

讲堂 □ [Go语言核心36讲](#) □ [文章详情](#)

18 | if语句、for语句和switch语句

2018-09-21 郝林



18 | if语句、for语句和switch语句

朗读人：黄洲君 15'34'' | 7.13M

在上两篇文章中，我主要为你讲解了与go语句、goroutine 和 Go 语言调度器有关的知识 and 技法。

内容很多，你不用急于完全消化，可以在编程实践过程中逐步理解和感悟，争取夯实它们。

现在，让我们暂时走下神坛，回归民间。我今天要讲的if语句、for语句和switch语句都属于 Go 语言的基本流程控制语句。它们的语法看起来很朴素，但实际上也会有一些使用技巧和注意事项。我在本篇文章中会以一系列面试题为线索，为你讲述它们的用法。

那么，今天的问题是：使用携带range子句的for语句时需要注意哪些细节？这是一个比较笼统的问题。我还是通过编程题来讲解吧。

本问题中的代码都被放在了命令源码文件 demo41.go 的main函数中的。为了专注问题本身，本篇文章中展示的编程题会省略掉一部分代码包声明语句、代码包导入语句和main函数本身的声明部分。

```
numbers1 := []int{1, 2, 3, 4, 5, 6}
for i := range numbers1 {
    if i == 3 {
        numbers1[i] |= i
    }
}
fmt.Println(numbers1)
```

□复制代码

我先声明了一个元素类型为int的切片类型的变量numbers1，在该切片中有 6 个元素值，分别是1到6的整数。我用一条携带range子句的for语句去迭代numbers1变量中的所有元素值。

在这条for语句中，只有一个迭代变量i。我在每次迭代时，都会先去判断i的值是否等于3，如果结果为true，那么就让numbers1的第i个元素值与i本身做按位或的操作，再把操作结果作为numbers1的新的第i个元素值。最后我会打印出numbers1的值。

所以具体的问题就是，这段代码执行后会打印出什么内容？

这里的典型回答是：打印的内容会是[1 2 3 7 5 6]。

问题解析

你心算得到的答案是这样吗？让我们一起来复现一下这个计算过程。

当for语句被执行的时候，在range关键字右边的numbers1会先被求值。这个位置上的代码被称为range表达式。range表达式的结果值可以是数组、数组的指针、切片、字符串、字典或者允许接收操作的通道中的某一个，并且结果值只能有一个。

对于不同种类的range表达式结果值，for语句的迭代变量的数量可以有所不同。就拿我们这里的numbers1来说，它是一个切片，那么迭代变量就可以有两个，右边的迭代变量代表当次迭代对应的某一个元素值，而左边的迭代变量则代表该元素值在切片中的索引值。

那么，如果像本题代码中的for语句那样，只有一个迭代变量的情况意味着什么呢？这意味着，该迭代变量只会代表当次迭代对应的元素值的索引值。

更宽泛地讲，当只有一个迭代变量的时候，数组、数组的指针、切片和字符串的元素值都是无处安放的，我们只能拿到按照从小到大顺序给出的一个个索引值。

因此，这里的迭代变量i的值会依次是从0到5的整数。当i的值等于3的时候，与之对应的是切片中的第4个元素值4。对4和3进行按位或操作得到的结果是7。这就是答案中的第4个整数是7的原因了。

现在，我稍稍修改一下上面的代码。我们再来估算一下打印内容。

```
numbers2 := [...]int{1, 2, 3, 4, 5, 6}
maxIndex2 := len(numbers2) - 1
for i, e := range numbers2 {
    if i == maxIndex2 {
        numbers2[0] += e
    } else {
        numbers2[i+1] += e
    }
}
fmt.Println(numbers2)
```

