11-“万金油”的String，为什么不好用了？

你好，我是蒋德钧。

从今天开始，我们就要进入“实践篇”了。接下来，我们会用5节课的时间学习“数据结构”。我会介绍节省内存开销以及保存和统计海量数据的数据类型及其底层数据结构，还会围绕典型的应用场景（例如地址位置查询、时间序列数据库读写和消息队列存取），跟你分享使用Redis的数据类型和module扩展功能来满足需求的具体方案。

今天，我们先了解下String类型的内存空间消耗问题，以及选择节省内存开销的数据类型的解决方案。

先跟你分享一个我曾经遇到的需求。

当时，我们要开发一个图片存储系统，要求这个系统能快速地记录图片ID和图片在存储系统中保存时的ID（可以直接叫作图片存储对象ID）。同时，还要能够根据图片ID快速查找到图片存储对象ID。

因为图片数量巨大，所以我们就用10位数来表示图片ID和图片存储对象ID，例如，图片ID为1101000051，它在存储系统中对应的ID号是3301000051。

```
photo_id: 1101000051
photo_obj_id: 3301000051
```

可以看到，图片ID和图片存储对象ID正好一一对应，是典型的“键-单值”模式。所谓的“单值”，就是指键值对中的值就是一个值，而不是一个集合，这和String类型提供的“一个键对应一个值的数据”的保存形式刚好契合。

而且，String类型可以保存二进制字节流，就像“万金油”一样，只要把数据转成二进制字节数组，就可以保存了。

所以，我们的第一个方案就是用String保存数据。我们把图片ID和图片存储对象ID分别作为键值对的key和value来保存，其中，图片存储对象ID用了String类型。

刚开始，我们保存了1亿张图片，大约用了6.4GB的内存。但是，随着图片数据量的不断增加，我们的Redis内存使用量也在增加，结果就遇到了大内存Redis实例因为生成RDB而响应变慢的问题。很显然，String类型并不是一种好的选择，我们还需要进一步寻找能节省内存开销的数据类型方案。

在这个过程中，我深入地研究了String类型的底层结构，找到了它内存开销大的原因，对“万金油”的String类型有了全新的认知：String类型并不是适用于所有场合的，它有一个明显的短板，就是它保存数据时所消耗的内存空间较多。

同时，我还仔细研究了集合类型的数据结构。我发现，集合类型有非常节省内存空间的底层实现结构，但是，集合类型保存的数据模式，是一个键对应一系列值，并不适合直接保存单值的键值对。所以，我们就使用二级编码的方法，实现了用集合类型保存单值键值对，Redis实例的内存空间消耗明显下降了。

这节课，我就把在解决这个问题时学到的经验和方法分享给你，包括String类型的内存空间消耗在哪儿了、用什么数据结构可以节省内存，以及如何用集合类型保存单值键值对。如果你在使用String类型时也遇到了内存空间消耗较多的问题，就可以尝试下今天的解决方案了。

接下来，我们先来看看String类型的内存都消耗在哪里了。

为什么String类型内存开销大？

在刚才的案例中，我们保存了1亿张图片的信息，用了约6.4GB的内存，一个图片ID和图片存储对象ID的记录平均用了64字节。

但问题是，一组图片ID及其存储对象ID的记录，实际只需要16字节就可以了。

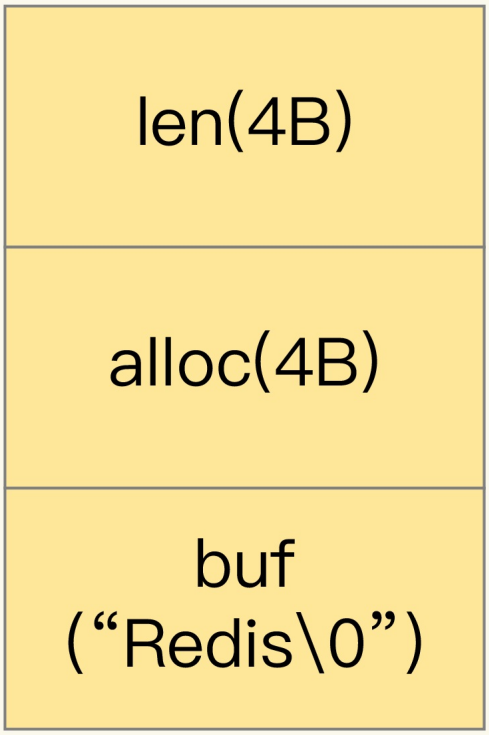
我们来分析一下。图片ID和图片存储对象ID都是10位数，我们可以用两个8字节的Long类型表示这两个ID。因为8字节的Long类型最大可以表示2的64次方的数值，所以肯定可以表示10位数。但是，为什么String类型却用了64字节呢？

