前面介绍了Redis用到的所有主要数据结构，比如简单动态字符串(SDS)、双端链表、字典、压缩列表、整数集合等。然而Redis并没有直接使用这些数据结构来实现键值对数据库，而是基于这些数据结构创建了一个对象系统，包括字符串对象、列表对象、哈希对象、集合对象和有序集合对象这五种类型的对象，每种对象都用到了至少一种前面所介绍的数据结构。

Redis在执行命令之前，根据对象的类型来判断一个对象是否可以执行给定的命令。还可以针对不同的使用场景，为对象设置多种不同的数据结构实现，从而优化对象在不同场景下的内存使用效率。

Redis的对象系统还实现了基于引用计数的内存回收机制，当某个对象的引用计数为0时，这个对象所占用的内存就会被自动释放；Redis还通过引用计数技术实现了对象共享机制，通过让多个键共享同一个对象来节约内存。

最后，Redis的对象带有访间时间记录信息，在服务器启用了maxmemory功能的情况下，长时间未使用的那些键可能会优先被服务器删除。

**一：对象的类型和编码**

Redis使用对象来表示数据库中的键和值，每当在Redis的数据库中新创建一个键值对时，至少会创建两个对象，一个对象用作键，另一个对象用作值。比如下面的命令：

127.0.0.1**:**6379**>** set msg "hello world"

OK

在数据库中创建了一个新的键值对，其中键是一个包含了字符串值"msg"的对象，而值则是一个包含了字符串值"hello world"的对象。

对于Redis中的键值对来说，键总是一个字符申对象，而值可以是字符串对象、列表对象、哈希对象、集合对象或者有序集合对象。

Redis中的每个对象都由一个redisObject结构表示，该结构在redis.h中定义：

**typedef** struct redisObject **{**

unsigned type**:**4**;**

unsigned encoding**:**4**;**

unsigned lru**:**REDIS\_LRU\_BITS**;** */\* lru time (relative to server.lruclock) \*/*

int refcount**;**

void **\***ptr**;**

**}** robj**;**

type字段表示对象的类型，取值可以是如下内容之一：

#define REDIS\_STRING 0

#define REDIS\_LIST 1

#define REDIS\_SET 2

#define REDIS\_ZSET 3

#define REDIS\_HASH 4

Redis中的TYPE命令可以返回对象的类型：

127.0.0.1**:**6379**>** type msg

string

redisObject的ptr指针指向对象底层的数据结构，而数据结构的类型由redisObject的encoding字段决定。取值可以是如下内容之一：

#define REDIS\_ENCODING\_RAW 0 /\* Raw representation \*/

#define REDIS\_ENCODING\_INT 1 /\* Encoded as integer \*/

#define REDIS\_ENCODING\_HT 2 /\* Encoded as hash table \*/

#define REDIS\_ENCODING\_ZIPMAP 3 /\* Encoded as zipmap \*/

#define REDIS\_ENCODING\_LINKEDLIST 4 /\* Encoded as regular linked list \*/

#define REDIS\_ENCODING\_ZIPLIST 5 /\* Encoded as ziplist \*/

#define REDIS\_ENCODING\_INTSET 6 /\* Encoded as intset \*/

#define REDIS\_ENCODING\_SKIPLIST 7 /\* Encoded as skiplist \*/

#define REDIS\_ENCODING\_EMBSTR 8 /\* Embedded sds string encoding \*/

其中，字符串对象可以使用的编码有：REDIS\_ENCODING\_RAW 、REDIS\_ENCODING\_INT和REDIS\_ENCODING\_EMBSTR，列表对象可以使用的编码有：REDIS\_ENCODING\_LINKEDLIST和REDIS\_ENCODING\_ZIPLIST，哈希对象可以使用的编码有：REDIS\_ENCODING\_HT、和REDIS\_ENCODING\_ZIPLIST，集合对象可以使用的编码有：REDIS\_ENCODING\_HT和REDIS\_ENCODING\_INTSET，有序集合可以使用的编码有：REDIS\_ENCODING\_ZIPLIST和REDIS\_ENCODING\_SKIPLIST。

在Redis的2.6版本之前，包含少量键值对的哈希对象使用REDIS\_ENCODING\_ZIPMAP编码。自2.6开始，开始使用REDIS\_ENCODING\_ZIPLIST，作为包含少量键值对的哈希对象的实现。

使用DBJECT ENCDDING命令可以查看一个值的编码：

127.0.0.1**:**6379**>** object encoding msg

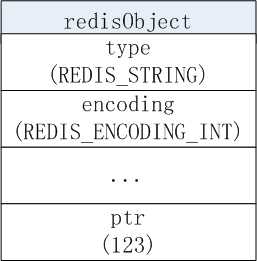
"embstr"

对象在不同的场景下，使用不同的数据结构来实现，极大地提升了Redis的灵活性和效率。比如列表对象包含的元素比较少时，Redis使用压缩列表作为列表对象的底层实现，这样做更加节约内存，并且在内存中以连续块方式保存的压缩列表比起双端链表，可以更快被载人到缓存中。随着列表对象包含的元素越来越多，使用压缩列表来保存元素的优势逐渐消失时，对象就会将底层实现从压缩列表转向双端链表上面。

**二：字符串对象**

字符串对象的编码可以是int、raw或者embstr。

如果一个字符串对象保存的是整数值，并且这个整数值可以用long类型来表示，那字符串对象会将整数值保存在字符串对象redisObject结构的ptr属性里面，也就是将void\*转换成long。内存布局如下：



