06-从ziplist到quicklist，再到listpack的启发

你好，我是蒋德钧。

在前面的第4讲，我介绍Redis优化设计数据结构来提升内存利用率的时候，提到可以使用压缩列表（ziplist）来保存数据。所以现在你应该也知道，ziplist的最大特点，就是它被设计成一种内存紧凑型的数据结构，占用一块连续的内存空间，以达到节省内存的目的。

但是，**在计算机系统中，任何一个设计都是有利有弊的**。对于ziplist来说，这个道理同样成立。

虽然ziplist节省了内存开销，可它也存在两个设计代价：一是不能保存过多的元素，否则访问性能会降低；二是不能保存过大的元素，否则容易导致内存重新分配，甚至可能引发连锁更新的问题。所谓的连锁更新，简单来说，就是ziplist中的每一项都要被重新分配内存空间，造成ziplist的性能降低。

因此，针对ziplist在设计上的不足，Redis代码在开发演进的过程中，新增设计了两种数据结构：**quicklist和listpack**。这两种数据结构的设计目标，就是尽可能地保持ziplist节省内存的优势，同时避免ziplist潜在的性能下降问题。

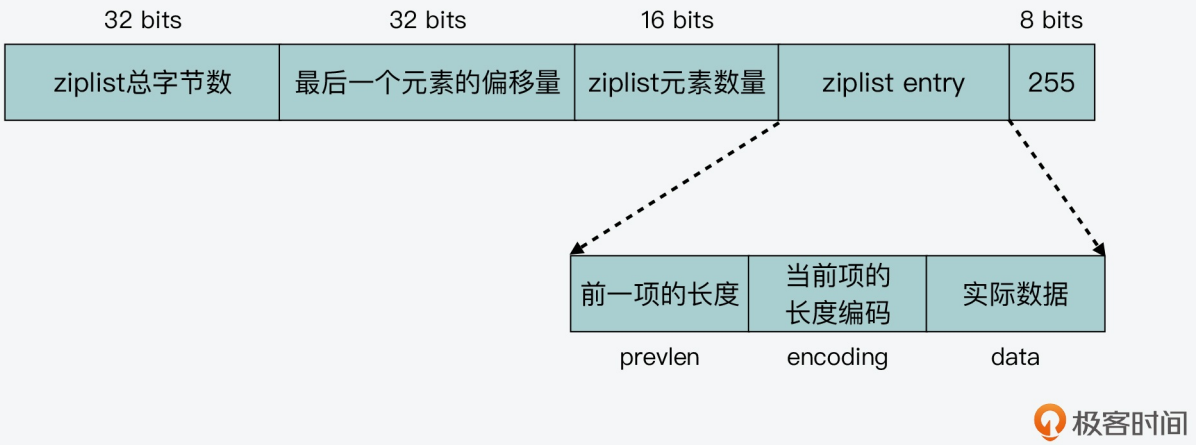
今天这节课，我就来给你详细介绍下quicklist和listpack的设计思想和实现思路，不过在具体讲解这两种数据结构之前，我想先带你来了解下为什么ziplist的设计会存在缺陷。这样一来，你在学习quicklist和listpack时，可以和ziplist的设计进行对比，进一步就能更加容易地掌握quicklist和listpack的设计考虑了。

而且，ziplist和quicklist的区别，也是经常被问到的面试题，而listpack数据结构因为比较新，你对它的设计实现可能了解得并不多。那在学完了这节课之后，你其实就可以很轻松地应对这三种数据结构的使用问题了。此外，你还可以从这三种数据结构的逐步优化设计中，学习到Redis数据结构在内存开销和访问性能之间，采取的设计取舍思想。如果你需要开发高效的数据结构，你就可以把这种设计思想应用起来。

好，那么接下来，我们就先来了解下ziplist在设计与实现上存在的缺陷。

ziplist的不足

你已经知道，一个ziplist数据结构在内存中的布局，就是一块连续的内存空间。这块空间的起始部分是大小固定的10字节元数据，其中记录了ziplist的总字节数、最后一个元素的偏移量以及列表元素的数量，而这10字节后面的内存空间则保存了实际的列表数据。在ziplist的最后部分，是一个1字节的标识（固定为255），用来表示ziplist的结束，如下图所示：



不过，虽然ziplist通过紧凑的内存布局来保存数据，节省了内存空间，但是ziplist也面临着随之而来的两个不足：查找复杂度高和潜在的连锁更新风险。那么下面，我们就分别来了解下这两个问题。

查找复杂度高

因为ziplist头尾元数据的大小是固定的，并且在ziplist头部记录了最后一个元素的位置，所以，当在ziplist中查找第一个或最后一个元素的时候，就可以很快找到。

但问题是，当要查找列表中间的元素时，ziplist就得从列表头或列表尾遍历才行。而当ziplist保存的元素过多时，查找中间数据的复杂度就增加了。更糟糕的是，如果ziplist里面保存的是字符串，ziplist在查找某个元素时，还需要逐一判断元素的每个字符，这样又进一步增加了复杂度。

也正因为如此，我们在使用ziplist保存Hash或Sorted Set数据时，都会在redis.conf文件中，通过hash-max-ziplist-entries和zset-max-ziplist-entries两个参数，来控制保存在ziplist中的元素个数。

不仅如此，除了查找复杂度高以外，ziplist在插入元素时，如果内存空间不够了，ziplist还需要重新分配一块连续的内存空间，而这还会进一步引发连锁更新的问题。

连锁更新风险

我们知道，因为ziplist必须使用一块连续的内存空间来保存数据，所以当新插入一个元素时，ziplist就需要计算其所需的空间大小，并申请相应的内存空间。这一系列操作，我们可以从ziplist的元素插入函数__ziplistInsert中看到。

__ziplistInsert函数首先会计算获得当前ziplist的长度，这个步骤通过ZIPLIST_BYTES宏定义就可以完成，如下所示。同时，该函数还声明了reqlen变量，用于记录插入元素后所需的新增空间大小。

```
//获取当前ziplist长度curlen；声明reqlen变量，用来记录新插入元素所需的长度
size_t curlen = intrev32ifbe(ZIPLIST_BYTES(zl)), reqlen;
```

然后，**__ziplistInsert函数会判断当前要插入的位置是否是列表末尾**。如果不是末尾，那么就需要获取位于当前插入位置的元素的prevlen和prevlensize。这部分代码如下所示：

```
//如果插入的位置不是ziplist末尾，则获取前一项长度
   if (p[0] != ZIP_END) {
   ZIP_DECODE_PREVLEN(p, prevlensize, prevlen);
   } else {
      …
   }
```