其实，除了记录实际数据，String类型还需要额外的内存空间记录数据长度、空间使用等信息，这些信息也叫作元数据。当实际保存的数据较小时，元数据的空间开销就显得比较大了，有点“喧宾夺主”的意思。

那么，String类型具体是怎么保存数据的呢？我来解释一下。

当你保存64位有符号整数时，String类型会把它保存为一个8字节的Long类型整数，这种保存方式通常也叫作int编码方式。

但是，当你保存的数据中包含字符时，String类型就会用简单动态字符串（Simple Dynamic String，SDS）结构体来保存，如下图所示：



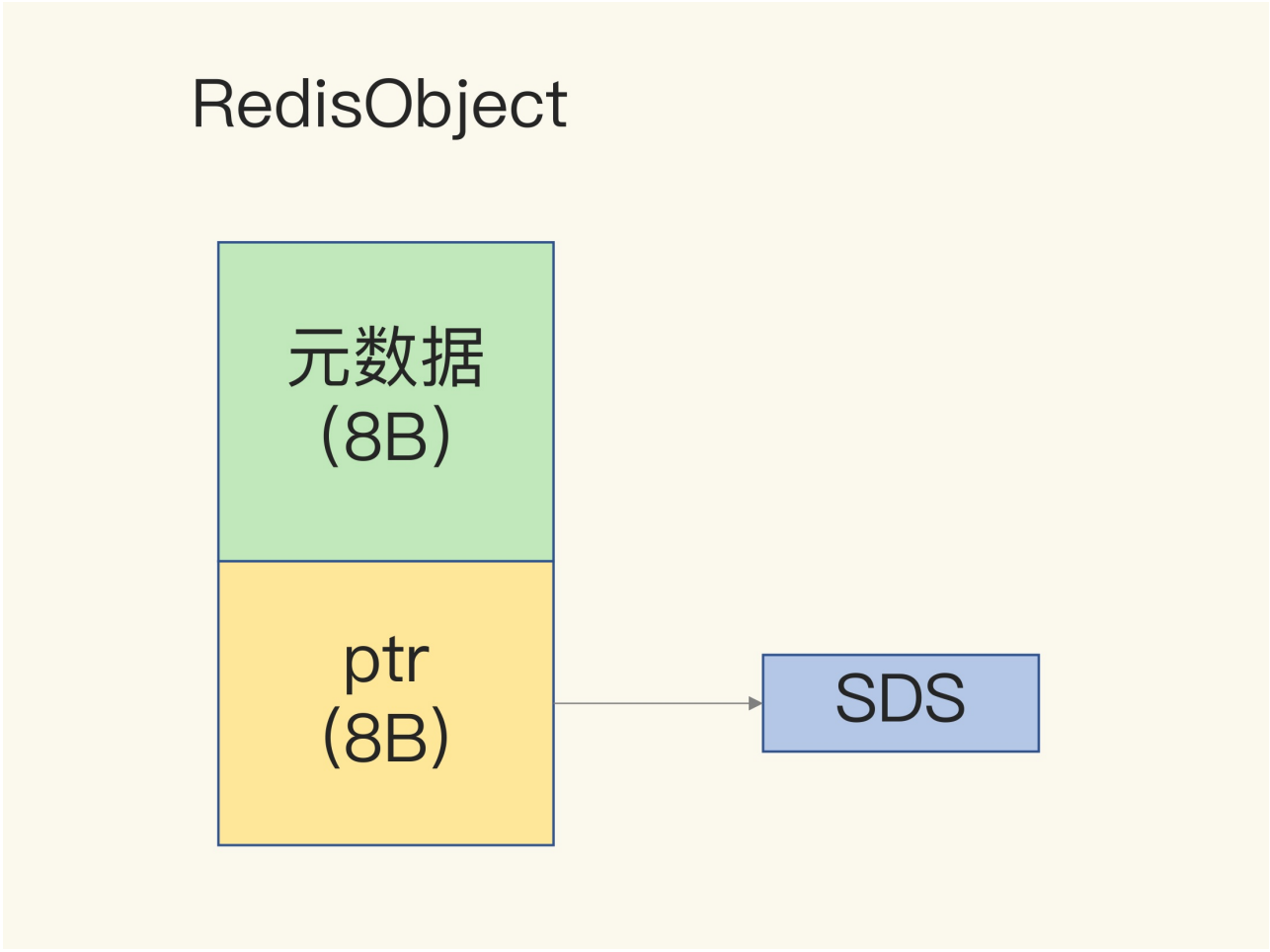
- **buf**：字节数组，保存实际数据。为了表示字节数组的结束，Redis会自动在数组最后加一个“\0”，这就会额外占用1个字节的开销。
- **len**：占4个字节，表示buf的已用长度。
- **alloc**：也占个4字节，表示buf的实际分配长度，一般大于len。