□复制代码

注意，我把迭代的对象换成了numbers2。numbers2中的元素值同样是从1到6的6个整数，并且元素类型同样是int，但它是一个数组而不是一个切片。

在for语句中，我总是会对紧挨在当次迭代对应的元素后边的那个元素，进行重新赋值，新的值会是这两个元素的值之和。当迭代到最后一个元素时，我会把此range表达式结果值中的第一个元素值，替换为它的原值与最后一个元素值的和，最后，我会打印出numbers2的值。

对于这段代码，我的问题依旧是：打印的内容会是什么？你可以先思考一下。

好了，我要公布答案了。打印的内容会是[7 3 5 7 9 11]。我先来重现一下计算过程。当for语句被执行的时候，在range关键字右边的numbers2会先被求值。

这里需要注意两点：

1. range表达式只会在for语句开始执行时被求值一次，无论后边会有多少次迭代；
2. range表达式的求值结果会被复制，也就是说，被迭代的对象是range表达式结果值的副本而不是原值。

基于这两个规则，我们接着往下看。在第一次迭代时，我改变的是numbers2的第二个元素的值，新值为3，也就是1和2之和。

但是，被迭代的对象的第二个元素却没有任何改变，毕竟它与numbers2已经是毫不相关的两个数组了。因此，在第二次迭代时，我会把numbers2的第三个元素的值修改为5，即被迭代对象的第二个元素值2和第三个元素值3的和。

以此类推，之后的numbers2的元素值依次会是7、9和11。当迭代到最后一个元素时，我会把numbers2的第一个元素的值修改为1和6之和。

好了，现在该你操刀了。你需要把numbers2的值由一个数组改成一个切片，其中的元素值都不要变。为了避免混淆，你还要把这个切片值赋给变量numbers3，并且把后边代码中所有的numbers2都改为numbers3。

问题是不变的，执行这段修改版的代码后打印的内容会是什么呢？如果你实在估算不出来，可以先实际执行一下，然后再尝试解释看到的答案。提示一下，切片与数组是不同的，前者是引用类型的，而后者是值类型的。

我们可以先接着讨论后边的内容，但是我强烈建议你一定要回来，再看看我留给你的这个问题，认真地思考和计算一下。

知识扩展

问题 1： switch语句中的switch表达式和case表达式之间有着怎样的联系？

先来看一段代码。

```
value1 := [...]int8{0, 1, 2, 3, 4, 5, 6}
switch 1 + 3 {
case value1[0], value1[1]:
    fmt.Println("0 or 1")
}
```

```
case value1[2], value1[3]:
    fmt.Println("2 or 3")
case value1[4], value1[5], value1[6]:
    fmt.Println("4 or 5 or 6")
}
```

❏复制代码

我先声明了一个数组类型的变量`value1`，该变量的元素类型是`int8`。在后边的`switch`语句中，被夹在`switch`关键字和左花括号`{`之间的是`1 + 3`，这个位置上的代码被称为`switch`表达式。这个`switch`语句还包含了三个`case`子句，而每个`case`子句又各包含了一个`case`表达式和一条打印语句。

所谓的`case`表达式一般由`case`关键字和一个表达式列表组成，表达式列表中的多个表达式之间需要有英文逗号,分割，比如，上面代码中的`case value1[0], value1[1]`就是一个`case`表达式，其中的两个子表达式都是由索引表达式表示的。

另外的两个`case`表达式分别是`case value1[2], value1[3]`和`case value1[4], value1[5], value1[6]`。

此外，在这里的每个`case`子句中的那些打印语句，会分别打印出不同的内容，这些内容用于表示`case`子句被选中的原因，比如，打印内容`0 or 1`表示当前`case`子句被选中是因为`switch`表达式的结果值等于`0`或`1`中的某一个。另外两条打印语句会分别打印出`2 or 3`和`4 or 5 or 6`。

现在问题来了，拥有这样三个`case`表达式的`switch`语句可以成功通过编译吗？如果不可以，原因是什么？如果可以，那么该`switch`语句被执行后会打印出什么内容。