比如：

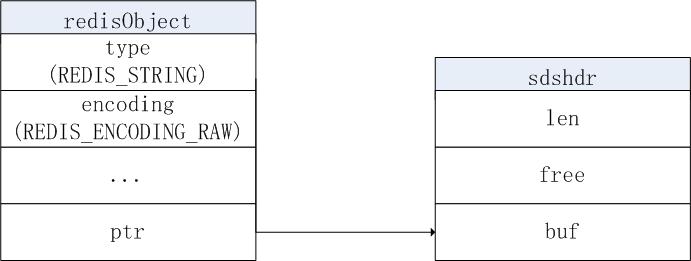
127.0.0.1**:**6379**>** set number 123

OK

127.0.0.1**:**6379**>** object encoding number

"int"

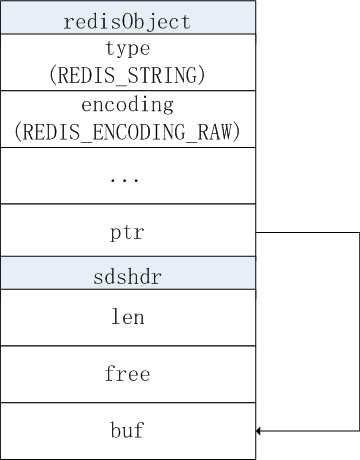
如果字符串对象保存的是一个字符串值，并且这个字符串值的长度大于39字节，那么字符串对象使用一个简单动态字符串(SDS)来保存这个字符串值，并将对象的编码设置为raw。内存布局如下，注意，对象的ptr指针，不是指向sdshdr的首地址，而是指向原始字符串的首地址：



如果字符串对象保存的是一个字符串值，并且这个字符串值的长度小于等于39字节，那么字符串对象将使用embstr编码的方式来保存这个字符串值。

embstr编码是专门用于保存短字符串的一种优化编码方式，这种编码和raw编码一样，都使用redisObject结构和sdshdr结构来表示字符串对象。

但raw编码会调用两次内存分配函数来分别创建redisObject结构和sdshdr结构，而embstr编码则通过调用一次内存分配函数来分配一块连续的空间，空间中依次包含redisObject和sdshdr结构。这样做的好处是：内存的分配和释放更加高效，而且embstr编码的字符串对象，使用连续的内存块，能够更好地利用缓存带来的优势。其内存布局如下：



比如：

127.0.0.1**:**6379**>** set msg "abcdefghijklmnopqrstuvwxyz0123456789012"

OK

127.0.0.1**:**6379**>** object encoding msg

"embstr"

127.0.0.1**:**6379**>** set msg "abcdefghijklmnopqrstuvwxyz01234567890123"

OK

127.0.0.1**:**6379**>** object encoding msg

"raw"

int编码的字符串对象和embstr编码的字符申对象，在满足一定条件的情况下，会被转换为raw编码的字符串对象。

int编码的字符串对象，如果向对象执行了一些命令，使得这个对象保存的不再是整数值，而是一个字符串值，那么字符串对象的编码将从int变为raw。比如下面的例子：

127.0.0.1**:**6379**>** set number 1

OK

127.0.0.1**:**6379**>** object encoding number

"int"

127.0.0.1**:**6379**>** append number " is a number"

**(**integer**)** 13

127.0.0.1**:**6379**>** object encoding number

"raw"

Redis没有为embstr编码的字符串对象编写任何相应的修改程序（只有int编码的字符串对象和raw编码的字符串对象有这些程序），所以embstr编码的字符串对象实际上是只读的。

当对embstr编码的字符串对象执行任何修改命令时，程序会先将对象的编码从embstr转换成raw，然后再执行修改命令。因为这个原因，embstr编码的字符串对象在执行修改命令之后，总会变成一个raw编码的字符串对象。比如：

127.0.0.1**:**6379**>** set msg "hello"

OK

127.0.0.1**:**6379**>** object encoding msg

"embstr"

127.0.0.1**:**6379**>** append msg " world"

**(**integer**)** 11

127.0.0.1**:**6379**>** object encoding msg

"raw"

**三：列表对象**

列表对象的编码可以是ziplist或者linkedlist。

当列表对象中的元素较少，且元素都是较短的字符串时，列表对象采用ziplist编码，也就是使用压缩列表(ziplist)作为底层实现，每个压缩列表节点（entry）保存一个列表元素。

比如：

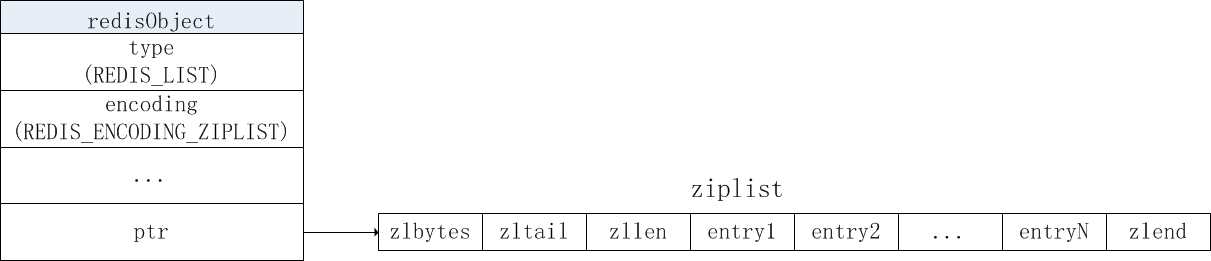
127.0.0.1**:**6379**>** rpush number 1 "three" 5

**(**integer**)** 3

127.0.0.1**:**6379**>** object encoding number

"ziplist"

采用ziplist编码时，内存布局如下，在ziplist中，每个节点中保存的是原始字符串(“three”)或者数字(123)，而非redisObject结构，或者sdshdr结构：



