

39 | bytes包与字节串操作（下）

2018-11-09 郝林



你好，我是郝林，今天我们继续分享bytes包与字节串操作的相关内容。

在上一篇文章中，我们分享了bytes.Buffer中已读计数的大致功用，并围绕着这个问题做了解析，下面我们来进行相关的知识扩展。

知识扩展

问题 1：bytes.Buffer的扩容策略是怎样的？

Buffer值既可以被手动扩容，也可以进行自动扩容。并且，这两种扩容方式的策略是基本一致的。所以，除非我们完全确定后续内容所需的字节数，否则让Buffer值自动去扩容就好了。

在扩容的时候，Buffer值中相应的代码（以下简称扩容代码）会先判断内容容器的剩余容量，是否可以满足调用方的要求，或者是否足够容纳新的内容。

如果可以，那么扩容代码会在当前的内容容器之上，进行长度扩充。更具体地说，如果内容容器的容量与其长度的差，大于或等于另需的字节数，那么扩容代码就会通过切片操作对原有的内容容器的长度进行扩充，就像下面这样：

```
b.buf = b.buf[:length+need]
```

反之，如果内容容器的剩余容量不够了，那么扩容代码可能就会用新的内容容器去替代原有的内容容器，从而实现扩容。不过，这里还进一步优化。

如果当前内容容器的容量的一半仍然大于或等于其现有长度再加上另需的字节数的和，即：

```
cap(b.buf)/2 >= len(b.buf)+need
```

那么，扩容代码就会复用现有的内容容器，并把容器中的未读内容拷贝到它的头部位置。这也意味着其中的已读内容，将会全部被未读内容和之后的新内容覆盖掉。

这样的复用预计可以至少节省掉一次后续的扩容所带来的内存分配，以及若干字节的拷贝。

若这一步优化未能达成，也就是说，当前内容容器的容量小于新长度的二倍，那么扩容代码就只能再创建一个新的内容容器，并把原有容器中的未读内容拷贝进去，最后再用新的容器替换掉原有的容器。这个新容器的容量将会等于原有容量的二倍再加上另需字节数的和。

新容器的容量=2*原有容量+所需字节数

通过上面这些步骤，对内容容器的扩充基本上就完成了。不过，为了内部数据的一致性，以及避免原有的已读内容可能造成的数据混乱，扩容代码还会把已读计数置为0，并再对内容容器做一下切片操作，以掩盖掉原有的已读内容。

顺便说一下，对于处在零值状态的Buffer值来说，如果第一次扩容时的另需字节数不大于64，那么该值就会基于一个预先定义好的、长度为64的字节数组来创建内容容器。

在这种情况下，这个内容容器的容量就是64。这样做的目的是为了让Buffer值在刚被真正使用的时候就可以快速地做好准备。

问题2：bytes.Buffer中的哪些方法可能会造成内容的泄露？

首先明确一点，什么叫内容泄露？这里所说的内容泄露是指，使用Buffer值的一方通过某种非标准的（或者说不正式的）方式得到了本不该得到的内容。

比如说，我通过调用Buffer值的某个用于读取内容的方法，得到了一部分未读内容。我应该，也只应该通过这个方法的结果值，拿到在那一时刻Buffer值中的未读内容。

但是，在这个Buffer值又有了一些新内容之后，我却可以通过当时得到的结果值，直接获得新的内容，而不需要再次调用相应的方法。

这就是典型的非标准读取方式。这种读取方式是不应该存在的，即使存在，我们也不应该使用。因为它是在无意中（或者说不小心）暴露出来的，其行为很可能是不稳定的。

在`bytes.Buffer`中，`Bytes`方法和`Next`方法都可能会造成内容的泄露。原因在于，它们都把基于内容容器的切片直接返回给了方法的调用方。

我们都知道，通过切片，我们可以直接访问和操纵它的底层数组。不论这个切片是基于某个数组得来的，还是通过对另一个切片做切片操作获得的，都是如此。

在这里，`Bytes`方法和`Next`方法返回的字节切片，都是通过对内容容器做切片操作得到的。也就是说，它们与内容容器共用了同一个底层数组，起码在一段时期之内是这样的。