我刚才说过，只要`switch`表达式的结果值与某个`case`表达式中的任意一个子表达式的结果值相等，该`case`表达式所属的`case`子句就会被选中。

并且，一旦某个`case`子句被选中，其中的附带在`case`表达式后边的那些语句就会被执行。与此同时，其他的所有`case`子句都会被忽略。

当然了，如果被选中的`case`子句附带的语句列表中包含了`fallthrough`语句，那么紧挨在它下边的那个`case`子句附带的语句也会被执行。

正因为存在上述判断相等的操作（以下简称判等操作），`switch`语句对`switch`表达式的结果类型，以及各个`case`表达式中子表达式的结果类型都是有要求的。毕竟，在 **Go** 语言中，只有类型相同的值之间才有可能被允许进行判等操作。

如果switch表达式的结果值是无类型的常量，比如`1 + 3`的求值结果就是无类型的常量4，那么这个常量会被自动地转换为此种常量的默认类型的值，比如整数4的默认类型是`int`，又比如浮点数3.14的默认类型是`float64`。

因此，由于上述代码中的switch表达式的结果类型是`int`，而那些case表达式中子表达式的结果类型却是`int8`，它们的类型并不相同，所以这条switch语句是无法通过编译的。

再来看一段很类似的代码：

```
value2 := [...]int8{0, 1, 2, 3, 4, 5, 6}
switch value2[4] {
case 0, 1:
    fmt.Println("0 or 1")
case 2, 3:
    fmt.Println("2 or 3")
case 4, 5, 6:
    fmt.Println("4 or 5 or 6")
}
```

□复制代码

其中的变量`value2`与`value1`的值是完全相同的。但不同的是，我把switch表达式换成了`value2[4]`，并把下边那三个case表达式分别换为了`case 0, 1`、`case 2, 3`和`case 4, 5, 6`。

如此一来，switch表达式的结果值是`int8`类型的，而那些case表达式中子表达式的结果值却是无类型的常量了。这与之前的情况恰恰相反。那么，这样的switch语句可以通过编译吗？

答案是肯定的。因为，如果case表达式中子表达式的结果值是无类型的常量，那么它的类型会被自动地转换为switch表达式的结果类型，又由于上述那几个整数都可以被转换为`int8`类型的值，所以对这些表达式的结果值进行判等操作是没有问题的。

当然了，如果这里说的自动转换没能成功，那么switch语句照样通不过编译。

通过上面这两道题，你应该可以搞清楚switch表达式和case表达式之间的联系了。由于需要进行判等操作，所以前者 and 后者中的子表达式的结果类型需要相同。

switch语句会进行有限的类型转换，但肯定不能保证这种转换可以统一它们的类型。还要注意，如果这些表达式的结果类型有某个接口类型，那么一定要小心检查它们的动态值是否都具有可比性（或者说是否允许判等操作）。因为，如果答案是否定的，虽然不会造成编译错误，但是后果会更加严重：引发 **panic**（也就是运行时恐慌）。

问题 2: switch语句对它的case表达式有哪些约束?

我在上一个问题的阐述中还重点表达了一点, 不知你注意到了没有, 那就是: switch语句在case子句的选择上是具有唯一性的。正因为如此, switch语句不允许case表达式中的子表达式结果值存在相等的情况, 不论这些结果值相等的子表达式, 是否存在于不同的case表达式中, 都会是这样的结果。具体请看这段代码:

```
value3 := [...]int8{0, 1, 2, 3, 4, 5, 6}
switch value3[4] {
case 0, 1, 2:
    fmt.Println("0 or 1 or 2")
case 2, 3, 4:
    fmt.Println("2 or 3 or 4")
case 4, 5, 6:
    fmt.Println("4 or 5 or 6")
}
```

[复制代码](#)

变量value3的值同value1, 依然是由从0到6的 7 个整数组成的数组, 元素类型是int8。switch表达式是value3[4], 三个case表达式分别是case 0, 1, 2、case 2, 3, 4和case 4, 5, 6。

