压缩列表(ziplist)是列表键和哈希键的底层实现之一。当列表键只包含少量列表项，并且每个列表项要么是小整数值，要么是长度较短的字符串时；或者当哈希键只包含少量键值对，并且每个键值对的键和值要么是小整数值，要么是长度较短的字符串时，那么Redis就会使用压缩列表来做为列表键或哈希键的底层实现。

压缩列表是Redis为了节约内存而开发的，可用于存储字符串和整数值。它是一个顺序型数据结构，由一系列特殊编码的连续内存块组成。一个压缩列表可以包含任意多个结点(entry)，每个entry的大小不定，每个entry可保存一个字符串或一个整数值。

ziplist的相关实现在都在ziplist.c中。

**一：ziplist结构**

ziplist的结构如下：

<zlbytes><zltail><zllen><entry1><entry2>...<entryN><zlend>

zlbytes是一个uint32\_t类型的整数，表示整个ziplist占用的内存字节数；

zltail是一个uint32\_t类型的整数，表示ziplist中最后一个entry的偏移量，通过该偏移量，无需遍历整个ziplist就可以确定尾结点的地址；

zllen是一个uint16\_t类型的整数，表示ziplist中的entry数目。如果该值小于UINT16\_MAX(65535)，则该属性值就是ziplist的entry数目，若该值等于UINT16\_MAX(65535)，则还需要遍历整个ziplist才能得到真正的entry数目；

entryX表示ziplist的结点，每个entry的长度不定；

zlend是一个uint8\_t类型的整数，其值为0xFF(255)，表示ziplist的结尾。

在ziplist.c中，定义了一系列的宏，可以分别获取ziplist中存储的各个属性，比如：

#define ZIPLIST\_BYTES(zl) (\*((uint32\_t\*)(zl)))

#define ZIPLIST\_TAIL\_OFFSET(zl) (\*((uint32\_t\*)((zl)+sizeof(uint32\_t))))

#define ZIPLIST\_LENGTH(zl) (\*((uint16\_t\*)((zl)+sizeof(uint32\_t)\*2)))

#define ZIPLIST\_HEADER\_SIZE (sizeof(uint32\_t)\*2+sizeof(uint16\_t))

#define ZIPLIST\_ENTRY\_HEAD(zl) ((zl)+ZIPLIST\_HEADER\_SIZE)

#define ZIPLIST\_ENTRY\_TAIL(zl) ((zl)+intrev32ifbe(ZIPLIST\_TAIL\_OFFSET(zl)))

#define ZIPLIST\_ENTRY\_END(zl) ((zl)+intrev32ifbe(ZIPLIST\_BYTES(zl))-1)

ZIPLIST\_BYTES获取ziplist中的zlbytes属性；ZIPLIST\_TAIL\_OFFSET获取ziplist的zltail属性；ZIPLIST\_LENGTH获取ziplist的zllen属性

ZIPLIST\_ENTRY\_HEAD得到ziplist头结点的地址；ZIPLIST\_ENTRY\_TAIL得到ziplist中尾节点的首地址，ZIPLIST\_ENTRY\_END得到ziplist结尾字节zlend的地址。

注意，ziplist中所有的属性值都是以小端的格式存储的。因此取得ziplist中保存的属性值后，还需要对内存做字节翻转才能得到真正的值。intrev32ifbe就是在大端系统下对内存进行字节翻转的函数。

**二：entry结构**

每个压缩列表节点都由previous\_entry\_length、encoding和content三部分组成。

previous\_entry\_length表示前一个entry的字节长度，根据该字段值就可以从后向前遍历ziplist。

previous\_entry\_length字段长度可以是1字节，也可以是5字节。如果前一个entry长度小于254字节，则该字段只占用一个字节；如果前一个entry长度大于等于254字节，则该字段占用5个字节：第一个字节置为0xFE(254)，之后的4个字节保存entry的实际长度（小端格式存储）。

