Go中的一些小细节

Go 夜读 SIG 小组
2019-12-05

打印出true还是false？

```
package main
func f() bool {return false}
func main() {
    switch f()
    {
    case true:  println(true)
    case false: println(false)
    }
}
```

请不要用go fmt自动格式化此段代码。一些IDE会在存盘时自动格式化代码，所以最好在记事本中输入此段代码。

Go语言分号自动插入规则

1. 在Go代码中，注释除外，如果一个代码行的最后一个语法词段(token)为下列所示之一，则一个分号将自动插入在此字段后(即行尾)：
   - 一个标识符；
   - 一个整数、浮点数、虚部、码点或者字符串字面表示形式；
   - 这几个跳转关键字之一：break、continue、fallthrough和return；
   - 自增运算符++或者自减运算符--；
   - 一个右括号：)] }

2. 为了允许一条复杂语句完全显示在一个代码行中，分号可能被插入在一个右小括号)或者右大括号}之前。

在编译之前，一些分号将被自动插入

```
package main;
func f() bool {return false;};
func main() {
    switch f();
    {
    case true:  println(true);
    case false: println(false);
    };
};
```

我们可以认为编译器在编译之前将自动插入一些分号，如左侧所示。

另一个细节：switch比较值的默认值

```
switch InitSimpleStatement; CompareOperand0 {
case CompareOperandList1:
    // do something
case CompareOperandList2:
    // do something
...
default:
    // do something
}
```

表达式 CompareOperand0 可以不出现（被省略）。如果它被省略，则它的值将被视为 true。

最终的等价形式

```
package main;
func f() bool {return false;};
func main() {
    switch f(); true
    {
    case true:  println(true);
    case false: println(false);
    };
};
```

到此，一目了然，此程序将打印出 **`true`**。

- 在分号(包括自动插入的分号)后断行
- 或者在除了break、continue、fallthrough和return之外的关键字后断行

均不会不影响程序行为。

开大括号有些时候可以放在下一行

```
func f() { // 此大括号不可放在下一行
    for
    {
        switch
        {
        }
        break
    }
}
```


一段合法的但看上去有些别扭的代码

```
for i := 0
i < 10
i++ {
    if n := i%2
    n == 0 {
            fmt.Println(i, "是个偶数")
    }
}
```


插入分号之后

```
for i := 0;
i < 10;
i++ {
    if n := i%2;
    n == 0 {
            fmt.Println(i, "是个偶数");
    };
};
```


最终等价于

```
for i := 0; i < 10; i++ {
    if n := i%2; n == 0 {
            fmt.Println(i, "是个偶数");
    };
};
```

看来使用go fmt来尽量统一

代码风格并非是一件坏事。

os.Exit调用之后defer调用将不会被执行

```
package main

import "os"

func main() {
    defer println("bye") // 不会打印出来
    os.Exit(1)
}
```

*注意:*log.Fatal*将调用*os.Exit

runtime.Goexit调用之后defer调用会被执行

```
package main

import "runtime"

func main() {
    c := make(chan int)
    go func() {
        defer close(c)
        // 会打印出来
        defer println("bye")
        runtime.Goexit()
    }()
    <-c
}
```

估值结果因是否为变量或常量而异

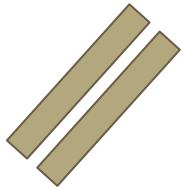
```
package main


func main() {
    var n uint = 10
    const N uint = 10
    var x byte = (1 << n) / 100 // 1被推断为byte，将溢出
    var y byte = (1 << N) / 100 // 1被认为是类型不确定整数
    println(x, y) // 0 10
}
```


编译行为因是否为变量或常量而异

```
const N = 2
var m = 2

var _ float64 = 1 << N  // 编译没问题
var _ = float64(1 << N) // 编译没问题

// 1被推断为float64类型
var _ float64 = 1 << m  // 编译失败：浮点数不可被移位
var _ = float64(1 << m) // 编译失败：浮点数不可被移位
```


多值赋值语句的执行规则(两阶段)

```
package main


import "fmt"


func main() {
    var a []int = nil
    a, a[0] = []int{1, 2}, 9
    fmt.Println(a)
}
```



```
// 第一阶段
x := &a; y := a; z := 0
// 第二阶段（不影响第一阶段）
*x = []int{1, 2}
y[z] = 9
```

```
a = []int{1, 2}
a[0] = 9
```

多值赋值语句的执行规则

```
package main

import "fmt"

func main() {
    x := []int{123}
    x, x[0] = nil, 456 // 此句不会发生恐慌
    fmt.Println(x)
}
```