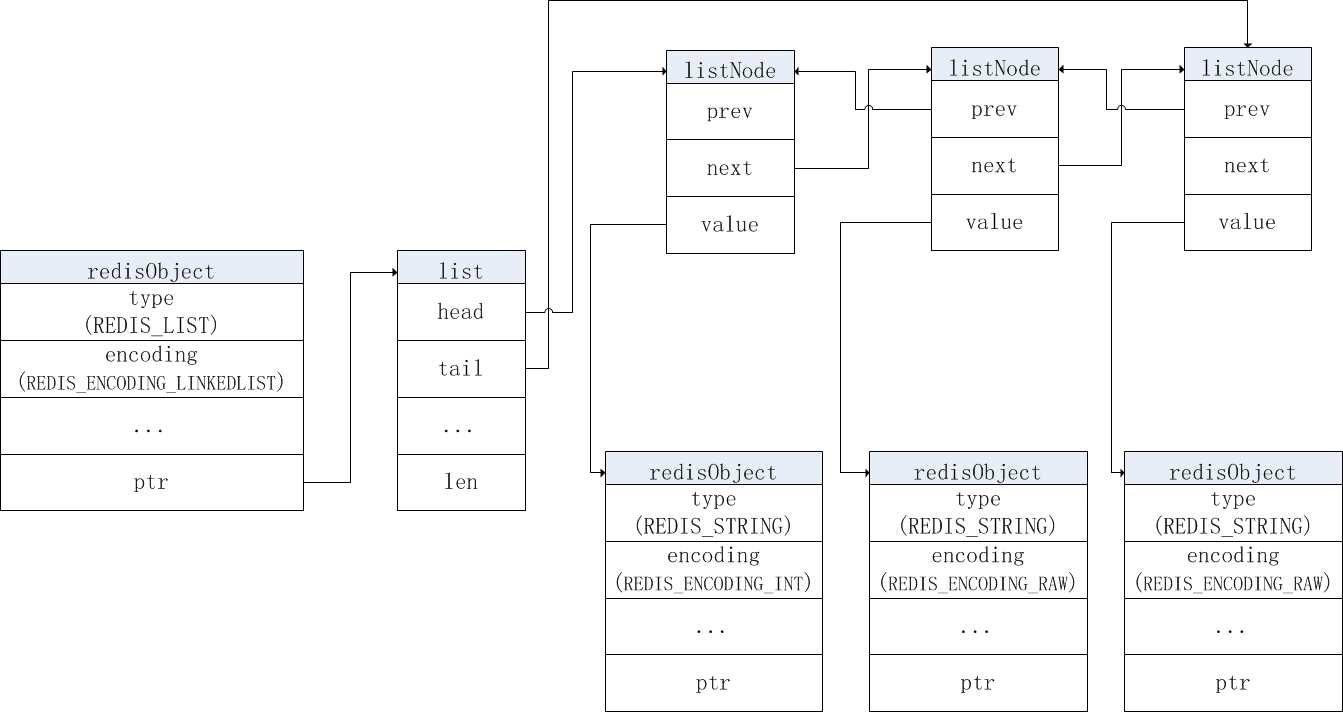
当列表对象可以同时满足以下两个条件时，才会使用ziplist编码：

a：列表对象中，每个元素保存的字符串元素的长度都小于64字节；

b：列表对象保存的元素数量小于512个。

当条件不满足时，列表对象使用linkedlist编码，也就是采用双端链表(list)作为底层实现，每个链表节点(listNode)保存一个列表元素（字符串对象）。

列表对象采用linkedlist编码时，内存布局如下，在list链表中，每个节点listNode结构中的value指针，指向一个redisObject结构：



linkedlist编码的列表对象，在底层的双端链表结构中包含了多个字符串对象，这种嵌套字符串对象的行为在稍后介绍的哈希对象、集合对象和有序集合对象中都会出现。字符串对象是Redis五种类型的对象中，唯一一种会被其他四种类型对象嵌套的对象。

对于使用ziplist编码的列表对象，当插入操作使得两个条件中的任意一个不满足时，就会执行编码转换。原本保存在压缩列表里的所有列表元素都会被转移并保存到双端链表中，对象的编码从ziplist变为linkedlist。

比如，下面分别演示了插入长度超过64字节的元素，以及元素数超过512的情况：

127.0.0.1**:**6379**>** rpush number "0123456789012345678901234567890123456789012345678901234567890123"

**(**integer**)** 4

127.0.0.1**:**6379**>** object encoding number

"ziplist"

127.0.0.1**:**6379**>** rpush number "01234567890123456789012345678901234567890123456789012345678901234"

**(**integer**)** 5

127.0.0.1**:**6379**>** object encoding number

"linkedlist"

127.0.0.1**:**6379**>** EVAL "for i=1, 512 do redis.call('RPUSH', KEYS[1], i) end" 1 "integers"

**(**nil**)**

127.0.0.1**:**6379**>** llen integers

**(**integer**)** 512

127.0.0.1**:**6379**>** object encoding integers

"ziplist"

127.0.0.1**:**6379**>** rpush integers 513

**(**integer**)** 513

127.0.0.1**:**6379**>** object encoding integers

"linkedlist"

这两个条件中的阈值，可以通过配置文件中的以下选项进行修改：

list**-**max**-**ziplist**-**entries 512

list**-**max**-**ziplist**-**value 64

**四：哈希对象**

哈希对象的编码可以是：ziplist或者hashtable。

当哈希对象中的键值对较少，且键值对中保存的都是较短的字符串时，哈希对象采用zipiist编码，也就是使用压缩列表(ziplist)作为底层实现。当有新的键值对要加人到哈希对象时，会分别将键和值中的原始字符串添加到压缩列表的表尾。