之所以用0xFE(254)这个值作为分界点，是因为0xFF(255)被用作ziplist的结束标志，一旦扫描到0xFF，就认为ziplist结束了。

entry的content字段保存节点的实际内容，它可以是一个字符串或者整数，值的类型和长度由encoding属性决定。

encoding字段记录了节点的content所保存的数据类型及长度。如果entry中保存的是字符串，则encoding字段的前2个二进制位可以是00、01和10，分别表示不同长度类型的字符串，剩下的二进制位就表示字符串的实际长度；如果entry中的内容为整数，则encoding字段的前2个二进制位都为11，剩下的2个二进制位表示整数的类型。encoding的形式如下：

00pppppp，1字节，表示长度小于等于63(2^6 - 1) 字节的字符串；

01pppppp|qqqqqqqq，2 字节，表示长度小于等于16383 (2^14 - 1) 字节的字符串；

10\_\_\_\_\_\_|qqqqqqqq|rrrrrrrr|ssssssss|tttttttt，5字节，表示长度小于等于4294967295(2^32 - 1) 字节的字符串

11000000，1字节，表示int16\_t类型的整数；

11010000，1字节，表示int32\_t类型的整数；

11100000，1字节，表示int64\_t类型的整数；

11110000，1字节，表示24位有符号整数；

11111110，1字节，表示8位有符号整数；

1111xxxx，1字节，表示0到12之间的值。使用这一编码的节点没有相应的oontont属性，因为xxxx就可以表示0到12之间的值。因为0000和1110不可用，所以xxxx的取值从0001到1101，也就是1到13之间的值。因此，需要从xxxx减去1，才能得到实际的值。

注意，所有的整数都是以小端的格式存储的。

**三：连锁更新**

每个节点的previous\_entry\_length字段记录了前一个节点的长度。如果前一节点的长度小于254字节，那么previous\_entry\_length字段占用1个字节；如果前一节点的长度大于等于254字节，那么previous\_entry\_length字段占用5个字节。

考虑这样一种情况：在一个压缩列表中，有多个连续的、长度介于250字节到253字节之间的节点e1至eN。因所有节点的长度都小于254字节，所以e1至eN节点的previous\_entry\_length字段都是1字节长。

这时，如果将一个长度大于等于254字节的新节点new设置为压缩列表的表头节点，那么new将成为e1的前置节点，但是因为e1的previous\_entry\_length字段仅长1字节，没办法保存新节点new的长度，所以需要对压缩列表执行空间重分配操作，将e1节点的previous\_entry\_length字段从原来的1字节扩展为5字节。

现在，麻烦的事情来了，原e1的长度介于250字节至253字节之间，e1的previous\_entry\_length字段变成5个字节后，e1的长度就大于等于254了。从而e2的previous\_entry\_length字段，也需要从原来的1字节长扩展为5字节长。

因此，需要再次对压缩列表执行空间重分配操作，并将e2节点的previous\_entry\_length属性从原来的1字节长扩展为5字节长。

以此类推，为了让每个节点的previous\_entry\_length属性都符合压缩列表对节点的要求，程序需要不断地对压缩列表执行空间重分配操作，直到eN为止。

Redis将这种在特殊情况下产生的连续多次空间扩展操作称之为“连锁更新”(cascade update)。除了添加新节点可能会引发连锁更新之外，删除节点也可能会引发连锁更新。

因为连锁更新在最坏情况下需要对压缩列表执行N次空间重分配操作，而每次空间重分配的最坏复杂度为O(N)，所以连锁更新的最坏复杂度为O(N^2)。

尽管连锁更新的复杂度较高，但它真正造成性能间题的几率是很低的。

首先，压缩列表里要恰好有多个连续的、长度介于250字节至253字节之间的节点，连锁更新才有可能触发。在实际中，这种情况并不多见；

其次，即使出现连锁更新，但只要被更新的节点数量不多，就不会对性能造成任何影响，比如对三五个节点进行连锁更新是绝对不会影响性能的；

因此，ziplistPush等函数的平均复杂度仅为O(N)。在实际中，我们可以放心地使用这些函数，而不必担心连锁更新会影响压缩列表的性能。

ziplist变动时，previous\_entry\_length字段长度可能需要从1字节扩展为5字节，从而会引起连锁更新，也可能需要从5字节收缩为1字节。这就有可能会发生抖动现象，也就是节点的previous\_entry\_length字段，不断的扩展和收缩的现象。Redis中，为了避免这种现象，允许previous\_entry\_length字段在需要收缩的情况下，保持5字节不变。