当估值一个表达式、赋值语句或者函数返回语句中的操作数时，所有的函数调用、方法调用和通道操作将按照它们在代码中的出现顺序进行估值。

Go白皮书未指定表达式中其它估值顺序。

表达式估值顺序

```
package main

import "fmt"

func f(p *int) int {
    *p++
    return *p
}
```

```
func main() {
    var x int
    x, z := f(&x), f(&x)
    fmt.Println(x, z) // 1 2
}
```

当前的gccgo编译器未正确实现此估值规则。(将打印出 2 1)

表达式估值顺序

```
package main


var x, i = []int{1, 2}, 0
func f() int {i = 1; return 9}


func main() {
    x[i] = f()
    println(x[0], x[1])
}
```

打印出 `1 9` *或者* `9 2` *都不违反Go语言白皮书。因为表达式* `f()` *和* `i` *之间的相对估值顺序未定义。*

表达式估值顺序

```
package main

import "fmt"

func f(p *int) int {
    *p = 123
    return *p
}
```

```
func g(x int) (a, b int) {
    return x, f(&x)
    // <=> a, b = x, f(&x);
    // return
}

func main() {
    fmt.Println(g(0))
}
```

gc编译器将打印出: `123 123`

gccgo编译器将打印出: `0 123`

表达式估值顺序

```
package main


var (
    _ = f("w", x)
    x = f("x", z)
    y = f("y", x)
    z = f("z")
)
```

```
func f(s string, deps ...int) int {
    print(s)
    return 0
}

func main() {}
```

将打印出 `zxwy`

如何高效地完美clone一个切片

```
// 方法一：
b = make([]T, len(a))
copy(b, a)
```

```
// 方法二：
b = append(a[:0:0], a...)
```

```
$ go version
go version go1.13.4 linux/amd64
$ go test -bench=.
Benchmark_MakeAndCopy-4        847    1385659 ns/op
Benchmark_Append-4             1576      723186 ns/op
```


数组和切片组合字面值中下标规则

```
package main

import "fmt"

var x = []int{2: 5, 6, 0: 7}

func main() {
  fmt.Println(x) // [7 0 5 6]
}
```

- *数组和切片组合字面值中的下标必须为整数常量；*
- *在一个数组或切片组合字面值中，如果一个元素的索引下标缺失，则编译器认为它的索引下标为出现在它之前的元素的索引下标加一。*

容器组合字面值中键值或者下标规则

一个容器组合字面值中的常量键值(包括索引下标)不可重复。

此规则可以被用来实现编译时刻断言。

```
var _ = map[bool]int{false: 0, aCompileTimeFact: 1}
```

```
const S1 = "Hello world"
const S2 = ""
var _ = map[bool]int{false: 0, len(S1) != 0: 1}
var _ = map[bool]int{false: 0, len(S2) != 0: 1} // 编译报错
```

[Sp|Fp|P]rintf函数支持位置参数

```
package main

import "fmt"

func main() {
    // 将打印出：coco
    fmt.Printf("%[2]v%[1]v%[2]v%[1]v", "o", "c")
}
```

官方编译器对字节切片的解读不一致

```
type MyByte byte

var s = "Golang"

var x = []byte(s)
var y = []MyByte(s)

var a = string(x) // 没问题
var b = string(y) // 编译报错
```

```
func f() {
    copy(x, s) // 没问题
    copy(y, s) // 编译报错
    _ = append(x, s...) // 没问题
    _ = append(y, s...) // 编译报错
}
```

gccgo编译器不会报错

https://github.com/golang/go/issues/23536

无法直接转换但是可以间接转换

```
type MyInt int
type IntPtr *int
type MyIntPtr *MyInt

var x IntPtr
var y *int
var z MyIntPtr
var w *MyInt
```

```
func main() {
    z = MyIntPtr(x) // 编译报错
    z = MyIntPtr((*MyInt)((*int)(x)))
}
```

*MyInt和*int的值可以相互转换是因为它们的基类型MyInt和int的底层类型一致。(这是一条特殊的指针类型转换规则。)

一个指针类型的基类型可以是它自己
一个切片类型的元素类型可以是它自己

```
type P *P

type S []S

func f() {
    var p P; p = &p
    p = ******************p
    var s = make(S, 1); s[0] = s
    s = s[0][0][0][0][0][0][0][0]
}
```


一个字符串衔接的编译器优化

```
import "testing"


var s string
var x = []byte{1023: 'x'}
var y = []byte{1023: 'y'}


func fc() {
    s = (" " + string(x) + string(y))[1:] // 更高效
}
```


一个字符串衔接的编译器优化(续)

```
func fd() {
    s = string(x) + string(y) // 较低效
}

func main() {
    fmt.Println(testing.AllocsPerRun(1, fc)) // 1
    fmt.Println(testing.AllocsPerRun(1, fd)) // 3
}
```


参考资料

1. Go 101项目：https://github.com/golang101/golang101
2. Go 101官网：https://gfw.go101.org
3. Go 101公众号