04-内存友好的数据结构该如何细化设计？

你好，我是蒋德钧。今天我们来聊聊，Redis中是如何通过优化设计数据结构，来提升内存利用率的。

我们知道Redis是内存数据库，所以，高效使用内存对Redis的实现来说非常重要。而实际上，Redis主要是通过两大方面的技术来提升内存使用效率的，分别是**数据结构的优化设计与使用**，以及**内存数据按一定规则淘汰**。

关于内存数据按规则淘汰，这是通过Redis内存替换策略实现的，也就是将很少使用的数据从内存中淘汰，从而把有限的内存空间用于保存会被频繁访问的数据。这部分的设计与实现，主要和内存替换策略有关，我会在后面的缓存模块给你详细介绍。

所以这节课，我主要是带你学习Redis数据结构在面向内存使用效率方面的优化，其中包括两方面的设计思路：一是**内存友好的数据结构设计**；二是**内存友好的数据使用方式**。

这两方面的设计思路和实现方法是具有通用性的，当你在设计系统软件时，如果需要对内存使用精打细算，以便节省内存开销，这两种设计方法和实现考虑就非常值得学习和掌握。

好，接下来，我们就先来学习下内存友好的数据结构设计。

内存友好的数据结构

首先要知道，在Redis中，有三种数据结构针对内存使用效率做了设计优化，分别是简单动态字符串（SDS）、压缩列表（ziplist）和整数集合（intset）。下面，我们就分别来学习一下。

SDS的内存友好设计

实际上，我在第2讲中就已经给你介绍过SDS的结构设计，这里我们先做个简单的回顾：SDS设计了不同类型的结构头，包括sdshdr8、sdshdr16、sdshdr32和sdshdr64。这些不同类型的结构头可以适配不同大小的字符串，从而避免了内存浪费。

不过，SDS除了使用精巧设计的结构头外，在保存较小字符串时，其实还使用了**嵌入式字符串**的设计方法。这种方法避免了给字符串分配额外的空间，而是可以让字符串直接保存在Redis的基本数据对象结构体中。

所以这也就是说，要想理解嵌入式字符串的设计与实现，我们就需要先来了解下，Redis使用的基本数据对象结构体redisObject是什么样的。

redisObject结构体与位域定义方法

redisObject结构体是在server.h文件中定义的，主要功能是用来保存键值对中的值。这个结构一共定义了4个元数据和一个指针。

- **type**：redisObject的数据类型，是应用程序在Redis中保存的数据类型，包括String、List、Hash等。
- **encoding**：redisObject的编码类型，是Redis内部实现各种数据类型所用的数据结构。
- **lru**：redisObject的LRU时间。
- **refcount**：redisObject的引用计数。
- **ptr**：指向值的指针。

下面的代码展示了redisObject结构体的定义：

```
typedef struct redisObject {
    unsigned type:4; //redisObject的数据类型，4个bits
    unsigned encoding:4; //redisObject的编码类型，4个bits
    unsigned lru:LRU_BITS;  //redisObject的LRU时间，LRU_BITS为24个bits
    int refcount; //redisObject的引用计数，4个字节
    void *ptr; //指向值的指针，8个字节
} robj;
```

从代码中我们可以看到，在type、encoding和lru三个变量后面都有一个冒号，并紧跟着一个数值，表示该元数据占用的比特数。其中，type和encoding分别占4bits。而lru占用的比特数，是由server.h中的宏定义LRU_BITS决定的，它的默认值是24bits，如下所示：

```
#define LRU_BITS 24
```

而这里我想让你学习掌握的，就是这种**变量后使用冒号和数值的定义方法**。这实际上是C语言中的**位域定义方法**，可以用来有效地节省内存开销。

这种方法比较适用的场景是，当一个变量占用不了一个数据类型的所有bits时，就可以使用位域定义方法，把一个数据类型中的bits，划分成多个位域，每个位域占一定的bit数。这样一来，一个数据类型的所有bits就可以定义多个变量了，从而也就有效节省了内存开销。

此外，你可能还会发现，对于type、encoding和lru三个变量来说，它们的数据类型都是unsigned。已知一个unsigned类型是4字节，但这三个变量，是分别占用了一个unsigned类型4字节中的4bits、4bits和24bits。因此，相较于三个变量，每个变量用一个4字节的unsigned类型定义来说，使用位域定义方法可以让三个变量只用4字节，最后就能节省8字节的开销。