**四：代码**

Redis中的ziplist实现，并未涉及难以理解的算法。但是因为ziplist本身的编码需求较多，因而代码需要处理各种细节，初看之下比较繁杂，分析之后，其实很容易理解。

1：连锁更新

下面就是处理连锁更新的代码，zl指向一个ziplist，p指向其中第一个不需要更新的节点（第一个已经更新过的节点），其后续的节点可能需要更新：

static unsigned char **\***\_\_ziplistCascadeUpdate**(**unsigned char **\***zl**,** unsigned char **\***p**)** **{**

size\_t curlen **=** intrev32ifbe**(**ZIPLIST\_BYTES**(**zl**)),** rawlen**,** rawlensize**;**

size\_t offset**,** noffset**,** extra**;**

unsigned char **\***np**;**

zlentry cur**,** next**;**

**while** **(**p**[**0**]** **!=** ZIP\_END**)** **{**

cur **=** zipEntry**(**p**);**

rawlen **=** cur**.**headersize **+** cur**.**len**;**

rawlensize **=** zipPrevEncodeLength**(NULL,**rawlen**);**

*/\* Abort if there is no next entry. \*/*

**if** **(**p**[**rawlen**]** **==** ZIP\_END**)** **break;**

next **=** zipEntry**(**p**+**rawlen**);**

*/\* Abort when "prevlen" has not changed. \*/*

**if** **(**next**.**prevrawlen **==** rawlen**)** **break;**

**if** **(**next**.**prevrawlensize **<** rawlensize**)** **{**

*/\* The "prevlen" field of "next" needs more bytes to hold*

*\* the raw length of "cur". \*/*

offset **=** p**-**zl**;**

extra **=** rawlensize**-**next**.**prevrawlensize**;**

zl **=** ziplistResize**(**zl**,**curlen**+**extra**);**

p **=** zl**+**offset**;**

*/\* Current pointer and offset for next element. \*/*

np **=** p**+**rawlen**;**

noffset **=** np**-**zl**;**

*/\* Update tail offset when next element is not the tail element. \*/*

**if** **((**zl**+**intrev32ifbe**(**ZIPLIST\_TAIL\_OFFSET**(**zl**)))** **!=** np**)** **{**

ZIPLIST\_TAIL\_OFFSET**(**zl**)** **=**

intrev32ifbe**(**intrev32ifbe**(**ZIPLIST\_TAIL\_OFFSET**(**zl**))+**extra**);**

**}**

*/\* Move the tail to the back. \*/*

memmove**(**np**+**rawlensize**,**

np**+**next**.**prevrawlensize**,**

curlen**-**noffset**-**next**.**prevrawlensize**-**1**);**

zipPrevEncodeLength**(**np**,**rawlen**);**

*/\* Advance the cursor \*/*

p **+=** rawlen**;**

curlen **+=** extra**;**

**}** **else** **{**

**if** **(**next**.**prevrawlensize **>** rawlensize**)** **{**

*/\* This would result in shrinking, which we want to avoid.*

*\* So, set "rawlen" in the available bytes. \*/*

zipPrevEncodeLengthForceLarge**(**p**+**rawlen**,**rawlen**);**

**}** **else** **{**

zipPrevEncodeLength**(**p**+**rawlen**,**rawlen**);**

**}**

*/\* Stop here, as the raw length of "next" has not changed. \*/*

