

## 08 | container包中的那些容器

2018-08-29 郝林



【Go语言代码较多，建议配合文章收听音频。】

我们在上次讨论了数组和切片，当我们提到数组的时候，往往会想起链表。Go语言的链表实现在标准库的`container/list`代码包中。

这个代码包中有两个公开的程序实体——`List`和`Element`，`List`实现了一个双向链表（以下简称链表），而`Element`则代表了链表中元素的结构。

那么，我今天的问题是：可以把自己生成的`Element`类型值传给链表吗？

我们在这里用到了`List`的四种方法。

`MoveBefore`方法和`MoveAfter`方法，它们分别用于把给定的元素移动到另一个元素的前面和后面。

`MoveToFront`方法和`MoveToBack`方法，分别用于把给定的元素移动到链表的最前端和最后端。

在这些方法中，“给定的元素”都是`*Element`类型的，`*Element`类型是`Element`类型的指针类型，`*Element`的值就是元素的指针。

```
func (l *List) MoveBefore(e, mark *Element)
func (l *List) MoveAfter(e, mark *Element)

func (l *List) MoveToFront(e *Element)
func (l *List) MoveToBack(e *Element)
```

具体问题是，如果我们自己生成这样的值，然后把它作为“给定的元素”传给链表的方法，那么会发生什么？链表会接受它吗？

这里，给出一个**典型回答**：不会接受，这些方法将不会对链表做出任何改动。因为我们自己生成的`Element`值并不在链表中，所以也就谈不上“在链表中移动元素”。更何况链表不允许我们把自己生成的`Element`值插入其中。

## 问题解析

在`List`包含的方法中，用于插入新元素的那些方法都只接受`interface{}`类型的值。这些方法在内部会使用`Element`值，包装接收到的新元素。

这样做正是为了避免直接使用我们自己生成的元素，主要原因是避免链表的内部关联，遭到外界破坏，这对于链表本身以及我们这些使用者来说都是有益的。

`List`的方法还有下面这几种：

`Front`和`Back`方法分别用于获取链表中最前端和最后端的元素，`InsertBefore`和`InsertAfter`方法分别用于在指定的元素之前和之后插入新元素，`PushFront`和`PushBack`方法则分别用于在链表的最前端和最后端插入新元素。

```
func (l *List) Front() *Element
func (l *List) Back() *Element

func (l *List) InsertBefore(v interface{}, mark *Element) *Element
func (l *List) InsertAfter(v interface{}, mark *Element) *Element

func (l *List) PushFront(v interface{}) *Element
func (l *List) PushBack(v interface{}) *Element
```

这些方法都会把一个`Element`值的指针作为结果返回，它们就是链表留给我们的安全“接口”。拿到这些内部元素的指针，我们就可以去调用前面提到的用于移动元素的方法了。

## 知识扩展

### 1. 问题：为什么链表可以做到开箱即用？

List和Element都是结构体类型。结构体类型有一个特点，那就是它们的零值都会是拥有特定结构，但是没有任何定制化内容的值，相当于一个空壳。值中的字段也都会被分别赋予各自类型的零值。

广义来讲，所谓的零值就是只做了声明，但还未做初始化的变量被给予的缺省值。每个类型的零值都会依据该类型的特性而被设定。

比如，经过语句`var a [2]int`声明的变量a的值，将会是一个包含了两个0的整数数组。又比如，经过语句`var s []int`声明的变量s的值将会是一个`[]int`类型的、值为`nil`的切片。

那么经过语句`var l list.List`声明的变量l的值将会是什么呢？**[1]**这个零值将会是一个长度为0的链表。这个链表持有的根元素也将会是一个空壳，其中只会包含缺省的内容。那这样的链表我们可以直接拿来使用吗？

答案是，可以的。这被称为“开箱即用”。Go语言标准库中很多结构体类型的程序实体都做到了开箱即用。这也是在编写可供别人使用的代码包（或者说程序库）时，我们推荐遵循的最佳实践之一。那么，语句`var l list.List`声明的链表l可以直接使用，这是怎么做到的呢？

关键在于它的“延迟初始化”机制。

所谓的**延迟初始化**，你可以理解为把初始化操作延后，仅在实际需要的时候才进行。延迟初始化的优点在于“延后”，它可以分散初始化操作带来的计算量和存储空间消耗。

例如，如果我们需要集中声明非常多的大容量切片的话，那么那时的CPU和内存空间的使用量肯定都会一个激增，并且只有设法让其中的切片及其底层数组被回收，内存使用量才会有所降低。

如果数组是可以被延迟初始化的，那么计算量和存储空间的压力就可以被分散到实际使用它们的时候。这些数组被实际使用的时间越分散，延迟初始化带来的优势就会越明显。

实际上，Go语言的切片就起到了延迟初始化其底层数组的作用，你可以想一想为什么会这么说的理由。

延迟初始化的缺点恰恰也在于“延后”。你可以想象一下，如果我在调用链表的每个方法的时候，它们都需要先去判断链表是否已经被初始化，那这也会是一个计算量上的浪费。在这些方法被非常频繁地调用的情况下，这种浪费的影响就开始显现了，程序的性能将会降低。

在这里的链表实现中，一些方法是无需对是否初始化做判断的。比如Front方法和Back方法，

一旦发现链表的长度为0,直接返回nil就好了。

又比如,在用于删除元素、移动元素,以及一些用于插入元素的方法中,只要判断一下传入的元素中指向所属链表的指针,是否与当前链表的指针相等就可以了。

如果不相等,就一定说明传入的元素不是这个链表中的,后续的操作就不用做了。反之,就一定说明这个链表已经被初始化了。

原因在于,链表的PushFront方法、PushBack方法、PushBackList方法以及PushFrontList方法总会先判断链表的状态,并在必要时进行初始化,这就是延迟初始化。