因此，压缩列表中，保存了同一键值对的两个节点总是紧挨在一起，保存键的节点在前，保存值的节点在后。先添加到哈希对象中的键值对在前，后添加的键值对在后。

比如：

127.0.0.1**:**6379**>** hset profile name "tom"

**(**integer**)** 1

127.0.0.1**:**6379**>** hset profile age 25

**(**integer**)** 1

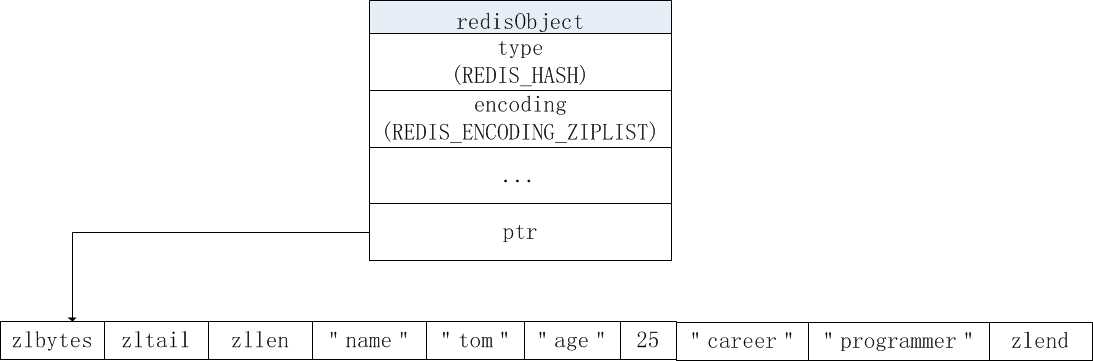
127.0.0.1**:**6379**>** hset profile career "programmer"

**(**integer**)** 1

127.0.0.1**:**6379**>** object encoding profile

"ziplist"

哈希对象采用ziplist编码时，内存布局与ziplist编码的列表对象类似。在压缩列表中，相邻两个节点中保存的，分别是key和value的原始字符串，而非redisObject结构，或者sdshdr结构，比如下图，就表示上面例子中，哈希对象profile的内存结构：



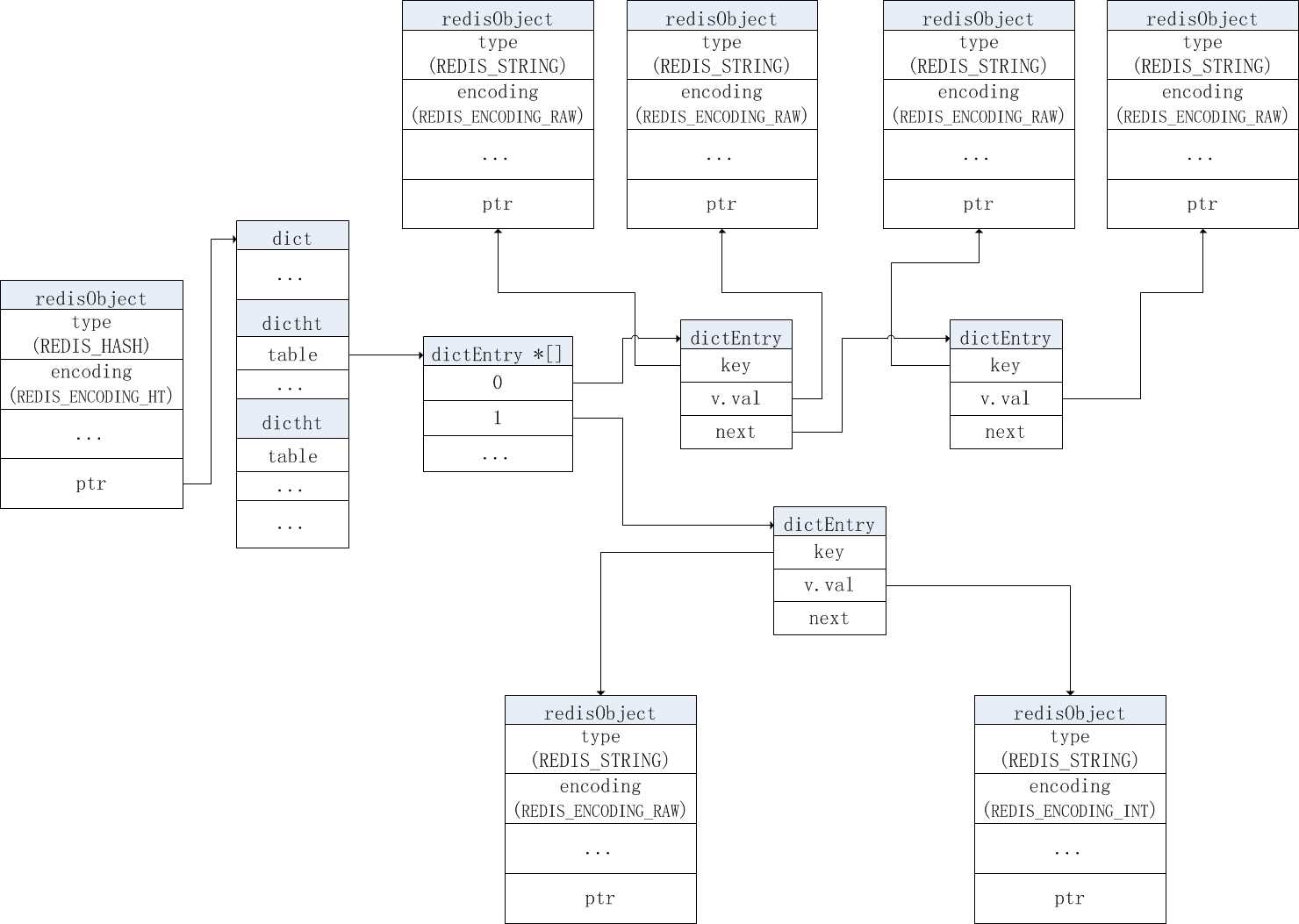
当哈希对象可以同时满足以下两个条件时，才会使用ziplist编码：

a：哈希对象中，所有键值对中，保存的字符串元素的长度都小于等于64字节；

b：列表对象保存的键值对数量小于等于512个。

当条件之一不满足时，哈希对象使用hashtable编码，也就是采用字典(dict)作为底层实现，字典的哈希表中，每个哈希节点(dictEntry)保存一个键值对（两个字符串对象）。