由于在这三个case表达式中存在结果值相等的子表达式, 所以这个switch语句无法通过编译。不过, 好在这个约束本身还有个约束, 那就是只针对结果值为常量的子表达式。

比如, 子表达式1+1和2不能同时出现, 1+3和4也不能同时出现。有了这个约束的约束, 我们就可以想办法绕过这个对子表达式的限制了。再看一段代码:

```
value5 := [...]int8{0, 1, 2, 3, 4, 5, 6}
switch value5[4] {
case value5[0], value5[1], value5[2]:
    fmt.Println("0 or 1 or 2")
case value5[2], value5[3], value5[4]:
    fmt.Println("2 or 3 or 4")
case value5[4], value5[5], value5[6]:
    fmt.Println("4 or 5 or 6")
}
```

[复制代码](#)

变量名换成了value5, 但这不是重点。重点是, 我把case表达式中的常量都换成了诸如value5[0]这样的索引表达式。

虽然第一个case表达式和第二个case表达式都包含了value5[2]，并且第二个case表达式和第三个case表达式都包含了value5[4]，但这已经不是问题了。这条switch语句可以成功通过编译。

不过，这种绕过方式对于类型判断的switch语句（以下简称为类型switch语句）就无效了。因为类型switch语句中的case表达式的子表达式，都必须直接由类型字面量表示，而无法通过间接的方式表示。代码如下：

```
value6 := interface{}(byte(127))
switch t := value6.(type) {
case uint8, uint16:
    fmt.Println("uint8 or uint16")
case byte:
    fmt.Printf("byte")
default:
    fmt.Printf("unsupported type: %T", t)
}
```

❏复制代码

变量value6的值是空接口类型的。该值包装了一个byte类型的值127。我在后面使用类型switch语句来判断value6的实际类型，并打印相应的内容。

这里有两个普通的case子句，还有一个default case子句。前者的case表达式分别是case uint8, uint16和case byte。你还记得吗？byte类型是uint8类型的别名类型。

因此，它们两个本质上是同一个类型，只是类型名称不同罢了。在这种情况下，这个类型switch语句是无法通过编译的，因为子表达式byte和uint8重复了。好了，以上说的就是case表达式的约束以及绕过方式，你学会了吗。

总结

我们今天主要讨论了for语句和switch语句，不过我并没有说明那些语法规则，因为它们太简单了。我们需要多加注意的往往是那些隐藏在 Go 语言规范和最佳实践里的细节。

这些细节其实就是我们很多技术初学者所谓的“坑”。比如，我在讲for语句的时候交代了携带range子句时只有一个迭代变量意味着什么。你必须知道在迭代数组或切片时只有一个迭代变量的话是无法迭代出其中的元素值的，否则你的程序可能就不会像你预期的那样运行了。

还有，range表达式的结果值是会被复制的，实际迭代时并不会使用原值。至于会影响到什么，那就要看这个结果值的类型是值类型还是引用类型了。

说到switch语句，你要明白其中的case表达式的所有子表达式的结果值都是要与switch表达式的结果值判等的，因此它们的类型必须相同或者能够都统一到switch表达式的结果类型。如果无法做到，那么这条switch语句就不能通过编译。

最后，同一条switch语句中的所有case表达式的子表达式的结果值不能重复，不过好在这只是对于由字面量直接表示的子表达式而言的。

请记住，普通case子句的编写顺序很重要，最上边的case子句中的子表达式总是会被最先求值，在判等的时候顺序也是这样。因此，如果某些子表达式的结果值有重复并且它们与switch表达式的结果值相等，那么位置靠上的case子句总会被选中。

思考题

1. 在类型switch语句中，我们怎样对被判断类型的那个值做相应的类型转换？
2. 在if语句中，初始化子句声明的变量的作用域是什么？