而且,我们在向一个空的链表中添加新元素的时候,肯定会调用这四个方法中的一个,这时新元素中指向所属链表的指针,一定会被设定为当前链表的指针。所以,指针相等是链表已经初始化的充分必要条件。

明白了吗? List利用了自身以及Element在结构上的特点,巧妙地平衡了延迟初始化的优缺点,使得链表可以开箱即用,并且在性能上可以达到最优。

## 问题 2: Ring与List的区别在哪儿?

container/ring包中的Ring类型实现的是一个循环链表,也就是我们俗称的环。其实List在内部就是一个循环链表。它的根元素永远不会持有任何实际的元素值,而该元素的存在就是为了连接这个循环链表的首尾两端。

所以也可以说, List的零值是一个只包含了根元素,但不包含任何实际元素值的空链表。那么,既然Ring和List在本质上都是循环链表,那它们到底有什么不同呢?

最主要的不同有下面几种。

1. Ring类型的数据结构仅由它自身即可代表,而List类型则需要由它以及Element类型联合表示。这是表示方式上的不同,也是结构复杂度上的不同。
2. 一个Ring类型的值严格来讲,只代表了其所属的循环链表中的一个元素,而一个List类型的值则代表了一个完整的链表。这是表示维度上的不同。
3. 在创建并初始化一个Ring值的时候,我们可以指定它包含的元素的数量,但是对于一个List值来说却不能这样做(也没有必要这样做)。循环链表一旦被创建,其长度是不可变的。这是两个代码包中的New函数在功能上的不同,也是两个类型在初始化值方面的第一个不同。
4. 仅通过`var r ring.Ring`语句声明的r将会是一个长度为1的循环链表,而List类型的零值则是一个长度为0的链表。别忘了List中的根元素不会持有实际元素值,因此计算长度时不会包含它。这是两个类型在初始化值方面的第二个不同。

5. Ring值的Len方法的算法复杂度是 $O(N)$ 的，而List值的Len方法的算法复杂度则是 $O(1)$ 的。这是两者在性能方面最显而易见的差别。

其他的不同基本上都是方法方面的了。比如，循环链表也有用于插入、移动或删除元素的方法，不过用起来都显得更抽象一些，等等。

## 总结

我们今天主要讨论了container/list包中的链表实现。我们详细讲解了链表的一些主要的使用技巧和实现特点。由于此链表实现在内部就是一个循环链表，所以我们还把它与container/ring包中的循环链表实现做了一番比较，包括结构、初始化以及性能方面。

## 思考题

1. container/ring包中的循环链表的适用场景都有哪些？
2. 你使用过container/heap包中的堆吗？它的适用场景又有哪些呢？

在这里，我们先不求对它们的实现了如指掌，能用对、用好才是我们进阶之前的第一步。好了，感谢你的收听，我们下次再见。

---

[1]: List这个结构体类型有两个字段，一个是Element类型的字段root，另一个是int类型的字段len。顾名思义，前者代表的就是那个根元素，而后者用于存储链表的长度。注意，它们都是包级私有的，也就是说使用者无法查看和修改它们。

像前面那样声明的l，其字段root和len都会被赋予相应的零值。len的零值是0，正好可以表明该链表还未包含任何元素。由于root是Element类型的，所以它的零值就是该类型的空壳，用字面量表示的话就是Element{}。

Element类型包含了几个包级私有的字段，分别用于存储前一个元素、后一个元素以及所属链表的指针值。另外还有一个名叫Value的公开的字段，该字段的作用就是持有元素的实际值，它是interface{}类型的。在Element类型的零值中，这些字段的值都会是nil。

## 参考阅读

### 切片与数组的比较

切片本身有着占用内存少和创建便捷等特点，但它的本质上还是数组。切片的一大好处是可以通过窗口快速地定位并获取，或者修改底层数组中的元素。

不过，当我们想删除切片中的元素的时候就没那么简单了。元素复制一般是免不了的，就算只删除一个元素，有时也会造成大量元素的移动。这时还要注意空出的元素槽位的“清空”，否则很可



能会造成内存泄漏。

另一方面，在切片被频繁“扩容”的情况下，新的底层数组会不断产生，这时内存分配的量以及元素复制的次数可能就很可观了，这肯定会对程序的性能产生负面的影响。

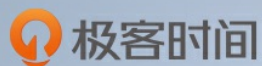
尤其是当我们没有一个合理、有效的“缩容”策略的时候，旧的底层数组无法被回收，新的底层数组中也会有大量无用的元素槽位。过度的内存浪费不但会降低程序的性能，还可能会使内存溢出并导致程序崩溃。

由此可见，正确地使用切片是多么的重要。不过，一个更重要的事实是，任何数据结构都不是银弹。不是吗？数组的自身特点和适用场景都非常鲜明，切片也是一样。它们都是Go语言原生的数据结构，使用起来也都很方便。不过，你的集合类工具箱中不应该只有它们。这就是我们使用链表的原因。

不过，对比来看，一个链表所占用的内存空间，往往要比包含相同元素的数组所占内存大得多。这是由于链表的元素并不是连续存储的，所以相邻的元素之间需要互相保存对方的指针。不但如此，每个元素还要存有它所属链表的指针。

有了这些关联，链表的结构反倒更简单了。它只持有头部元素（或称为根元素）基本上就可以了。当然了，为了防止不必要的遍历和计算，链表的长度记录在内也是必须的。

[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