以`Bytes`方法为例。它会返回在调用那一刻其所属值中的所有未读内容。示例代码如下：

```
contents := "ab"
buffer1 := bytes.NewBufferString(contents)
fmt.Printf("The capacity of new buffer with contents %q: %d\n",
    contents, buffer1.Cap()) // 内容容器的容量为：8。
unreadBytes := buffer1.Bytes()
fmt.Printf("The unread bytes of the buffer: %v\n", unreadBytes) // 未读内容为：[97 98]。
```

我用字符串值`"ab"`初始化了一个`Buffer`值，由变量`buffer1`代表，并打印了当时该值的一些状态。你可能会有疑惑，我只在这个`Buffer`值中放入了一个长度为2的字符串值，但为什么该值的容量却变为了8。

虽然这与我们当前的主题无关，但是我可以提示你一下：你可以去阅读`runtime`包中一个名叫`stringtoslicebyte`的函数，答案就在其中。

接着说`buffer1`。我又向该值写入了字符串值`"cdefg"`，此时，其容量仍然是8。我在前面通过调用`buffer1`的`Bytes`方法得到的结果值`unreadBytes`，包含了在那时其中的所有未读内容。

但是，由于这个结果值与`buffer1`的内容容器在此时还共用着同一个底层数组，所以，我只需通过简单的再切片操作，就可以利用这个结果值拿到`buffer1`在此时的所有未读内容。如此一来，`buffer1`的新内容就被泄露出来了。

```
buffer1.WriteString("cdefg")
fmt.Printf("The capacity of buffer: %d\n", buffer1.Cap()) // 内容容器的容量仍为：8。
unreadBytes = unreadBytes[:cap(unreadBytes)]
fmt.Printf("The unread bytes of the buffer: %v\n", unreadBytes) // 基于前面获取到的结果值可得，
```

如果我当时把unreadBytes的值传到了外界，那么外界就可以通过该值操纵buffer1的内容了，就像下面这样：

```
unreadBytes[len(unreadBytes)-2] = byte('X') // 'X'的ASCII编码为88。  
fmt.Printf("The unread bytes of the buffer: %v\n", buffer1.Bytes()) // 未读内容变为了: [97 98 9
```

现在，你应该能够体会到，这里的内容泄露可能造成的严重后果了吧？对于Buffer值的Next方法，也存在相同的问题。

不过，如果经过扩容，Buffer值的内容容器或者它的底层数组被重新设定了，那么之前的内容泄露问题就无法再进一步发展了。我在demo80.go文件中写了一个比较完整的示例，你可以去看一看，并揣摩一下。

总结

我们结合两篇内容总结一下。与strings.Builder类型不同，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。bytes.Buffer类型使用字节切片作为其内容容器，并且会用一个字段实时地记录已读字节的计数。

虽然我们无法直接计算出这个已读计数，但是由于它在Buffer值中起到的作用非常关键，所以我们很有必要去理解它。无论是读取、写入、截断、导出还是重置，已读计数都是功能实现中的重要一环。

与strings.Builder类型的值一样，Buffer值既可以被手动扩容，也可以进行自动的扩容。除非我们完全确定后续内容所需的字节数，否则让Buffer值自动去扩容就好了。

Buffer值的扩容方法并不一定会为了获得更大的容量，替换掉现有的内容容器，而是先会本着尽量减少内存分配和内容拷贝的原则，对当前的内容容器进行重用。并且，只有在容量实在无法满足要求的时候，它才会去创建新的内容容器。

此外，你可能并没有想到，Buffer值的某些方法可能会造成内容的泄露。这主要是由于这些方法返回的结果值，在一段时期内会与其所属值的内容容器共用同一个底层数组。

如果我们有意或无意地把这些结果值传到了外界，那么外界就有可能通过它们操纵相关Buffer值的内容。

这属于很严重的数据安全问题。我们一定要避免这种情况的发生。最彻底的做法是，在传出切片这类值之前要做好隔离。比如，先对它们进行深度拷贝，然后再把副本传出去。

思考题

今天的思考题是：对比`strings.Builder`和`bytes.Buffer`的`String`方法，并判断哪一个更高效？原因是什么？

[戳此查看Go语言专栏文章配套详细代码。](#)



极客时间

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