

38 | bytes包与字节串操作（上）

2018-11-07 郝林



我相信，经过上一次的学习，你已经对`strings.Builder`和`strings.Reader`这两个类型足够熟悉了。

我上次还建议你去自行查阅`strings`代码包中的其他程序实体。如果你认真去看了，那么肯定会对我们今天要讨论的`bytes`代码包，有种似曾相识的感觉。

`strings`包和`bytes`包可以说是一对孪生兄弟，它们在API方面非常的相似。单从它们提供的函数的数量和功能上讲，差别微乎其微。

只不过，`strings`包主要面向的是Unicode字符和经过UTF-8编码的字符串，而`bytes`包面对的则主要是字节和字节切片。

我今天会主要讲`bytes`包中最有特色的类型`Buffer`。顾名思义，`bytes.Buffer`类型的用途主要是作为字节序列的缓冲区。

与`strings.Builder`类型一样，`bytes.Buffer`也是开箱即用的。但不同的是，`strings.Builder`只能拼接和导出字符串，而`bytes.Buffer`不但可以拼接、截断其中的字节序列，以各种形式导出其中的内容，还可以顺序地读取其中的子序列。

可以说，`bytes.Buffer`是集读、写功能于一身的数据类型。当然了，这些也基本上都是作为一个缓冲区应该拥有的功能。

在内部，`bytes.Buffer`类型同样也是使用字节切片作为内容容器的。并且，与`strings.Reader`类型类似，`bytes.Buffer`有一个`int`类型的字段，用于代表已读字节的计数，可以简称为已读计数。

不过，这里的已读计数就无法通过`bytes.Buffer`提供的方法计算出来了。

我们先来看下面的代码：

```
var buffer1 bytes.Buffer
contents := "Simple byte buffer for marshaling data."
fmt.Printf("Writing contents %q ...\n", contents)
buffer1.WriteString(contents)
fmt.Printf("The length of buffer: %d\n", buffer1.Len())
fmt.Printf("The capacity of buffer: %d\n", buffer1.Cap())
```

我先声明了一个`bytes.Buffer`类型的变量`buffer1`，并写入了一个字符串。然后，我想打印出这个`bytes.Buffer`类型的值（以下简称`Buffer`值）的长度和容量。在运行这段代码之后，我们将会看到如下的输出：

```
Writing contents "Simple byte buffer for marshaling data." ...
The length of buffer: 39
The capacity of buffer: 64
```

乍一看这没什么问题。长度39和容量64的含义看起来与我们已知的概念是一致的。我向缓冲区中写入了一个长度为39的字符串，所以`buffer1`的长度就是39。

根据切片的自动扩容策略，64这个数字也是合理的。另外，可以想象，这时的已读计数的值应该是0，这是因为我还没有调用任何用于读取其中内容的方法。

可实际上，与`strings.Reader`类型的`Len`方法一样，`buffer1`的`Len`方法返回的也是内容容器中未被读取部分的长度，而不是其中已存内容的总长度（以下简称内容长度）。示例如下：

```
p1 := make([]byte, 7)
n, _ := buffer1.Read(p1)
fmt.Printf("%d bytes were read. (call Read)\n", n)
fmt.Printf("The length of buffer: %d\n", buffer1.Len())
fmt.Printf("The capacity of buffer: %d\n", buffer1.Cap())
```

当我从`buffer1`中读取一部分内容，并用它们填满长度为7的字节切片`p1`之后，`buffer1`的`Len`方法返回的结果值也会随即发生变化。如果运行这段代码，我们会发现，这个缓冲区的长度已经变为了32。

另外，因为我们并没有再向该缓冲区中写入任何内容，所以它的容量会保持不变，仍是64。

总之，在这里，你需要记住的是，`Buffer`值的长度是未读内容的长度，而不是已存内容的总长度。它与在当前值之上的读操作和写操作都有关系，并会随着这两种操作的进行而改变，它可能会变得更小，也可能会变得更大。

而`Buffer`值的容量指的是它的内容容器（也就是那个字节切片）的容量，它只与在当前值之上的写操作有关，并会随着内容的写入而不断增长。

再说已读计数。由于`strings.Reader`还有一个`Size`方法可以给出内容长度的值，所以我们用内容长度减去未读部分的长度，就可以很方便地得到它的已读计数。

然而，`bytes.Buffer`类型却没有这样一个方法，它只有`Cap`方法。可是`Cap`方法提供的是内容容器的容量，也不是内容长度。

并且，这里的内容容器容量在很多时候都与内容长度不相同。因此，没有了现成的计算公式，只要遇到稍微复杂些的情况，我们就很难估算出`Buffer`值的已读计数。

一旦理解了已读计数这个概念，并且能够在读写的过程中，实时地获得已读计数和内容长度的值，我们就可以很直观地了解到当前`Buffer`值各种方法的行为了。不过，很可惜，这两个数字我们都无法直接拿到。

虽然，我们无法直接得到一个`Buffer`值的已读计数，并且有时候也很难估算它，但是我们绝对不能就此作罢，而应该通过研读`bytes.Buffer`和文档和源码，去探究已读计数在其中起到的关键作用。