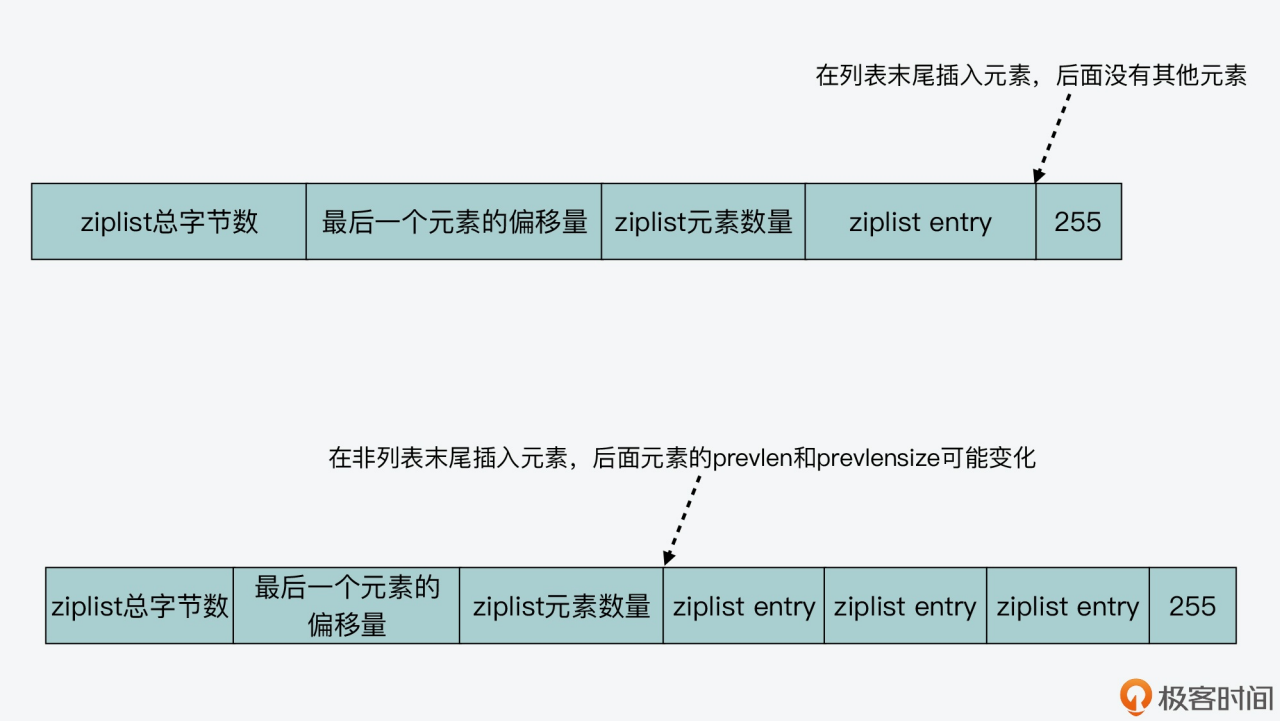
实际上，在ziplist中，每一个元素都会记录其**前一项的长度，也就是prevlen**。然后，为了节省内存开销，ziplist会使用不同的空间记录prevlen，这个**prevlen空间大小就是prevlensize**。

举个简单的例子，当在一个元素A前插入一个新的元素B时，A的prevlen和prevlensize都要根据B的长度进

行相应的变化。

那么现在，我们假设A的prevlen原本只占用1字节（也就是prevlensize等于1），而能记录的前一项长度最大为253字节。此时，如果B的长度超过了253字节，A的prevlen就需要使用5个字节来记录（prevlen具体的编码方式，你可以复习回顾下第4讲），这样就需要申请额外的4字节空间了。不过，如果元素B的插入位置是列表末尾，那么插入元素B时，我们就不用考虑后面元素的prevlen了。

我画了下面这张图，以便于你理解数据插入过程对插入位置元素的影响。



因此，为了保证ziplist有足够的内存空间，来保存插入元素以及插入位置元素的prevlen信息，**__ziplistInsert函数在获得插入位置元素的prevlen和prevlensize后，紧接着就会计算插入元素的长度**。

现在我们已知，一个ziplist元素包括了prevlen、encoding和实际数据data三个部分。所以，在计算插入元素的所需空间时，__ziplistInsert函数也会分别计算这三个部分的长度。这个计算过程一共可以分成四步来完成。

- 第一步，计算实际插入元素的长度。

首先你要知道，这个计算过程和插入元素是整数还是字符串有关。__ziplistInsert函数会先调用zipTryEncoding函数，这个函数会判断插入元素是否为整数。如果是整数，就按照不同的整数大小，计算encoding和实际数据data各自所需的空间；如果是字符串，那么就先把字符串长度记录为所需的新增空间大小。这一过程的代码如下所示：

```
if (zipTryEncoding(s,slen,&value,&encoding)) {
        reqlen = zipIntSize(encoding);
    } else {
        reqlen = slen;
    }
```

- 第二步，调用zipStorePrevEntryLength函数，将插入位置元素的prevlen也计算到所需空间中。

这是因为在插入元素后，__ziplistInsert函数可能要为插入位置的元素分配新增空间。这部分代码如下所示：

```
reqlen += zipStorePrevEntryLength(NULL,prevlen);
```

- 第三步，调用zipStoreEntryEncoding函数，根据字符串的长度，计算相应encoding的大小。

在刚才的第一步中，__ziplistInsert函数对于字符串数据，只是记录了字符串本身的长度，所以在第三步中，__ziplistInsert函数还会调用zipStoreEntryEncoding函数，根据字符串的长度来计算相应的encoding大小，如下所示：

```
reqlen += zipStoreEntryEncoding(NULL,encoding,slen);
```

好了，到这里，__ziplistInsert函数就已经在reqlen变量中，记录了插入元素的prevlen长度、encoding大小，以及实际数据data的长度。这样一来，插入元素的整体长度就有了，这也是插入位置元素的prevlen所要记录的大小。

- 第四步，调用zipPrevLenByteDiff函数，判断插入位置元素的prevlen和实际所需的prevlen大小。

最后，__ziplistInsert函数会调用zipPrevLenByteDiff函数，用来判断插入位置元素的prevlen和实际所需的prevlen，这两者间的大小差别。这部分代码如下所示，prevlen的大小差别是使用nextdiff来记录的：

