

37 | strings包与字符串操作

2018-11-05 郝林

Go语言核心36讲

[进入课程 >](#)



讲述：黄洲君

时长 12:58 大小 5.94M



在上一篇文章中，我介绍了 Go 语言与 Unicode 编码规范、UTF-8 编码格式的渊源及运用。

Go 语言不但拥有可以独立代表 Unicode 字符的类型 `rune`，而且还有可以对字符串值进行 Unicode 字符拆分的 `for` 语句。

除此之外，标准库中的 `unicode` 包及其子包还提供了很多的函数和数据类型，可以帮助我们解析各种内容中的 Unicode 字符。

这些程序实体都很好用，也都很简单明了，而且有效地隐藏了 Unicode 编码规范中的一些复杂的细节。我就不在这里对它们进行专门的讲解了。

我们今天主要来说一说标准库中的strings代码包。这个代码包也用到了不少unicode包和unicode/utf8包中的程序实体。

比如，strings.Builder类型的WriteRune方法。

又比如，strings.Reader类型的ReadRune方法，等等。

下面这个问题就是针对strings.Builder类型的。**我们今天的问题是：与string值相比，strings.Builder类型的值有哪些优势？**

这里的**典型回答**是这样的。

strings.Builder类型的值（以下简称Builder值）的优势有下面的三种：

已存在的内容不可变，但可以拼接更多的内容；

减少了内存分配和内容拷贝的次数；

可将内容重置，可重用值。

问题解析

先来说说string类型。 我们都知道，在 Go 语言中，string类型的值是不可变的。如果我们想获得一个不一样的字符串，那么就只能基于原字符串进行裁剪、拼接等操作，从而生成一个新的字符串。

裁剪操作可以使用切片表达式；

拼接操作可以用操作符+实现。

在底层，一个string值的内容会被存储到一块连续的内存空间中。同时，这块内存容纳的字节数量也会被记录下来，并用于表示该string值的长度。

你可以把这块内存的内容看成一个字节数组，而相应的string值则包含了指向字节数组头部的指针值。如此一来，我们在一个string值上应用切片表达式，就相当于在对其底层的字节数组做切片。

另外，我们在进行字符串拼接的时候，Go 语言会把所有被拼接的字符串依次拷贝到一个崭新且足够大的连续内存空间中，并把持有相应指针值的`string`值作为结果返回。

显然，当程序中存在过多的字符串拼接操作的时候，会对内存的分配产生非常大的压力。

注意，虽然`string`值在内部持有一个指针值，但其类型仍然属于值类型。不过，由于`string`值的不可变，其中的指针值也为内存空间的节省做出了贡献。

更具体地说，一个`string`值会在底层与它的所有副本共用同一个字节数组。由于这里的字节数组永远不会被改变，所以这样做是绝对安全的。

与`string`值相比，`Builder`值的优势其实主要体现在字符串拼接方面。

`Builder`值中有一个用于承载内容的容器（以下简称内容容器）。它是一个以`byte`为元素类型的切片（以下简称字节切片）。

由于这样的字节切片的底层数组就是一个字节数组，所以我们可以说它与`string`值存储内容的方式是一样的。

实际上，它们都是通过一个`unsafe.Pointer`类型的字段来持有那个指向了底层字节数组的指针值的。

正是因为这样的内部构造，`Builder`值同样拥有高效利用内存的前提条件。虽然，对于字节切片本身来说，它包含的任何元素值都可以被修改，但是`Builder`值并不允许这样做，其中的内容只能够被拼接或者完全重置。

这就意味着，已存在于`Builder`值中的内容是不可变的。因此，我们可以利用`Builder`值提供的方法拼接更多的内容，而丝毫不用担心这些方法会影响到已存在的内容。