否则，我们想用好`bytes.Buffer`的意愿，恐怕就不会那么容易实现了。

下面的这个问题，如果你认真地阅读了`bytes.Buffer`的源码之后，就可以很好地回答出来。

我们今天的问题是：`bytes.Buffer`类型的值记录的已读计数，在其中起到了怎样的作用？

这道题的典型回答是这样的。

`bytes.Buffer`中的已读计数的大致功用如下所示。

1. 读取内容时，相应方法会依据已读计数找到未读部分，并在读取后更新计数。

2. 写入内容时，如需扩容，相应方法会根据已读计数实现扩容策略。
3. 截断内容时，相应方法截掉的是已读计数代表索引之后的未读部分。
4. 读回退时，相应方法需要用已读计数记录回退点。
5. 重置内容时，相应方法会把已读计数置为0。
6. 导出内容时，相应方法只会导出已读计数代表的索引之后的未读部分。
7. 获取长度时，相应方法会依据已读计数和内容容器的长度，计算未读部分的长度并返回。

问题解析

通过上面的典型回答，我们已经能够体会到已读计数在`bytes.Buffer`类型，及其方法中的重要性了。没错，`bytes.Buffer`的绝大多数方法都用到了已读计数，而且都是非用不可。

在读取内容的时候，相应方法会先根据已读计数，判断一下内容容器中是否还有未读的内容。如果有，那么它就会从已读计数代表的索引处开始读取。

在读取完成后，它还会及时地更新已读计数。也就是说，它会记录一下又有多少个字节被读取了。**这里所说的相应方法包括了所有名称以Read开头的方法，以及Next方法和WriteTo方法。**

在写入内容的时候，绝大多数的相应方法都会先检查当前的内容容器，是否有足够的容量容纳新的内容。如果没有，那么它们就会对内容容器进行扩容。

在扩容的时候，方法会在必要时，依据已读计数找到未读部分，并把其中的内容拷贝到扩容后内容容器的头部位置。

然后，方法将会把已读计数的值置为0，以表示下一次读取需要从内容容器的第一个字节开始。**用于写入内容的相应方法，包括了所有名称以Write开头的方法，以及ReadFrom方法。**

用于截断内容的方法Truncate，会让很多对`bytes.Buffer`不太了解的程序开发者迷惑。它会接受一个`int`类型的参数，这个参数的值代表了：在截断时需要保留头部的多少个字节。

不过，需要注意的是，这里说的头部指的并不是内容容器的头部，而是其中的未读部分的头部。头部的起始索引正是由已读计数的值表示的。因此，在这种情况下，已读计数的值再加上参数值后得到的和，就是内容容器新的总长度。

在`bytes.Buffer`中，**用于读回退的方法有UnreadByte和UnreadRune**。这两个方法分别用于回退一个字节和回退一个Unicode字符。调用它们一般都是为了退回在上一次被读取内容末尾的那个分隔符，或者为重新读取前一个字节或字符做准备。

不过，退回的前提是，在调用它们之前的那一个操作必须是“读取”，并且是成功的读取，否则这

些方法就只能忽略后续操作并返回一个非nil的错误值。

UnreadByte方法的做法比较简单，把已读计数的值减1就好了。而UnreadRune方法需要从已读计数中减去的，是上一次被读取的Unicode字符所占用的字节数。

这个字节数由bytes.Buffer的另一个字段负责存储，它在这里的有效取值范围是[1, 4]。只有ReadRune方法才会把这个字段的值设定在此范围之内。

由此可见，只有紧接在调用ReadRune方法之后，对UnreadRune方法的调用才能够成功完成。该方法明显比UnreadByte方法的适用面更窄。

我在前面说过，bytes.Buffer的Len方法返回的是内容容器中未读部分的长度，而不是其中已存内容的总长度（即：内容长度）。

而该类型的Bytes方法和String方法的行为，与Len方法是保持一致的。前两个方法只会去访问未读部分中的内容，并返回相应的结果值。

在我们剖析了所有的相关方法之后，可以这样来总结：在已读计数代表的索引之前的那些内容，永远都是已经被读过的，它们几乎没有机会再次被读取。

不过，这些已读内容所在的内存空间可能会被存入新的内容。这一般都是由于重置或者扩充内容容器导致的。这时，已读计数一定会被置为0，从而再次指向内容容器中的第一个字节。这有时候也是为了避免内存分配和重用内存空间。

总结

总结一下，bytes.Buffer是一个集读、写功能于一身的数据类型。它非常适合作为字节序列的缓冲区。我们会在下一篇文章中继续对bytes.Buffer的知识进行延展。如果你对于这部分内容有什么样问题，欢迎给我留言，我们一起讨论。

感谢你的收听，我们下次再见。

[戳此查看Go语言专栏文章配套详细代码。](#)

GO语言核心36讲

3个月带你通关 GO 语言

郝林

《Go 并发编程实战》作者
GoHackers 技术社群发起人
前轻松筹大数据负责人