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

那么在这里，如果nextdiff大于0，就表明插入位置元素的空间不够，需要新增nextdiff大小的空间，以便能保存新的prevlen。然后，**__ziplistInsert函数在新增空间时，就会调用ziplistResize函数，来重新分配ziplist所需的空间**。

ziplistResize函数接收的参数分别是待重新分配的ziplist和重新分配的空间大小。而__ziplistInsert函数传入的重新分配大小的参数，是三个长度之和。

那么是哪三个长度之和呢？

这三个长度分别是ziplist现有大小（curlen）、待插入元素自身所需的新增空间（reqlen），以及插入位置元素prevlen所需的新增空间（nextdiff）。下面的代码显示了ziplistResize函数的调用和参数传递逻辑：

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

进一步，那么ziplistResize函数在获得三个长度总和之后，具体是如何扩容呢？

我们可以进一步看下ziplistResize函数的实现，这个函数会调用**zrealloc函数**，来完成空间的重新分配，而重新分配的空间大小就是由**传入参数len**决定的。这样，我们就了解到了ziplistResize函数涉及到内存分配操作，因此如果我们往ziplist频繁插入过多数据的话，就可能引起多次内存分配，从而会对Redis性能造成影响。

下面的代码显示了ziplistResize函数的部分实现，你可以看下。

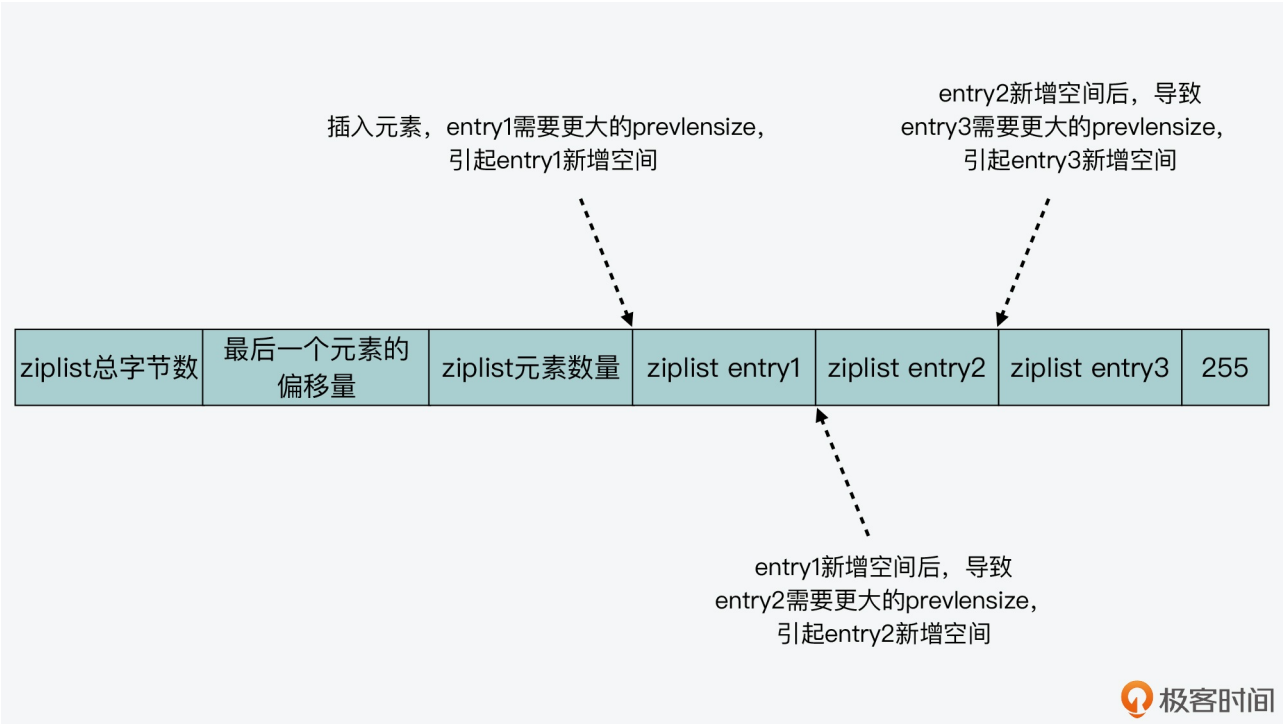
```
unsigned char *ziplistResize(unsigned char *zl, unsigned int len) {
    //对zl进行重新内存空间分配，重新分配的大小是len
    zl = zrealloc(zl,len);
    …
    zl[len-1] = ZIP_END;
    return zl;
}
```

好了，到这里，我们就了解了ziplist在新插入元素时，会计算其所需的新增空间，并进行重新分配。而当新插入的元素较大时，就会引起插入位置的元素prevlensize增加，进而就会导致插入位置的元素所占空间也增加。

而如此一来，这种空间新增就会引起连锁更新的问题。

实际上，所谓的**连锁更新**，就是指当一个元素插入后，会引起当前位置元素新增prevlensize的空间。而当前位置元素的空间增加后，又会进一步引起该元素的后续元素，其prevlensize所需空间的增加。

这样，一旦插入位置后续的所有元素，都会因为前序元素的prevlenszie增加，而引起自身空间也要增加，这种每个元素的空间都需要增加的现象，就是连锁更新。我画了下面这张图，你可以看下。



连锁更新一旦发生，就会导致ziplist占用的内存空间要多次重新分配，这就会直接影响到ziplist的访问性能。

所以说，虽然ziplist紧凑型的内存布局能节省内存开销，但是如果保存的元素数量增加了，或是元素变大了，ziplist就会面临性能问题。那么，有没有什么方法可以避免ziplist的问题呢？

这就是接下来我要给你介绍的quicklist和listpack，这两种数据结构的设计思想了。

quicklist设计与实现

我们先来学习下quicklist的实现思路。

quicklist的设计，其实是结合了链表和ziplist各自的优势。简单来说，**一个quicklist就是一个链表，而链表中的每个元素又是一个ziplist。**

我们来看下quicklist的数据结构，这是在quicklist.h文件中定义的，而quicklist的具体实现是在quicklist.c文件中。

首先，quicklist元素的定义，也就是quicklistNode。因为quicklist是一个链表，所以每个quicklistNode中，都包含了分别指向它前序和后序节点的**指针*prev和*next**。同时，每个quicklistNode又是一个ziplist，所以，在quicklistNode的结构体中，还有指向ziplist的**指针*zl**。

