存分配器，在编译时指定；可以是 libc 、jemalloc或者tcmalloc，默认是jemalloc；

二、Redis内存划分

Redis作为内存数据库，在内存中存储的内容主要是数据（键值对）；通过前面的叙述可以知道，除了数据以外，Redis的其他部分也会占用内存。

Redis的内存占用主要可以划分为以下几个部分：

1、数据内存

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；此外，Redis在存储对象时，并不是直接将数据扔进内存，而是会对对象进行各种包装：如redisObject、SDS等；这篇文章后面将重点介绍Redis中数据存储的细节。

2、进程内存

Redis主进程本身运行肯定需要占用内存，如代码、常量池等等；这部分内存大约几兆，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。

3、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，客户端缓冲存储客户端连接的输入输出缓冲；复制积压缓冲用于部分复制功能；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。

4、内存碎片

set s1 111

Lpush s1 1 2 3 4 5

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。后面将要说到的jemalloc便在控制内存碎片方面做的很好。

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。

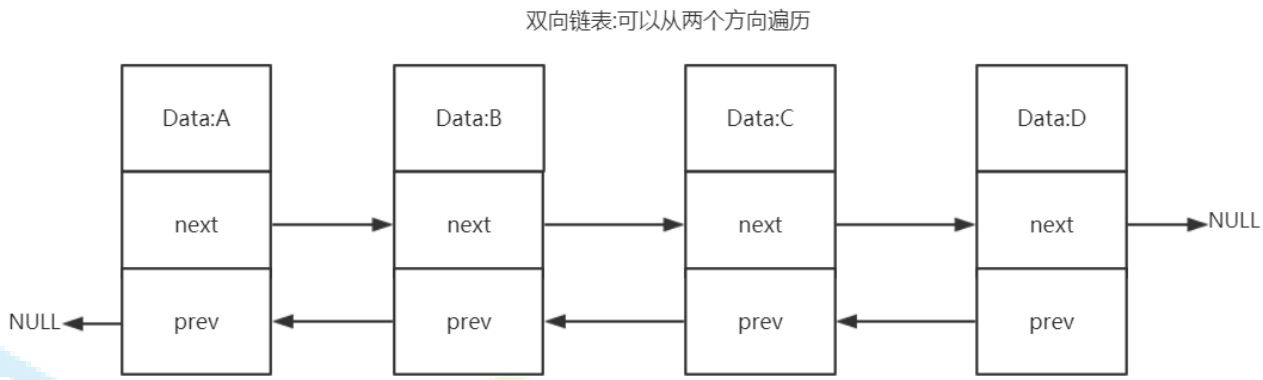
六、Redis数据结构

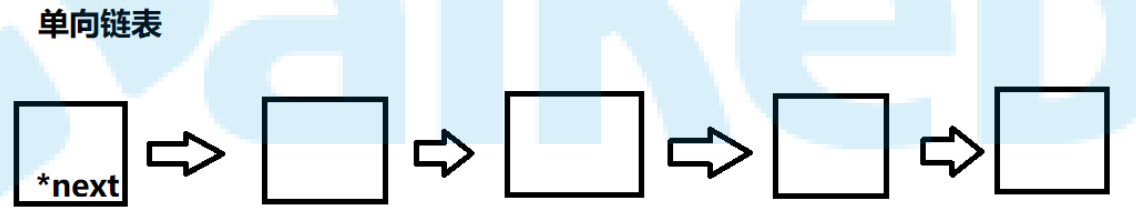
链表

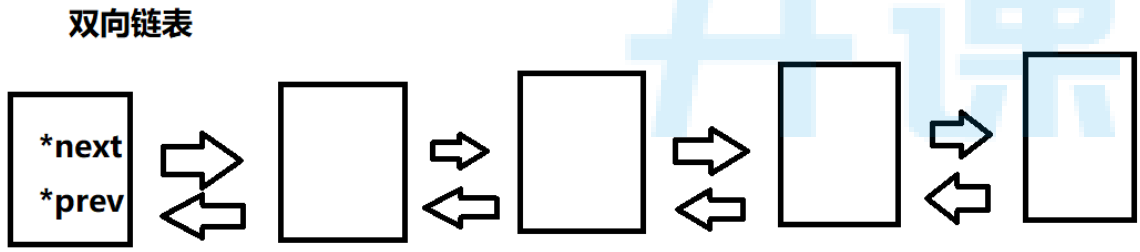
```
typedef struct listNode {
    //前置节点
    struct listNode *prev;
    //后置节点
    struct listNode *next;
    //节点的值
    void *value;
}listNode


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;
```







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

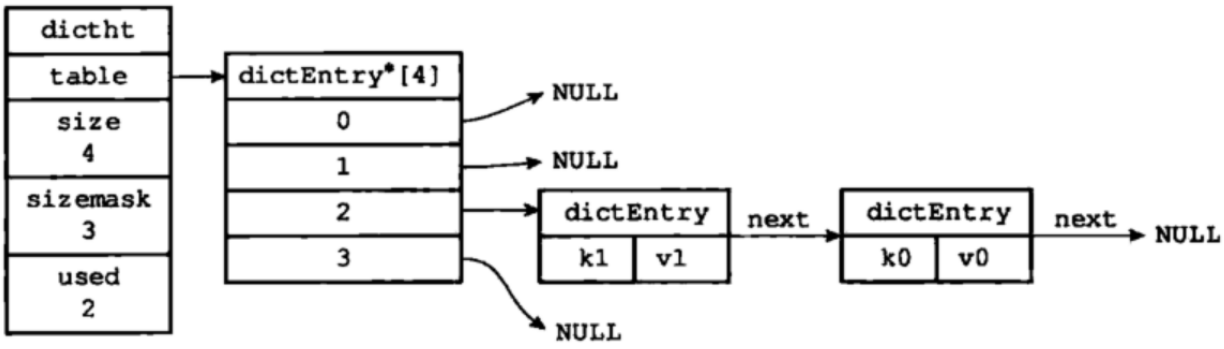


图 4-2　连接在一起的键 K1 和键 K0

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;
```

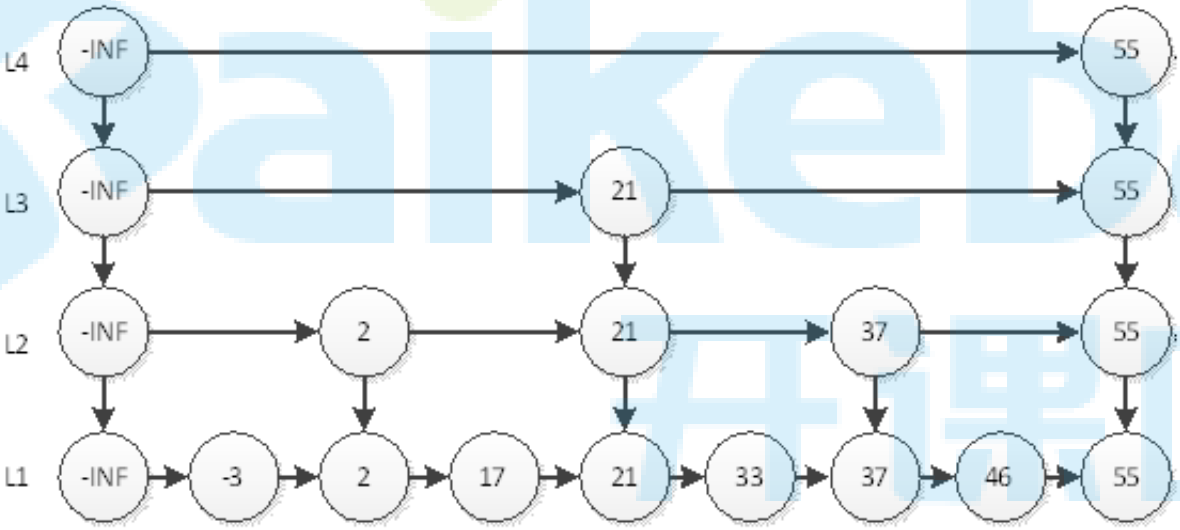
```
            int64_ts64;
        }v;

