ziplsit的特点简单介绍：

ziplist其实就是分配一块连续的内存，用指针和位操作来操作内存的一种高效的数据结构。

1. Ziplist 能存储strings和integer值，整型值被存储为实际的整型值而不是字符数组
2. Ziplist 是为了尽可能节约内存而设计相当特殊的双端队列
3. Ziplist 在头部和尾部的操作时间0（1），ziplist的操作都需要重新分配内存，所以实际的复杂度和ziplist的使用和内存有关。

ziplist压缩列表存储结构：

```
 +---------+--------+-------+--------+--------+--------+--------+------
-+
 | zlbytes | zltail | zllen | entry1 | entry2 |  ...   | entryN | zlend
|
 +---------+--------+-------+--------+--------+--------+--------+------
-+
```

zlbytes: 使用的内存数量。通过这个值，程序可以直接对 ziplist 的内存大小进行调整，而无须为了计算ziplist的内存大小而遍历整个列表。

zltail: 4字节，保存着到达列表中最后一个节点的偏移量。这个偏移量使得对表尾的操作可以在无须遍历整个列表的情况下进行。

zllen: 2字节，保存着列表中的节点数量。当 zllen 保存的值大于 2**16-1= 65535 时程序需要遍历整个列表才能知道列表实际包含了多少个节点。

zlend: 1字节，值为 255 = 0xFF，标识列表的末尾。

```c
// 判断encoding是否是字符串编码
#define ZIP_IS_STR(enc) (((enc) & ZIP_STR_MASK) < ZIP_STR_MASK)
```

```
//返回整个压缩列表的大小  即求zlbytes的值
/* Return total bytes a ziplist is composed of. */
#define ZIPLIST_BYTES(zl)       ( *(  (uint32_t*)(zl) ) )

/* Return the offset of the last item inside the ziplist. */
//返回压缩列表起始位置到最后一个元素的偏移量
#define ZIPLIST_TAIL_OFFSET(zl) (*((uint32_t*)((zl)+sizeof(uint32_t))))

//返回压缩列表的元素的个数
#define ZIPLIST_LENGTH(zl)      (*((uint16_t*)
((zl)+sizeof(uint32_t)*2)))

//返回压缩列表首部的长度 8字节
#define ZIPLIST_HEADER_SIZE     (sizeof(uint32_t)*2+sizeof(uint16_t))

//zlend  0xFF 一字节
#define ZIPLIST_END_SIZE        (sizeof(uint8_t))

//返回指向压缩列表的第一个元素的指针
#define ZIPLIST_ENTRY_HEAD(zl)  ((zl)+ZIPLIST_HEADER_SIZE)

/* Return the pointer to the last entry of a ziplist, using the
 * last entry offset inside the ziplist header. */
 //返回指向最后一个元素的指针
#define ZIPLIST_ENTRY_TAIL(zl)
((zl)+intrev32ifbe(ZIPLIST_TAIL_OFFSET(zl)))

//返回指针zlend的指针
/* Return the pointer to the last byte of a ziplist, which is, the
 * end of ziplist FF entry. */
#define ZIPLIST_ENTRY_END(zl)
((zl)+intrev32ifbe(ZIPLIST_BYTES(zl))-1)
```

压缩列表每个元素 entryX（entry1、entry2 .....）的结构：

```
 +----------+-----------+-------+
```

```
| prevlen  | encoding  | value |
+----------+-----------+-------+
```

prevlen：用来表示前一个元素entry的字节长度，prevlen元素本身存储在计算机所需要占用内存大小要么是1个字节，要么是5个字节，如果前一个元素的字节长度（占用的内存）小于154，那么prevlen占用1一个字节，如果前一个元素>=254字节，prevlen占用5个字节，此时prevlen第一个字节固定为0XFE(254),prevlen的后四字节才真正用来表示前一个元素的长度。

