# Go语言流程控制

< [Go语言make和new关键字的区别及实现原理](http://c.biancheng.net/view/vip_7307.html)[Go语言分支结构](http://c.biancheng.net/view/44.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/golang/flow_control/_blank)

流程控制是每种编程语言控制逻辑走向和执行次序的重要部分，流程控制可以说是一门语言的“经脉”。  
  
Go 语言的常用流程控制有 if 和 for，而 switch 和 goto 主要是为了简化代码、降低重复代码而生的结构，属于扩展类的流程控制。  
  
本章主要介绍了 Go 语言中的基本流程控制语句，包括分支语句（if 和 switch）、循环（for）和跳转（goto）语句。另外，还有循环控制语句（break 和 continue），前者的功能是中断循环或者跳出 switch 判断，后者的功能是继续 for 的下一个循环。

# Go语言if else（分支结构）

< [流程控制](http://c.biancheng.net/golang/flow_control/)[Go语言循环结构](http://c.biancheng.net/view/45.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

在Go语言中，关键字 if 是用于测试某个条件（布尔型或逻辑型）的语句，如果该条件成立，则会执行 if 后由大括号{}括起来的代码块，否则就忽略该代码块继续执行后续的代码。

1. if condition {
2. // do something
3. }

如果存在第二个分支，则可以在上面代码的基础上添加 else 关键字以及另一代码块，这个代码块中的代码只有在条件不满足时才会执行，if 和 else 后的两个代码块是相互独立的分支，只能执行其中一个。

1. if condition {
2. // do something
3. } else {
4. // do something
5. }

如果存在第三个分支，则可以使用下面这种三个独立分支的形式：

1. if condition1 {
2. // do something
3. } else if condition2 {
4. // do something else
5. }else {
6. // catch-all or default
7. }

else if 分支的数量是没有限制的，但是为了代码的可读性，还是不要在 if 后面加入太多的 else if 结构，如果必须使用这种形式，则尽可能把先满足的条件放在前面。  
  
关键字 if 和 else 之后的左大括号{必须和关键字在同一行，如果你使用了 else if 结构，则前段代码块的右大括号}必须和 else if 关键字在同一行，这两条规则都是被编译器强制规定的。  
  
非法的 Go 代码:

1. if x{
2. }
3. else { // 无效的
4. }

要注意的是，在使用 gofmt 格式化代码之后，每个分支内的代码都会缩进 4 个或 8 个空格，或者是 1 个 tab，并且右大括号}与对应的 if 关键字垂直对齐。  
  
在有些情况下，条件语句两侧的括号是可以被省略的，当条件比较复杂时，则可以使用括号让代码更易读，在使用 &&、|| 或 ! 时可以使用括号来提升某个表达式的运算优先级，并提高代码的可读性。

## 举例

通过下面的例子来了解 if 的写法：

1. var ten int = 11
2. if ten > 10 {
3. fmt.Println(">10")
4. } else {
5. fmt.Println("<=10")
6. }

代码输出如下：

>10

代码说明如下：

* 第 1 行，声明整型变量并赋值 11。
* 第 2 行，判断当 ten 的值大于 10 时执行第 3 行，否则执行第 4 行。
* 第 3 和第 5 行，分别打印大于 10 和小于等于 10 时的输出。

## 特殊写法

if 还有一种特殊的写法，可以在 if 表达式之前添加一个执行语句，再根据变量值进行判断，代码如下：

1. if err := Connect(); err != nil {
2. fmt.Println(err)
3. return
4. }

Connect 是一个带有返回值的函数，err:=Connect() 是一个语句，执行 Connect 后，将错误保存到 err 变量中。  
  
err != nil 才是 if 的判断表达式，当 err 不为空时，打印错误并返回。  
  
这种写法可以将返回值与判断放在一行进行处理，而且返回值的作用范围被限制在 if、else 语句组合中。

#### 提示

在编程中，变量的作用范围越小，所造成的问题可能性越小，每一个变量代表一个状态，有状态的地方，状态就会被修改，函数的局部变量只会影响一个函数的执行，但全局变量可能会影响所有代码的执行状态，因此限制变量的作用范围对代码的稳定性有很大的帮助。

# Go语言for（循环结构）

< [Go语言分支结构](http://c.biancheng.net/view/44.html)[输出九九乘法表](http://c.biancheng.net/view/46.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

与多数语言不同的是，Go语言中的循环语句只支持 for 关键字，而不支持 while 和 do-while 结构，关键字 for 的基本使用方法与C语言和 [C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) 中非常接近：

1. sum := 0
2. for i := 0; i < 10; i++ {
3. sum += i
4. }

可以看到比较大的一个不同在于 for 后面的条件表达式不需要用圆括号()括起来，Go语言还进一步考虑到无限循环的场景，让开发者不用写无聊的 for(;;){}和do{} while(1);，而直接简化为如下的写法：

1. sum := 0
2. for {
3. sum++
4. if sum > 100 {
5. break
6. }
7. }

使用循环语句时，需要注意的有以下几点：

* 左花括号{必须与 for 处于同一行。
* Go语言中的 for 循环与C语言一样，都允许在循环条件中定义和初始化变量，唯一的区别是，Go语言不支持以逗号为间隔的多个赋值语句，必须使用平行赋值的方式来初始化多个变量。
* Go语言的 for 循环同样支持 continue 和 break 来控制循环，但是它提供了一个更高级的 break，可以选择中断哪一个循环，如下例：

1. for j := 0; j < 5; j++ {
2. for i := 0; i < 10; i++ {
3. if i > 5 {
4. break JLoop
5. }
6. fmt.Println(i)
7. }
8. }
9. JLoop:
10. // ...