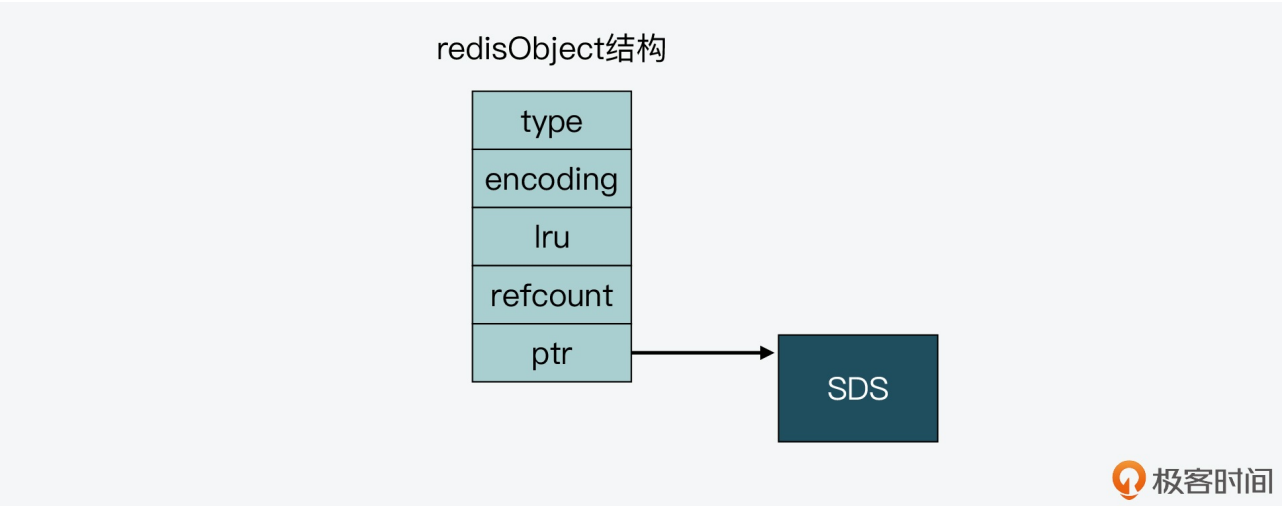
所以，当你在设计开发内存敏感型的软件时，就可以把这种位域定义方法使用起来。

好，了解了redisObject结构体和它使用的位域定义方法以后，我们再来看嵌入式字符串是如何实现的。

嵌入式字符串

前面我说过，SDS在保存比较小的字符串时，会使用嵌入式字符串的设计方法，将字符串直接保存在redisObject结构体中。然后在redisObject结构体中，存在一个指向值的指针ptr，而一般来说，这个ptr指针会指向值的数据结构。

这里我们就以创建一个String类型的值为例，Redis会调用**createStringObject函数**，来创建相应的redisObject，而这个redisObject中的ptr指针，就会指向SDS数据结构，如下图所示。



在Redis源码中，createStringObject函数会根据要创建的字符串的长度，决定具体调用哪个函数来完成创建。

那么针对这个createStringObject函数来说，它的参数是**字符串ptr**和**字符串长度len**。当len的长度大于OBJ_ENCODING_EMBSTR_SIZE_LIMIT这个宏定义时，createStringObject函数会调用createRawStringObject函数，否则就调用createEmbeddedStringObject函数。而在我们分析的Redis 5.0.8源码版本中，这个OBJ_ENCODING_EMBSTR_SIZE_LIMIT默认定义为44字节。

这部分代码如下所示：

```
#define OBJ_ENCODING_EMBSTR_SIZE_LIMIT 44
robj *createStringObject(const char *ptr, size_t len) {
    //创建嵌入式字符串，字符串长度小于等于44字节
    if (len <= OBJ_ENCODING_EMBSTR_SIZE_LIMIT)
        return createEmbeddedStringObject(ptr,len);
    //创建普通字符串，字符串长度大于44字节
    else
        return createRawStringObject(ptr,len);
}
```

现在，我们就来分析一下createStringObject函数的源码实现，以此了解大于44字节的普通字符串和小于等于44字节的嵌入式字符串分别是如何创建的。

首先，对于**createRawStringObject函数**来说，它在创建String类型的值的时候，会调用createObject函数。

> 补充：createObject函数主要是用来创建Redis的数据对象的。因为Redis的数据对象有很多类型，比如String、List、Hash等，所以在createObject函数的两个参数中，有一个就是用来表示所要创建的数据对象类型，而另一个是指向数据对象的指针。