此外，quicklistNode结构体中还定义了一些属性，比如ziplist的字节大小、包含的元素个数、编码格式、存储方式等。下面的代码显示了quicklistNode的结构体定义，你可以看下。

```
typedef struct quicklistNode {
    struct quicklistNode *prev;     //前一个quicklistNode
    struct quicklistNode *next;     //后一个quicklistNode
    unsigned char *zl;              //quicklistNode指向的ziplist
    unsigned int sz;                //ziplist的字节大小
    unsigned int count : 16;        //ziplist中的元素个数
```

```
    unsigned int encoding : 2;   //编码格式，原生字节数组或压缩存储
    unsigned int container : 2;  //存储方式
    unsigned int recompress : 1; //数据是否被压缩
    unsigned int attempted_compress : 1; //数据能否被压缩
    unsigned int extra : 10; //预留的bit位
} quicklistNode;
```

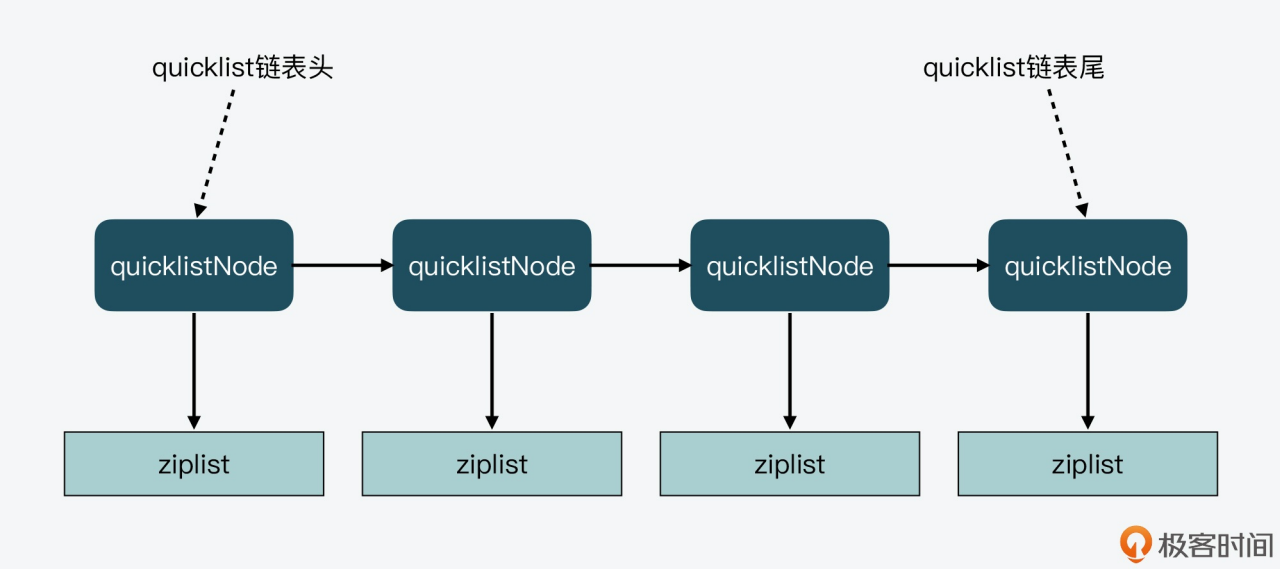
了解了quicklistNode的定义，我们再来看下quicklist的结构体定义。

quicklist作为一个链表结构，在它的数据结构中，是定义了**整个quicklist的头、尾指针**，这样一来，我们就可以通过quicklist的数据结构，来快速定位到quicklist的链表头和链表尾。

此外，quicklist中还定义了quicklistNode的个数、所有ziplist的总元素个数等属性。quicklist的结构定义如下所示：

```
typedef struct quicklist {
    quicklistNode *head;      //quicklist的链表头
    quicklistNode *tail;      //quicklist的链表尾
    unsigned long count;     //所有ziplist中的总元素个数
    unsigned long len;       //quicklistNodes的个数
    ...
} quicklist;
```

然后，从quicklistNode和quicklist的结构体定义中，我们就能画出下面这张quicklist的示意图。



而也正因为quicklist采用了链表结构，所以当插入一个新的元素时，quicklist首先就会检查插入位置的ziplist是否能容纳该元素，这是通过 **_quicklistNodeAllowInsert函数**来完成判断的。

_quicklistNodeAllowInsert函数会计算新插入元素后的大小（new_sz），这个大小等于quicklistNode的当前大小（node->sz）、插入元素的大小（sz），以及插入元素后ziplist的prevlen占用大小。

在计算完大小之后，_quicklistNodeAllowInsert函数会依次判断新插入的数据大小（sz）是否满足要求，即

单个ziplist是否不超过8KB，或是单个ziplist里的元素个数是否满足要求。

只要这里面的一个条件能满足，quicklist就可以在当前的quicklistNode中插入新元素，否则quicklist就会新建一个quicklistNode，以此来保存新插入的元素。

下面代码显示了是否允许在当前quicklistNode插入数据的判断逻辑，你可以看下。

```
unsigned int new_sz = node->sz + sz + ziplist_overhead;
if (likely(_quicklistNodeSizeMeetsOptimizationRequirement(new_sz, fill)))
    return 1;
else if (!sizeMeetsSafetyLimit(new_sz))
    return 0;
else if ((int)node->count < fill)
    return 1;
else
    return 0;
```

这样一来，quicklist通过控制每个quicklistNode中，ziplist的大小或是元素个数，就有效减少了在ziplist中新增或修改元素后，发生连锁更新的情况，从而提供了更好的访问性能。

而Redis除了设计了quicklist结构来应对ziplist的问题以外，还在5.0版本中新增了listpack数据结构，用来彻底避免连锁更新。下面我们就继续来学习下它的设计实现思路。

listpack设计与实现

listpack也叫紧凑列表，它的特点就是**用一块连续的内存空间来紧凑地保存数据**，同时为了节省内存空间，**listpack列表项使用了多种编码方式，来表示不同长度的数据**，这些数据包括整数和字符串。

和listpack相关的实现文件是listpack.c，头文件包括listpack.h和listpack_malloc.h。我们先来看下listpack的**创建函数lpNew**，因为从这个函数的代码逻辑中，我们可以了解到listpack的整体结构。