encoding： value的编码（编码元素的类型int还是表示字符串的字节数组/value的长度）

value： 元素的值：要么是整数，要么是字节数组表示字符串

压缩列表元素的encoding编码规则介绍：

ziplist的元素能存储int和字符串类型
先介绍字符串编码：此时encoding 存贮类型和len

encoding	占用字节	存贮结构encode/len	字符串长度范围	len取值
ZIP_STR_06B	1字节	00XXXXXX	长度<64	后6位
ZIP_STR_14B	2字节	01XXXXXX XXXXXXXX	长度<16384	后14位 2^14-1
ZIP_STR_32B	5字节	10000000 XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX	长度=<2^32-1	32位

int编码：由于整型的长度是固定的，因此 只需存储encoding信息，length值可根据编码进行计算得出。

encoding	占用字节	存储结构	取值范围
ZIP_INT_XX	1字节	11 11 0001~11111101	0~12
ZIP_INT_8B	1字节	11 11 1110	-2^{8}~2^{8}-1
ZIP_INT_16B	2字节	11 00 0000	-2^{16}~2^{16}-1
ZIP_INT_24B	3字节	11 11 0000	-2^{24}~2^{24}-1
ZIP_INT_32B	4字节	11 01 0000	-2^{32}~2^{32}-1
ZIP_INT_64B	8字节	11 10 0000	-2^{64}~2^{64}-1

注意到没有：

- **encoding字段第一个字节的前2位 00 01 10的时候表示元素entryx类型是字符串，11的时候表示是整型，因此通过encoding可以知道元素value字段的类型是否是整型或者字节数组（以及字节数组的长度）**
- 1111 0001~1111 1101表示0到12的的编码，其后四位的取值减去1就是0到12， 例如0001-1 = 0 、 1101-1= 12

redis预定义以下常量对应encoding字段的各编码类型：

```
#define ZIP_STR_06B (0 << 6)
#define ZIP_STR_14B (1 << 6)
#define ZIP_STR_32B (2 << 6)
#define ZIP_INT_16B (0xc0 | 0<<4)   ----> 11 00 0000  -2^16~2^16-1
#define ZIP_INT_32B (0xc0 | 1<<4)   ----> 11 01 0000  -2^32~2^32-1
#define ZIP_INT_64B (0xc0 | 2<<4)   ----> 11 10 0000  -2^64~2^64-1
#define ZIP_INT_24B (0xc0 | 3<<4)   ----> 11 11 0000  -2^24~2^24-1
#define ZIP_INT_8B 0xfe             ----> 11 11 1110  -2^8~2^8-1
```

来看一个例子：zl表中存贮2和5两个元素

```
[0f 00 00 00] [0c 00 00 00] [02 00] [00 f3] [02 f6] [ff]
      |             |          |       |       |     |
   zlbytes        zltail    entries   "2"     "5"   end
```

1. **zlbytes**： 上面的压缩列表zl总长度为15字节：所以**zlbytes = 0x0f** 值为15

- [ ] **注意了：redis里面都是采用小端模式，0f 00 00 00 按照小端取值为0x00 00 00 0f**

2. **ztail**： 取值0x0000000c;所以ztail=12 ztail就是压缩列表zl起始位置到尾部元素5的偏移量

3. **entries**：0x0002表示压缩列表元素的个数为2

4. 第一个元素00 f3 因为是第一个元素所以prevlen=0：第一个元素前面没有元素所以prevlen所用所需要的存贮空间占用一个字，所以00 f3的第一个字节为00 值为0，接下来是encoding编码，查看接下来第一个字节f3 11110011 的前两位11 所以是整数，整数的类型看前四位1111 所以第一个元素的值是3-1=2

5. 第二个元素02 f6，因为第一个元素的长度为2字节小于254，所以第二个元素的prevlen所需要的内存只需要一个字节，所以02 f6的第一个字节02表示前一个元素的长度 0x02=2， ，0xf6= 1111 0110 其中1111编码表示整型，0110=6，6-1=5，所以第二个元素的值为5