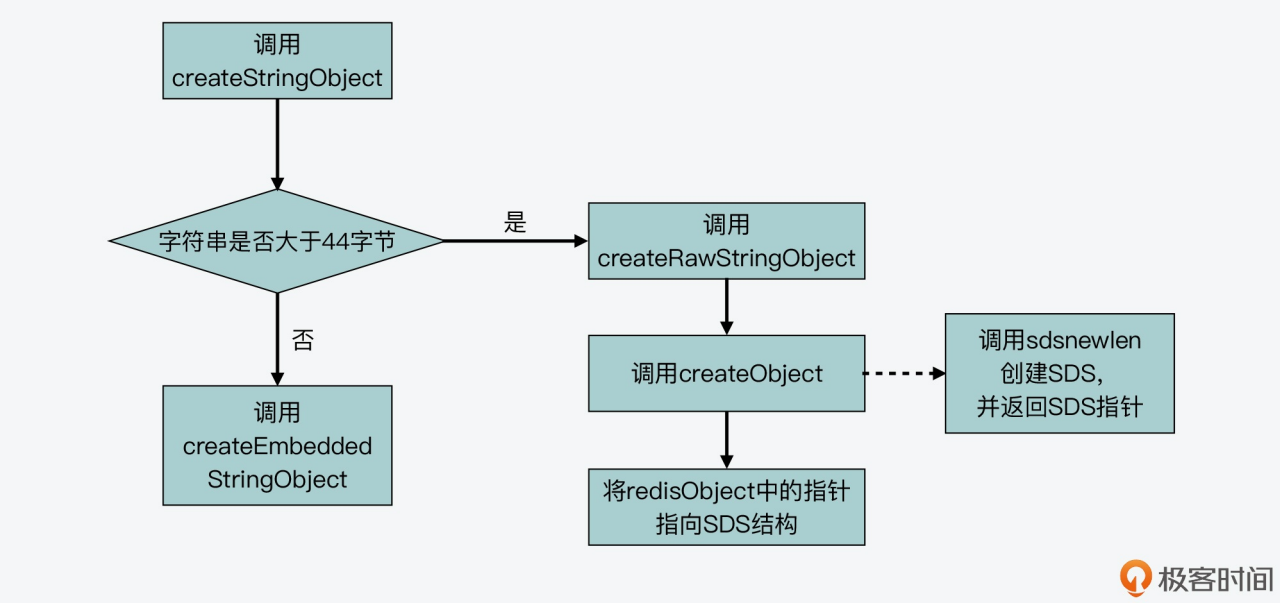
然后，createRawStringObject函数在调用createObject函数时，会传递OBJ_STRING类型，表示要创建String类型的对象，以及传递指向SDS结构的指针，如以下代码所示。这里**需要注意**的是，指向SDS结构的指针是由sdsnewlen函数返回的，而sdsnewlen函数正是用来创建SDS结构的。

```
robj *createRawStringObject(const char *ptr, size_t len) {
    return createObject(OBJ_STRING, sdsnewlen(ptr,len));
}
```

最后，我们再来进一步看下**createObject函数**。这个函数会把参数中传入的、指向SDS结构体的指针直接赋值给redisObject中的ptr，这部分的代码如下所示：

```
robj *createObject(int type, void *ptr) {
    //给redisObject结构体分配空间
   robj *o = zmalloc(sizeof(*o));
   //设置redisObject的类型
   o->type = type;
   //设置redisObject的编码类型，此处是OBJ_ENCODING_RAW，表示常规的SDS
   o->encoding = OBJ_ENCODING_RAW;
   //直接将传入的指针赋值给redisObject中的指针。
    o->ptr = ptr;
    o->refcount = 1;
    …
    return o;
}
```

为了方便理解普通字符串创建方法，我画了一张图，你可以看下。



这也就是说，在创建普通字符串时，Redis需要分别给redisObject和SDS分别分配一次内存，这样就既带来了内存分配开销，同时也会导致内存碎片。因此，当字符串小于等于44字节时，Redis就使用了嵌入式字符串的创建方法，以此减少内存分配和内存碎片。

而这个创建方法，就是由我们前面提到的**createEmbeddedStringObject函数**来完成的，该函数会使用一块连续的内存空间，来同时保存redisObject和SDS结构。这样一来，内存分配只有一次，而且也避免了内存碎片。

createEmbeddedStringObject函数的原型定义如下，它的参数就是从createStringObject函数参数中获得

的字符串指针ptr，以及字符串长度len。

```
robj *createEmbeddedStringObject(const char *ptr, size_t len)
```

那么下面，我们就来具体看看，createEmbeddedStringObject函数是如何把redisObject和SDS放置在一起的。

首先，createEmbeddedStringObject函数会**分配一块连续的内存空间**，这块内存空间的大小等于redisObject结构体的大小、SDS结构头sdshdr8的大小和字符串大小的总和，并且再加上1字节。注意，这里最后的1字节是SDS中加在字符串最后的结束字符“\0”。

这块连续内存空间的分配情况如以下代码所示：

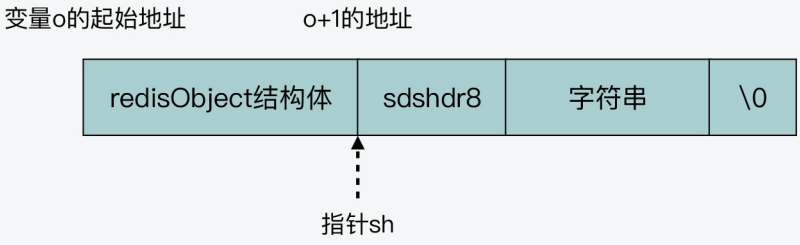
```
robj *o = zmalloc(sizeof(robj)+sizeof(struct sdshdr8)+len+1);
```

你也可以参考下图，其中展示了这块内存空间的布局。

好，那么createEmbeddedStringObject函数在分配了内存空间之后，就会**创建SDS结构的指针sh，并把sh指向这块连续空间中SDS结构头所在的位置**，下面的代码显示了这步操作。其中，o是redisObject结构体的变量，o+1表示将内存地址从变量o开始移动一段距离，而移动的距离等于redisObject这个结构体的大小。

```
struct sdshdr8 *sh = (void*)(o+1);
```