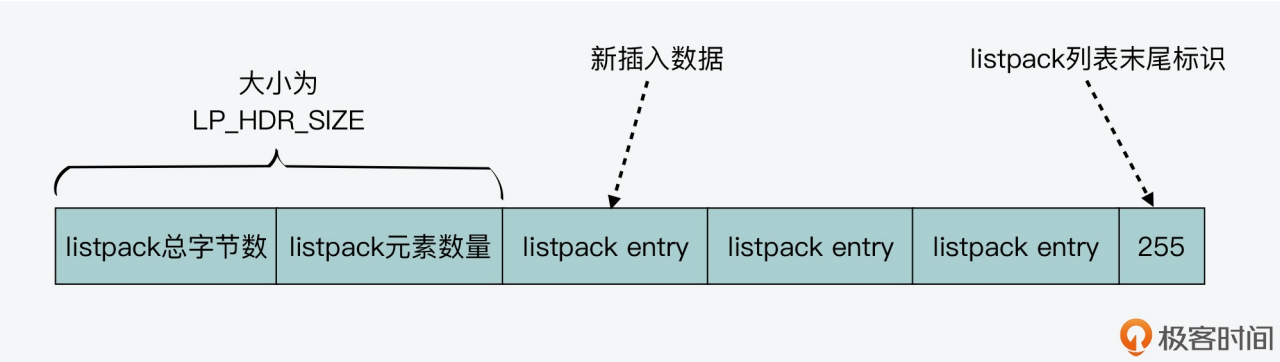
lpNew函数创建了一个空的listpack，一开始分配的大小是LP_HDR_SIZE再加1个字节。LP_HDR_SIZE宏定义是在listpack.c中，它默认是6个字节，其中4个字节是记录listpack的总字节数，2个字节是记录listpack的元素数量。

此外，listpack的最后一个字节是用来标识listpack的结束，其默认值是宏定义LP_EOF。和ziplist列表项的结束标记一样，LP_EOF的值也是255。

```
unsigned char *lpNew(void) {
    //分配LP_HRD_SIZE+1
    unsigned char *lp = lp_malloc(LP_HDR_SIZE+1);
    if (lp == NULL) return NULL;
    //设置listpack的大小
    lpSetTotalBytes(lp,LP_HDR_SIZE+1);
    //设置listpack的元素个数，初始值为0
    lpSetNumElements(lp,0);
    //设置listpack的结尾标识为LP_EOF，值为255
    lp[LP_HDR_SIZE] = LP_EOF;
```

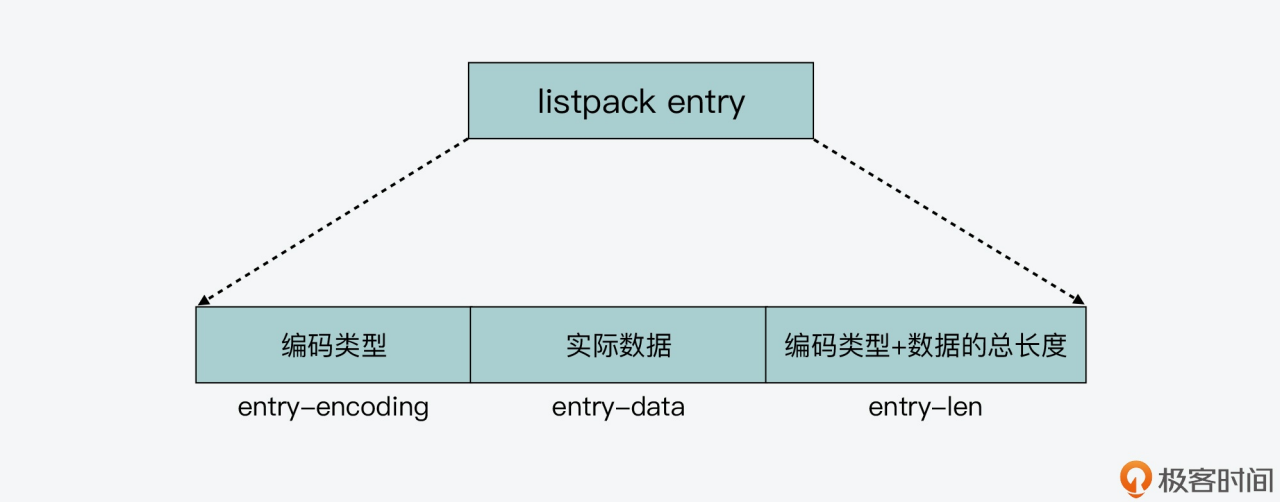
```
    return lp;
}
```

你可以看看下面这张图，展示的就是大小为LP_HDR_SIZE的listpack头和值为255的listpack尾。当有新元素插入时，该元素会被插在listpack头和尾之间。



好了，了解了listpack的整体结构后，我们再来看下listpack列表项的设计。

和ziplist列表项类似，listpack列表项也包含了元数据信息和数据本身。不过，为了避免ziplist引起的连锁更新问题，listpack中的每个列表项不再像ziplist列表项那样，保存其前一个列表项的长度，**它只会包含三个方面内容**，分别是当前元素的编码类型（entry-encoding）、元素数据(entry-data)，以及编码类型和元素数据这两部分的长度(entry-len)，如下图所示。



这里，关于listpack列表项的设计，你需要重点掌握两方面的要点，分别是列表项元素的编码类型，以及列表项避免连锁更新的方法。下面我就带你具体了解下。

listpack列表项编码方法

我们先来看下listpack元素的编码类型。如果你看了listpack.c文件，你会发现该文件中有大量类似LP_ENCODING__XX_BIT_INT和LP_ENCODING__XX_BIT_STR的宏定义，如下所示：

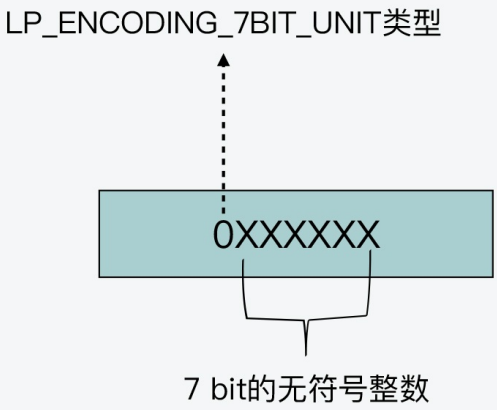
```
#define LP_ENCODING_7BIT_UINT 0
#define LP_ENCODING_6BIT_STR 0x80
#define LP_ENCODING_13BIT_INT 0xC0
...
```

```
#define LP_ENCODING_64BIT_INT 0xF4
#define LP_ENCODING_32BIT_STR 0xF0
```