**break;**

**}**

**}**

**return** zl**;**

**}**

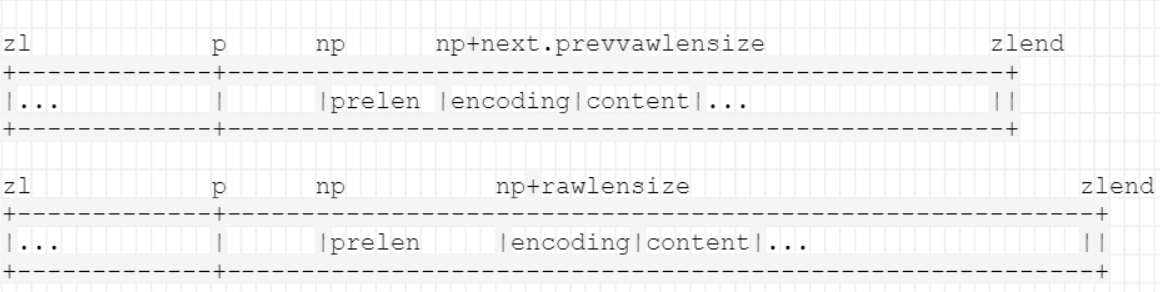
首先获取ziplist当前的长度curlen；

然后从p开始轮训，获取p指向节点的总长度rawlen，以及编码该字节长度需要的字节数rawlensize；

如果p指向的结点没有后继结点，直接退出返回；否则，next表示p的后继节点，next节点的previous\_entry\_length字段值为next.prevrawlen，该字段长度是next.prevrawlensize。如果next.prevrawlen与rawlen相等，则表示next节点的前继节点的长度未发生变化，直接退出返回；

如果next.prevrawlensize小于rawlensize，表示next节点的前继节点的长度，原来小于254字节，现在大于等于254字节了。因此next节点的previous\_entry\_length字段需要扩展长度，扩展的字节数extra为rawlensize-next.prevrawlensize，利用ziplistResize扩展ziplist的内存空间。注意，扩容前要保存p的偏移量，扩容后利用该偏移量，可以重新得到p的位置。

如下图，分别表示扩容前和扩容后的情况：



如果next节点不是尾节点，则需要更新ziplist的zltail属性；如果next是尾结点，因为next之前的内容没有变化，因此无需更新zltail属性；

然后开始移动内存，移动的内容是，从next节点的previous\_entry\_length字段之后的内存开始，到ziplist末尾字节zlend之前的内容：

memmove(np+rawlensize, np+next.prevrawlensize, curlen-noffset-next.prevrawlensize-1);

然后更新next中的previous\_entry\_length字段为rawlen；

然后p指向next节点，依次遍历下一个节点；

如果next.prevrawlensize大于rawlensize，表示next节点的前继节点的长度，原来大于等于254字节，现在小于254字节了。为了避免“抖动”，调用zipPrevEncodeLengthForceLarge保持next节点的previous\_entry\_length字段长度不变，并强制编码为rawlen，然后退出返回；

如果next.prevrawlensize等于rawlensize，表示next节点的前继节点的长度，原来小于254字节，现在还是小于254字节，或者原来大于等于254字节，现在还是大于等于254字节。这种情况下，直接将next节点的previous\_entry\_length字段编码为rawlen，然后退出返回。

2：删除节点

下面是删除节点的代码，从ziplist中，p指向的节点开始，最多删除num个节点：

static unsigned char **\***\_\_ziplistDelete**(**unsigned char **\***zl**,** unsigned char **\***p**,** unsigned int num**)** **{**

unsigned int i**,** totlen**,** deleted **=** 0**;**

size\_t offset**;**

int nextdiff **=** 0**;**

zlentry first**,** tail**;**

first **=** zipEntry**(**p**);**

**for** **(**i **=** 0**;** p**[**0**]** **!=** ZIP\_END **&&** i **<** num**;** i**++)** **{**

p **+=** zipRawEntryLength**(**p**);**

deleted**++;**

**}**

totlen **=** p**-**first**.**p**;**

**if** **(**totlen **>** 0**)** **{**

**if** **(**p**[**0**]** **!=** ZIP\_END**)** **{**

*/\* Storing `prevrawlen` in this entry may increase or decrease the*

*\* number of bytes required compare to the current `prevrawlen`.*

*\* There always is room to store this, because it was previously*

*\* stored by an entry that is now being deleted. \*/*

nextdiff **=** zipPrevLenByteDiff**(**p**,**first**.**prevrawlen**);**

p **-=** nextdiff**;**

zipPrevEncodeLength**(**p**,**first**.**prevrawlen**);**

*/\* Update offset for tail \*/*

ZIPLIST\_TAIL\_OFFSET**(**zl**)** **=**

intrev32ifbe**(**intrev32ifbe**(**ZIPLIST\_TAIL\_OFFSET**(**zl**))-**totlen**);**