经过这步操作后，sh指向的位置就如下图所示：

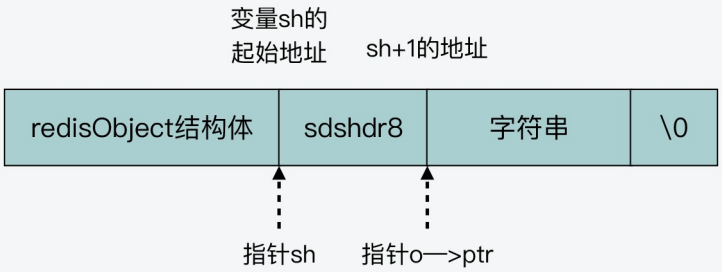


紧接着，createEmbeddedStringObject函数会**把redisObject中的指针ptr，指向SDS结构中的字符数组**。

如以下代码所示，其中sh是刚才介绍的指向SDS结构的指针，属于sdshdr8类型。而sh+1表示把内存地址从sh起始地址开始移动一定的大小，移动的距离等于sdshdr8结构体的大小。

```
o->ptr = sh+1;
```

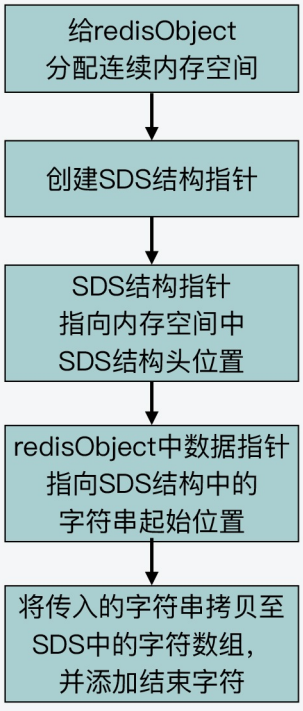
这步操作完成后，redisObject结构体中的指针ptr的指向位置就如下图所示，它会指向SDS结构头的末尾，同时也是字符数组的起始位置：



最后，createEmbeddedStringObject函数会**把参数中传入的指针ptr指向的字符串，拷贝到SDS结构体中的字符数组，并在数组最后添加结束字符**。这部分代码如下所示：

```
memcpy(sh->buf,ptr,len);
sh->buf[len] = '\0';
```

下面这张图，也展示了createEmbeddedStringObject创建嵌入式字符串的过程，你可以再整体来看看。



总之，你可以记住，Redis会通过设计实现一块连续的内存空间，把redisObject结构体和SDS结构体紧凑地放置在一起。这样一来，对于不超过44字节的字符串来说，就可以避免内存碎片和两次内存分配的开销了。

而除了嵌入式字符串之外，Redis还设计了压缩列表和整数集合，这也是两种紧凑型的内存数据结构，所以下面我们再来学习下它们的设计思路。

压缩列表和整数集合的设计

首先你要知道，List、Hash和Sorted Set这三种数据类型，都可以使用压缩列表（ziplist）来保存数据。压缩列表的函数定义和实现代码分别在ziplist.h和ziplist.c中。

不过，我们在ziplist.h文件中其实根本看不到压缩列表的结构体定义。这是因为压缩列表本身就是一块连续的内存空间，它通过使用不同的编码来保存数据。

这里为了方便理解压缩列表的设计与实现，我们先来看看它的**创建函数ziplistNew**，如下所示：

```
unsigned char *ziplistNew(void) {
    //初始分配的大小
    unsigned int bytes = ZIPLIST_HEADER_SIZE+ZIPLIST_END_SIZE;
    unsigned char *zl = zmalloc(bytes);
    …
   //将列表尾设置为ZIP_END
    zl[bytes-1] = ZIP_END;
    return zl;
}
```

实际上，ziplistNew函数的逻辑很简单，就是创建一块连续的内存空间，大小为ZIPLIST_HEADER_SIZE和ZIPLIST_END_SIZE的总和，然后再把该连续空间的最后一个字节赋值为ZIP_END，表示列表结束。

另外你要注意的是，在上面代码中定义的三个宏ZIPLIST_HEADER_SIZE、ZIPLIST_END_SIZE和ZIP_END，在ziplist.c中也分别有定义，分别表示ziplist的列表头大小、列表尾大小和列表尾字节内容，如下所示。