上述代码中，break 语句终止的是 JLoop 标签处的外层循环。

## for 中的初始语句——开始循环时执行的语句

初始语句是在第一次循环前执行的语句，一般使用初始语句执行变量初始化，如果变量在此处被声明，其作用域将被局限在这个 for 的范围内。  
  
初始语句可以被忽略，但是初始语句之后的分号必须要写，代码如下：

1. step := 2
2. for ; step > 0; step-- {
3. fmt.Println(step)
4. }

这段代码将 step 放在 for 的前面进行初始化，for 中没有初始语句，此时 step 的作用域就比在初始语句中声明 step 要大。

## for 中的条件表达式——控制是否循环的开关

每次循环开始前都会计算条件表达式，如果表达式为 true，则循环继续，否则结束循环，条件表达式可以被忽略，忽略条件表达式后默认形成无限循环。

#### 1) 结束循环时带可执行语句的无限循环

下面代码忽略条件表达式，但是保留结束语句，代码如下：

1. var i int
2. for ; ; i++ {
3. if i > 10 {
4. break
5. }
6. }

代码说明如下：

* 第 3 行，无须设置 i 的初始值，因此忽略 for 的初始语句，两个分号之间是条件表达式，也被忽略，此时循环会一直持续下去，for 的结束语句为 i++，每次结束循环前都会调用。
* 第 5 行，判断 i 大于 10 时，通过 break 语句跳出 for 循环到第 9 行。

#### 2) 无限循环

上面的代码还可以改写为更美观的写法，代码如下：

1. var i int
2. for {
3. if i > 10 {
4. break
5. }
6. i++
7. }

代码说明如下：

* 第 3 行，忽略 for 的所有语句，此时 for 执行无限循环。
* 第 9 行，将 i++ 从 for 的结束语句放置到函数体的末尾是等效的，这样编写的代码更具有可读性。

无限循环在收发处理中较为常见，但需要无限循环有可控的退出方式来结束循环。

#### 3) 只有一个循环条件的循环

在上面代码的基础上进一步简化代码，将 if 判断整合到 for 中，变为下面的代码：

1. var i int
2. for i <= 10 {
3. i++
4. }

在代码第 3 行中，将之前使用if i>10{}判断的表达式进行取反，变为判断 i 小于等于 10 时持续进行循环。  
  
上面这段代码其实类似于其他编程语言中的 while，在 while 后添加一个条件表达式，满足条件表达式时持续循环，否则结束循环。

## for 中的结束语句——每次循环结束时执行的语句

在结束每次循环前执行的语句，如果循环被 break、goto、return、panic 等语句强制退出，结束语句不会被执行。

# Go语言输出九九乘法表

< [Go语言循环结构](http://c.biancheng.net/view/45.html)[Go语言键值循环](http://c.biancheng.net/view/47.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

熟悉了Go语言的基本循环格式后，让我们用一个例子来温习一遍吧。  
  
输出九九乘法表：

1. package main
2. import "fmt"
3. func main() {
4. // 遍历, 决定处理第几行
5. for y := 1; y <= 9; y++ {
6. // 遍历, 决定这一行有多少列
7. for x := 1; x <= y; x++ {
8. fmt.Printf("%d\*%d=%d ", x, y, x\*y)
9. }
10. // 手动生成回车
11. fmt.Println()
12. }
13. }

结果输出如下：

1\*1=1

1\*2=2 2\*2=4

1\*3=3 2\*3=6 3\*3=9

1\*4=4 2\*4=8 3\*4=12 4\*4=16

1\*5=5 2\*5=10 3\*5=15 4\*5=20 5\*5=25

1\*6=6 2\*6=12 3\*6=18 4\*6=24 5\*6=30 6\*6=36

1\*7=7 2\*7=14 3\*7=21 4\*7=28 5\*7=35 6\*7=42 7\*7=49

1\*8=8 2\*8=16 3\*8=24 4\*8=32 5\*8=40 6\*8=48 7\*8=56 8\*8=64

1\*9=9 2\*9=18 3\*9=27 4\*9=36 5\*9=45 6\*9=54 7\*9=63 8\*9=72 9\*9=81

代码说明如下：

* 第 8 行，生成 1～9 的数字，对应乘法表的每一行，也就是被乘数。
* 第 11 行，乘法表每一行中的列数随着行数的增加而增加，这一行的 x 表示该行有多少列。
* 第 16 行，打印一个空行，实际作用就是换行。

这段程序按行优先打印，打印完一行，换行（第 16 行），接着执行下一行乘法表直到整个数值循环完毕。

# Go语言for range（键值循环）

< [输出九九乘法表](http://c.biancheng.net/view/46.html)[Go语言switch语句](http://c.biancheng.net/view/48.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

for range 结构是Go语言特有的一种的迭代结构，在许多情况下都非常有用，for range 可以遍历数组、切片、字符串、map 及通道（channel），for range 语法上类似于其它语言中的 foreach 语句，一般形式为：

for key, val := range coll {  
    ...  
}

需要要注意的是，val 始终为集合中对应索引的值拷贝，因此它一般只具有只读性质，对它所做的任何修改都不会影响到集合中原有的值。一个字符串是 Unicode 编码的字符（或称之为 rune ）集合，因此也可以用它来迭代字符串：

for pos, char := range str {  
    ...  
}

每个 rune 字符和索引在 for range 循环中是一一对应的，它能够自动根据 UTF-8 规则识别 Unicode 编码的字符。  
  
通过 for range 遍历的返回值有一定的规律：

* 数组、切片、字符串返回索引和值。
* map 返回键和值。
* 通道（channel）只返回通道内的值。

## 遍历数组、切片——获得索引和值

在遍历代码中，key 和 value 分别代表切片的下标及下标对应的值，下面的代码展示如何遍历切片，数组也是类似的遍历方法：

1. for key, value := range []int{1, 2, 3, 4} {
2. fmt.Printf("key:%d value:%d\n", key, value)
3. }

