链滴

压缩列表的源码实现

作者：zeekling

原文链接：https://ld246.com/article/1670598813861

来源网站：链滴

简介

压缩列表ziplist本质上就是一个字节数组，是Redis为了节约内存而设计的一种线性数据结构，可以包多个元素，每个元素可以是一个字节数组或一个整数。

Redis的有序集合、散列和列表都直接或者间接使用了压缩列表。当有序集合或散列表的元素个数比少，且元素都是短字符串时，Redis便使用压缩列表作为其底层数据存储结构。列表使用快速链表（quicklist）数据结构存储，而快速链表就是双向链表与压缩列表的组合。

ziplist 压缩列表是一个特殊编码的双端链表（内存上连续），为了尽可能节省内存而设计的。ziplist 以存储字符串或者整数值，其中整数被编码保存为实际的整数，而不是字符数组。ziplist 支持 O(1) 时间复杂度在列表的两端进行 push 和 pop 操作。然而因为这些操作都需要对整个 ziplist 进行内存分配（因为是一块连续的内存），所以操作的实际复杂度和 ziplist 占用的内存大小有关。在 7.0 版本，ziplist 已经全面被 listpack 替换了（主要是因为连锁更新较影响性能）

压缩列表的存储结构

Redis使用字节数组表示一个压缩列表，压缩列表结构如下所示：

<zlbytes> <zltail> <zllen> <entry> <entry> ... <entry> <zlend>

压缩列表各字段的含义如下：

● zlbytes：是一个 32 位无符号整数（4 bytes），记录整个 ziplist 占用的内存字节数，包含 4 个字的 zlbytes 本身。

● zltail：是一个 32 位无符号整数（4 bytes），记录 ziplist 到尾节点的位置偏移量。通过这个偏移我们可以直接定位到表尾节点，例如进行表尾的 pop 操作，不然得完整遍历 ziplist。

● zllen：是一个 16 位无符号整数（2 bytes），记录 ziplist 里的节点数量。由于它设计只用 2 个字进行存储，2 字节实际最大可以表示为 2^16 - 1 即: 65535。当数字小于它时，则 zllen 的值就是实的节点数量（O(1) 时间复杂度），也就是注释里的 2^16 - 2 的含义。否则当 zllen 值为 65535 时即 ^16-1，用它作为一个标识，表示需要完整遍历整个压缩列表 O(N) 时间复杂度才能计算出真实的节数量。所以 ziplist 不适合存储过多元素（遍历计算节点数量开销很大，且我们假设它只用于元素数量少的场景）。

● entryX：压缩列表存储的元素，可以是字节数组或者整数，长度不限。entry的编码结构将在后面细介绍。

● zlend： 是一个 8 位无符号整数（1 byte），是一个特殊的标志位来标记压缩列表的结尾，0xFF(进制表示为: 255)。其它正常节点不会有以这个字节开头的，在遍历 ziplist 的时候通过这个标记来判是否遍历结束。

元素的存储结构

压缩列表元素的存储结构如下所示：

<prevlen> <encoding> <entry-data>

每一个 ziplist entry 压缩列表节点在实际的节点数据之前都会包含两部分元数据，也叫 entry heade 。

1. prevlen: 前置节点的字节长度，以支持我们从后往前遍历（通过指针偏移量定位前一个节点）

2. encoding: 当前节点 entry-data 节点数据部分的类型和编码，例如存储的是整数还是字符串，类下还会细分多种编码。

有时候节点可以不用有 entry-data，可以在 encoding 部分直接存储节点数据。例如一些小整数，以直接在 encoding 部分用几位来存储表示，对每一位都物尽其用。此时元素的存储结构为：

<prevlen> <encoding>

当前节点的前节点字节长度，prevlen 的编码方式如下（同时我们将存储 prevlen 所需的字节数为 prvlensize，即下面的 1 或者 5 字节）

- 如果前节点的字节长度 小于 254 字节，那么 prevlen 使用 1 个字节来保存它，一个 8 位无符号的数。

<prevlen from 0 to 253> <encoding> <entry>

- prevlen：前一个entry长度，介于(0,253)
- encoding:当前节点的实际数据类型以及长度
- entry:当前节点的实际数据
- 如果前节点的字节长度 大于等于254 字节，那么 prevlen 使用 5 个字节来保存它：

0xFE <4 bytes unsigned little endian prevlen> <encoding> <entry>

- 0xFE: zlend 标识，值为 254（1 字节）
- <4 bytes unsigned little endian prevlen>: 当前节点的实际长度（4 字节）
- <encoding>: 当前节点的实际数据类型以及长度
- <entry>: 当前节点的实际数据