可以看到，在SDS中，buf保存实际数据，而len和alloc本身其实是SDS结构体的额外开销。

另外，对于String类型来说，除了SDS的额外开销，还有一个来自于RedisObject结构体的开销。

因为Redis的数据类型有很多，而且，不同数据类型都有些相同的元数据要记录（比如最后一次访问的时间、被引用的次数等），所以，Redis会用一个RedisObject结构体来统一记录这些元数据，同时指向实际数据。

一个RedisObject包含了8字节的元数据和一个8字节指针，这个指针再进一步指向具体数据类型的实际数据所在，例如指向String类型的SDS结构所在的内存地址，可以看一下下面的示意图。关于RedisObject的具体结构细节，我会在后面的课程中详细介绍，现在你只要了解它的基本结构和元数据开销就行了。



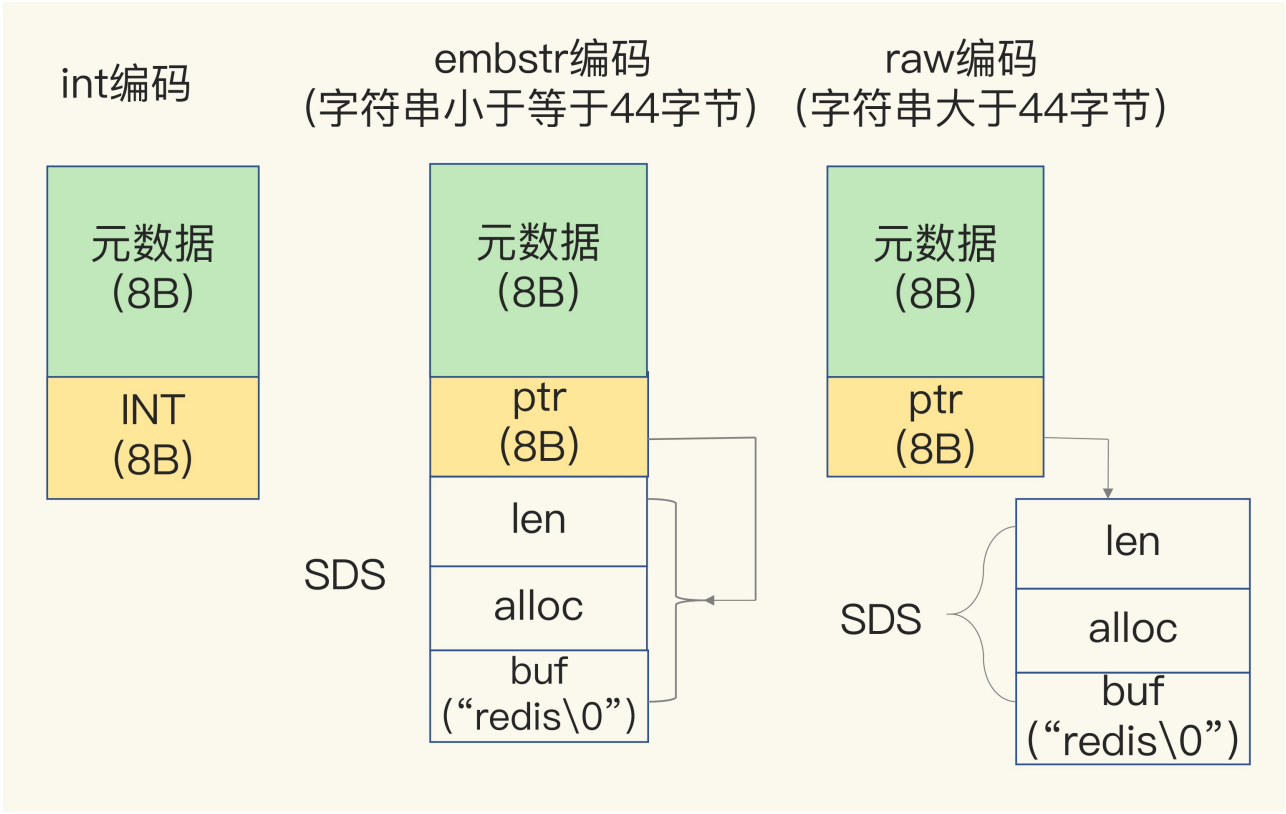
为了节省内存空间，Redis还对Long类型整数和SDS的内存布局做了专门的设计。