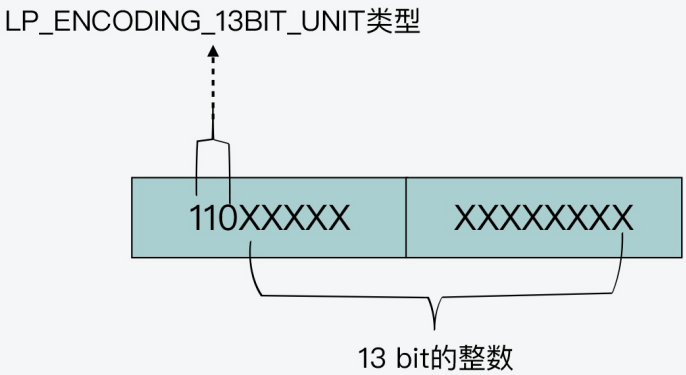
这些宏定义其实就对应了listpack的元素编码类型。具体来说，**listpack元素会对不同长度的整数和字符串进行编码**，这里我们分别来看下。

首先，对于**整数编码**来说，当listpack元素的编码类型为LP_ENCODING_7BIT_UINT时，表示元素的实际数据是一个7 bit的无符号整数。又因为LP_ENCODING_7BIT_UINT本身的宏定义值为0，所以编码类型的值也相应为0，占1个bit。

此时，编码类型和元素实际数据共用1个字节，这个字节的最高位为0，表示编码类型，后续的7位用来存储7 bit的无符号整数，如下图所示：



而当编码类型为LP_ENCODING_13BIT_INT时，这表示元素的实际数据是13 bit的整数。同时，因为LP_ENCODING_13BIT_INT的宏定义值为0xC0，转换为二进制值是1100 0000，所以，这个二进制值中的后5位和后续的1个字节，共13位，会用来保存13bit的整数。而该二进制值中的前3位110，则用来表示当前的编码类型。我画了下面这张图，你可以看下。



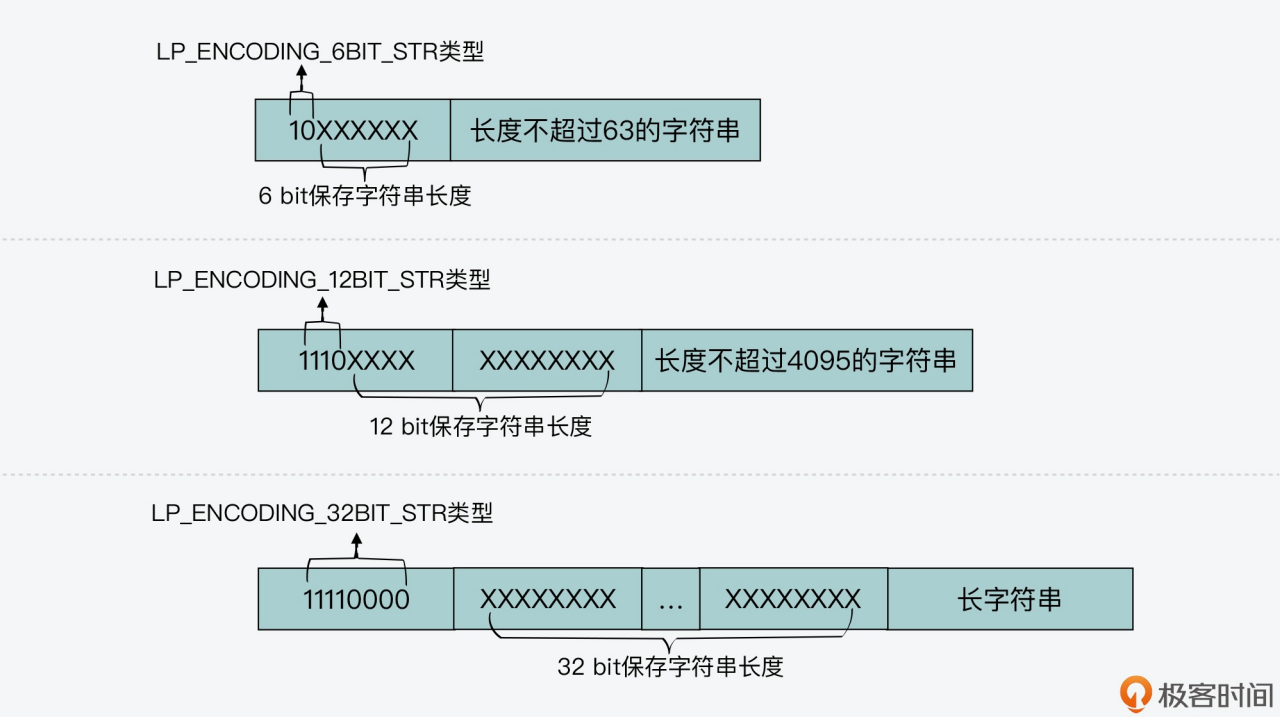
好，在了解了LP_ENCODING_7BIT_UINT和LP_ENCODING_13BIT_INT这两种编码类型后，剩下的LP_ENCODING_16BIT_INT、LP_ENCODING_24BIT_INT、LP_ENCODING_32BIT_INT和LP_ENCODING_64BIT_INT，你应该也就能知道它们的编码方式了。

这四种类型是分别用2字节（16 bit）、3字节（24 bit）、4字节（32 bit）和8字节（64 bit）来保存整数数据。同时，它们的编码类型本身占1字节，编码类型值分别是它们的宏定义值。

然后，对于**字符串编码**来说，一共有三种类型，分别是LP_ENCODING_6BIT_STR、LP_ENCODING_12BIT_STR和LP_ENCODING_32BIT_STR。从刚才的介绍中，你可以看到，整数编码类型名称中BIT前面的数字，表示的是整数的长度。因此类似的，字符串编码类型名称中BIT前的数字，表示的就是字符串的长度。

比如，当编码类型为LP_ENCODING_6BIT_STR时，编码类型占1字节。该类型的宏定义值是0x80，对应的二进制值是1000 0000，这其中的前2位是用来标识编码类型本身，而后6位保存的是字符串长度。然后，列表项中的数据部分保存了实际的字符串。