压缩列表元素编码：

encoding编码 ontext类型	encoding 长度	
00 pppppp 度为63字节，pppppp的长度为6	1 bytes	value最大
01 pppppp qqqqqqqq alue的最大长度16383字节	2 bytes	
10000000 qqqqqqqq rrrrrrrr ssssssss ttttttt alue的最大长度2^32-1。第一个字节的6个低位不被使用，并且被设置为零。	5 bytes	
11000000 s)	3 bytes	int 16 (2 byt
11010000 s)	5 bytes	int 32 (4 byt
11100000 s)	9 bytes	int 64 (8 byt
11110000 tes)	4 bytes	24位整数(3 b
11111110 e)	2 bytes	8位整数(1 by
1111xxxx 数	1 byte	0-12的无符号

11111111	1 byte	ziplist结束

Redis 常见的encoding：

```
#define ZIP_STR_MASK 0xc0
#define ZIP_INT_MASK 0x30
#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)
#define ZIP_INT_32B (0xc0 | 1<<4)
#define ZIP_INT_64B (0xc0 | 2<<4)
#define ZIP_INT_24B (0xc0 | 3<<4)
#define ZIP_INT_8B 0xfe
```

解码结构体

对于压缩列表中的任意元素，获取前一个元素的长度、判断存储的数据类型、获取数据内容等都需要过复杂的解码运算。解码后的结果应该被缓存起来，为此定义了结构体zlentry，用于表示解码后的压列表元素，单纯的用来临时存储解码之后的元素信息。

```
typedef struct zlentry {
    unsigned int prevrawlensize;
    unsigned int prevrawlen;
    unsigned int lensize;
    unsigned int len;
    unsigned int headersize;
    unsigned char encoding;
    unsigned char *p;
} zlentry;
```

- prevrawlensize: 存储下面 prevrawlen 所需要的字节数
- prevrawlen: 存储前一个节点的字节长度
- len: 存储当前节点的字节长度
- headersize: prevrawlensize + lensize 当前节点的头部字节，其实是 prevlen + encoding 两项用的字节数
- encoding: 存储当前节点的数据编码格式
- p: 指向当前节点开头第一个字节的指针

函数zipEntry用来解码压缩列表的元素，填充为zlentry结构体。

```
static inline void zipEntry(unsigned char *p, zlentry *e) {
    ZIP_DECODE_PREVLEN(p, e->prevrawlensize, e->prevrawlen);
    ZIP_ENTRY_ENCODING(p + e->prevrawlensize, e->encoding);
    ZIP_DECODE_LENGTH(p + e->prevrawlensize, e->encoding, e->lensize, e->len);
    assert(e->lensize != 0);
    e->headersize = e->prevrawlensize + e->lensize;
    e->p = p;
}
```

解码主要分为下面几个步骤：

解码前节点长度

根据 p 目前的指针，获取 entry 的 prevlen 的值；

- 如果prevlen一个字节编码，对应字节 (ptr)[0] 的值就是 prevlen。
- 如果prevlen五个字节编码，具体的 prevlen 是存储在后四个字节，后四个字节进行位运算获得实的 prevlen

```
#define ZIP_DECODE_PREVLEN(ptr, prevlensize, prevlen) do {                     \
    ZIP_DECODE_PREVLENSIZE(ptr, prevlensize);                                  \
    if ((prevlensize) == 1) {                                                  \
        (prevlen) = (ptr)[0];                                                  \
    } else { /* prevlensize == 5 */                                           \
        (prevlen) = ((ptr)[4] << 24) |                                         \
                    ((ptr)[3] << 16) |                                         \
                    ((ptr)[2] <<  8) |                                         \
                    ((ptr)[1]);                                                \
    }                                                                          \
} while(0)
```

根据prevlensize的长度判断prevlen的长度，如果(ptr)[0] < ZIP_BIG_PREVLEN时（ZIP_BIG_PREVLN=254），则prevlen长度为1，否则长度为5。

```
#define ZIP_DECODE_PREVLENSIZE(ptr, prevlensize) do {                          \
    if ((ptr)[0] < ZIP_BIG_PREVLEN) {                                          \
        (prevlensize) = 1;                                                     \
    } else {                                                                   \
        (prevlensize) = 5;                                                     \
    }                                                                          \
  } while(0)
```