一方面，当保存的是Long类型整数时，RedisObject中的指针就直接赋值为整数数据了，这样就不用额外的指针再指向整数了，节省了指针的空间开销。

另一方面，当保存的是字符串数据，并且字符串小于等于44字节时，RedisObject中的元数据、指针和SDS是一块连续的内存区域，这样就可以避免内存碎片。这种布局方式也被称为embstr编码方式。

当然，当字符串大于44字节时，SDS的数据量就开始变多了，Redis就不再把SDS和RedisObject布局在一起了，而是会给SDS分配独立的空间，并用指针指向SDS结构。这种布局方式被称为raw编码模式。

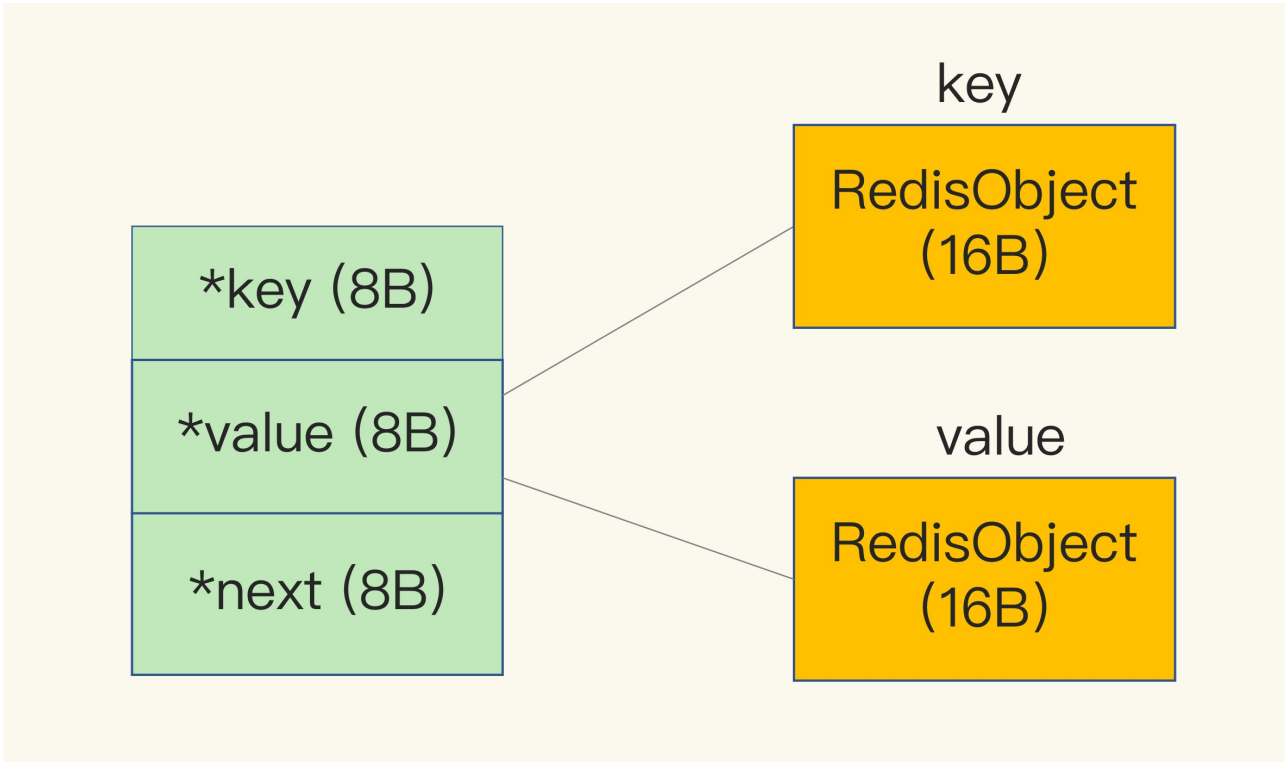
为了帮助你理解int、embstr和raw这三种编码模式，我画了一张示意图，如下所示：