代码输出如下：

key:0  value:1  
key:1  value:2  
key:2  value:3  
key:3  value:4

## 遍历字符串——获得字符

Go语言和其他语言类似，可以通过 for range 的组合，对字符串进行遍历，遍历时，key 和 value 分别代表字符串的索引和字符串中的每一个字符。  
  
下面这段代码展示了如何遍历字符串：

1. var str = "hello 你好"
2. for key, value := range str {
3. fmt.Printf("key:%d value:0x%x\n", key, value)
4. }

代码输出如下：

key:0 value:0x68  
key:1 value:0x65  
key:2 value:0x6c  
key:3 value:0x6c  
key:4 value:0x6f  
key:5 value:0x20  
key:6 value:0x4f60  
key:9 value:0x597d

代码中的变量 value，实际类型是 rune 类型，以十六进制打印出来就是字符的编码。

## 遍历 map——获得 map 的键和值

对于 map 类型来说，for range 遍历时，key 和 value 分别代表 map 的索引键 key 和索引对应的值，一般被称为 map 的键值对，因为它们是一对一对出现的，下面的代码演示了如何遍历 map。

1. m := map[string]int{
2. "hello": 100,
3. "world": 200,
4. }
5. for key, value := range m {
6. fmt.Println(key, value)
7. }

代码输出如下：

hello 100  
world 200

#### 注意

对 map 遍历时，遍历输出的键值是无序的，如果需要有序的键值对输出，需要对结果进行排序。

## 遍历通道（channel）——接收通道数据

for range 可以遍历通道（channel），但是通道在遍历时，只输出一个值，即管道内的类型对应的数据。  
  
下面代码为我们展示了通道的遍历：

1. c := make(chan int)
2. go func() {
3. c <- 1
4. c <- 2
5. c <- 3
6. close(c)
7. }()
8. for v := range c {
9. fmt.Println(v)
10. }

代码说明如下：

* 第 1 行创建了一个整型类型的通道。
* 第 3 行启动了一个 goroutine，其逻辑的实现体现在第 5～8 行，实现功能是往通道中推送数据 1、2、3，然后结束并关闭通道。
* 这段 goroutine 在声明结束后，在第 9 行马上被执行。
* 从第 11 行开始，使用 for range 对通道 c 进行遍历，其实就是不断地从通道中取数据，直到通道被关闭。

## 在遍历中选择希望获得的变量

在使用 for range 循环遍历某个对象时，一般不会同时需要 key 或者 value，这个时候可以采用一些技巧，让代码变得更简单，下面将前面的例子修改一下，参考下面的代码示例：

1. m := map[string]int{
2. "hello": 100,
3. "world": 200,
4. }
5. for \_, value := range m {
6. fmt.Println(value)
7. }

代码输出如下：

100  
200

在上面的例子中将 key 变成了下划线\_，这里的下划线就是匿名变量。

* 可以理解为一种占位符。
* 匿名变量本身不会进行空间分配，也不会占用一个变量的名字。
* 在 for range 可以对 key 使用匿名变量，也可以对 value 使用匿名变量。

再看一个匿名变量的例子：

1. for key, \_ := range []int{1, 2, 3, 4} {
2. fmt.Printf("key:%d \n", key)
3. }

代码输出如下：

key:0  
key:1  
key:2  
key:3

在该例子中，value 被设置为匿名变量，只使用 key，而 key 本身就是切片的索引，所以例子输出索引。  
  
我们总结一下 for 的功能：

* Go语言的 for 包含初始化语句、条件表达式、结束语句，这 3 个部分均可缺省。
* for range 支持对数组、切片、字符串、map、通道进行遍历操作。
* 在需要时，可以使用匿名变量对 for range 的变量进行选取。

# Go语言switch case语句

< [Go语言键值循环](http://c.biancheng.net/view/47.html)[Go语言goto语句](http://c.biancheng.net/view/49.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

Go语言的 switch 要比C语言的更加通用，表达式不需要为常量，甚至不需要为整数，case 按照从上到下的顺序进行求值，直到找到匹配的项，如果 switch 没有表达式，则对 true 进行匹配，因此，可以将 if else-if else 改写成一个 switch。  
  
相对于C语言和 [Java](http://c.biancheng.net/java/" \t "http://c.biancheng.net/view/_blank) 等其它语言来说，Go语言中的 switch 结构使用上更加灵活，语法设计尽量以使用方便为主。

## 基本写法

Go语言改进了 switch 的语法设计，case 与 case 之间是独立的代码块，不需要通过 break 语句跳出当前 case 代码块以避免执行到下一行，示例代码如下：

1. var a = "hello"
2. switch a {
3. case "hello":
4. fmt.Println(1)
5. case "world":
6. fmt.Println(2)
7. default:
8. fmt.Println(0)
9. }

代码输出如下：

1

上面例子中，每一个 case 均是字符串格式，且使用了 default 分支，Go语言规定每个 switch 只能有一个 default 分支。

#### 1) 一分支多值

当出现多个 case 要放在一起的时候，可以写成下面这样：

1. var a = "mum"
2. switch a {
3. case "mum", "daddy":
4. fmt.Println("family")
5. }