解码encoding

p+prevrawlensize 位置的第一个字节，获取 entry 当前的 encoding 属性，保存在 encoding 变量时间复杂度 O(1)。

```
#define ZIP_ENTRY_ENCODING(ptr, encoding) do {  \
    (encoding) = ((ptr)[0]); \
    if ((encoding) < ZIP_STR_MASK) (encoding) &= ZIP_STR_MASK; \
} while(0)
```

解码长度

p+prevrawlensize 根据 encoding 获取 entry 的 len 相关属性。 ptr[0]<11000000说明是字节数组前两个比特为字节数组编码类型

进制转换：echo "ibase=16;obase=2;C0" | bc

```
#define ZIP_DECODE_LENGTH(ptr, encoding, lensize, len) do {                    \
    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) = ((uint32_t)(ptr)[1] << 24) |                        \
                ((uint32_t)(ptr)[2] << 16) |                        \
                ((uint32_t)(ptr)[3] <<  8) |                        \
                ((uint32_t)(ptr)[4]);                               \
    } else {                                                        \
        (lensize) = 0;                                              \
        (len) = 0;                                                  \
    }                                                               \
} else {                                                            \
    (lensize) = 1;                                                  \
    if ((encoding) == ZIP_INT_8B)  (len) = 1;                       \
    else if ((encoding) == ZIP_INT_16B) (len) = 2;                  \
    else if ((encoding) == ZIP_INT_24B) (len) = 3;                  \
    else if ((encoding) == ZIP_INT_32B) (len) = 4;                  \
    else if ((encoding) == ZIP_INT_64B) (len) = 8;                  \
    else if (encoding >= ZIP_INT_IMM_MIN && encoding <= ZIP_INT_IMM_MAX)   \
        (len) = 0;                                                  \
    else                                                            \
        (lensize) = (len) = 0;                                      \
}                                                                   \
} while(0)
```

基本操作

主要介绍压缩列表的基本操作，包括创建压缩列表，遍历元素，插入元素，删除元素，修改元素等。

创建压缩列表

创建一个空的压缩列表:只对 lbytes、zltail、zllen、zlend四个字段进行初始化。初始化过程如下：

- 计算空ziplist的长度并且申请内存， zlbytes和zltail的类型是32位无符号整数，zllen是16位无符整数，所以总长度为：zlbytes(4) + zltail(4) + zllen(2) = 10 bytes
- 将总字节数写入内存。zl 既为 ziplist 的起始地址，其中值又负责记录 ziplist 的总字节长度，zlbytes 编码存储固定 4 字节，也就代表了一个 ziplist 总字节最大为为 (2^32)-1 字节。
- 将到尾节点的偏移量写进内存，因为是刚初始化的 ziplist，偏移量其实就是 HEADER_SIZE 值，此它刚好指向 zlend，因此能够以 O(1) 时间复杂度快速在尾部进行 push 或 pop 操作。
- 写入节点数量：0
- 最后一个字节设置为 ZIP_END，标识 ziplist 结尾。

实现代码如下：

```
unsigned char *ziplistNew(void) {
    unsigned int bytes = ZIPLIST_HEADER_SIZE+ZIPLIST_END_SIZE;
    unsigned char *zl = zmalloc(bytes);
    ZIPLIST_BYTES(zl) = intrev32ifbe(bytes);
```

```
    ZIPLIST_TAIL_OFFSET(zl) = intrev32ifbe(ZIPLIST_HEADER_SIZE);
    ZIPLIST_LENGTH(zl) = 0;
    zl[bytes-1] = ZIP_END;
    return zl;
}
```

插入元素

压缩列表实现函数如下,其中:

- zl：压缩列表。
- p: 元素插入位置
- s: 插入元素内容
- slen: 元素数据长度。

```
unsigned char *__ziplistInsert(unsigned char *zl, unsigned char *p, unsigned char *s, unsigned
nt slen)
```

插入元素可以简要分为3个步骤：① 将元素内容编码；② 重新分配空间；③ 复制数据。

编码

编码就是计算前节点的prelen字段，encoding字段和content字段的内容。计算prelen的前提条件就明确元素的插入位置。

元素的插入位置主要包含两种场景:

- 元素插入到中间位置。
- 元素插入到末尾。

场景一：元素插入到中间位置

当插入到ziplist的中间节点时,解码插入节点p的prevlen（函数ZIP_DECODE_PREVLEN）。

场景二：元素插入到末尾

当插入到ziplist的尾部时，通过zltail计算出ziplist的最后一个节点，再计算prevlen。首先我们应当获最后一个节点。

可以通过zltail获取最后一个节点的内容。zl偏移zltail的偏移量就可以获取最后一个节点的指针。

```
#define ZIPLIST_ENTRY_TAIL(zl)  ((zl)+intrev32ifbe(ZIPLIST_TAIL_OFFSET(zl)))
```