好了，知道了RedisObject所包含的额外元数据开销，现在，我们就可以计算String类型的内存使用量了。

因为10位数的图片ID和图片存储对象ID是Long类型整数，所以可以直接用int编码的RedisObject保存。每个int编码的RedisObject元数据部分占8字节，指针部分被直接赋值为8字节的整数了。此时，每个ID会使用16字节，加起来一共是32字节。但是，另外的32字节去哪儿了呢？

我在第2讲中说过，Redis会使用一个全局哈希表保存所有键值对，哈希表的每一项是一个dictEntry的结构体，用来指向一个键值对。dictEntry结构中有三个8字节的指针，分别指向key、value以及下一个dictEntry，三个指针共24字节，如下图所示：



但是，这三个指针只有24字节，为什么会占用了32字节呢？这就要提到Redis使用的内存分配库jemalloc

了。

jemalloc在分配内存时，会根据我们申请的字节数N，找一个比N大，但是最接近N的2的幂次数作为分配的空间，这样可以减少频繁分配的次数。

举个例子。如果你申请6字节空间，jemalloc实际会分配8字节空间；如果你申请24字节空间，jemalloc则会分配32字节。所以，在我们刚刚说的场景里，dictEntry结构就占用了32字节。

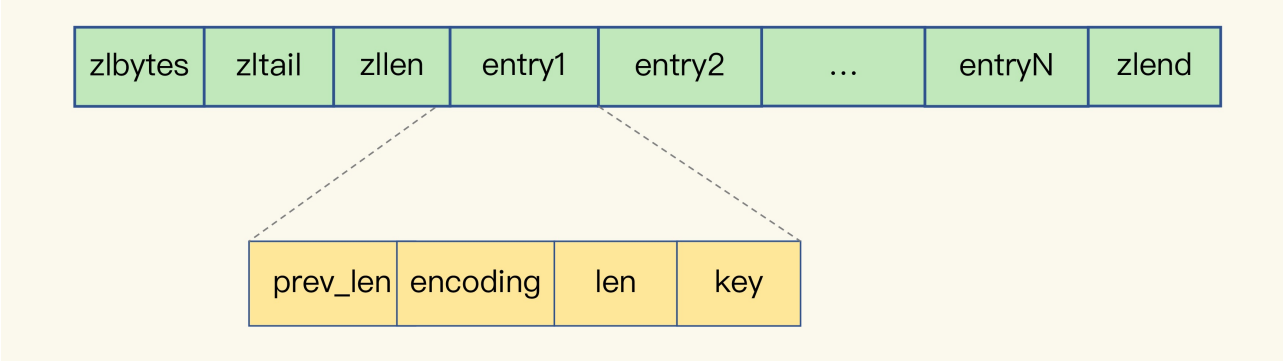
好了，到这儿，你应该就能理解，为什么用String类型保存图片ID和图片存储对象ID时需要用64个字节了。

你看，明明有效信息只有16字节，使用String类型保存时，却需要64字节的内存空间，有48字节都没有用于保存实际的数据。我们来换算下，如果要保存的图片有1亿张，那么1亿条的图片ID记录就需要6.4GB内存空间，其中有4.8GB的内存空间都用来保存元数据了，额外的内存空间开销很大。那么，有没有更加节省内存的方法呢?

用什么数据结构可以节省内存?