不同的 case 表达式使用逗号分隔。

#### 2) 分支表达式

case 后不仅仅只是常量，还可以和 if 一样添加表达式，代码如下：

1. var r int = 11
2. switch {
3. case r > 10 && r < 20:
4. fmt.Println(r)
5. }

注意，这种情况的 switch 后面不再需要跟判断变量。

## 跨越 case 的 fallthrough——兼容C语言的 case 设计

在Go语言中 case 是一个独立的代码块，执行完毕后不会像C语言那样紧接着执行下一个 case，但是为了兼容一些移植代码，依然加入了 fallthrough 关键字来实现这一功能，代码如下：

1. var s = "hello"
2. switch {
3. case s == "hello":
4. fmt.Println("hello")
5. fallthrough
6. case s != "world":
7. fmt.Println("world")
8. }

代码输出如下：

hello  
world

新编写的代码，不建议使用 fallthrough。

# Go语言goto语句——跳转到指定的标签

< [Go语言switch语句](http://c.biancheng.net/view/48.html)[Go语言break（跳出循环）](http://c.biancheng.net/view/50.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

Go语言中 goto 语句通过标签进行代码间的无条件跳转，同时 goto 语句在快速跳出循环、避免重复退出上也有一定的帮助，使用 goto 语句能简化一些代码的实现过程。

## 使用 goto 退出多层循环

下面这段代码在满足条件时，需要连续退出两层循环，使用传统的编码方式如下：

1. package main
2. import "fmt"
3. func main() {
4. var breakAgain bool
5. // 外循环
6. for x := 0; x < 10; x++ {
7. // 内循环
8. for y := 0; y < 10; y++ {
9. // 满足某个条件时, 退出循环
10. if y == 2 {
11. // 设置退出标记
12. breakAgain = true
13. // 退出本次循环
14. break
15. }
16. }
17. // 根据标记, 还需要退出一次循环
18. if breakAgain {
19. break
20. }
21. }
22. fmt.Println("done")
23. }

代码说明如下：

* 第 10 行，构建外循环。
* 第 13 行，构建内循环。
* 第 16 行，当 y==2 时需要退出所有的 for 循环。
* 第 19 行，默认情况下循环只能一层一层退出，为此就需要设置一个状态变量 breakAgain，需要退出时，设置这个变量为 true。
* 第 22 行，使用 break 退出当前循环，执行后，代码调转到第 28 行。
* 第 28 行，退出一层循环后，根据 breakAgain 变量判断是否需要再次退出外层循环。
* 第 34 行，退出所有循环后，打印 done。

将上面的代码使用Go语言的 goto 语句进行优化：

1. package main
2. import "fmt"
3. func main() {
4. for x := 0; x < 10; x++ {
5. for y := 0; y < 10; y++ {
6. if y == 2 {
7. // 跳转到标签
8. goto breakHere
9. }
10. }
11. }
12. // 手动返回, 避免执行进入标签
13. return
14. // 标签
15. breakHere:
16. fmt.Println("done")
17. }

代码说明如下：

* 第 13 行，使用 goto 语句跳转到指明的标签处，标签在第 23 行定义。
* 第 20 行，标签只能被 goto 使用，但不影响代码执行流程，此处如果不手动返回，在不满足条件时，也会执行第 24 行代码。
* 第 23 行，定义 breakHere 标签。

使用 goto 语句后，无须额外的变量就可以快速退出所有的循环。

## 使用 goto 集中处理错误

多处错误处理存在代码重复时是非常棘手的，例如：

1. err := firstCheckError()
2. if err != nil {
3. fmt.Println(err)
4. exitProcess()
5. return
6. }
7. err = secondCheckError()
8. if err != nil {
9. fmt.Println(err)
10. exitProcess()
11. return
12. }
13. fmt.Println("done")

代码说明如下：

* 第 1 行，执行某逻辑，返回错误。
* 第 2～6 行，如果发生错误，打印错误退出进程。
* 第 8 行，执行某逻辑，返回错误。
* 第 10～14 行，发生错误后退出流程。
* 第 16 行，没有任何错误，打印完成。

在上面代码中，有一部分都是重复的错误处理代码，如果后期在这些代码中添加更多的判断，就需要在这些雷同的代码中依次修改，极易造成疏忽和错误。  
  
使用 goto 语句来实现同样的逻辑：

1. err := firstCheckError()
2. if err != nil {
3. goto onExit
4. }
5. err = secondCheckError()
6. if err != nil {
7. goto onExit
8. }
9. fmt.Println("done")
10. return
11. onExit:
12. fmt.Println(err)
13. exitProcess()

代码说明如下：

* 第 3 行和第 9 行，发生错误时，跳转错误标签 onExit。
* 第 17 行和第 18 行，汇总所有流程进行错误打印并退出进程。

# Go语言break（跳出循环）

< [Go语言goto语句](http://c.biancheng.net/view/49.html)[Go语言continue](http://c.biancheng.net/view/51.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

Go语言中 break 语句可以结束 for、switch 和 select 的代码块，另外 break 语句还可以在语句后面添加标签，表示退出某个标签对应的代码块，标签要求必须定义在对应的 for、switch 和 select 的代码块上。  
  
跳出指定循环：

1. package main
2. import "fmt"
3. func main() {
4. OuterLoop:
5. for i := 0; i < 2; i++ {
6. for j := 0; j < 5; j++ {
7. switch j {
8. case 2:
9. fmt.Println(i, j)
10. break OuterLoop
11. case 3:
12. fmt.Println(i, j)
13. break OuterLoop
14. }
15. }
16. }
17. }