6. end的编码0xFF

entryX元素解码之后存储结构

```
typedef struct zlentry {
    unsigned int prevrawlensize; /* Bytes used to encode the previous
entry len*/
    unsigned int prevrawlen;     /* Previous entry len. */
    unsigned int lensize;        /* Bytes used to encode this entry
type/len.
                                    For example strings have a 1, 2 or
5 bytes
                                    header. Integers always use a
```

```
single byte.*/
    unsigned int len;           /* Bytes used to represent the actual
entry.
                                   For strings this is just the string
length
                                   while for integers it is 1, 2, 3,
4, 8 or
                                   0 (for 4 bit immediate) depending
on the
                                   number range. */
    unsigned int headersize;    /* prevrawlensize + lensize. */
    unsigned char encoding;     /* Set to ZIP_STR_* or ZIP_INT_*
depending on
                                   the entry encoding. However for 4
bits
                                   immediate integers this can assume
a range
                                   of values and must be range-
checked. */
    unsigned char *p;           /* Pointer to the very start of the
entry, that
                                   is, this points to prev-entry-len
field. */
} zlentry;
```

再看一下entryX在ziplist中存储结构，然后分析zlentry

```
+----------+-----------+-------+
| prevlen  | encoding  | value |
+----------+-----------+-------+
```

其上面三个字段通过zipEntry函数解析存储结构为zlentry

- **prevrawlen**： 就是上面提到prevlen，表示前一个元素的长度。
- **prevrawlensize**： prevrawlen本身编码的字节数，也就是prevrawlen本身存贮所需要的内存空间，redis中规定前一个元素的长度小于254就占用1字节，大于等于254占用5字节。
- **len**：表示元素的长度。

- **lensize**：表示encoding的长度，也就是encoding所需要占用的内存。
- **headersize**：表示本元素entryX的首部长度，即prevlen和encoding两个字段所占用的内存之和 headersize= prevrawlensize + lensize

zipEntry用来解码压缩列表元素entryX，存储于zlentry结构体。

```
/* Return a struct with all information about an entry. */
void zipEntry(unsigned char *p, zlentry *e) {
    //p 为encoding的起始地址，即p指向列表元素
    ZIP_DECODE_PREVLEN(p, e->prevrawlensize, e->prevrawlen);
    ZIP_DECODE_LENGTH(p + e->prevrawlensize, e->encoding, e->lensize,
e->len);
    e->headersize = e->prevrawlensize + e->lensize;
    e->p = p;
}
```

ZIP_DECODE_PREVLEN 解析prevlen字段 --- 传入指针ptr 求prevlensize和prevlen的值

```
//传入指针ptr 求prevlensize和prevlen的值
#define ZIP_DECODE_PREVLEN(ptr, prevlensize, prevlen) do {
\
    ZIP_DECODE_PREVLENSIZE(ptr, prevlensize);
\
    if ((prevlensize) == 1) {
\
        (prevlen) = (ptr)[0];
\
    } else if ((prevlensize) == 5) {
\
        assert(sizeof((prevlen)) == 4);
\
        memcpy(&(prevlen), ((char*)(ptr)) + 1, 4);
\
        memrev32ifbe(&prevlen);
\
    }
\
} while(0);
```

ZIP_DECODE_LENGTH 用来解码encoding字段--传入指针ptr 求encoding 、lensize 、len

```
#define ZIP_DECODE_LENGTH(ptr, encoding, lensize, len) do {
\
    //ZIP_ENTRY_ENCODING求encoding
    ZIP_ENTRY_ENCODING((ptr), (encoding));
\
    if ((encoding) < ZIP_STR_MASK) {
\
        if ((encoding) == ZIP_STR_06B) {
\
            (lensize) = 1;
\
            (len) = (ptr)[0] & 0x3f;
\
        } else if ((encoding) == ZIP_STR_14B) {
\
            (lensize) = 2;
\
            (len) = (((ptr)[0] & 0x3f) << 8) | (ptr)[1];
\
        } else if ((encoding) == ZIP_STR_32B) {
\
            (lensize) = 5;
\
            (len) = ((ptr)[1] << 24) |
\
                    ((ptr)[2] << 16) |
\
                    ((ptr)[3] <<  8) |
\
                    ((ptr)[4]);
\
        } else {
\
            panic("Invalid string encoding 0x%02X", (encoding));
\
        }
\
    } else {
\
```

```
        (lensize) = 1;
\
        (len) = zipIntSize(encoding);
\
    }
\
} while(0);
```