哈希对象采用hashtable编码时，内存布局如下，在dict字典的哈希表dictht中，每个哈希表节点dictEntry结构中，key指针指向一个redisObject结构，表示键对象，v.val指针指向一个redisObject结构，表示值对象：



对于使用ziplist编码的哈希对象来说，当插入操作使得两个条件中的任意一个不能被满足时，就会执行编码转换。原本保存在压缩列表里的所有键值对都会被转移并保存到字典中，对象的编码从ziplist变为hashtable。

比如，下面分别演示了插入长度超过64字节的键、值，以及键值对数目超过512的情况：

127.0.0.1**:**6379**>** hset testhash 0123012345678901234567890123456789012345678901234567890123456789 "hehe2"

**(**integer**)** 1

127.0.0.1**:**6379**>** object encoding testhash

"ziplist"

127.0.0.1**:**6379**>** hset testhash 01234012345678901234567890123456789012345678901234567890123456789 "hehe2"

**(**integer**)** 1

127.0.0.1**:**6379**>** object encoding testhash

"hashtable"

127.0.0.1**:**6379**>** hset testhash2 name "0123012345678901234567890123456789012345678901234567890123456789"

**(**integer**)** 1

127.0.0.1**:**6379**>** object encoding testhash2

"ziplist"

127.0.0.1**:**6379**>** hset testhash2 name2 "01234012345678901234567890123456789012345678901234567890123456789"

**(**integer**)** 1

127.0.0.1**:**6379**>** object encoding testhash2

"hashtable"

127.0.0.1**:**6379**>** EVAL "for i=1, 512 do redis.call('HSET', KEYS[1], i, i) end" 1 "testhash3"

**(**nil**)**

127.0.0.1**:**6379**>** hlen testhash3

**(**integer**)** 512

127.0.0.1**:**6379**>** object encoding testhash3

"ziplist"

127.0.0.1**:**6379**>** hset testhash3 name "tom"

**(**integer**)** 1

127.0.0.1**:**6379**>** hlen testhash3

**(**integer**)** 513

127.0.0.1**:**6379**>** object encoding testhash3

"hashtable"

这两个条件中的阈值，可以通过配置文件中的以下选项进行修改：

hash**-**max**-**ziplist**-**entries 512

hash**-**max**-**ziplist**-**value 64

**五：集合对象**

集合对象的编码可以是intset或者hashtable。

当集合对象中的元素较少，且都是整数时，集合对象采用intset编码，也就是使用整数集合(intset)作为底层实现。比如：

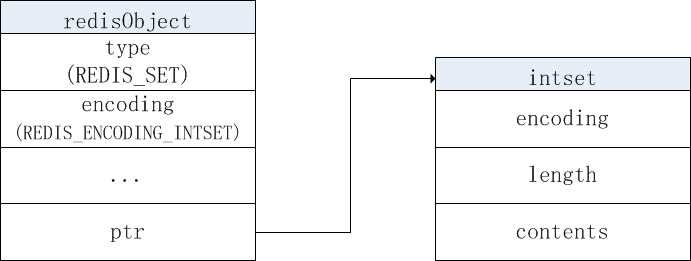
127.0.0.1**:**6379**>** sadd numbers 1 2 3

**(**integer**)** 3

127.0.0.1**:**6379**>** object encoding numbers

"intset"

集合对象采用intset编码时，内存布局如下，redisObject中的ptr指向一个整数集合，在整数集合的contents中直接保存整数，而非redisObject结构，或者sdshdr结构：