代码输出如下：

0 2

代码说明如下：

* 第 7 行，外层循环的标签。
* 第 8 行和第 9 行，双层循环。
* 第 10 行，使用 switch 进行数值分支判断。
* 第 13 和第 16 行，退出 OuterLoop 对应的循环之外，也就是跳转到第 20 行。

# Go语言continue（继续下一次循环）

< [Go语言break（跳出循环）](http://c.biancheng.net/view/50.html)[示例：聊天机器人](http://c.biancheng.net/view/vip_7309.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

Go语言中 continue 语句可以结束当前循环，开始下一次的循环迭代过程，仅限在 for 循环内使用，在 continue 语句后添加标签时，表示开始标签对应的循环，例如：

1. package main
2. import "fmt"
3. func main() {
4. OuterLoop:
5. for i := 0; i < 2; i++ {
6. for j := 0; j < 5; j++ {
7. switch j {
8. case 2:
9. fmt.Println(i, j)
10. continue OuterLoop
11. }
12. }
13. }
14. }

代码输出结果如下：

0 2  
1 2

代码说明：第 14 行将结束当前循环，开启下一次的外层循环，而不是第 10 行的循环。

# 示例：聊天机器人

< [Go语言continue](http://c.biancheng.net/view/51.html)[示例：词频统计](http://c.biancheng.net/view/vip_7310.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

结合咱们之前的学习，本节带领大家来编写一个聊天机器人的雏形，下面的代码中展示了一个简单的聊天程序。

1. package main
2. import (
3. "bufio"
4. "fmt"
5. "os"
6. "strings"
7. )
8. func main() {
9. // 准备从标准输入读取数据。
10. inputReader := bufio.NewReader(os.Stdin)
11. fmt.Println("Please input your name:")
12. // 读取数据直到碰到 \n 为止。
13. input, err := inputReader.ReadString('\n')
14. if err != nil {
15. fmt.Printf("An error occurred: %s\n", err)
16. // 异常退出。
17. os.Exit(1)
18. } else {
19. // 用切片操作删除最后的 \n 。
20. name := input[:len(input)-2]
21. fmt.Printf("Hello, %s! What can I do for you?\n", name)
22. }
23. for {
24. input, err = inputReader.ReadString('\n')
25. if err != nil {
26. fmt.Printf("An error occurred: %s\n", err)
27. continue
28. }
29. input = input[:len(input)-2]
30. // 全部转换为小写。
31. input = strings.ToLower(input)
32. switch input {
33. case "":
34. continue
35. case "nothing", "bye":
36. fmt.Println("Bye!")
37. // 正常退出。
38. os.Exit(0)
39. default:
40. fmt.Println("Sorry, I didn't catch you.")
41. }
42. }
43. }

这个聊天程序在问候用户之后会不断地询问“是否可以帮忙”，但是实际上它什么忙也帮不上，因为它现在什么也听不懂，除了 nothing 和 bye，一看到这两个词，它就会与用户“道别”，停止运行，现在试运行一下这个命令源码文件：

D:\code>go run test.go  
Please input your name:  
->Robert  
Hello, Robert! What can I do for you?  
->A piece of cake, please.  
Sorry, I didn't catch you.  
->Bye  
Bye!

注意，其中的“->”符号之后的内容是我们输入的。

# Go语言词频统计

< [示例：聊天机器人](http://c.biancheng.net/view/vip_7309.html)[示例：缩进排序](http://c.biancheng.net/view/vip_7311.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

从数据挖掘到语言学习本身，文本分析功能的应用非常广泛，本一节我们来分析一个例子，它是文本分析最基本的一种形式：统计出一个文件里单词出现的频率。  
  
示例中频率统计后的结果以两种不同的方式显示，一种是将单词按照字母顺序把单词和频率排列出来，另一种是按照有序列表的方式把频率和对应的单词显示出来，完整的示例代码如下所示：

