

源码 3：挨肩迭背 —— 探索「压缩列表」内部

Redis 为了节约内存空间使用，zset 和 hash 容器对象在元素个数较少的时候，采用压缩列表 (ziplist) 进行存储。压缩列表是一块连续的内存空间，元素之间紧挨着存储，没有任何冗余空隙。

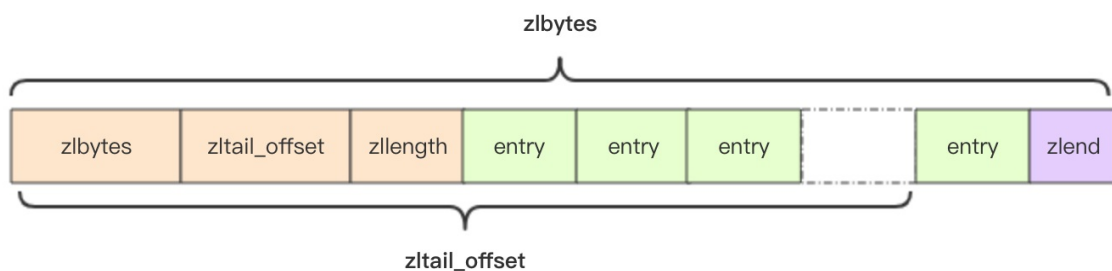
```
> zadd programmings 1.0 go 2.0 python 3.0 java
(integer) 3
> debug object programmings
Value at:0x7fec2de00020 refcount:1
encoding:ziplist serializedlength:36 lru:6022374
lru_seconds_idle:6
> hmset books go fast python slow java fast
OK
> debug object books
Value at:0x7fec2de000c0 refcount:1
encoding:ziplist serializedlength:48 lru:6022478
lru_seconds_idle:1
```

这里，注意观察 debug object 输出的 encoding 字段都是 ziplist，这就表示内部采用压缩列表结构进行存储。

```

struct ziplist<T> {
    int32 zlbytes; // 整个压缩列表占用字节数
    int32 zltail_offset; // 最后一个元素距离压缩列表起始位置的偏移量，用于快速定位到最后一个节点
    int16 zllength; // 元素个数
    T[] entries; // 元素内容列表，挨个挨个紧凑存储
    int8 zlend; // 标志压缩列表的结束，值恒为 0xFF
}

```



压缩列表为了支持双向遍历，所以才会有 `zltail_offset` 这个字段，用来快速定位到最后一个元素，然后倒着遍历。

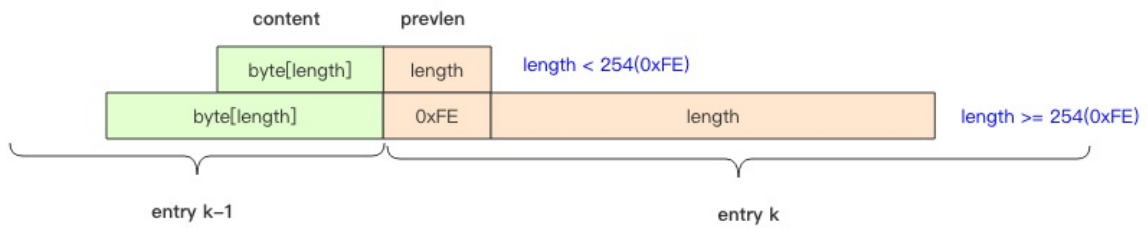
`entry` 块随着容纳的元素类型不同，也会有不一样的结构。

```

struct entry {
    int<var> prevlen; // 前一个 entry 的字节长度
    int<var> encoding; // 元素类型编码
    optional byte[] content; // 元素内容
}