zipRawEntryLength：返回p的指向的元素长度

```
/* Return the total number of bytes used by the entry pointed to by
'p'. */
unsigned int zipRawEntryLength(unsigned char *p) {
    unsigned int prevlensize, encoding, lensize, len;
    ZIP_DECODE_PREVLENSIZE(p, prevlensize);
    ZIP_DECODE_LENGTH(p + prevlensize, encoding, lensize, len);
    return prevlensize + lensize + len;
}
```

上文中已经讲到过字节数组只需要根据ptr[0]的前2位即可判断类型：00 10 10 表示字符串，而判断整数需要ptr[0]的前四位.

```
/* Return bytes needed to store integer encoded by 'encoding'. */
unsigned int zipIntSize(unsigned char encoding) {
    switch(encoding) {
    case ZIP_INT_8B:  return 1;
    case ZIP_INT_16B: return 2;
    case ZIP_INT_24B: return 3;
    case ZIP_INT_32B: return 4;
    case ZIP_INT_64B: return 8;
    }
    //0----12
    if (encoding >= ZIP_INT_IMM_MIN && encoding <= ZIP_INT_IMM_MAX)
        return 0; /* 4 bit immediate */
    panic("Invalid integer encoding 0x%02X", encoding);
    return 0;
}
```

zipStoreEntryEncoding 此函数传入encoding 和 length，把encoidng的值存入p 然后

返回编码prawlen的字节数，即prawlen所需要的内存

```
unsigned int zipStoreEntryEncoding(unsigned char *p, unsigned char
encoding, unsigned int rawlen) {
    unsigned char len = 1, buf[5];

    if (ZIP_IS_STR(encoding)) {
        /* Although encoding is given it may not be set for strings,
         * so we determine it here using the raw length. */
        if (rawlen <= 0x3f) { //长度小于等于63 编码为 00xx xxxx 00表
示ZIP_STR_06B编码 xxxxxx 表示长度length
            if (!p) return len;
            buf[0] = ZIP_STR_06B | rawlen;
        } else if (rawlen <= 0x3fff) { //长度小于16383  编码为 01XXXXXX
XXXXXXXX
            len += 1;
            if (!p) return len;
            buf[0] = ZIP_STR_14B | ((rawlen >> 8) & 0x3f);
            buf[1] = rawlen & 0xff;
        } else {  // 编码为 ZIP_STR_32B 10000000 XXXXXXXX XXXXXXXX
XXXXXXXX XXXXXXXX
            len += 4;
            if (!p) return len;
            buf[0] = ZIP_STR_32B;
            buf[1] = (rawlen >> 24) & 0xff;
            buf[2] = (rawlen >> 16) & 0xff;
            buf[3] = (rawlen >> 8) & 0xff;
            buf[4] = rawlen & 0xff;
        }
    } else {
        /* Implies integer encoding, so length is always 1. */
        //整型的时候 整数的长度 只需要一个字节就可以编码
        if (!p) return len;
        buf[0] = encoding;
    }

    /* Store this length at p. */
    memcpy(p,buf,len);
    return len;
}
```

```
//只用len >= ZIP_BIG_PREVLEN
/* Encode the length of the previous entry and write it to "p". This
only
 * uses the larger encoding (required in __ziplistCascadeUpdate). */
int zipStorePrevEntryLengthLarge(unsigned char *p, unsigned int len) {
    if (p != NULL) {
        p[0] = ZIP_BIG_PREVLEN;
        memcpy(p+1,&len,sizeof(len));
        memrev32ifbe(p+1);
    }
    return 1+sizeof(len);
}

//编码前一个元素length的写入到p 返回编码前一个元素的length所需空间即占用的内存字节
数
/* Encode the length of the previous entry and write it to "p". Return
the
 * number of bytes needed to encode this length if "p" is NULL. */
unsigned int zipStorePrevEntryLength(unsigned char *p, unsigned int
len) {
    if (p == NULL) {
        return (len < ZIP_BIG_PREVLEN) ? 1 : sizeof(len)+1;
    } else {
        if (len < ZIP_BIG_PREVLEN) {
            p[0] = len;
            return 1;
        } else {
            return zipStorePrevEntryLengthLarge(p,len);
        }
    }
}
```