*/\* When the tail contains more than one entry, we need to take*

*\* "nextdiff" in account as well. Otherwise, a change in the*

*\* size of prevlen doesn't have an effect on the \*tail\* offset. \*/*

tail **=** zipEntry**(**p**);**

**if** **(**p**[**tail**.**headersize**+**tail**.**len**]** **!=** ZIP\_END**)** **{**

ZIPLIST\_TAIL\_OFFSET**(**zl**)** **=**

intrev32ifbe**(**intrev32ifbe**(**ZIPLIST\_TAIL\_OFFSET**(**zl**))+**nextdiff**);**

**}**

*/\* Move tail to the front of the ziplist \*/*

memmove**(**first**.**p**,**p**,**

intrev32ifbe**(**ZIPLIST\_BYTES**(**zl**))-(**p**-**zl**)-**1**);**

**}** **else** **{**

*/\* The entire tail was deleted. No need to move memory. \*/*

ZIPLIST\_TAIL\_OFFSET**(**zl**)** **=**

intrev32ifbe**((**first**.**p**-**zl**)-**first**.**prevrawlen**);**

**}**

*/\* Resize and update length \*/*

offset **=** first**.**p**-**zl**;**

zl **=** ziplistResize**(**zl**,** intrev32ifbe**(**ZIPLIST\_BYTES**(**zl**))-**totlen**+**nextdiff**);**

ZIPLIST\_INCR\_LENGTH**(**zl**,-**deleted**);**

p **=** zl**+**offset**;**

*/\* When nextdiff != 0, the raw length of the next entry has changed, so*

*\* we need to cascade the update throughout the ziplist \*/*

**if** **(**nextdiff **!=** 0**)**

zl **=** \_\_ziplistCascadeUpdate**(**zl**,**p**);**

**}**

**return** zl**;**

**}**

首先将p转化为zlentry结构的first，然后从p开始向后遍历num个节点，这样，就得到了需要删除的节点数deleted，以及要删除的总字节数totlen。只有当totlen大于0时，才进行删除动作，否则直接返回zl；此时p指向最后一个删除节点的后继节点；

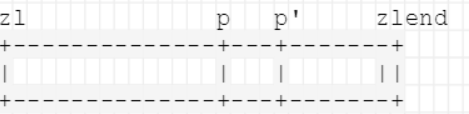
若此时p不是ziplist的结尾字节，则其指向节点的previous\_entry\_length字段，需要设置为新值first.prevrawlen。首先计算p指向节点，其新旧previous\_entry\_length字段长度的差值nextdiff，nextdiff为正，表示p节点的previous\_entry\_length字段需要扩容，从要删除的内存中选择扩容的部分，因此p-=nextdiff，p向前走nextdiff个字节，表示少删除nextdiff个字节。如果nextdiff为负，表示p节点的previous\_entry\_length字段需要收缩，因此p-=nextdiff，表示p向后走nextdiff个字节，表示多删除nextdiff个字节；

然后重新设置p指向节点的previous\_entry\_length字段值为first.prevrawlen；

接下来就是调整zltail属性，分情况讨论：如果遍历num个节点之后，未做nextdiff调整之前，p指向的节点就是尾结点，如下图所示（仅以nextdiff为正为例）：



上图中，p'表示未做nextdiff调整之前的指针，也就是指向原尾结点的指针，p表示做出nextdiff调整之后的指针。删除num个节点之后，zltail的值，也就是first.p-zl，因此有：new\_zltail = old\_zltail - totlen; 删除后如下图所示：



上面的情况，对于nextdiff为负也是成立的，不再赘述。

如果遍历num个节点之后，未做nextdiff调整之前，p指向的节点不是尾结点，则新的zltail，等于旧的zltail，减去删除的总字节数，删除的总字节数为totlen-nextdiff，因此有：new\_zltail = old\_zltail - totlen + nextdiff;