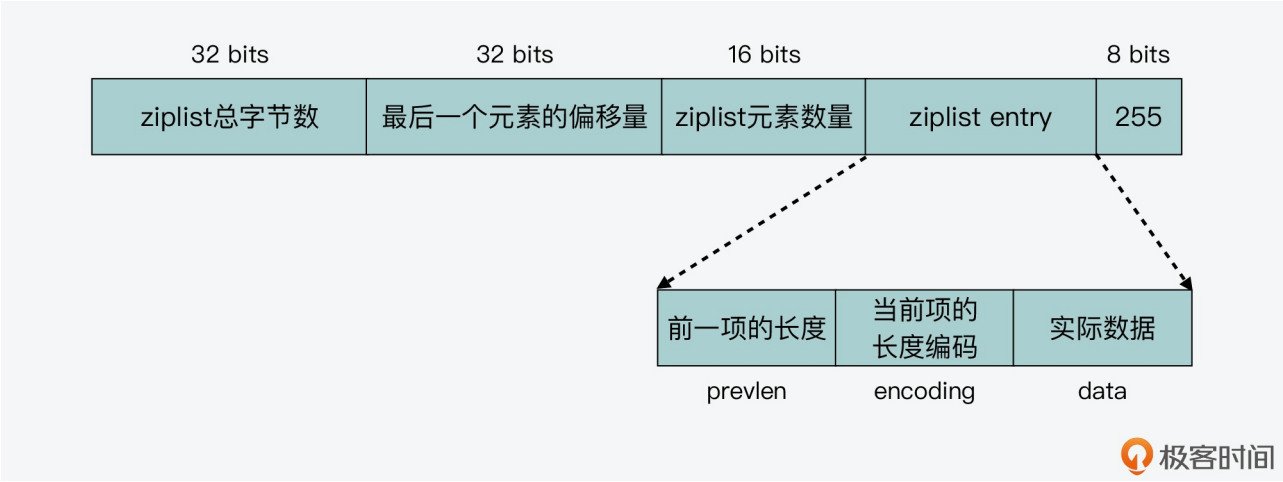
```
//ziplist的列表头大小，包括2个32 bits整数和1个16bits整数，分别表示压缩列表的总字节数，列表最后一个元素的离列表头的偏移，以及
#define ZIPLIST_HEADER_SIZE     (sizeof(uint32_t)*2+sizeof(uint16_t))
//ziplist的列表尾大小，包括1个8 bits整数，表示列表结束。
#define ZIPLIST_END_SIZE        (sizeof(uint8_t))
//ziplist的列表尾字节内容
#define ZIP_END 255
```

那么，在创建一个新的ziplist后，该列表的内存布局就如下图所示。注意，此时列表中还没有实际的数据。

然后，当我们往ziplist中插入数据时，ziplist就会根据数据是字符串还是整数，以及它们的大小进行不同的编码。这种根据数据大小进行相应编码的设计思想，正是Redis为了节省内存而采用的。

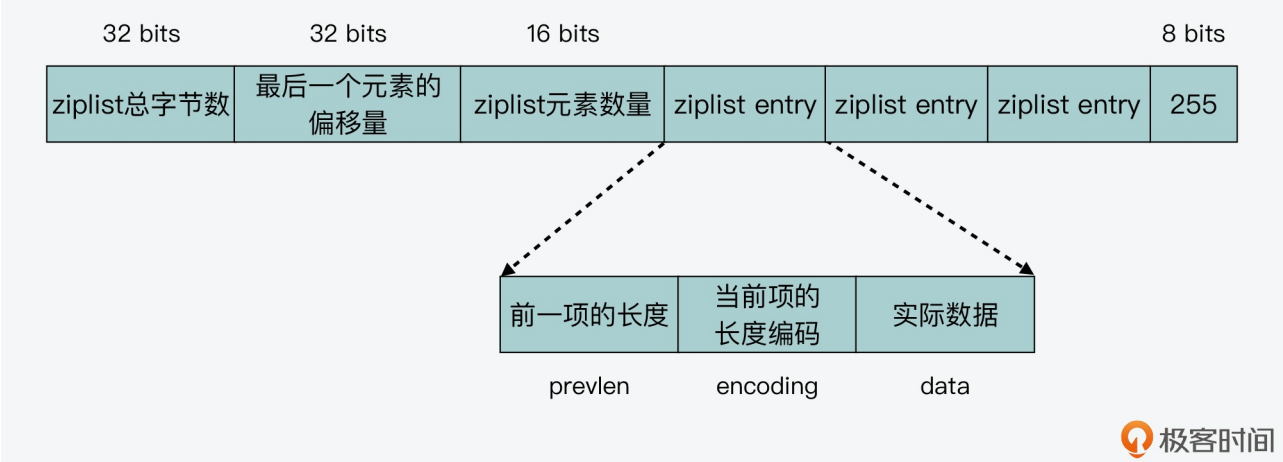
那么，ziplist是如何进行编码呢？要学习编码的实现，我们要先了解ziplist中列表项的结构。

ziplist列表项包括三部分内容，分别是**前一项的长度（prevlen）**、**当前项长度信息的编码结果（encoding）**，以及**当前项的实际数据（data）**。下面的图展示了列表项的结构（图中除列表项之外的内容分别是ziplist内存空间的起始和尾部）。



实际上，所谓的编码技术，就是指**用不同数量的字节来表示保存的信息**。在ziplist中，编码技术主要应用在列表项中的prevlen和encoding这两个元数据上。而当前项的实际数据data，则正常用整数或是字符串来表示。

所以这里，我们就先来看下**prevlen的编码设计**。ziplist中会包含多个列表项，每个列表项都是紧挨着彼此存放的，如下图所示。



而为了方便查找，每个列表项中都会记录前一项的长度。因为每个列表项的长度不一样，所以如果使用相同的字节大小来记录prevlen，就会造成内存空间浪费。

我给你举个例子，假设我们统一使用4字节记录prevlen，如果前一个列表项只是一个字符串“redis”，长度为5个字节，那么我们用1个字节（8 bits）就能表示256字节长度（2的8次方等于256）的字符串了。此时，prevlen用4字节记录，其中就有3字节是浪费掉了。