Redis有一种底层数据结构，叫压缩列表（ziplist），这是一种非常节省内存的结构。

我们先回顾下压缩列表的构成。表头有三个字段zlbytes、zltail和zllen，分别表示列表长度、列表尾的偏移量，以及列表中的entry个数。压缩列表尾还有一个zlend，表示列表结束。



压缩列表之所以能节省内存，就在于它是用一系列连续的entry保存数据。每个entry的元数据包括下面几部分。

- **prev_len**，表示前一个entry的长度。prev_len有两种取值情况：1字节或5字节。取值1字节时，表示上一个entry的长度小于254字节。虽然1字节的值能表示的数值范围是0到255，但是压缩列表中zlend的取值默认是255，因此，就默认用255表示整个压缩列表的结束，其他表示长度的地方就不能再用255这个值了。所以，当上一个entry长度小于254字节时，prev_len取值为1字节，否则，就取值为5字节。
- **len**：表示自身长度，4字节；
- **encoding**：表示编码方式，1字节；
- **content**：保存实际数据。

这些entry会挨个儿放置在内存中，不需要再用额外的指针进行连接，这样就可以节省指针所占用的空间。

我们以保存图片存储对象ID为例，来分析一下压缩列表是如何节省内存空间的。

每个entry保存一个图片存储对象ID（8字节），此时，每个entry的prev_len只需要1个字节就行，因为每个entry的前一个entry长度都只有8字节，小于254字节。这样一来，一个图片的存储对象ID所占用的内存大小是14字节（1+4+1+8=14），实际分配16字节。

Redis基于压缩列表实现了List、Hash和Sorted Set这样的集合类型，这样做的最大好处就是节省了dictEntry的开销。当你用String类型时，一个键值对就有一个dictEntry，要用32字节空间。但采用集合类型时，一个key就对应一个集合的数据，能保存的数据多了很多，但也只用了一个dictEntry，这样就节省了内存。

这个方案听起来很好，但还存在一个问题：在用集合类型保存键值对时，一个键对应了一个集合的数据，但是在我们的场景中，一个图片ID只对应一个图片的存储对象ID，我们该怎么用集合类型呢？换句话说，在一个键对应一个值（也就是单值键值对）的情况下，我们该怎么用集合类型来保存这种单值键值对呢？

如何用集合类型保存单值的键值对？

在保存单值的键值对时，可以采用基于Hash类型的二级编码方法。这里说的二级编码，就是把一个单值的数据拆分成两部分，前一部分作为Hash集合的key，后一部分作为Hash集合的value，这样一来，我们就可以把单值数据保存到Hash集合中了。

以图片ID 1101000060和图片存储对象ID 3302000080为例，我们可以把图片ID的前7位（1101000）作为Hash类型的键，把图片ID的最后3位（060）和图片存储对象ID分别作为Hash类型值中的key和value。

按照这种设计方法，我在Redis中插入了一组图片ID及其存储对象ID的记录，并且用info命令查看了内存开销，我发现，增加一条记录后，内存占用只增加了16字节，如下所示：

```
127.0.0.1:6379> info memory
# Memory
used_memory:1039120
127.0.0.1:6379> hset 1101000 060 3302000080
(integer) 1
127.0.0.1:6379> info memory
# Memory
used_memory:1039136
```

在使用String类型时，每个记录需要消耗64字节，这种方式却只用了16字节，所使用的内存空间是原来的1/4，满足了我们节省内存空间的需求。

不过，你可能也会有疑惑：“二级编码一定要把图片ID的前7位作为Hash类型的键，把最后3位作为Hash类型值中的key吗？”**其实，二级编码方法中采用的ID长度是有讲究的**。

在第2讲中，我介绍过Redis Hash类型的两种底层实现结构，分别是压缩列表和哈希表。

那么，Hash类型底层结构什么时候使用压缩列表，什么时候使用哈希表呢？其实，Hash类型设置了用压缩列表保存数据时的两个阈值，一旦超过了阈值，Hash类型就会用哈希表来保存数据了。

这两个阈值分别对应以下两个配置项：

- hash-max-ziplist-entries：表示用压缩列表保存时哈希集合中的最大元素个数。
- hash-max-ziplist-value：表示用压缩列表保存时哈希集合中单个元素的最大长度。

如果我们往Hash集合中写入的元素个数超过了hash-max-ziplist-entries，或者写入的单个元素大小超过了hash-max-ziplist-value，Redis就会自动把Hash类型的实现结构由压缩列表转为哈希表。