zipRawEntryLength p指向元素的长度

```
/* Return the total number of bytes used by the entry pointed to by
'p'. */
unsigned int zipRawEntryLength(unsigned char *p) {
    unsigned int prevlensize, encoding, lensize, len;
    ZIP_DECODE_PREVLENSIZE(p, prevlensize);
    ZIP_DECODE_LENGTH(p + prevlensize, encoding, lensize, len);
    return prevlensize + lensize + len;
```

```
}
```

__ziplistInsert函数中nextdiff的

```
int zipPrevLenByteDiff(unsigned char *p, unsigned int len) {
    unsigned int prevlensize;
    //宏,展开之后根据p[0]处的值计算出prevlensize,如果p[0]<254,prevlensize为1,否则为5
    ZIP_DECODE_PREVLENSIZE(p, prevlensize);
    //zipStorePrevEntryLength函数如果第一个参数为NULL,则根据len字段计算需要的字节数,同理,len<254为1个字节,否则为5个字节
    return zipStorePrevEntryLength(NULL, len) - prevlensize;
}
```

如上函数计算nextdiff,可以看出,根据插入位置p当前保存prev_entry_len字段的字节数和即将插入的entry需要的字节数相减得出nextdiff.值有三种类型

0: 空间相等
4：需要更多空间
-4：空间富余

__ziplistInsert： 在p位置插入节点

```
/* Insert item at "p". */
unsigned char *__ziplistInsert(unsigned char *zl, unsigned char *p,
unsigned char *s, unsigned int slen) {
    /*
        zl：指向压缩列表
        p：指向元素s插入的位置，  要插入的元素s插入列表后 s是p的元素的前一个元素
        s：要插入元素
        slen：插入元素s的长度，即所占用的内存
    */

    //curlen 当前压缩列表的长度
    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. */
    //三种情况:
    /*
    1、当压缩列表为空，不存在前一个元素，即前一个元素的长度为0
    2、插入位置p元素存在，能根据p求出前一个元素的长度
    3、插入的位置为最后一个元素：即插入的元素成为压缩列表最后一个元素
    */
    if (p[0] != ZIP_END) {
        //插入的位置不在尾部
        ZIP_DECODE_PREVLEN(p, prevlensize, prevlen);
    } else {
        //插入的位置在尾部
        unsigned char *ptail = ZIPLIST_ENTRY_TAIL(zl);//指向最后一个元素
        if (ptail[0] != ZIP_END) {
            prevlen = zipRawEntryLength(ptail);//zipRawEntryLength 求节
点长度
        }
    }

    /* See if the entry can be encoded */
    //s 指向新节点数据的指针 slen为数据的长度
    /* 判断长度为entrylen的entry字符串能否转换为数值，转换结果保存在v中 编码方式
保存在encoding中 */
    if (zipTryEncoding(s,slen,&value,&encoding)) {
        /* 'encoding' is set to the appropriate integer encoding */
        reqlen = zipIntSize(encoding);
    } else {
        /* 'encoding' is untouched, however zipStoreEntryEncoding 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 += zipStorePrevEntryLength(NULL,prevlen); //求prevlen字段所占
```

```
用内存大小要么是1 要么是5
    reqlen += zipStoreEntryEncoding(NULL,encoding,slen);//求encoding字段所占用的内存大小

    /* 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. */
    int forcelarge = 0;

    /*
        zipPrevLenByteDiff求p指向节点prevlensize的变化 因为在p位置插入长度为reqlen字节之后
        p指向的节点prevlen=reqlen prevlensize就是reqlen的编码字节数 变化的值为0、4、-4

    */
    nextdiff = (p[0] != ZIP_END) ? zipPrevLenByteDiff(p,reqlen) : 0;
    //修复由于连锁更新造成的bug情况
    if (nextdiff == -4 && reqlen < 4) {
        nextdiff = 0;
        forcelarge = 1;
    }

    /* Store offset because a realloc may change the address of zl. */
    offset = p-zl;//zl压缩列表头到p的位置偏移量
    zl = ziplistResize(zl,curlen+reqlen+nextdiff);//重新分配 调整内存大小
    p = zl+offset;

    /* Apply memory move when necessary and update tail offset. */
    //非空列表插入
    if (p[0] != ZIP_END) {