取出最后一个节点的长度，作为新插入节点p的prevlen,最后一个节点的prevlen是节点headersize和节点内容长度之和。

```
static inline unsigned int zipRawEntryLengthSafe(unsigned char* zl, size_t zlbytes, unsigned c
ar *p) {
    zlentry e;
    assert(zipEntrySafe(zl, zlbytes, p, &e, 0));
    return e.headersize + e.len;
```

```
}
```

元素编码

编码时尝试将输入字符串转为整数：若可以转为整数，则按照压缩列表整数类型编码存储，reqlen根encoding确定保存节点值需要的字节数；

若不可以转为整数，则按照字节数组方式存储，reqlen为字符串的长度。

reqlen字段为存储当前元素需要的空间大小，所以由prevlen占用空间、当前节点的encoding和lengt、当前节点值占用的空间三部分之和构成。

计算公式：reqlen = prevlenSize + encodingSize + dataSize

```
if (zipTryEncoding(s,slen,&value,&encoding)) {
  reqlen = zipIntSize(encoding);
} else {
  reqlen = slen;
}
reqlen += zipStorePrevEntryLength(NULL,prevlen);
reqlen += zipStoreEntryEncoding(NULL,encoding,slen);
```

重新分配空间

当元素不是在尾部插入时，我们需要确保插入位置的后一个节点有足够的空间来保存新插入节点的长，即保证后节点的prevlen足够，例如，

AC插入后变成ABC，我们需要跟新C节点的prevlen信息。计算方法如下：

```
nextdiff = (p[0] != ZIP_END) ? zipPrevLenByteDiff(p,reqlen) :0;
```

nextdiff 的值代表的含义如下:

- 0： 空间相等（刚好，不需要扩展）；
- 4： 需要更多空间（需要扩展）。
- -4 空间富余（可以缩容）

函数zipPrevLenByteDiff实现如下：

```
int zipPrevLenByteDiff(unsigned char *p, unsigned int len) {
    unsigned int prevlensize;
    ZIP_DECODE_PREVLENSIZE(p, prevlensize);
    return zipStorePrevEntryLength(NULL, len) - prevlensize;
}
```

数据复制

内存空间分配使用memmove函数实现：

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

当空间富裕且下一个节点的pevlen使用一个字节存储长度时，需要防止出现大量的连锁跟新，后一个点的prevlen不能直接缩成1，

还是需要使用4个字节保存。保存新插入节点的长度到后一节点的代码如下：

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

维护 zltail, 将新插入的节点长度算到zltail里面，如果 nextdiff == 4 > 0，说明后个节点的 prevSize 其实变大了的，

也需要把这部分增加落实到 zltail 算在偏移量里，这样才能真的对齐表尾节点

```
ZIPLIST_TAIL_OFFSET(zl) = intrev32ifbe(intrev32ifbe(ZIPLIST_TAIL_OFFSET(zl))+reqlen);
assert(zipEntrySafe(zl, newlen, p+reqlen, &tail, 1));
if (p[reqlen+tail.headersize+tail.len] != ZIP_END) {
   ZIPLIST_TAIL_OFFSET(zl) = intrev32ifbe(intrev32ifbe(ZIPLIST_TAIL_OFFSET(zl))+nextdiff);
}
```

删除元素

压缩列表的函数定定义如下，返回值为删除元素之后压缩列表的首地址，接口入参如下：

- zl: 压缩列表的首地址。
- p: 待删除的元素的首地址。

```
unsigned char *ziplistDelete(unsigned char *zl, unsigned char **p)
```

函数ziplistDelete只是简单调用了__ziplistDelete函数实现，其中num表示需要删除的元素数目，包元素p。

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

删除元素需要是三个步骤：

1. 计算待删除元素的总长度。
2. 数据复制。
3. 重新分配空间。

1. 计算待删除元素的总长度

计算长度的函数zipRawEntryLengthSafe之前已经讲过。将需要删除的元素解密之后，计算需要删除第num个元素的长度。

```
zipEntry(p, &first);
for (i = 0; p[0] != ZIP_END && i < num; i++) {
    p += zipRawEntryLengthSafe(zl, zlbytes, p);
    deleted++;
```

```
}
totlen = p-first.p;
```