1. package main
2. import (
3. "bufio"
4. "fmt"
5. "io"
6. "log"
7. "os"
8. "path/filepath"
9. "runtime"
10. "sort"
11. "strings"
12. "unicode"
13. "unicode/utf8"
14. )
15. func main() {
16. if len(os.Args) == 1 || os.Args[1] == "-h" || os.Args[1] == "--help" {
17. fmt.Printf("usage: %s <file1> [<file2> [... <fileN>]]\n",
18. filepath.Base(os.Args[0]))
19. os.Exit(1)
20. }
21. frequencyForWord := map[string]int{} // 与:make(map[string]int)相同
22. for \_, filename := range commandLineFiles(os.Args[1:]) {
23. updateFrequencies(filename, frequencyForWord)
24. }
25. reportByWords(frequencyForWord)
26. wordsForFrequency := invertStringIntMap(frequencyForWord)
27. reportByFrequency(wordsForFrequency)
28. }
29. func commandLineFiles(files []string) []string {
30. if runtime.GOOS == "windows" {
31. args := make([]string, 0, len(files))
32. for \_, name := range files {
33. if matches, err := filepath.Glob(name); err != nil {
34. args = append(args, name) // 无效模式
35. } else if matches != nil {
36. args = append(args, matches...)
37. }
38. }
39. return args
40. }
41. return files
42. }
43. func updateFrequencies(filename string, frequencyForWord map[string]int) {
44. var file \*os.File
45. var err error
46. if file, err = os.Open(filename); err != nil {
47. log.Println("failed to open the file: ", err)
48. return
49. }
50. defer file.Close()
51. readAndUpdateFrequencies(bufio.NewReader(file), frequencyForWord)
52. }
53. func readAndUpdateFrequencies(reader \*bufio.Reader,
54. frequencyForWord map[string]int) {
55. for {
56. line, err := reader.ReadString('\n')
57. for \_, word := range SplitOnNonLetters(strings.TrimSpace(line)) {
58. if len(word) > utf8.UTFMax ||
59. utf8.RuneCountInString(word) > 1 {
60. frequencyForWord[strings.ToLower(word)] += 1
61. }
62. }
63. if err != nil {
64. if err != io.EOF {
65. log.Println("failed to finish reading the file: ", err)
66. }
67. break
68. }
69. }
70. }
71. func SplitOnNonLetters(s string) []string {
72. notALetter := func(char rune) bool { return !unicode.IsLetter(char) }
73. return strings.FieldsFunc(s, notALetter)
74. }
75. func invertStringIntMap(intForString map[string]int) map[int][]string {
76. stringsForInt := make(map[int][]string, len(intForString))
77. for key, value := range intForString {
78. stringsForInt[value] = append(stringsForInt[value], key)
79. }
80. return stringsForInt
81. }
82. func reportByWords(frequencyForWord map[string]int) {
83. words := make([]string, 0, len(frequencyForWord))
84. wordWidth, frequencyWidth := 0, 0
85. for word, frequency := range frequencyForWord {
86. words = append(words, word)
87. if width := utf8.RuneCountInString(word); width > wordWidth {
88. wordWidth = width
89. }
90. if width := len(fmt.Sprint(frequency)); width > frequencyWidth {
91. frequencyWidth = width
92. }
93. }
94. sort.Strings(words)
95. gap := wordWidth + frequencyWidth - len("Word") - len("Frequency")
96. fmt.Printf("Word %\*s%s\n", gap, " ", "Frequency")
97. for \_, word := range words {
98. fmt.Printf("%-\*s %\*d\n", wordWidth, word, frequencyWidth,
99. frequencyForWord[word])
100. }
101. }
102. func reportByFrequency(wordsForFrequency map[int][]string) {
103. frequencies := make([]int, 0, len(wordsForFrequency))
104. for frequency := range wordsForFrequency {
105. frequencies = append(frequencies, frequency)
106. }
107. sort.Ints(frequencies)
108. width := len(fmt.Sprint(frequencies[len(frequencies)-1]))
109. fmt.Println("Frequency → Words")
110. for \_, frequency := range frequencies {
111. words := wordsForFrequency[frequency]
112. sort.Strings(words)
113. fmt.Printf("%\*d %s\n", width, frequency, strings.Join(words, ", "))
114. }
115. }

程序的运行结果如下所示。

PS D:\code> go run .\main.go small-file.txt  
Word       Frequency  
ability                     1  
about                     1  
above                     3  
years                      1  
you                    128  
  
  
Frequency → Words  
    1 ability, about, absence, absolute, absolutely, abuse, accessible, ...  
    2 accept, acquired, after, against, applies, arrange, assumptions, ...  
...  
128    you  
151    or  
192    to  
221    of  
345    the

其中，small-file.txt 为待统计的文件名，它不是固定的，可以根据实际情况自行调整。由于输出的结果太多，所以上面只截取了部分内容。  
  
通过上面的输出结果可以看出，第一种输出是比较直接的，我们可以使用一个map[string]int类型的结构来保存每一个单词的频率，但是要得到第二种输出结果我们需要将整个映射反转成多值类型的映射，如map[int][]string，也就是说，键是频率而值则是所有具有这个频率的单词。  
  
接下来我们将从程序的 main() 函数开始，从上到下分析。

1. func main() {
2. if len(os.Args) == 1 || os.Args[1] == "-h" || os.Args[1] == "--help" {
3. fmt.Printf("usage: %s <file1> [<file2> [... <fileN>]]\n",
4. filepath.Base(os.Args[0]))
5. os.Exit(1)
6. }
7. frequencyForWord := map[string]int{} // 与:make(map[string]int)相同
8. for \_, filename := range commandLineFiles(os.Args[1:]) {
9. updateFrequencies(filename, frequencyForWord)
10. }
11. reportByWords(frequencyForWord)
12. wordsForFrequency := invertStringIntMap(frequencyForWord)
13. reportByFrequency(wordsForFrequency)
14. }

main() 函数首先分析命令行参数，之后再进行相应处理。  
  
我们使用复合语法创建一个空的映射，用来保存从文件读到的每一个单词和对应的频率，接着我们遍历从命令行得到的每一个文件，分析每一个文件后更新 frequencyForWord 的数据。  
  
得到第一个映射之后，我们就可以输出第一个报告了（按照字母顺序排列的列表），然后我们创建一个反转的映射，输出第二个报告（按出现频率统计并排序的列表）。

1. func commandLineFiles(files []string) []string {
2. if runtime.GOOS == "windows" {
3. args := make([]string, 0, len(files))
4. for \_, name := range files {
5. if matches, err := filepath.Glob(name); err != nil {
6. args = append(args, name) // 无效模式
7. } else if matches != nil {
8. args = append(args, matches...)
9. }
10. }
11. return args
12. }
13. return files
14. }

因为 Unix 类系统（如 Linux 或 Mac OS X 等）的命令行工具默认会自动处理通配符（也就是说，\*.txt 能匹配任意后缀为 .txt 的文件，如 README.txt 和 INSTALL.txt 等），而 Windows 平台的命令行工具（CMD）不支持通配符，所以如果用户在命令行输入 \*.txt，那么程序只能接收到 \*.txt。  
  