下面的图展示了三种字符串编码类型和数据的布局，你可以看下。



listpack避免连锁更新的实现方式

最后，我们再来了解下listpack列表项是如何避免连锁更新的。

在listpack中，因为每个列表项只记录自己的长度，而不会像ziplist中的列表项那样，会记录前一项的长度。所以，当我们在listpack中新增或修改元素时，实际上只会涉及每个列表项自己的操作，而不会影响后续列表项的长度变化，这就避免了连锁更新。

不过，你可能会有疑问：**如果listpack列表项只记录当前项的长度，那么listpack支持从左向右正向查询列表，或是从右向左反向查询列表吗？**

其实，listpack是能支持正、反向查询列表的。

当应用程序从左向右正向查询listpack时，我们可以先调用lpFirst函数。该函数的参数是指向listpack头的指针，它在执行时，会让指针向右偏移LP_HDR_SIZE大小，也就是跳过listpack头。你可以看下lpFirst函数的代码，如下所示：

```
unsigned char *lpFirst(unsigned char *lp) {
    lp += LP_HDR_SIZE; //跳过listpack头部6个字节
    if (lp[0] == LP_EOF) return NULL;  //如果已经是listpack的末尾结束字节，则返回NULL
    return lp;
}
```

然后，再调用lpNext函数，该函数的参数包括了指向listpack某个列表项的指针。lpNext函数会进一步调用lpSkip函数，并传入当前列表项的指针，如下所示：

```
unsigned char *lpNext(unsigned char *lp, unsigned char *p) {
    ...
    p = lpSkip(p);  //调用lpSkip函数，偏移指针指向下一个列表项
    if (p[0] == LP_EOF) return NULL;
    return p;
}
```

最后，lpSkip函数会先后调用lpCurrentEncodedSize和lpEncodeBacklen这两个函数。

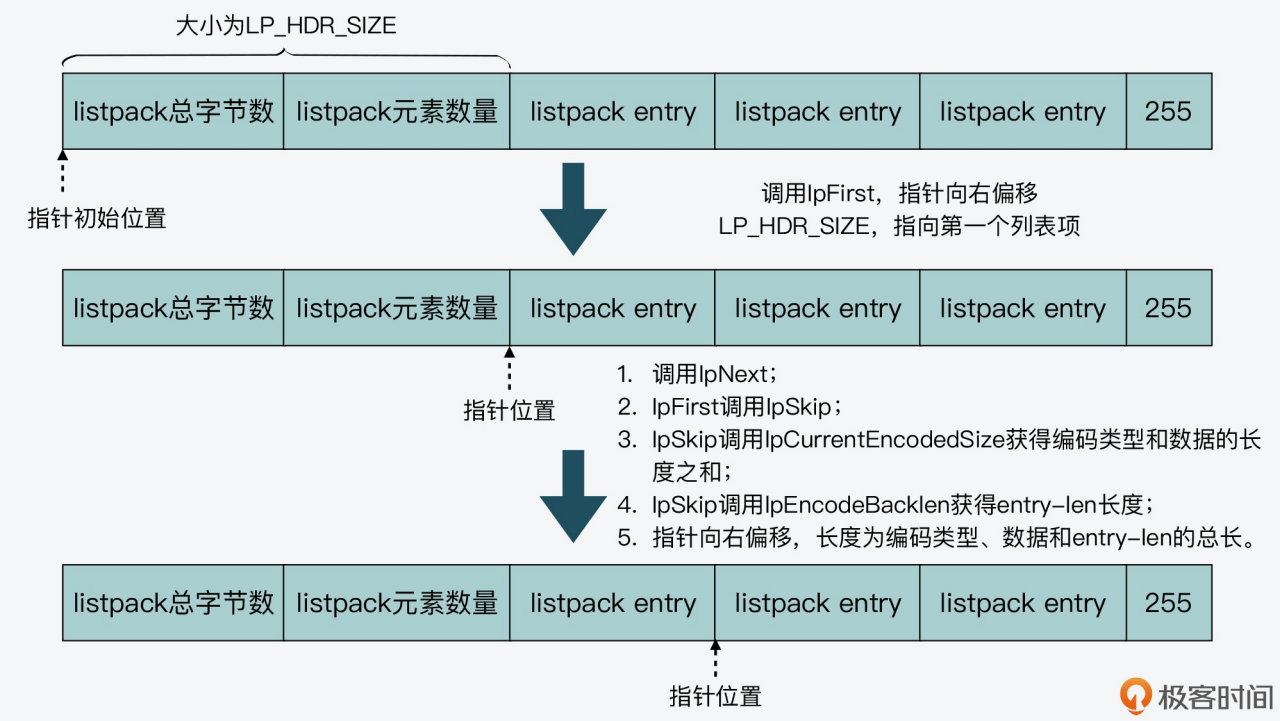
lpCurrentEncodedSize函数是根据当前列表项第1个字节的取值，来计算当前项的编码类型，并根据编码类型，计算当前项编码类型和实际数据的总长度。然后，lpEncodeBacklen函数会根据编码类型和实际数据的长度之和，进一步计算列表项最后一部分entry-len本身的长度。

这样一来，lpSkip函数就知道当前项的编码类型、实际数据和entry-len的总长度了，也就可以将当前项指针向右偏移相应的长度，从而实现查到下一个列表项的目的。

下面代码展示了lpEncodeBacklen函数的基本计算逻辑，你可以看下。

```
unsigned long lpEncodeBacklen(unsigned char *buf, uint64_t l) {
    //编码类型和实际数据的总长度小于等于127，entry-len长度为1字节
    if (l <= 127) {
        ...
        return 1;
    } else if (l < 16383) { //编码类型和实际数据的总长度大于127但小于16383，entry-len长度为2字节
       ...
        return 2;
    } else if (l < 2097151) {//编码类型和实际数据的总长度大于16383但小于2097151，entry-len长度为3字节
       ...
        return 3;
    } else if (l < 268435455) { //编码类型和实际数据的总长度大于2097151但小于268435455，entry-len长度为4字节
        ...
        return 4;
    } else { //否则，entry-len长度为5字节
       ...
        return 5;
    }
}
```

我也画了一张图，展示了从左向右遍历listpack的基本过程，你可以再回顾下。



好，了解了从左向右正向查询listpack，我们再来看下**从右向左反向查询listpack**。

首先，我们根据listpack头中记录的listpack总长度，就可以直接定位到listapck的尾部结束标记。然后，我们可以调用lpPrev函数，该函数的参数包括指向某个列表项的指针，并返回指向当前列表项前一项的指针。

lpPrev函数中的关键一步就是调用lpDecodeBacklen函数。lpDecodeBacklen函数会从右向左，逐个字节地读取当前列表项的entry-len。