这里所说的方法指的是，`Builder`值拥有的一系列指针方法，包括：

`Write`、`WriteByte`、`WriteRune`和`WriteString`。我们可以把它们统称为拼接方法。

我们可以通过调用上述方法把新的内容拼接到已存在的内容的尾部（也就是右边）。这时，如有必要，`Builder`值会自动地对自身的内容容器进行扩容。这里的自动扩容策略与切片的扩容策略一致。

换句话说，我们在向`Builder`值拼接内容的时候并不一定会引起扩容。只要内容容器的容量够用，扩容就不会进行，针对于此的内存分配也不会发生。同时，只要没有扩容，`Builder`值中已存在的内容就不会再被拷贝。


除了`Builder`值的自动扩容，我们还可以选择手动扩容，这通过调用`Builder`值的`Grow`方法就可以做到。`Grow`方法也可以被称为扩容方法，它接受一个`int`类型的参数`n`，该参数用于代表将要扩充的字节数量。

如有必要，`Grow`方法会把其所属值中内容容器的容量增加`n`个字节。更具体地讲，它会生成一个字节切片作为新的内容容器，该切片的容量会是原容器容量的二倍再加上`n`。之后，它会把原容器中的所有字节全部拷贝到新容器中。

 复制代码

```
1 var builder1 strings.Builder
2 // 省略若干代码。
3 fmt.Println("Grow the builder ...")
4 builder1.Grow(10)
5 fmt.Printf("The length of contents in the builder is %d.\n", builder1.Len())
```

当然，`Grow`方法还可能什么都不做。这种情况的前提条件是：当前的内容容器中的未用容量已经够用了，即：未用容量大于或等于`n`。这里的前提条件与前面提到的自动扩容策略中的前提条件是类似的。

 复制代码

```
1 fmt.Println("Reset the builder ...")
2 builder1.Reset()
3 fmt.Printf("The third output(%d):\n%q\n", builder1.Len(), builder1.String())
```

最后，`Builder`值是可以被重用的。通过调用它的`Reset`方法，我们可以让`Builder`值重新回到零值状态，就像它从未被使用过那样。

一旦被重用，`Builder`值中原有的内容容器会被直接丢弃。之后，它和其中的所有内容，将会被 Go 语言的垃圾回收器标记并回收掉。

知识扩展

问题 1：`strings.Builder`类型在使用上有约束吗？

答案是：有约束，概括如下：


在已被真正使用后就不可再被复制；

由于其内容不是完全不可变的，所以需要使用者自行解决操作冲突和并发安全问题。

我们只要调用了`Builder`值的拼接方法或扩容方法，就意味着开始真正使用它了。显而易见，这些方法都会改变其所属值中的内容容器的状态。

一旦调用了它们，我们就不能再以任何的方式对其所属值进行复制了。否则，只要在任何副本上调用上述方法就都会引发 `panic`。

这种 `panic` 会告诉我们，这样的使用方式是并不合法的，因为这里的`Builder`值是副本而不是原值。顺便说一句，这里所说的复制方式，包括但不限于在函数间传递值、通过通道传递值、把值赋予变量等等。


 复制代码

```
1 var builder1 strings.Builder
2 builder1.Grow(1)
3 builder3 := builder1
4 //builder3.Grow(1) // 这里会引发 panic。
5 _ = builder3
```

虽然这个约束非常严格，但是如果我们仔细思考一下的话，就会发现它还是有好处的。

正是由于已使用的`Builder`值不能再被复制，所以肯定不会出现多个`Builder`值中的内容容器（也就是那个字节切片）共用一个底层字节数组的情况。这样也就避免了多个同源的`Builder`值在拼接内容时可能产生的冲突问题。

不过，虽然已使用的`Builder`值不能再被复制，但是它的指针值却可以。无论什么时候，我们都可以通过任何方式复制这样的指针值。注意，这样的指针值指向的都会是同一个`Builder`值。