为了保持平台之间的一致性，这里使用 commandLineFiles() 函数来实现跨平台的处理，当程序运行在 Windows 平台时，实现文件名通配功能。

1. func updateFrequencies(filename string, frequencyForWord map[string]int) {
2. var file \*os.File
3. var err error
4. if file, err = os.Open(filename); err != nil {
5. log.Println("failed to open the file: ", err)
6. return
7. }
8. defer file.Close()
9. readAndUpdateFrequencies(bufio.NewReader(file), frequencyForWord)
10. }

updateFrequencies() 函数纯粹就是用来处理文件的，它打开给定的文件，并使用 defer 在函数返回时关闭文件，这里我们将文件作为一个 \*bufio.Reader（使用 bufio.NewReader() 函数创建）传给 readAndUpdateFrequencies() 函数，因为这个函数是以字符串的形式一行一行地读取数据的，所以实际的工作都是在 readAndUpdateFrequencies() 函数里完成的，代码如下。

1. func readAndUpdateFrequencies(reader \*bufio.Reader, frequencyForWord map[string]int) {
2. for {
3. line, err := reader.ReadString('\n')
4. for \_, word := range SplitOnNonLetters(strings.TrimSpace(line)) {
5. if len(word) > utf8.UTFMax || utf8.RuneCountInString(word) > 1 {
6. frequencyForWord[strings.ToLower(word)] += 1
7. }
8. }
9. if err != nil {
10. if err != io.EOF {
11. log.Println("failed to finish reading the file: ", err)
12. }
13. break
14. }
15. }
16. }

第一部分的代码我们应该很熟悉了，用了一个无限循环来一行一行地读一个文件，当读到文件结尾或者出现错误的时候就退出循环，将错误报告给用户但并不退出程序，因为还有很多其他的文件需要去处理。  
  
任意一行都可能包括标点、数字、符号或者其他非单词字符，所以我们需要逐个单词地去读，将每一行分隔成若干个单词并使用 SplitOnNonLetters() 函数忽略掉非单词的字符，并且过滤掉字符串开头和结尾的空白。  
  
只需要记录含有两个以上（包括两个）字母的单词，可以通过使用 if 语句，如 utf8.RuneCountlnString(word) > 1 来完成。  
  
上面描述的 if 语句有一点性能损耗，因为它会分析整个单词，所以在这个程序里我们增加了一个判断条件，用来检査这个单词的字节数是否大于 utf8.UTFMax（utf8.UTFMax 是一个常量，值为 4，用来表示一个 UTF-8 字符最多需要几个字节）。

1. func SplitOnNonLetters(s string) []string {
2. notALetter := func(char rune) bool { return !unicode.IsLetter(char) }
3. return strings.FieldsFunc(s, notALetter)
4. }

SplitOnNonLetters() 函数用来在非单词字符上对一个字符串进行切分，首先我们为 strings.FieldsFunc() 函数创建一个匿名函数 notALetter，如果传入的是字符那就返回 false，否则返回 true，然后返回调用函数 strings.FieldsFunc() 的结果，调用的时候将给定的字符串和 notALetter 作为它的参数。

1. func reportByWords(frequencyForWord map[string]int) {
2. words := make([]string, 0, len(frequencyForWord))
3. wordWidth, frequencyWidth := 0, 0
4. for word, frequency := range frequencyForWord {
5. words = append(words, word)
6. if width := utf8.RuneCountInString(word); width > wordWidth {
7. wordWidth = width
8. }
9. if width := len(fmt.Sprint(frequency)); width > frequencyWidth {
10. frequencyWidth = width
11. }
12. }
13. sort.Strings(words)
14. gap := wordWidth + frequencyWidth - len("Word") - len("Frequency")
15. fmt.Printf("Word %\*s%s\n", gap, " ", "Frequency")
16. for \_, word := range words {
17. fmt.Printf("%-\*s %\*d\n", wordWidth, word, frequencyWidth,
18. frequencyForWord[word])
19. }
20. }

计算出了 frequencyForWord 之后，调用 reportByWords() 将它的数据打印出来，因为我们需要将输出结果按照字母顺序排序好，所以首先要创建一个空的容量足够大的 []string 切片来保存所有在 frequencyForWord 里的单词。  
  
第一个循环遍历映射里的所有项，把每个单词追加到 words 字符串切片里去，使用 append() 函数只需要把给定的单词追加到第 len(words) 个索引位置上即可，words 的长度会自动增加 1。  
  
得到了 words 切片之后，对它进行排序，这个在 readAndUpdateFrequencies() 函数中已经处理好了。  
  
经过排序之后我们打印两列标题，第一个是 "Word"，为了能让 Frequency 最后一个字符 y 右对齐，需要在 "Word" 后打印一些空格，通过%\*s可以实现的打印固定长度的空白，也可以使用 %s来打印 strings.Repeat(" ", gap) 返回的字符串。  
  
最后，我们将单词和它们的频率用两列方式按照字母顺序打印出来。

1. func invertStringIntMap(intForString map[string]int) map[int][]string {
2. stringsForInt := make(map[int][]string, len(intForString))
3. for key, value := range intForString {
4. stringsForInt[value] = append(stringsForInt[value], key)
5. }
6. return stringsForInt
7. }

上面的函数首先创建一个空的映射，用来保存反转的结果，但是我们并不知道它到底要保存多少个项，因此我们假设它和原来的映射容量一样大，然后简单地遍历原来的映射，将它的值作为键保存到反转的映射里，并将键增加到对应的值里去，新的映射的值就是一个字符串切片，即使原来的映射有多个键对应同一个值，也不会丢掉任何数据。