更新完zltail属性后，接下来就是移动内存了，将当前p指向的内存，移动到first.p指向的位置上，移动的内存总数就是，从当前p指向的内存开始，到ziplist结尾字节之前的内容；

如果在遍历num个节点之后，p指向的字节就是ziplist的结尾字节，则无需移动内存，仅需要重新设置zltail属性即可，此时的尾结点，是first.p的前继节点，因此有：

new\_zltail = first.p - zl - first.prevrawlen;

移动内存，以及设置新的zltail之后，剩下的就是重新为ziplist分配空间，并且设置新的zllen属性。注意，重新分配ziplist内存之前，保存p的偏移量offset，这样，在分配空间之后，就可以利用offset重新得到p的位置了；

如果nextdiff不为0，表示p指向的节点的大小发生了变化，调用\_\_ziplistCascadeUpdate处理后续的节点。

3：插入节点

下面是插入节点的代码，根据原始数据s，及其长度slen，新建一个ziplist节点，将该节点插入到p之前的位置，也就是，p指向的节点，成为新结点的后继节点。

static unsigned char **\***\_\_ziplistInsert**(**unsigned char **\***zl**,** unsigned char **\***p**,** unsigned char **\***s**,** unsigned int slen**)** **{**

size\_t curlen **=** intrev32ifbe**(**ZIPLIST\_BYTES**(**zl**)),** reqlen**;**

unsigned int prevlensize**,** prevlen **=** 0**;**

size\_t offset**;**

int nextdiff **=** 0**;**

unsigned char encoding **=** 0**;**

long long value **=** 123456789**;** */\* initialized to avoid warning. Using a value*

*that is easy to see if for some reason*

*we use it uninitialized. \*/*

zlentry tail**;**

*/\* Find out prevlen for the entry that is inserted. \*/*

**if** **(**p**[**0**]** **!=** ZIP\_END**)** **{**

ZIP\_DECODE\_PREVLEN**(**p**,** prevlensize**,** prevlen**);**

**}** **else** **{**

unsigned char **\***ptail **=** ZIPLIST\_ENTRY\_TAIL**(**zl**);**

**if** **(**ptail**[**0**]** **!=** ZIP\_END**)** **{**

prevlen **=** zipRawEntryLength**(**ptail**);**

**}**

**}**

*/\* See if the entry can be encoded \*/*

**if** **(**zipTryEncoding**(**s**,**slen**,&**value**,&**encoding**))** **{**

*/\* 'encoding' is set to the appropriate integer encoding \*/*

reqlen **=** zipIntSize**(**encoding**);**

**}** **else** **{**

*/\* 'encoding' is untouched, however zipEncodeLength will use the*

*\* string length to figure out how to encode it. \*/*

reqlen **=** slen**;**

**}**

*/\* We need space for both the length of the previous entry and*

*\* the length of the payload. \*/*

reqlen **+=** zipPrevEncodeLength**(NULL,**prevlen**);**

reqlen **+=** zipEncodeLength**(NULL,**encoding**,**slen**);**

*/\* When the insert position is not equal to the tail, we need to*

*\* make sure that the next entry can hold this entry's length in*

*\* its prevlen field. \*/*

nextdiff **=** **(**p**[**0**]** **!=** ZIP\_END**)** **?** zipPrevLenByteDiff**(**p**,**reqlen**)** **:** 0**;**

*/\* Store offset because a realloc may change the address of zl. \*/*

offset **=** p**-**zl**;**

zl **=** ziplistResize**(**zl**,**curlen**+**reqlen**+**nextdiff**);**

p **=** zl**+**offset**;**

*/\* Apply memory move when necessary and update tail offset. \*/*

**if** **(**p**[**0**]** **!=** ZIP\_END**)** **{**

*/\* Subtract one because of the ZIP\_END bytes \*/*

memmove**(**p**+**reqlen**,**p**-**nextdiff**,**curlen**-**offset**-**1**+**nextdiff**);**