 复制代码

```
1 f2 := func(bp *strings.Builder) {
2     (*bp).Grow(1) // 这里虽然不会引发 panic，但不是并发安全的。
3     builder4 := *bp
4     //builder4.Grow(1) // 这里会引发 panic。
5     _ = builder4
6 }
7 f2(&builder1)
```

正因为如此，这里就产生了一个问题，即：如果`Builder`值被多方同时操作，那么其中的内容就很可能产生混乱。这就是我们所说的操作冲突和并发安全问题。

`Builder`值自己是无法解决这些问题的。所以，我们在通过传递其指针值共享`Builder`值的时候，一定要确保各方对它的使用是正确、有序的，并且是并发安全的；而最彻底的解决方案是，绝不共享`Builder`值以及它的指针值。

我们可以在各处分别声明一个`Builder`值来使用，也可以先声明一个`Builder`值，然后在真正使用它之前，便将它的副本传到各处。另外，我们还可以先使用再传递，只要在传递之前调用它的`Reset`方法即可。

 复制代码

```
1 builder1.Reset()
2 builder5 := builder1
3 builder5.Grow(1) // 这里不会引发 panic。
```

总之，关于复制`Builder`值的约束是有意义的，也是很有必要的。虽然我们仍然可以通过某些方式共享`Builder`值，但最好还是不要以身犯险，“各自为政”是最好的解决方案。不过，对于处在零值状态的`Builder`值，复制不会有任何问题。


问题 2：为什么说`strings.Reader`类型的值可以高效地读取字符串？

与`strings.Builder`类型恰恰相反，`strings.Reader`类型是为了高效读取字符串而存在的。后者的高效主要体现在它对字符串的读取机制上，它封装了很多用于在`string`值上读取内容的最佳实践。

`strings.Reader`类型的值（以下简称`Reader`值）可以让我们很方便地读取一个字符串中的内容。在读取的过程中，`Reader`值会保存已读取的字节的计数（以下简称已读计数）。

已读计数也代表着下一次读取的起始索引位置。`Reader`值正是依靠这样一个计数，以及针对字符串值的切片表达式，从而实现快速读取。

此外，这个已读计数也是读取回退和位置设定时的重要依据。虽然它属于`Reader`值的内部结构，但我们还是可以通过该值的`Len`方法和`Size`把它计算出来的。代码如下：

 复制代码

```
1 var reader1 strings.Reader
2 // 省略若干代码。
3 readingIndex := reader1.Size() - int64(reader1.Len()) // 计算出的已读计数。
```

`Reader`值拥有的大部分用于读取的方法都会及时地更新已读计数。比如，`ReadByte`方法会在读取成功后将这个计数的值加1。


又比如，`ReadRune`方法在读取成功之后，会把被读取的字符所占用的字节数作为计数的增量。

不过，`ReadAt`方法算是一个例外。它既不会依据已读计数进行读取，也不会读取后更新它。正因为如此，这个方法可以自由地读取其所属的`Reader`值中的任何内容。

除此之外，`Reader`值的`Seek`方法也会更新该值的已读计数。实际上，这个`Seek`方法的主要作用正是设定下一次读取的起始索引位置。

另外，如果我们把常量`io.SeekCurrent`的值作为第二个参数值传给该方法，那么它还会依据当前的已读计数，以及第一个参数`offset`的值来计算新的计数值。

由于Seek方法会返回新的计数值，所以我们可以很容易地验证这一点。比如像下面这样：

 复制代码

```
1 offset2 := int64(17)
2 expectedIndex := reader1.Size() - int64(reader1.Len()) + offset2
3 fmt.Printf("Seek with offset %d and whence %d ...\\n", offset2, io.SeekCurrent)
4 readingIndex, _ := reader1.Seek(offset2, io.SeekCurrent)
5 fmt.Printf("The reading index in reader: %d (returned by Seek)\\n", readingIndex)
6 fmt.Printf("The reading index in reader: %d (computed by me)\\n", expectedIndex)
```