        /* Subtract one because of the ZIP_END bytes */
        // 数据移动，根据nextdiff的值移动数据 例如nextdiff=4时 p-nextdiff 即从p后移4字节的位置
        // 开始移动curlen-offset-1+nextdiff长度内存数据，减一是zlend不需要移动。
        memmove(p+reqlen,p-nextdiff,curlen-offset-1+nextdiff);

        /* Encode this entry's raw length in the next entry. */
        //在p位置插入元素reqlen长度后，需要更新p位置的元素的prevlen=reqlen的值
        //写入p节点前一节点的信息长度（要插入节点的长度）
        if (forcelarge)
            //插入元素reqlen < 254，但是p的位置的元素的prevlen依然占用5字节
```

```
            zipStorePrevEntryLengthLarge(p+reqlen,reqlen);
        else
            zipStorePrevEntryLength(p+reqlen,reqlen);

        /* Update offset for tail */
        //更新ztail字段的值
        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.
*/
        zipEntry(p+reqlen, &tail);
        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. */
        //元素插入后成为列表最后一个元素  空列表插入，只更新尾节点偏移量
        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 */
    //考虑连锁更新
    if (nextdiff != 0) {
        offset = p-zl;
        zl = __ziplistCascadeUpdate(zl,p+reqlen);
        p = zl+offset;
    }

    /* Write the entry */
    // 写入前一节点长度信息
    p += zipStorePrevEntryLength(p,prevlen);
    // 写入节点编码与长度信息
    p += zipStoreEntryEncoding(p,encoding,slen);
     // 写入数据
    if (ZIP_IS_STR(encoding)) {
        memcpy(p,s,slen);
    } else {
```

```
        zipSaveInteger(p,value,encoding);
    }
    // 增加列表长度
    ZIPLIST_INCR_LENGTH(zl,1);
    return zl;
}
```

连锁更新

```
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) {
        // 解析当前节点信息
        zipEntry(p, &cur);
        // 当前节点总长
        rawlen = cur.headersize + cur.len;
        // 保存当前节点长度信息所需长度
        rawlensize = zipStorePrevEntryLength(NULL,rawlen);

        // 列表末尾，停止遍历
        if (p[rawlen] == ZIP_END) break;
        // 解析下一节点信息
        zipEntry(p+rawlen, &next);

        /* 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;
            // 下一节点因 前一节点长度信息 字段长度变更引发的自身长度变化大小
            extra = rawlensize-next.prevrawlensize;
            // 内存重新分配
```

```
            zl = ziplistResize(zl,curlen+extra);
            p = zl+offset;

            /* Current pointer and offset for next element. */
            np = p+rawlen;
            noffset = np-zl;

            // 如果下一节点不是尾节点，则需要更新 尾部节点偏移量
            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);
            zipStorePrevEntryLength(np,rawlen);


            p += rawlen;
            curlen += extra;
        } else {
            // 如果 next节点原本的 前一节点长度信息 字段长度可以容纳新插入节点的
长度信息，则直接写入并退出遍历
            if (next.prevrawlensize > rawlensize) {
                /* This would result in shrinking, which we want to
avoid.
                 * So, set "rawlen" in the available bytes. */
                zipStorePrevEntryLengthLarge(p+rawlen,rawlen);
            } else {
                zipStorePrevEntryLength(p+rawlen,rawlen);
            }

            /* Stop here, as the raw length of "next" has not changed.
*/
            break;
        }
    }
    return zl;
}
```