好，我们再回过头来看，ziplist在对prevlen编码时，会先调用**zipStorePrevEntryLength函数**，用于判断前一个列表项是否小于254字节。如果是的话，那么prevlen就使用1字节表示；否则，zipStorePrevEntryLength函数就调用zipStorePrevEntryLengthLarge函数进一步编码。这部分代码如下所示：

```
//判断prevlen的长度是否小于ZIP_BIG_PREVLEN，ZIP_BIG_PREVLEN等于254
if (len < ZIP_BIG_PREVLEN) {
   //如果小于254字节，那么返回prevlen为1字节
   p[0] = len;
   return 1;
} else {
   //否则，调用zipStorePrevEntryLengthLarge进行编码
   return zipStorePrevEntryLengthLarge(p,len);
}
```

也就是说，**zipStorePrevEntryLengthLarge函数**会先将prevlen的第1字节设置为254，然后使用内存拷贝函数memcpy，将前一个列表项的长度值拷贝至prevlen的第2至第5字节。最后，zipStorePrevEntryLengthLarge函数返回prevlen的大小，为5字节。

```
if (p != NULL) {
    //将prevlen的第1字节设置为ZIP_BIG_PREVLEN，即254
    p[0] = ZIP_BIG_PREVLEN;
 //将前一个列表项的长度值拷贝至prevlen的第2至第5字节，其中sizeof(len)的值为4
    memcpy(p+1,&len,sizeof(len));
    …
}
//返回prevlen的大小，为5字节
return 1+sizeof(len);
```

好，在了解了prevlen使用1字节和5字节两种编码方式后，我们再来学习下**encoding的编码方法**。

我们知道，一个列表项的实际数据，既可以是整数也可以是字符串。整数可以是16、32、64等字节长度，同时字符串的长度也可以大小不一。

所以，ziplist在zipStoreEntryEncoding函数中，针对整数和字符串，就分别使用了不同字节长度的编码结果。下面的代码展示了zipStoreEntryEncoding函数的部分代码，你可以看到当数据是不同长度字符串或是整数时，编码结果的长度len大小不同。

```
//默认编码结果是1字节
 unsigned char len = 1;
 //如果是字符串数据
 if (ZIP_IS_STR(encoding)) {
```

```
     //字符串长度小于等于63字节（16进制为0x3f）
        if (rawlen <= 0x3f) {
            //默认编码结果是1字节
            …
        }
    //字符串长度小于等于16383字节（16进制为0x3fff）
        else if (rawlen <= 0x3fff) {
            //编码结果是2字节
            len += 1;
            …
        }
    //字符串长度大于16383字节

        else {
            //编码结果是5字节
            len += 4;
            …
        }
    } else {
        /* 如果数据是整数，编码结果是1字节*/
        if (!p) return len;
        ...
    }
```

简而言之，针对不同长度的数据，使用不同大小的元数据信息（prevlen和encoding），这种方法可以有效地节省内存开销。当然，除了ziplist之外，Redis还设计了一个内存友好的数据结构，这就是**整数集合（intset）**，它是作为底层结构来实现Set数据类型的。

和SDS嵌入式字符串、ziplist类似，整数集合也是一块连续的内存空间，这一点我们从整数集合的定义中就可以看到。intset.h和intset.c分别包括了整数集合的定义和实现。

下面的代码展示了intset的结构定义。我们可以看到，整数集合结构体中记录数据的部分，就是一个int8_t类型的整数数组contents。从内存使用的角度来看，整数数组就是一块连续内存空间，所以这样就避免了内存碎片，并提升了内存使用效率。

```
typedef struct intset {
    uint32_t encoding;
    uint32_t length;
    int8_t contents[];
} intset;
```

好了，到这里，我们就已经了解了Redis针对内存开销所做的数据结构优化，分别是SDS嵌入式字符串、压缩列表和整数集合。

而除了对数据结构做优化，Redis在数据访问上，也会尽量节省内存开销，接下来我们就一起来学习下。

节省内存的数据访问

我们知道，在Redis实例运行时，有些数据是会被经常访问的，比如常见的整数，Redis协议中常见的回复信息，包括操作成功（“OK”字符串）、操作失败（ERR），以及常见的报错信息。

所以，为了避免在内存中反复创建这些经常被访问的数据，Redis就采用了**共享对象**的设计思想。这个设计思想很简单，就是把这些常用数据创建为共享对象，当上层应用需要访问它们时，直接读取就行。

现在我们就来做个假设。有1000个客户端，都要保存“3”这个整数。如果Redis为每个客户端，都创建了一个值为3的redisObject，那么内存中就会有大量的冗余。而使用了共享对象方法后，Redis在内存中只用保存一个3的redisObject就行，这样就有效节省了内存空间。