综上所述，Reader值实现高效读取的关键就在于它内部的已读计数。计数的值就代表着下一次读取的起始索引位置。它可以很容易地被计算出来。Reader值的Seek方法可以直接设定该值中的已读计数值。

总结

今天，我们主要讨论了strings代码包中的两个重要类型，即：Builder和Reader。前者用于构建字符串，而后者则用于读取字符串。


与string值相比，Builder值的优势主要体现在字符串拼接方面。它可以在保证已存在的内容不变的前提下，拼接更多的内容，并且会在拼接的过程中，尽量减少内存分配和内容拷贝的次数。

不过，这类值在使用上也是有约束的。它在被真正使用之后就不能再被复制了，否则就会引发panic。虽然这个约束很严格，但是也可以带来一定的好处。它可以有效地避免一些操作冲突。虽然我们可以通过一些手段（比如传递它的指针值）绕过这个约束，但这是弊大于利的。最好的解决方案就是分别声明、分开使用、互不干涉。

Reader值可以让我们很方便地读取一个字符串中的内容。它的高效主要体现在它对字符串的读取机制上。在读取的过程中，Reader值会保存已读取的字节的计数，也称已读计数。

这个计数代表着下一次读取的起始索引位置，同时也是高效读取的关键所在。我们可以利用这类值的Len方法和Size方法，计算出其中的已读计数的值。有了它，我们就可以更加灵活地进行字符串读取了。

我只在本文介绍了上述两个数据类型，但并不意味着strings包中有用的程序实体只有这两个。实际上，strings包还提供了大量的函数。比如：

 复制代码

```
1 `Count`、`IndexRune`、`Map`、`Replace`、`SplitN`、`Trim`，等等。
```

它们都是非常易用和高效的。你可以去看看它们的源码，也许会因此有所感悟。

思考题

今天的思考题是：`*strings.Builder`和`*strings.Reader`都分别实现了哪些接口？这样做有什么好处吗？

[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



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

精选留言 (8)

写留言



Realm

2018-11-05

5

1 string拼接的结果是生成新的string，需要把原字符串拷贝到新的string中；Builder底层有个[]byte,按需扩容，不必每次拼接都需要拷贝；

2 Reader的优势是维护一个已读计数器，知道下一次读的位置，读得更快。

展开

作者回复: 嗯，是的。



jimmy

2019-01-17

1

strings.Builder里边的String方法是

// String returns the accumulated string.

```
func (b *Builder) String() string {  
    return *(*string)(unsafe.Pointer(&b.buf))  
}...
```

展开

作者回复: 省去了类型转换的开销，效率会高很多。



南方有嘉木

2018-11-27

1

请问容量增加n个字节，为什么是原来的2倍再加上n呢

展开



Cloud

2018-11-05

1

很实用！

展开 ▾



Geek_37cea...

2019-04-02

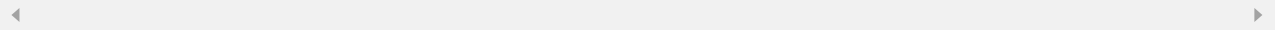


老师，我在看strings 源码的时候发现了

```
func noescape(p unsafe.Pointer) unsafe.Pointer {  
    x := uintptr(p)  
    return unsafe.Pointer(x ^ 0)  
}...
```

展开 ▾

作者回复: 为了产生一个新值啊，要跟这个函数的参数值划清界限。



Geek_1ed70...

2019-03-14



读源代码讲得好深....

展开 ▾



王小勃

2019-03-04



打卡

展开 ▾



kingkang

2019-01-04



请问byte数组转string出现乱码怎么处理？

展开 ▾

作者回复: 如果字节数组的内容不是UTF-8编码的Unicode字符，这样直接转就会出现乱码。先要搞清楚两个问题：1. 这个字节数组的内容会是可打印的字符吗？2. 如果是可打印的字符，那它使用什么编码的？