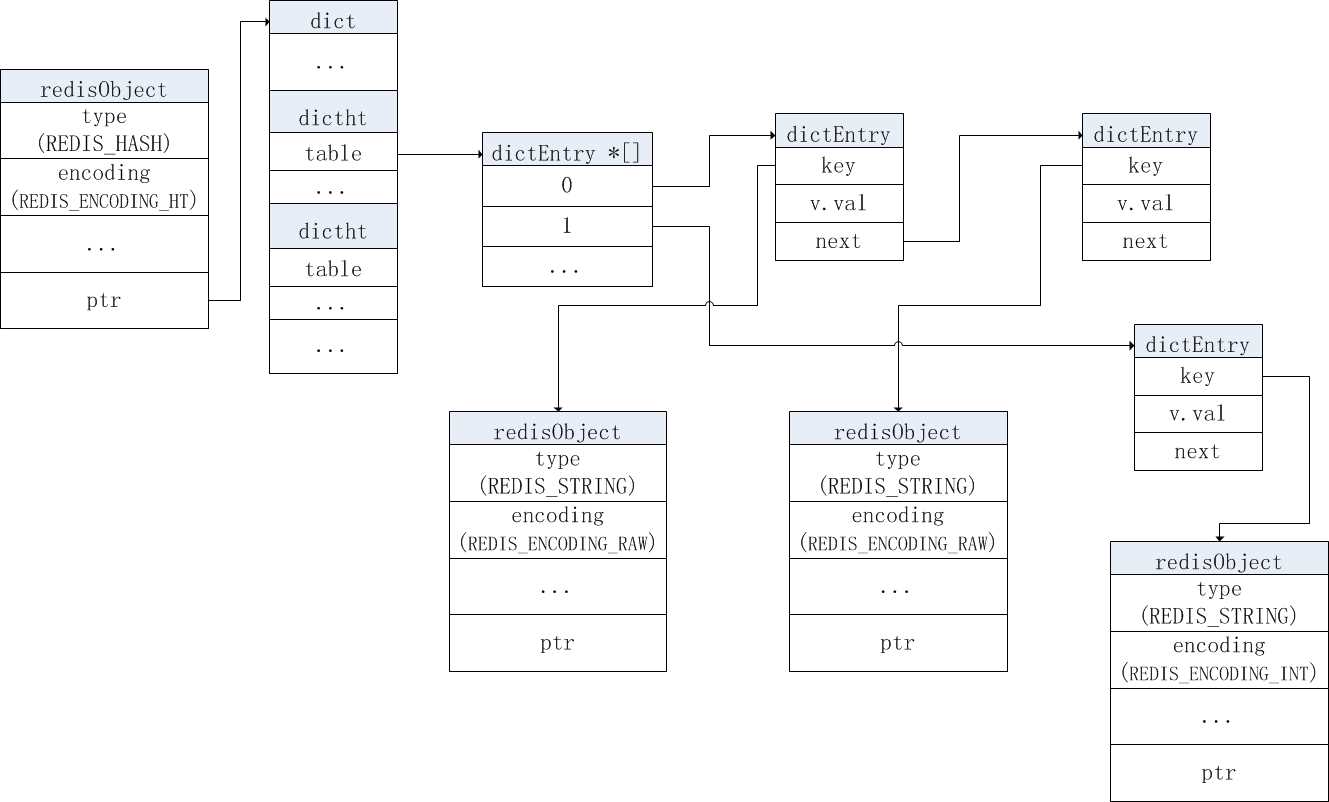
当集合对象可以同时满足以下两个条件时，才会使用intset编码：

a：集合对象保存的所有元素都是整数值；

b：集合对象保存的元素数量不超过512个。

当条件之一不满足时，集合对象使用hashtable编码，也就是采用字典(dict)作为底层实现，字典的哈希表中，每个哈希节点(dictEntry)的key保存一个元素（字符串对象），并将哈希节点的value置为NULL。

集合对象采用hashtable编码时，内存布局如下，在dict字典的哈希表dictht中，每个哈希表节点dictEntry结构中，key指针指向一个redisObject结构，表示集合元素，v.val直接置为NULL：



对于使用intset编码的集合对象来说，当插入操作使得两个条件中的任意一个不能被满足时，就会执行编码转换。原本保存在整数集合里的所有元素都会被转移并保存到字典中，对象的编码从intset变为hashtable。

比如，下面分别演示了插入非整数的元素，以及元素数目超过512的情况：

127.0.0.1**:**6379**>** sadd numbers 1 2 3

**(**integer**)** 3

127.0.0.1**:**6379**>** object encoding numbers

"intset"

127.0.0.1**:**6379**>** sadd numbers hehe

**(**integer**)** 1

127.0.0.1**:**6379**>** object encoding numbers

"hashtable"

127.0.0.1**:**6379**>** EVAL "for i=1, 512 do redis.call('SADD', KEYS[1], i) end" 1 integers

**(**nil**)**

127.0.0.1**:**6379**>** scard integers

**(**integer**)** 512

127.0.0.1**:**6379**>** object encoding integers

"intset"

127.0.0.1**:**6379**>** sadd integers 1000

**(**integer**)** 1

127.0.0.1**:**6379**>** scard integers

**(**integer**)** 513

127.0.0.1**:**6379**>** object encoding integers

"hashtable"

条件中的阈值，可以通过配置文件中的以下选项进行修改：

set-max-intset-entries 512

**五：有序集合对象**

有序集合的编码可以是ziplist或者skiplist。

当集合对象中的元素较少，且元素中的字符串长度较小时，有序集合对象采用ziplist编码，也就是使用压缩列表(ziplist)作为底层实现。

每个集合元素使用两个紧挨的压缩列表节点来保存，第一个节点保存元素的成员(member)，而第二个元素则保存元素的分值(score)。压缩列表内的集合元素按分值从小到大进行排序，分值较小的元素被放置在靠近表头的方向，而分值较大的元素则被放置在靠近表尾的方向。

比如：

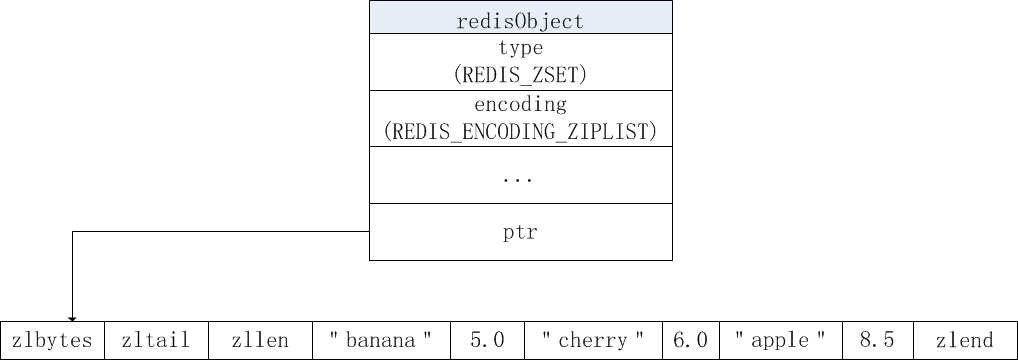
127.0.0.1**:**6379**>** zadd price 8.5 apple 5.0 banana 6.0 cherry