以下代码展示的是server.c文件中，**创建共享对象的函数createSharedObjects**，你可以看下。

```
void createSharedObjects(void) {
   …
   //常见回复信息
   shared.ok = createObject(OBJ_STRING,sdsnew("+OK\r\n"));
   shared.err = createObject(OBJ_STRING,sdsnew("-ERR\r\n"));
   …
   //常见报错信息
 shared.nokeyerr = createObject(OBJ_STRING,sdsnew("-ERR no such key\r\n"));
 shared.syntaxerr = createObject(OBJ_STRING,sdsnew("-ERR syntax error\r\n"));
   //0到9999的整数
   for (j = 0; j < OBJ_SHARED_INTEGERS; j++) {
        shared.integers[j] =
          makeObjectShared(createObject(OBJ_STRING,(void*)(long)j));
        …
    }
   …
}
```

小结

降低内存开销，对于Redis这样的内存数据库来说非常重要。今天这节课，我们了解了Redis用于优化内存使用效率的两种方法：内存优化的数据结构设计和节省内存的共享数据访问。

那么，对于实现数据结构来说，如果想要节省内存，Redis就给我们提供了两个优秀的设计思想：一个是**使用连续的内存空间**，避免内存碎片开销；二个是**针对不同长度的数据，采用不同大小的元数据**，以避免使用统一大小的元数据，造成内存空间的浪费。

另外在数据访问方面，你也要知道，**使用共享对象**其实可以避免重复创建冗余的数据，从而也可以有效地节省内存空间。不过，共享对象主要适用于**只读场景**，如果一个字符串被反复地修改，就无法被多个请求共享访问了。所以这一点，你在应用时也需要注意一下。

每课一问

SDS判断是否使用嵌入式字符串的条件是44字节，你知道为什么是44字节吗？

欢迎在留言区分享你的思考过程，我们一起交流讨论。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。

精选留言：

- Kaito 2021-08-03 02:47:42

1、要想理解 Redis 数据类型的设计，必须要先了解 redisObject。

Redis 的 key 是 String 类型，但 value 可以是很多类型（String/List/Hash/Set/ZSet等），所以 Redis 要想存储多种数据类型，就要设计一个通用的对象进行封装，这个对象就是 redisObject。

```
// server.h
typedef struct redisObject {
unsigned type:4;
unsigned encoding:4;
unsigned lru:LRU_BITS;
int refcount;
void *ptr;
} robj;
```

其中，最重要的 2 个字段：

- type：面向用户的数据类型（String/List/Hash/Set/ZSet等）
- encoding：每一种数据类型，可以对应不同的底层数据结构来实现（SDS/ziplist/intset/hashtable/skiplist等）

例如 String，可以用 embstr（嵌入式字符串，redisObject 和 SDS 一起分配内存），也可以用 rawstr（redisObject 和 SDS 分开存储）实现。

又或者，当用户写入的是一个「数字」时，底层会转成 long 来存储，节省内存。

同理，Hash/Set/ZSet 在数据量少时，采用 ziplist 存储，否则就转为 hashtable 来存。

所以，redisObject 的作用在于：

1) 为多种数据类型提供统一的表示方式
2) 同一种数据类型，底层可以对应不同实现，节省内存
3）支持对象共享和引用计数，共享对象存储一份，可多次使用，节省内存

redisObject 更像是连接「上层数据类型」和「底层数据结构」之间的桥梁。

2、关于 String 类型的实现，底层对应 3 种数据结构：

- embstr：小于 44 字节，嵌入式存储，redisObject 和 SDS 一起分配内存，只分配 1 次内存
- rawstr：大于 44 字节，redisObject 和 SDS 分开存储，需分配 2 次内存
- long：整数存储（小于 10000，使用共享对象池存储，但有个前提：Redis 没有设置淘汰策略，详见 object.c 的 tryObjectEncoding 函数）

3、ziplist 的特点：

1) 连续内存存储：每个元素紧凑排列，内存利用率高
2) 变长编码：存储数据时，采用变长编码（满足数据长度的前提下，尽可能少分配内存）
3）寻找元素需遍历：存放太多元素，性能会下降（适合少量数据存储）
4) 级联更新：更新、删除元素，会引发级联更新（因为内存连续，前面数据膨胀/删除了，后面要跟着一起动）

List、Hash、Set、ZSet 底层都用到了 ziplist。

4、intset 的特点：

1) Set 存储如果都是数字，采用 intset 存储
2) 变长编码：数字范围不同，intset 会选择 int16/int32/int64 编码（intset.c 的 _intsetValueEncoding 函数）
3）有序：intset 在存储时是有序的，这意味着查找一个元素，可使用「二分查找」（intset.c 的 intsetSearch 函数）
4) 编码升级/降级：添加、更新、删除元素，数据范围发生变化，会引发编码长度升级或降级

课后题：SDS 判断是否使用嵌入式字符串的条件是 44 字节，你知道为什么是 44 字节吗？

嵌入式字符串会把 redisObject 和 SDS 一起分配内存，那在存储时结构是这样的：

- redisObject：16 个字节
- SDS：sdshdr8（3 个字节）+ SDS 字符数组（N 字节 + \0 结束符 1 个字节）

Redis 规定嵌入式字符串最大以 64 字节存储，所以 N = 64 - 16(redisObject) - 3(sdshr8) - 1(\0)， N = 44 字节。 [20赞]