ziplistDelete:

```
unsigned char *ziplistDelete(unsigned char *zl, unsigned char **p) {
    size_t offset = *p-zl;
    zl = __ziplistDelete(zl,*p,1);

    /* Store pointer to current element in p, because ziplistDelete
will
     * do a realloc which might result in a different "zl"-pointer.
     * When the delete direction is back to front, we might delete the
last
     * entry and end up with "p" pointing to ZIP_END, so check this. */
    *p = zl+offset;
    return zl;
}

/* Delete a range of entries from the ziplist. */
unsigned char *ziplistDeleteRange(unsigned char *zl, int index,
unsigned int num) {
    unsigned char *p = ziplistIndex(zl,index);
    return (p == NULL) ? zl : __ziplistDelete(zl,p,num);
}
```

__ziplistDelete:因为可能会触发连锁更新，所以删除操作最坏复杂度为O(n^2)，平均复杂度为O(n)

```
/* Delete "num" entries, starting at "p". Returns pointer to the
ziplist. */
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;

    //解码第一个删除的元素
    zipEntry(p, &first);
```

```
    //遍历所有待删除的删除的元素
    for (i = 0; p[0] != ZIP_END && i < num; i++) {
        p += zipRawEntryLength(p);
        deleted++;
    }

    //待删除所有元素的总长度
    totlen = p-first.p; /* Bytes taken by the element(s) to delete. */
    if (totlen > 0) {
        if (p[0] != ZIP_END) {//如果p指向zlend 不需要进行数据复制
            /* 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. */
            // 计算元素entryN长度的变化量 删除后p指向元素的prelen信息有所变化,
            // 导致prevlensize 大小发生变化0 4 -4三种情况
            nextdiff = zipPrevLenByteDiff(p,first.prevrawlen);

            /* Note that there is always space when p jumps backward:
if
             * the new previous entry is large, one of the deleted
elements
             * had a 5 bytes prevlen header, so there is for sure at
least
             * 5 bytes free and we need just 4. */
            /*根据prevlen信息变化移动p指针*/
            p -= nextdiff;
            //p指针移动后把prevrawlen信息存贮到p处
            zipStorePrevEntryLength(p,first.prevrawlen);

            /* Update offset for tail */
            //更新ztail信息
            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. */
            //解码p指向的元素
            zipEntry(p, &tail);
            /* 如果p节点不是尾节点，则尾节点偏移量需要加上nextdiff的变更量
               因为尾节点偏移量是指列表首地址到尾节点首地址的距离
               p节点的 【前一节点长度信息】 字段的长度变化只影响它字段之后的信息
地址。
               p节点为尾节点时，为节点首地址在【前一节点长度信息】字段前边，所以不
受影响。*/
            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 */
            //数据复制
            memmove(first.p,p,
                intrev32ifbe(ZIPLIST_BYTES(zl))-(p-zl)-1);
        } else {
            /* The entire tail was deleted. No need to move memory. */
            // 一直删除到尾节点，不需要变更中间节点，只需要调整下尾节点偏移量
            ZIPLIST_TAIL_OFFSET(zl) =
                intrev32ifbe((first.p-zl)-first.prevrawlen);
        }

        /* Resize and update length */
        offset = first.p-zl;
        // 重新分配内存大小
        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 */
        // 如果最后一个被删除节点的下一节点的【前一个节点长度信息】字段长度 需要变
更，则可能会触发连锁更新
        if (nextdiff != 0)
            zl = __ziplistCascadeUpdate(zl,p);
    }
    return zl;
}
```