**(**integer**)** 3

127.0.0.1**:**6379**>** object encoding price

"ziplist"

有序集合对象采用ziplist编码时，内存布局与ziplist编码的列表对象类似。在压缩列表中，相邻两个节点中保存的，分别是元素成员和分值的原始字符串和double浮点数，而非redisObject结构，或者sdshdr结构，比如下图，就表示上例中，有序集合price的内存结构：



当有序集合对象可以同时满足以下两个条件时，才会使用ziplist编码：

a：有序集合保存的元素数量小于等于128个；

b：有序集合保存的所有元素中，原始字符串的长度都小于等于64字节。

当条件之一不满足时，有序集合对象使用skiplist编码，也就是采用zset结构作为底层实现，zset由跳跃表(skiplist)和字典(dict)两种数据结构组成：

**typedef** struct zset **{**

dict **\***dict**;**

zskiplist **\***zsl**;**

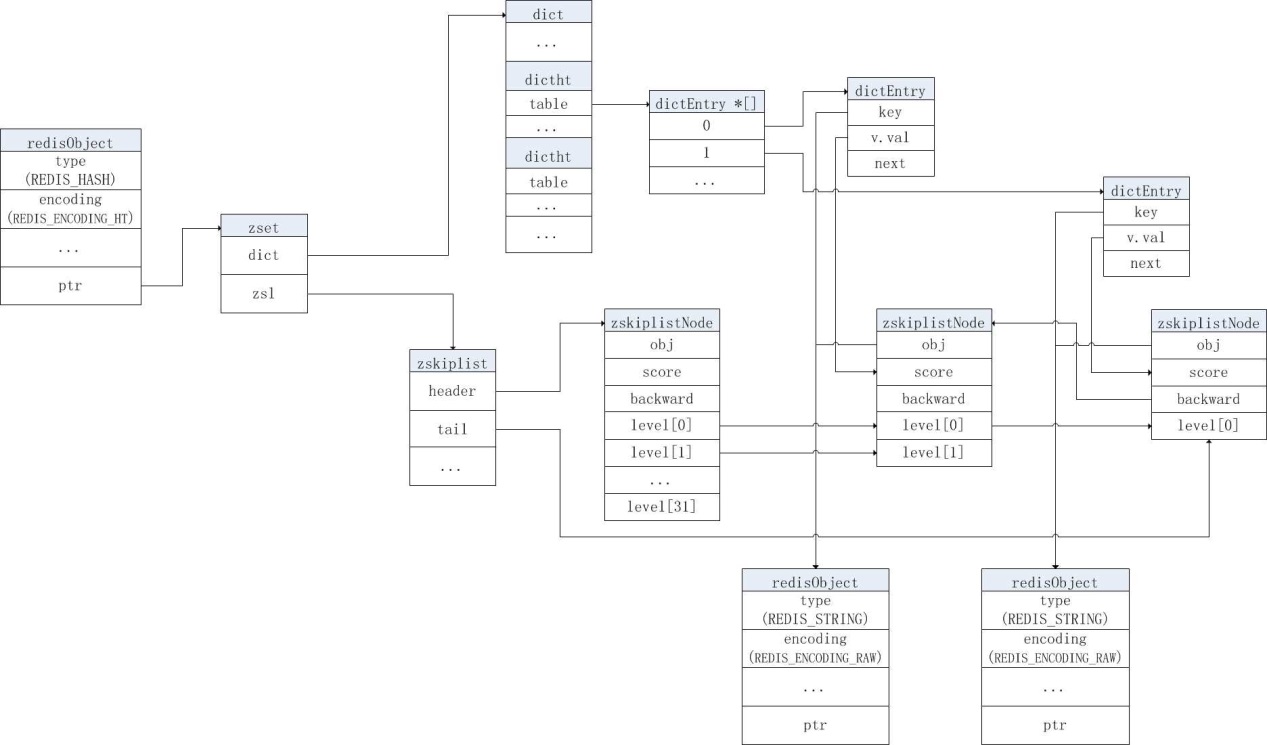
**}** zset**;**

zset结构中的跳跃表zskiplist，按分值从小到大保存所有集合元素，每个跳跃表节点zskiplistNode中，obj属性保存了元素的成员，而score属性则保存了元素的分值。通过这个跳跃表，程序可以对有序集合进行范围型操作，比如ZRANK，ZRANGE等命令就是基于跳跃表API来实现的。

除此之外，zset结构中的dict字典，为有序集合创建了一个从成员到分值的映射，字典中的每个键值对都保存了一个集合元素：字典的键保存了元素的成员，而字典的值则保存了元素的分值。通过这个字典，程序可以用O(1)复杂度查找给定成员的分值，ZSCORE命令就是根据这一特性实现的。

有序集合每个元素的成员都是一个字符串对象，而每个元素的分值都是一个double类型的浮点数。虽然zset结构同时使用跳跃表和字典来保存有序集合元素，但这两种数据结构会通过指针来共享相同元素的成员和分值，所以同时使用跳跃表和字典来保存集合元素不会产生任何重复成员或者分值，也不会因此而浪费额外的内存。

有序集合对象采用skiplist编码时，内存布局如下，在skiplist的节点zskiplistNode中，obj指针指向一个redisObject结构，表示有序集合元素的成员，在dict字典的哈希表dictht中，每个哈希表节点dictEntry结构中，key指针指向一个同一个redisObject结构；zskiplistNode中的score记录有序集合元素的分数，在dict字典的哈希表dictht中，每个哈希表节点dictEntry结构中，在v.val指向该zskiplistNode中的score：