- Darren 2021-08-03 11:42:03
Kaito大佬描述的已经很详细了，44是因为 N = 64 - 16(redisObject) - 3(sdshr8) - 1(\0)， N = 44 字节。那么为什么是64减呢，为什么不是别的，因为在目前的x86体系下，一般的缓存行大小是64字节，redis为了一次能加载完成，因此采用64自己作为embstr类型(保存redisObject)的最大长度。 [5赞]

- 曾轼麟 2021-08-03 11:09:51
先回答老师的问题：为什么嵌入式字符串是以44字节为边界？

在了解这个问题之前，我们来了解一下jemalloc 分配内存机制，jemalloc 为了减少分配的内存空间大小不是2的幂次，在每次分配内存的时候都会返回2的幂次的空间大小，比如我需要分配5字节空间，jemalloc 会返回8字节，15字节会返回16字节。其常见的分配空间大小有： 8, 16, 32, 64, ..., 2kb, 4kb, 8kb。

但是这种方式也可能会造成，空间的浪费，比如我需要33字节，结果给我64字节，为了解决这个问题jemalloc将内存分配划分为，小内存（small_class）和大内存（large_class）通过不同的内存大小使用不同阶级策略，比如小内存允许存在48字节等方式。

Redis的嵌入式字符串，头部空间大小（redisObject + sdshdr8 + 1）已经去到了20字节，为了仍然能够满足jemalloc的64字节范围(48的太小了)，所以限制为44字节大小

此外总结一下阅读本文后的理解：

redis为了充分提高内存利用率，从几个方面入手：
1、淘汰不在使用的内存空间（后面章节会详细说明）
2、紧凑型的内存设计
3、实例内存共享

在为了提高内存利用率，redis做出了以下努力：

1、设计实现了SDS
2、设计实现了ziplist
3、设计实现了intset
4、搭配redisObject设计了嵌入式字符串
5、设计了共享对象（共享内存大部是常量实例）

此外补充一下老师文章中的内容，ziplist虽然能带来内存的节省，但是本质上是时间换空间的结果，当插入或者删除元素的时候由于内存使用率的变化，每次都有可能导致previous_entry_length 等字段需要扩展/缩小字节大小，从而导致一种现象【连锁更新】，就是每次更新或者删除的时候都要取重新修改head中的字节大小，从而带来性能开销，当然这种情况比较极端基本上不会触发。 [4赞]

- 可怜大灰狼 2021-08-03 11:16:36
原先embstr的限制长度是39，现在提升到了44，还是归功于sdshdr变成sdshdr8，头减少了5字节。
注释：The current limit of 44 is chosen so that the biggest string object. we allocate as EMBSTR will s till fit into the 64 byte arena of jemalloc. [2赞]

- lzh2nix 2021-08-07 09:21:02
内存友好可以从以下三个方面考虑：

1. 内存碎片的优化。redis作为内存大户使用了对内存碎片更友好内存分配器(jemalloc)

2. 内部数据结构对内存做很多的特殊优化
1. 使用更利于内存的数据结构，在变大之后又可以自动切换到其他数据结构
a. sds在初始化时使用最小的数据结构随着变大可以扩大到sdshdr8/sdshdr16/sdshdr32/sdshdr64
b. hash/set优先使用zipilst，之后扩展为skiplist
2. 对小字符符串在结构体中直接内嵌字符串的方式来避免一次内存分配
3. 尽量使用连续内存(ziplist/intset),首先避免了避免内存碎片，这种数据结构对排序也很友好

3. 针对公共对象使用sharedObject来避免相同对象的多次内存分配

- 悟空聊架构 2021-08-06 17:27:24
这篇确实难度有增加，涉及到了内存分配，而且对于 C 语言的功底也是有要求的。

课后题，44 怎么来的，objet.c 文件中有定义：
* The current limit of 44 is chosen so that the biggest string object
* we allocate as EMBSTR will still fit into the 64 byte arena of jemalloc. */

就是说用 64 字节存放连续的内存空间，这个内存空间包含以下部分：
redisObject
SDS 结构头
sdshdr8
字符串大小
1 字节空字符

而这个字符串大小就是我们要找的 44 字节。

下面看下怎么算出来的：

RedisObject 占用 16 字节。由4个部分组成：
type 占用 半个字节

encoding 占用半个字节
LRU_BITS占用 3 个字节
*ptr 占用 8个字节

sdshdr8 占用 3 字节
还有一个空字符 占用 1 字节

64-16-3-1=44

- 阿梵杰～ 2021-08-04 23:05:23
  而 sh+1 表示把内存地址从 sh 起始地址开始移动一定的大小，移动的距离等于 sdshdr8 结构体的大小。

  o->ptr = sh+1;

  【请教】为啥 +1 移动的距离就等于 sdshdr8 结构体的大小呢？ 有大佬赐教下吗？？

- 董宗磊 2021-08-04 11:41:39
  这个专栏的难度感觉有点高了，有些对着代码也看不太懂，更不要说能记住了。老师有什么好的学习思路建议吗?