一旦从压缩列表转为了哈希表，Hash类型就会一直用哈希表进行保存，而不会再转回压缩列表了。在节省内存空间方面，哈希表就没有压缩列表那么高效了。

为了能充分使用压缩列表的精简内存布局，我们一般要控制保存在Hash集合中的元素个数。所以，在刚才的二级编码中，我们只用图片ID最后3位作为Hash集合的key，也就保证了Hash集合的元素个数不超过1000，同时，我们把hash-max-ziplist-entries设置为1000，这样一来，Hash集合就可以一直使用压缩列表来节省内存空间了。

小结

这节课，我们打破了对String的认知误区，以前，我们认为String是“万金油”，什么场合都适用，但是，在保存的键值对本身占用的内存空间不大时（例如这节课里提到的的图片ID和图片存储对象ID），String类型的元数据开销就占据主导了，这里面包括了RedisObject结构、SDS结构、dictEntry结构的内存开销。

针对这种情况，我们可以使用压缩列表保存数据。当然，使用Hash这种集合类型保存单值键值对的数据时，我们需要将单值数据拆分成两部分，分别作为Hash集合的键和值，就像刚才案例中用二级编码来表示图片ID，希望你能把这个方法用到自己的场景中。

最后，我还想再给你提供一个小方法：如果你想知道键值对采用不同类型保存时的内存开销，可以在这个网址里输入你的键值对长度和使用的数据类型，这样就能知道实际消耗的内存大小了。建议你把这个小工具用起来，它可以帮助你充分地节省内存。

每课一问

按照惯例，给你提个小问题：除了String类型和Hash类型，你觉得，还有其他合适的类型可以应用在这节课所说的保存图片的例子吗？

欢迎在留言区写下你的思考和答案，我们一起交流讨论，也欢迎你把今天的内容分享给你的朋友。

精选留言：

- Kaito 2020-08-31 11:36:57
  保存图片的例子，除了用String和Hash存储之外，还可以用Sorted Set存储（勉强）。

  Sorted Set与Hash类似，当元素数量少于zset-max-ziplist-entries，并且每个元素内存占用小于zset-max-ziplist-value时，默认也采用ziplist结构存储。我们可以把zset-max-ziplist-entries参数设置为1000，这样Sorted Set默认就会使用ziplist存储了，member和score也会紧凑排列存储，可以节省内存空间。

  使用zadd 1101000 3302000080 060命令存储图片ID和对象ID的映射关系，查询时使用zscore 1101000 060获取结果。

  但是Sorted Set使用ziplist存储时的缺点是，这个ziplist是需要按照score排序的（为了方便zrange和zrevr

ange命令的使用），所以在插入一个元素时，需要先根据score找到对应的位置，然后把member和score插入进去，这也意味着Sorted Set插入元素的性能没有Hash高（这也是前面说勉强能用Sorte Set存储的原因）。而Hash在插入元素时，只需要将新的元素插入到ziplist的尾部即可，不需要定位到指定位置。

不管是使用Hash还是Sorted Set，当采用ziplist方式存储时，虽然可以节省内存空间，但是在查询指定元素时，都要遍历整个ziplist，找到指定的元素。所以使用ziplist方式存储时，虽然可以利用CPU高速缓存，但也不适合存储过多的数据（hash-max-ziplist-entries和zset-max-ziplist-entries不宜设置过大），否则查询性能就会下降比较厉害。整体来说，这样的方案就是时间换空间，我们需要权衡使用。

当使用ziplist存储时，我们尽量存储int数据，ziplist在设计时每个entry都进行了优化，针对要存储的数据，会尽量选择占用内存小的方式存储（整数比字符串在存储时占用内存更小），这也有利于我们节省Redis的内存。还有，因为ziplist是每个元素紧凑排列，而且每个元素存储了上一个元素的长度，所以当修改其中一个元素超过一定大小时，会引发多个元素的级联调整（前面一个元素发生大的变动，后面的元素都要重新排列位置，重新分配内存），这也会引发性能问题，需要注意。

另外，使用Hash和Sorted Set存储时，虽然节省了内存空间，但是设置过期变得困难（无法控制每个元素的过期，只能整个key设置过期，或者业务层单独维护每个元素过期删除的逻辑，但比较复杂）。而使用String虽然占用内存多，但是每个key都可以单独设置过期时间，还可以设置maxmemory和淘汰策略，以这种方式控制整个实例的内存上限。