对于使用ziplist编码的有序集合对象来说，当插入操作使得两个条件中的任意一个不能被满足时，就会执行编码转换。原本保存在压缩列表里的所有元素都会被转移并保存到zset结构中，对象的编码从ziplist变为skiplist。

比如，下面分别演示了插入长度超过64字节的元素，以及键值对数目超过128的情况：

127.0.0.1**:**6379**>** zadd testzset 1.0 0123012345678901234567890123456789012345678901234567890123456789

**(**integer**)** 1

127.0.0.1**:**6379**>** object encoding testzset

"ziplist"

127.0.0.1**:**6379**>** zadd testzset 2.0 01234012345678901234567890123456789012345678901234567890123456789

**(**integer**)** 1

127.0.0.1**:**6379**>** object encoding testzset

"skiplist"

127.0.0.1**:**6379**>** EVAL "for i=1, 128 do redis.call('ZADD', KEYS[1], i, i) end" 1 testzset2

**(**nil**)**

127.0.0.1**:**6379**>** zcard testzset2

**(**integer**)** 128

127.0.0.1**:**6379**>** object encoding testzset2

"ziplist"

127.0.0.1**:**6379**>** zadd testzset2 3.14 pi

**(**integer**)** 1

127.0.0.1**:**6379**>** zcard testzset2

**(**integer**)** 129

127.0.0.1**:**6379**>** object encoding testzset2

"skiplist"

以上两个条件中的阈值，可以通过配置文件中的以下选项进行修改：

zset-max-ziplist-entries 128

zset-max-ziplist-value 64

**六：命令的执行**

Redis中用于操作键的命令基本上可以分为两种类型：

一种命令可以对任何类型的键执行，比如DEL、EXPIRE、TYPE、OBJECT命令等；

另一种命令只能对特定类型的键执行，比如：SET，GET，APPEND，STRLEN等命令只能对字符串键执行。

使用一种数据类型的特定命令操作另一种数据类型的健会提示错误：”(error) WRONG TYPE Operation against a key holding the wrong kind of value”。

因此，在执行一个类型特定的命令之前，Redis会先检查输人键的类型是否正确，然后再决定是否执行给定的命令。类型检查是通过redisObject结构的type属性来实现的。

Redis除了会根据值对象的类型来判断键是否能够执行指定命令之外，还会根据值对象的编码方式，选择正确的函数来执行命令。

比如，列表对象的编码有ziplist和linkedlist两种，如果对一个列表键执行LLEN命令，那么服务器需要根据键的值对象所使用的编码来选择正确的LLEN命令实现，如果编码为ziplist，将调用ziplistLen函数来返回压缩列表的长度；如果编码为linkedlist，将调用listLength函数来返回双端链表的长度。

**七：引用计数**

利用redisObject结构的引用计数，也就是refcount属性，可以实现内存回收，以及对象共享机制。

C语言不具备内存自动回收功能，所以Redis在自己的对象系统中构建了一个引用计数技术，实现内存回收机制。通过这一机制，程序可以通过跟踪对象的引用计数信息，在适当的时候自动释放对象并进行内存回收。

对象的引用计数会随着对象的使用状态而不断变化：

创建一个新对象时，引用计数的值会被初始化为1；

当对象被一个新程序使用时，它的引用计数值会加1；

当对象不再被一个程序使用时，它的引用计数值会减1；

当对象的引用计数值变为0时，对象所占用的内存会被释放。

除了用于实现内存回收机制之外，对象的引用计数属性还带有对象共享的作用。比如，假设键A创建了一个整数值100的字符串对象作为值对象，如果键B也要创建一个整数值100的字符串对象作为值对象，那么Redis会让键A和键B共享同一个字符串对象。这样做可以节约内存。

在 Redis中，让多个键共享同一个值对象，只需要将共享的值对象的引用计数加1即可。像前面介绍有序集合对象的skiplist编码时，dict和skiplist两种数据结构就共享同一个redisObject对象作为元素的成员。

默认情况下，Redis会在初始化服务器时，会创建10000个字符串对象，这些对象包含了从0到9999的所有整数值，当需要用到值为0到9 999的字符串对象时，服务器就会使用这些共享对象.而不是新创建对象。

比如：如果创建一个值为100的键a，并使用OBJECT REFCOUNT命令查看键a的值对象的引用计数，会发现值对象的引用计数为2，这就是因为键a的值，共享了Redis预先创建好的对象的缘故：

127.0.0.1**:**6379**>** set a 100

OK

127.0.0.1**:**6379**>** object refcount a

**(**integer**)** 2

**八：对象的空转时间**

除了前面介绍过的type，encoding，ptr和refcount四个属性之外，redisObject结构包含的最后一个属性为lru属性，该属性记录了对象最后一次被程序访问的时间。

OBJECT IDLETIME命令可以打印出给定键的空转时长，这一空转时长就是通过将当前时间减去键的值对象的lru时间计算得出的：

127.0.0.1**:**6379**>** set msg "hello"

OK

127.0.0.1**:**6379**>** object idletime msg

**(**integer**)** 6

127.0.0.1**:**6379**>** object idletime msg

**(**integer**)** 23

127.0.0.1**:**6379**>** get msg

"hello"

127.0.0.1**:**6379**>** object idletime msg

**(**integer**)** 1

键的空转时长还有另外一个作用，如果服务器打开了maxmemoxy选项，并且服务器用于回收内存的算法为volatile-lru或者allkeys-lru，那么当服务器占用的内存数超过了maxmemory选项所设置的阈值时，空转时长较高的那部分键会优先被服务器释放，从而回收内存。

对象相关的代码，可以参阅：

https://github.com/gqtc/redis-3.0.5/blob/master/redis-3.0.5/src/object.c