*/\* Encode this entry's raw length in the next entry. \*/*

zipPrevEncodeLength**(**p**+**reqlen**,**reqlen**);**

*/\* Update offset for tail \*/*

ZIPLIST\_TAIL\_OFFSET**(**zl**)** **=**

intrev32ifbe**(**intrev32ifbe**(**ZIPLIST\_TAIL\_OFFSET**(**zl**))+**reqlen**);**

*/\* When the tail contains more than one entry, we need to take*

*\* "nextdiff" in account as well. Otherwise, a change in the*

*\* size of prevlen doesn't have an effect on the \*tail\* offset. \*/*

tail **=** zipEntry**(**p**+**reqlen**);**

**if** **(**p**[**reqlen**+**tail**.**headersize**+**tail**.**len**]** **!=** ZIP\_END**)** **{**

ZIPLIST\_TAIL\_OFFSET**(**zl**)** **=**

intrev32ifbe**(**intrev32ifbe**(**ZIPLIST\_TAIL\_OFFSET**(**zl**))+**nextdiff**);**

**}**

**}** **else** **{**

*/\* This element will be the new tail. \*/*

ZIPLIST\_TAIL\_OFFSET**(**zl**)** **=** intrev32ifbe**(**p**-**zl**);**

**}**

*/\* When nextdiff != 0, the raw length of the next entry has changed, so*

*\* we need to cascade the update throughout the ziplist \*/*

**if** **(**nextdiff **!=** 0**)** **{**

offset **=** p**-**zl**;**

zl **=** \_\_ziplistCascadeUpdate**(**zl**,**p**+**reqlen**);**

p **=** zl**+**offset**;**

**}**

*/\* Write the entry \*/*

p **+=** zipPrevEncodeLength**(**p**,**prevlen**);**

p **+=** zipEncodeLength**(**p**,**encoding**,**slen**);**

**if** **(**ZIP\_IS\_STR**(**encoding**))** **{**

memcpy**(**p**,**s**,**slen**);**

**}** **else** **{**

zipSaveInteger**(**p**,**value**,**encoding**);**

**}**

ZIPLIST\_INCR\_LENGTH**(**zl**,**1**);**

**return** zl**;**

**}**

首先获取ziplist当前长度curlen；

然后得到p指向位置的前继节点的长度prevlen；

然后计算新结点中content部分的长度：如果s中的内容能够表示为整数值，则得到该整数值的长度reqlen，否则，reqlen等于slen；

根据prevlen，得到新结点的previous\_entry\_length字段长度，加到reqlen中；

根据encoding和slen，得到新结点的encoding字段长度，加到reqlen中，至此，reqlen就表示新结点的总长度；

因新结点将成为p的前继节点，因此，只要p并非ziplist的结尾字节，就利用zipPrevLenByteDiff，计算p中新旧previous\_entry\_length字段的长度之差nextdiff，nextdiff为正，表示因新结点的长度大于等于254，p的previous\_entry\_length字段需要扩容nextdiff个字节；如果nextdiff为负，表示因新结点的长度小于254，p的previous\_entry\_length字段需要收缩nextdiff个字节；

接下来，利用ziplistResize对ziplist进行扩容，扩容长度为reqlen+nextdiff，扩容之前，保存p的偏移量offset，这样扩容后根据offset，就可以重新得到p的位置；

如果p并非ziplist的结尾字节，则接下来开始移动内存，从p-nextdiff指向的位置，移动到p+reqlen的位置，移动的字节数为curlen-offset-1+nextdiff；

然后设置新结点的后继节点（也就是插入之前，p指向的节点）的previous\_entry\_length字段；

然后设置zltail属性，这里根据p是否是尾结点而区分对待，情况类似与\_\_ziplistDelete中的讨论，不再赘述。

如果p就是ziplist的结尾字节，则只需要更新zltail属性即可；

如果nextdiff非0，则需要调用\_\_ziplistCascadeUpdate，处理p+reqlen节点开始的连锁升级或是收缩情况；

最后，设置插入的新结点的各种字段值，并更新zllen属性。

其他关于ziplist的代码，可参阅：

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