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

跳跃表

普通单向链表：

跳跃表（跳表）：



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：

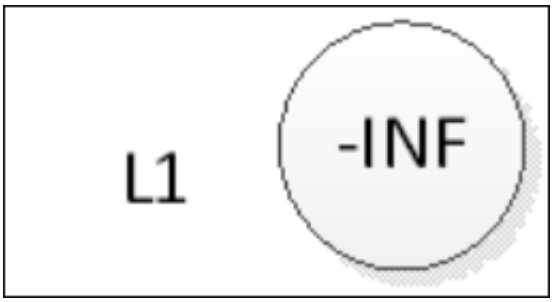
查找46： 55---21---55--37--55--46

插入

L1 层

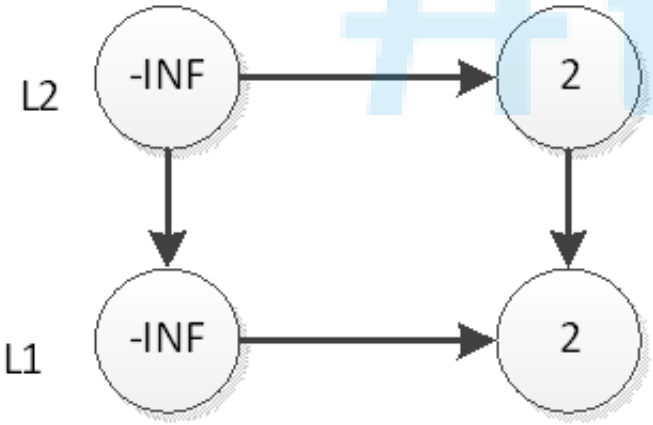
概率算法

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：

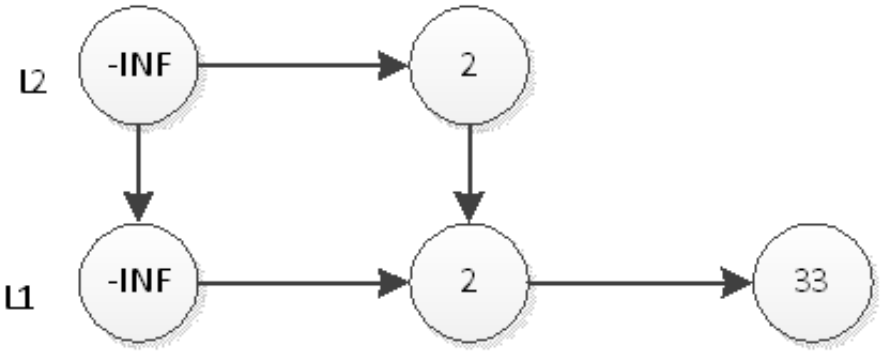