```

它的 `prevlen` 字段表示前一个 `entry` 的字节长度，当压缩列表倒着遍历时，需要通过这个字段来快速定位到下一个元素的位置。它是一个变长的整数，当字符串长度小于 254(0xFE) 时，使用一个字节表示；如果达到或超出 254(0xFE) 那就使用 5 个字节来表示。第一个字节是 0xFE(254)，剩余四个字节表示字符串长度。你可能会觉得用 5 个字节来表示字符串长度，是不是太浪费了。我们可以算一下，当字符串长度比较长的时候，其实 5 个字节也只占用了不到 $(5/(254+5)) < 2\%$ 的空间。



encoding字段存储了元素内容的编码类型信息，ziplist 通过这个字段来决定后面的 content 内容的形式。

Redis 为了节约存储空间，对 encoding 字段进行了相当复杂的设计。Redis 通过这个字段的前缀位来识别具体存储的数据形式。下面我们来看看 Redis 是如何根据encoding的前缀位来区分内容的：

1. 00xxxxxx 最大长度位 63 的短字符串，后面的 6 个位存储字符串的位数，剩余的字节就是字符串的内容。
2. 01xxxxxx xxxxxxxx 中等长度的字符串，后面 14 个位来表示字符串的长度，剩余的字节就是字符串的内容。
3. 10000000 aaaaaaaaaa bbbbbbbb cccccccc dddddddd 特大字符串，需要使用额外 4 个字节来表示长度。第一个字节前缀是10，剩余 6 位没有使用，统一置为零。后面跟着字符串内容。不过这样的大字符串是没有机会使用的，压缩列表通常只是用来存储小数据的。
4. 11000000 表示 int16，后跟两个字节表示整数。
5. 11010000 表示 int32，后跟四个字节表示整数。
6. 11100000 表示 int64，后跟八个字节表示整数。
7. 11110000 表示 int24，后跟三个字节表示整数。
8. 11111110 表示 int8，后跟一个字节表示整数。
9. 11111111 表示 ziplist 的结束，也就是 zlend 的值 0xFF。
10. 1111xxxx 表示极小整数，xxxx 的范围只能是 (0001~1101)，也就是1~13，因为0000、1110、1111都被占用了。读取到的 value 需要将 xxxx 减 1，也就是整数0~12就是最终的 value。

注意到 content 字段在结构体中定义为 optional 类型，表示这个字段是可选的，对于很小的整数而言，它的内容已经内联到 encoding 字段的尾部了。

增加元素

因为 ziplist 都是紧凑存储，没有冗余空间 (对比一下 Redis 的字符串结构)。意味着插入一个新的元素就需要调用 realloc 扩展内存。取决于内存分配器算法和当前的 ziplist 内存大小，realloc 可能会重新分配新的内存空间，并将之前的内容一次性拷贝到新的地址，也可能在原有的地址上进行扩展，这时就不需要进行旧内容的内存拷贝。

如果 ziplist 占据内存太大，重新分配内存和拷贝内存就会有很大的消耗。所以 ziplist 不适合存储大型字符串，存储的元素也不宜过多。

级联更新

```
/* When an entry is inserted, we need to set the
prevlen field of the next
 * entry to equal the length of the inserted
entry. It can occur that this
 * length cannot be encoded in 1 byte and the
next entry needs to be grow
 * a bit larger to hold the 5-byte encoded
prevlen. This can be done for free,
 * because this only happens when an entry is
already being inserted (which
 * causes a realloc and memmove). However,
encoding the prevlen may require
 * that this entry is grown as well. This effect
```

may cascade throughout

- * the ziplist when there are consecutive entries with a size close to

- * ZIP_BIG_PREVLEN, so we need to check that the prevlen can be encoded in

- * every consecutive entry.

- *

- * Note that this effect can also happen in reverse, where the bytes required

- * to encode the prevlen field can shrink. This effect is deliberately ignored,

- * because it can cause a "flapping" effect where a chain prevlen fields is

- * first grown and then shrunk again after consecutive inserts. Rather, the

- * field is allowed to stay larger than necessary, because a large prevlen

- * field implies the ziplist is holding large entries anyway.