2. 数据复制

在上一步之后，first和p之间的数据就是需要删除的，其中first指向第一个需要删除的节点，p指向最一个需要保留

第一个节点或者列表尾。

删除节点为中间节点

当删除节点为中间节点时，在删除当前节点之后，后面节点的prevlen存储可能不够。p指向的后面节需要判断是否有

足够的prevlen是否够。因为是删除节点，包含删除 first 节点，这里删除的空间是肯定够 p 节点 prev enSize 扩展的

将 p 向后移动 nextdiff 差值的长度，减少需要删除的内存，用来扩展第一个节点（都删除后的）的 pr vlen。

```
nextdiff = zipPrevLenByteDiff(p,first.prevrawlen);
p -= nextdiff;
```

将 first 的前一个节点的长度编码扩展到 p 当前的位置。

```
zipStorePrevEntryLength(p,first.prevrawlen);
```

更新列表末尾的偏移量，原本的 减去 所有被删除的内存。

```
set_tail = intrev32ifbe(ZIPLIST_TAIL_OFFSET(zl))-totlen;
```

如果被删除节点后有多于一个节点，那么需要将 nextdiff 也计算到表尾偏移量中。因为当前 p 指向不是尾节点，

因此要加上 nextdiff 才能让表尾偏移量正确。

```
assert(zipEntrySafe(zl, zlbytes, p, &tail, 1));
if (p[tail.headersize+tail.len] != ZIP_END) {
   set_tail = set_tail + nextdiff;
}
```

数据复制： 末尾向前面移动数据，覆盖被删除节点。

```
size_t bytes_to_move = zlbytes-(p-zl)-1;
memmove(first.p,p,bytes_to_move);
```

删除节点为末尾节点

p[0] == ZIP_END 到达末尾，说明后面其实没有节点，无需移动内存,更新尾节点偏移量到前一个节的地址，因为此时

first 前一个节点是尾节点。

```
set_tail = (first.p-zl)-first.prevrawlen;
```

3. 重新分配空间

重新分配空间与插入元素逻辑相似，代码如下：

```
offset = first.p-zl;
zlbytes -= totlen - nextdiff;
zl = ziplistResize(zl, zlbytes);
p = zl+offset;
```

遍历元素

压缩列表遍历分为前向(从头到尾)和后向遍历(从尾到头)。前向遍历的API定义如下：

- zl: 压缩列表的首地址。
- p: 为需要查找的字符串的首地址。
- 返回p的前一个节点。

```
unsigned char *ziplistPrev(unsigned char *zl, unsigned char *p)
```

后向遍历的API定义如下：

```
unsigned char *ziplistNext(unsigned char *zl, unsigned char *p)
```

前向遍历

- 如果 p[0]==ZIP_END，则说明前一个节点是压缩列表的尾节点。
- 如果 p == ZIPLIST_ENTRY_HEAD(zl) 说明p是压缩列表的第一个节点，前一个节点为NULL。
- 如果是查找的是中间节点，需要解析出 prevlen 往前进行偏移，然后解析出对应节点。

```
unsigned char *ziplistPrev(unsigned char *zl, unsigned char *p) {
    unsigned int prevlensize, prevlen = 0;
    if (p[0] == ZIP_END) {
        p = ZIPLIST_ENTRY_TAIL(zl);
        return (p[0] == ZIP_END) ? NULL : p;
    } else if (p == ZIPLIST_ENTRY_HEAD(zl)) {
        return NULL;
    } else {
        ZIP_DECODE_PREVLEN(p, prevlensize, prevlen);
        assert(prevlen > 0);
        p-=prevlen;
        size_t zlbytes = intrev32ifbe(ZIPLIST_BYTES(zl));
        zipAssertValidEntry(zl, zlbytes, p);
        return p;
    }
}
```

后向遍历

- 如果 p[0] == ZIP_END，则表示当前ziplist为空，范围NULL。
- 在 p 当前节点的基础上，往后偏移 p 节点的字节长度，即指向下一个节点。如果之后的 p 指向 ZIP

END 说明已经达到 ziplist 尾部，没有下一个节点

```
unsigned char *ziplistNext(unsigned char *zl, unsigned char *p) {
    ((void) zl);
    size_t zlbytes = intrev32ifbe(ZIPLIST_BYTES(zl));

    if (p[0] == ZIP_END) {
        return NULL;
    }

    p += zipRawEntryLength(p);
    if (p[0] == ZIP_END) {
        return NULL;
    }

    zipAssertValidEntry(zl, zlbytes, p);
    return p;
}
```