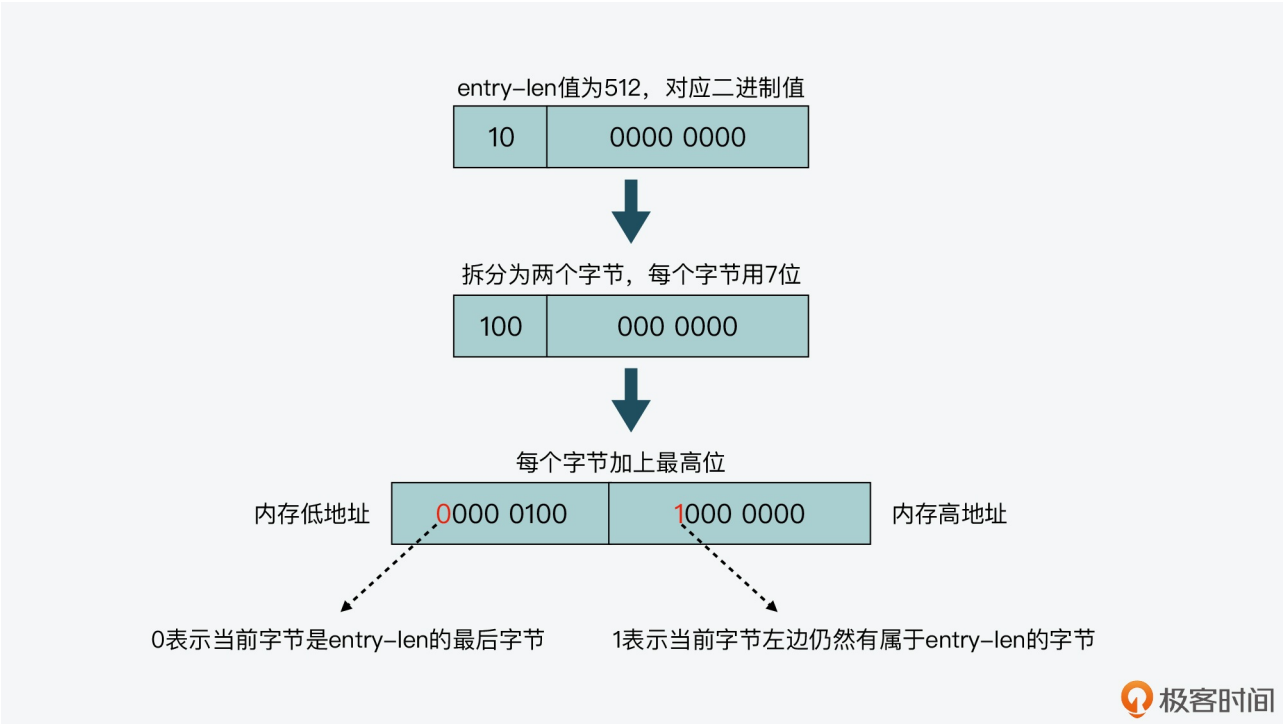
那么，**lpDecodeBacklen函数如何判断entry-len是否结束了呢?**

这就依赖于entry-len的编码方式了。entry-len每个字节的最高位，是用来表示当前字节是否为entry-len的最后一个字节，这里存在两种情况，分别是：

- 最高位为1，表示entry-len还没有结束，当前字节的左边字节仍然表示entry-len的内容；
- 最高位为0，表示当前字节已经是entry-len最后一个字节了。

而entry-len每个字节的低7位，则记录了实际的长度信息。这里你需要注意的是，entry-len每个字节的低7位采用了**大端模式存储**，也就是说，entry-len的低位字节保存在内存高地址上。

我画了下面这张图，展示了entry-len这种特别的编码方式，你可以看下。



实际上，正是因为有了entry-len的特别编码方式，lpDecodeBacklen函数就可以从当前列表项起始位置的指针开始，向左逐个字节解析，得到前一项的entry-len值。这也是lpDecodeBacklen函数的返回值。而从刚才的介绍中，我们知道entry-len记录了编码类型和实际数据的长度之和。

因此，lpPrev函数会再调用lpEncodeBacklen函数，来计算得到entry-len本身长度，这样一来，我们就可以得到前一项的总长度，而lpPrev函数也就可以将指针指向前一项的起始位置了。所以按照这个方法，listpack就实现了从右向左的查询功能。

小结

这节课，我从ziplist的设计不足出发，依次给你介绍了quicklist和listpack的设计思想。

你要知道，ziplist的不足主要在于**一旦ziplist中元素个数多了，它的查找效率就会降低**。而且如果在ziplist里新增或修改数据，ziplist占用的内存空间还需要重新分配；更糟糕的是，ziplist新增某个元素或修改某个元素时，可能会导致后续元素的prevlen占用空间都发生变化，从而引起连锁更新问题，导致每个元素的空间都要重新分配，这就会导致ziplist的访问性能下降。

所以，为了应对ziplist的问题，Redis先是在3.0版本中设计实现了quicklist。quicklist结构在ziplist基础上，使用链表将ziplist串联起来，链表的每个元素就是一个ziplist。这种设计**减少了数据插入时内存空间的重新分配，以及内存数据的拷贝**。同时，quicklist限制了每个节点上ziplist的大小，一旦一个ziplist过大，就会采用新增quicklist节点的方法。

不过，又因为quicklist使用quicklistNode结构指向每个ziplist，无疑增加了内存开销。为了**减少内存开销，并进一步避免ziplist连锁更新问题**，Redis在5.0版本中，就设计实现了listpack结构。listpack结构沿用了ziplist紧凑型的内存布局，把每个元素都紧挨着放置。

listpack中每个列表项不再包含前一项的长度了，因此当某个列表项中的数据发生变化，导致列表项长度变化时，其他列表项的长度是不会受影响的，因而这就避免了ziplist面临的连锁更新问题。

总而言之，Redis在内存紧凑型列表的设计与实现上，从ziplist到quicklist，再到listpack，你可以看到

Redis在内存空间开销和访问性能之间的设计取舍，这一系列的设计变化，是非常值得你学习的。

每课一问

ziplist会使用zipTryEncoding函数计算插入元素所需的新增内存空间，假设插入的一个元素是整数，你知道ziplist能支持的最大整数是多大吗？

欢迎在留言区分享你的答案和思考过程，如果觉得有收获，也欢迎你把今天的内容分享给更多的朋友。