如果我们要插入元素2，首先是在底部插入元素2，如下图：

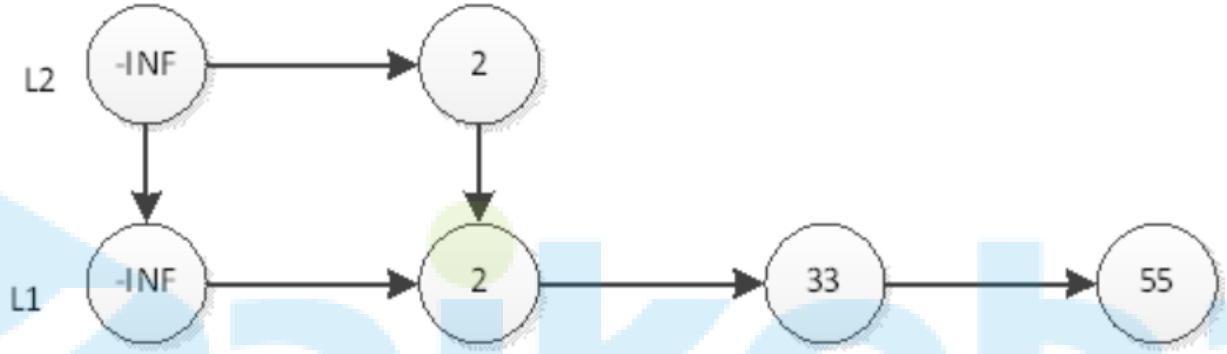
然后我们抛硬币，结果是正面，那么我们要将2插入到L2层，如下图



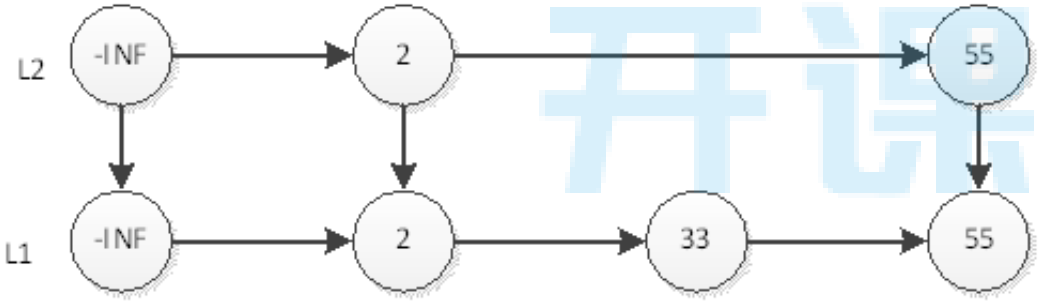
继续抛硬币，结果是反面，那么元素2的插入操作就停止了，插入后的表结构就是上图所示。接下来，我们插入元素33，跟元素2的插入一样，现在L1层插入33，如下图：



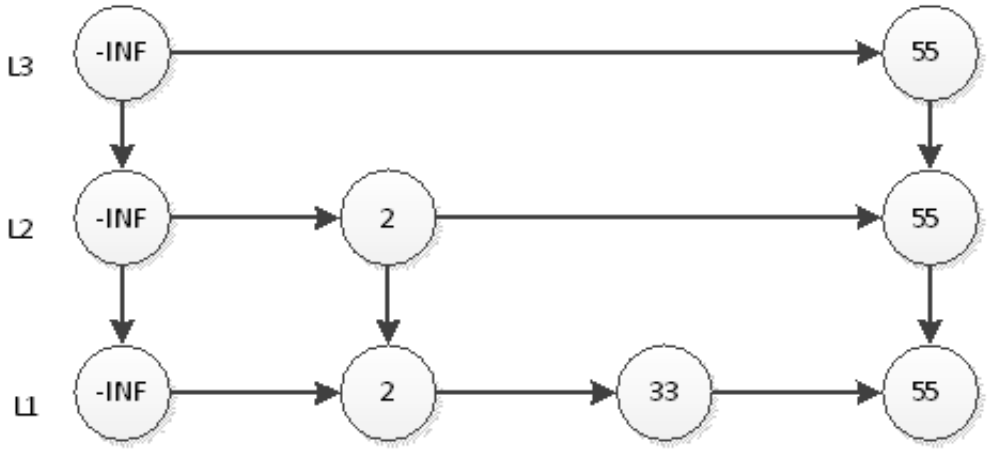
然后抛硬币，结果是反面，那么元素33的插入操作就结束了，插入后的表结构就是上图所示。接下来，我们插入元素55，首先在L1插入55，插入后如下图：