所以在选用Hash和Sorted Set存储时，意味着把Redis当做数据库使用，这样就需要务必保证Redis的可靠性（做好备份、主从副本），防止实例宕机引发数据丢失的风险。而采用String存储时，可以把Redis当做缓存使用，每个key设置过期时间，同时设置maxmemory和淘汰策略，控制整个实例的内存上限，这种方案需要在数据库层（例如MySQL）也存储一份映射关系，当Redis中的缓存过期或被淘汰时，需要从数据库中重新查询重建缓存，同时需要保证数据库和缓存的一致性，这些逻辑也需要编写业务代码实现。

总之，各有利弊，我们需要根据实际场景进行选择。 [8赞]

- Geek1185 2020-08-31 09:48:18
  老师能否讲解一下hash表这种redis数据结构，底层在用压缩列表的时候是如何根据二级的键找到对应的值的呢。是一个entry里会同时保存键和值吗 [2赞]

- 伟伟哦 2020-08-31 11:34:26
  老师今天讲的可以给个代码，配置了选项 如何实现把图片 ID 的最后 3 位（060）和图片存储对象 ID 分别作为 Hash 类型值中的 key 和 value。 代码操作下 [1赞]

- MClink 2020-08-31 08:24:45
  老师，底层数据结构的转换是怎么实现的呢？是单纯的开一个新的数据结构再把数据复制过去吗？再释放之前的数据结构的内存，复制过程中有修改值的话要怎么处理，复制过程中不就两倍内存消耗了 [1赞]

- 慎独明强 2020-08-31 07:59:02
  看了Redis设计与实现，有讲SDS这一块，对于老师分析的内容，自己心里有印象，再结合老师今天的实践案例，前面的知识还没有吃透
  😂😂 [1赞]

- 一大只🐶 2020-08-31 18:10:05
  不懂就问，老师，以photo_id:1101000051为例，key（photo_id）是字符串，那应该按SDS算，那元数据（8）+ptr（8）+SDS（4+4+8+1）应该是33，key+value+hash表 应该是49+32=81

- Wangxi 2020-08-31 18:06:30
  实测老师的例子，长度7位数，共100万条数据。使用string占用70mb，使用hash ziplist只占用9mb。效果非常明显。redis版本6.0.6

- 拥有两个端点是线段 2020-08-31 14:16:34
  老师，在字符串长度不超过44字节时是使用embstr编码，那为何规定是44字节呢？

- 叶子。 2020-08-31 13:52:34
  我记得之前讲的是
  1）字典中保存的键和值的大小都小于64字节
  2）字典中键值对的个数小于512个
  这两个是配置的默认值吗，为什么这里又可以设置为1000呢？

- 土豆白菜 2020-08-31 13:46:27
  1101000 060 3302000080
  前七位相同的图片id放进一个hash,也就是说每个hash有999个key-value
  键是1101000
  key-value是:001-3302000080
  key-value是:002-.....
  .....
  key-value是:060-.....
  key-value是:062-.....
  .....
  key-value是:998-.....
  key-value是:999-.....

  下一个hash就是1101001了

- 可怜大灰狼 2020-08-31 13:41:34
  老师今天说的，在看源码时都发现了，很高兴。同时也看到redis在计算used_memory，使用了一个小手段来对齐8字节。
  zmalloc.c中有这样的代码：if (_n&(sizeof(long)-1)) _n += sizeof(long)-(_n&(sizeof(long)-1));

- Spring4J 2020-08-31 10:01:51
  老师，dictEntry中的next指针是指向全局哈希表中为了解决哈希冲突而生成的拉链的next吗？

- jinjunzhu 2020-08-31 09:24:34
  我觉得hash就是最好的保存方式了，列表、集合、有序集合也都可以存，但是查找的时间复杂度都高于hash

- 服务器宕机了 2020-08-31 08:11:28
  请问如果使用二级编码这种方式，为了一定程度减少hash冲突。是不是又需要适当改变图片ID的生成方式，比如加长或者加入字符串的形式

- Spring4J 2020-08-31 01:45:08
  压缩列表实现哈希类型的数据时，一个键值对应该占用压缩列表的两个entry吧老师？