- *

- * The pointer "p" points to the first entry that does NOT need to be

- * updated, i.e. consecutive fields MAY need an update. */

```
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) {
    zipEntry(p, &cur);
    rawlen = cur.headersize + cur.len;
    rawlensize =
zipStorePrevEntryLength(NULL,rawlen);

    /* Abort if there is no next entry. */
    if (p[rawlen] == ZIP_END) break;
    zipEntry(p+rawlen, &next);

    /* Abort when "prevlen" has not changed.
*/
    // prevlen 的长度没有变，中断级联更新
    if (next.prevrawlen == rawlen) break;

    if (next.prevrawlensize < rawlensize) {
        /* The "prevlen" field of "next"
needs more bytes to hold
        * the raw length of "cur". */
        // 级联扩展
        offset = p-zl;
        extra = rawlensize-
next.prevrawlensize;
        // 扩大内存
        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. */
    // 更新 zltail_offset 指针
    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. */
    // 移动内存
    memmove(np+rawlensize,
            np+next.prevrawlensize,
            curlen-noffset-
next.prevrawlensize-1);
    zipStorePrevEntryLength(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. */
        // 级联收缩, 不过这里可以不用收缩了, 因
为 5 个字节也是可以存储 1 个字节的内容的
        // 虽然有点浪费, 但是级联更新实在是太可
怕了, 所以浪费就浪费吧

```

```

zipStorePrevEntryLengthLarge(p+rawlen,rawlen);
        } else {
            // 大小没变，改个长度值就完事了

zipStorePrevEntryLength(p+rawlen,rawlen);
        }

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

```

前面提到每个 entry 都会有一个 prevlen 字段存储前一个 entry 的长度。如果内容小于 254 字节，prevlen 用 1 字节存储，否则就是 5 字节。这意味着如果某个 entry 经过了修改操作从 253 字节变成了 254 字节，那么它的下一个 entry 的 prevlen 字段就要更新，从 1 个字节扩展到 5 个字节；如果这个 entry 的长度本来也是 253 字节，那么后面 entry 的 prevlen 字段还得继续更新。

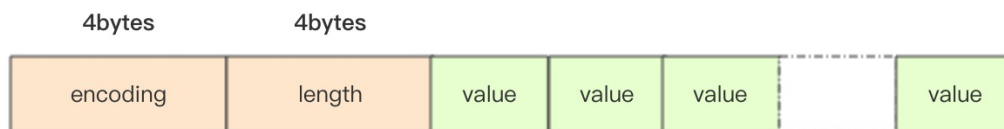
如果 ziplist 里面每个 entry 恰好都存储了 253 字节的内容，那么第一个 entry 内容的修改就会导致后续所有 entry 的级联更新，这就是一个比较耗费计算资源的操作。

删除中间的某个节点也可能会导致级联更新，读者可以思考一下为什么？

IntSet 小整数集合

当 set 集合容纳的元素都是整数并且元素个数较小时，Redis 会使用 intset 来存储集合元素。intset 是紧凑的数组结构，同时支持 16 位、32 位和 64 位整数。

```
struct intset<T> {  
    int32 encoding; // 决定整数位宽是 16 位、32 位还是 64 位  
    int32 length; // 元素个数  
    int<T> contents; // 整数数组，可以是 16 位、32 位和 64 位  
}
```



老钱也不是很能理解为什么 intset 的 encoding 字段和 length 字段使用 32 位整数存储。毕竟它只是用来存储小整数的，长度不应该很长，而且 encoding 只有 16 位、32 位和 64 位三个类型，用一个字节存储就绰绰有余。关于这点，读者们可以进一步讨论。

```
> sadd codehole 1 2 3  
(integer) 3  
> debug object codehole  
Value at:0x7fec2dc2bde0 refcount:1  
encoding:intset serializedlength:15 lru:6065795  
lru_seconds_idle:4  
> sadd codehole go java python  
(integer) 3  
> debug object codehole  
Value at:0x7fec2dc2bde0 refcount:1  
encoding:hashtable serializedlength:22  
lru:6065810 lru_seconds_idle:5
```

注意观察 debug object 的输出字段 encoding 的值，可以发现当 set 里面放进去了非整数值时，存储形式立即从 intset 转变成了 hash 结构。

思考

1. 为什么 set 集合在数量很小的时候不使用 ziplist 来存储？
2. 执行 `push codehole 1 2 3` 命令后，请写出列表内容的 16 进制形式。