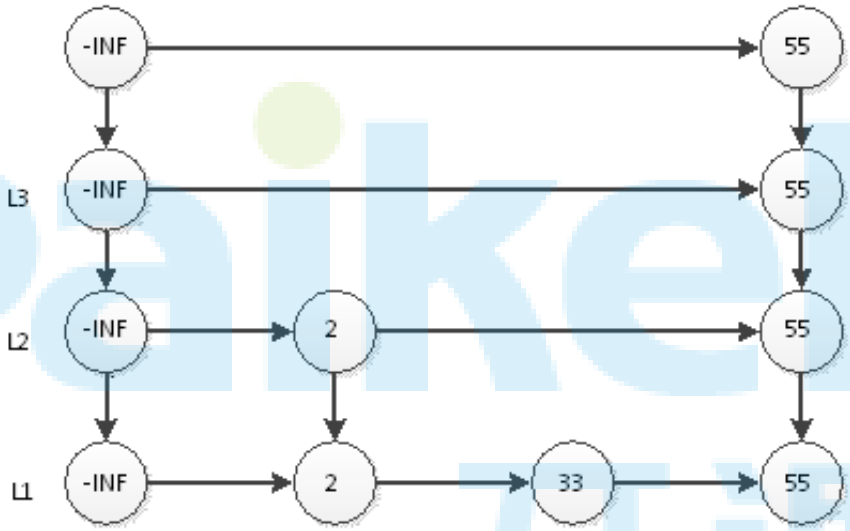
然后抛硬币，结果是正面，那么L2层需要插入55，如下图：



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
```

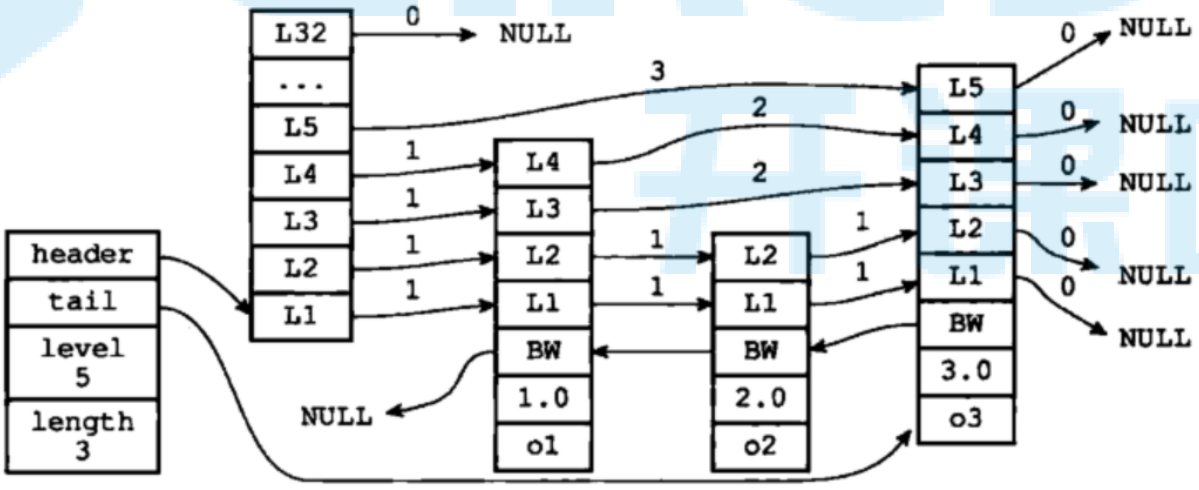
```
            struct zskiplistNode *forward;
            //跨度
            unsigned int span;
      }level[];

      //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

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：

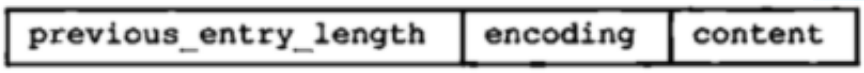


图 7-4　压缩列表节点的各个组成部分

> 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

开课吧