1. func reportByFrequency(wordsForFrequency map[int][]string) {
2. frequencies := make([]int, 0, len(wordsForFrequency))
3. for frequency := range wordsForFrequency {
4. frequencies = append(frequencies, frequency)
5. }
6. sort.Ints(frequencies)
7. width := len(fmt.Sprint(frequencies[len(frequencies)-1]))
8. fmt.Println("Frequency → Words")
9. for \_, frequency := range frequencies {
10. words := wordsForFrequency[frequency]
11. sort.Strings(words)
12. fmt.Printf("%\*d %s\n", width, frequency, strings.Join(words, ", "))
13. }
14. }

这个函数的结构和 reportByWords() 函数很相似，它首先创建一个切片用来保存频率，并按照频率升序排列，然后再计算需要容纳的最大长度并以此作为第一列的宽度，之后输出报告的标题，最后，遍历输出所有的频率并按照字母升序输出对应的单词，如果一个频率有超过两个对应的单词则单词之间使用逗号分隔开。

# Go语言缩进排序

< [示例：词频统计](http://c.biancheng.net/view/vip_7310.html)[示例：二分查找算法](http://c.biancheng.net/view/vip_7312.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

本节将通过实例为大家演示如何将字符串按照等级（缩进级别）进行排序，完整代码如下所示。

1. package main
2. import (
3. "fmt"
4. "sort"
5. "strings"
6. )
7. var original = []string{
8. "Nonmetals",
9. " Hydrogen",
10. " Carbon",
11. " Nitrogen",
12. " Oxygen",
13. "Inner Transitionals",
14. " Lanthanides",
15. " Europium",
16. " Cerium",
17. " Actinides",
18. " Uranium",
19. " Plutonium",
20. " Curium",
21. "Alkali Metals",
22. " Lithium",
23. " Sodium",
24. " Potassium",
25. }
26. func main() {
27. fmt.Println("| Original | Sorted |")
28. fmt.Println("|-------------------|-------------------|")
29. sorted := SortedIndentedStrings(original) // 最初是 []string
30. for i := range original { // 在全局变量中设置
31. fmt.Printf("|%-19s|%-19s|\n", original[i], sorted[i])
32. }
33. }
34. func SortedIndentedStrings(slice []string) []string {
35. entries := populateEntries(slice)
36. return sortedEntries(entries)
37. }
38. func populateEntries(slice []string) Entries {
39. indent, indentSize := computeIndent(slice)
40. entries := make(Entries, 0)
41. for \_, item := range slice {
42. i, level := 0, 0
43. for strings.HasPrefix(item[i:], indent) {
44. i += indentSize
45. level++
46. }
47. key := strings.ToLower(strings.TrimSpace(item))
48. addEntry(level, key, item, &entries)
49. }
50. return entries
51. }
52. func computeIndent(slice []string) (string, int) {
53. for \_, item := range slice {
54. if len(item) > 0 && (item[0] == ' ' || item[0] == '\t') {
55. whitespace := rune(item[0])
56. for i, char := range item[1:] {
57. if char != whitespace {
58. i++
59. return strings.Repeat(string(whitespace), i), i
60. }
61. }
62. }
63. }
64. return "", 0
65. }
66. func addEntry(level int, key, value string, entries \*Entries) {
67. if level == 0 {
68. \*entries = append(\*entries, Entry{key, value, make(Entries, 0)})
69. } else {
70. addEntry(level-1, key, value,
71. &((\*entries)[entries.Len()-1].children))
72. }
73. }
74. func sortedEntries(entries Entries) []string {
75. var indentedSlice []string
76. sort.Sort(entries)
77. for \_, entry := range entries {
78. populateIndentedStrings(entry, &indentedSlice)
79. }
80. return indentedSlice
81. }
82. func populateIndentedStrings(entry Entry, indentedSlice \*[]string) {
83. \*indentedSlice = append(\*indentedSlice, entry.value)
84. sort.Sort(entry.children)
85. for \_, child := range entry.children {
86. populateIndentedStrings(child, indentedSlice)
87. }
88. }
89. type Entry struct {
90. key string
91. value string
92. children Entries
93. }
94. type Entries []Entry
95. func (entries Entries) Len() int { return len(entries) }
96. func (entries Entries) Less(i, j int) bool {
97. return entries[i].key < entries[j].key
98. }
99. func (entries Entries) Swap(i, j int) {
100. entries[i], entries[j] = entries[j], entries[i]
101. }

注意 SortedIndentedStrings() 函数有一个很重要的前提就是，字符串的缩进是通过读到的空格或缩进的个数来决定的，下面来看一下输出结果，为了方便对比，这里将排序前的结果放在左边，排序后的结果放在右边。

| Original | Sorted |

|-------------------|-------------------|

|Nonmetals |Alkali Metals |

| Hydrogen | Lithium |

| Carbon | Potassium |

| Nitrogen | Sodium |

| Oxygen |Inner Transitionals|

|Inner Transitionals| Actinides |

| Lanthanides | Curium |

| Europium | Plutonium |

| Cerium | Uranium |

| Actinides | Lanthanides |

| Uranium | Cerium |

| Plutonium | Europium |

| Curium |Nonmetals |

|Alkali Metals | Carbon |

| Lithium | Hydrogen |

| Sodium | Nitrogen |

| Potassium | Oxygen |

其中，SortedIndentedStrings() 函数和它的辅助函数使用到了递归、函数引用以及指向切片的指针等。

1. type Entry struct {
2. key string
3. value string
4. children Entries
5. }
6. type Entries []Entry
7. func (entries Entries) Len() int { return len(entries) }
8. func (entries Entries) Less(i, j int) bool {
9. return entries[i].key < entries[j].key
10. }
11. func (entries Entries) Swap(i, j int) {
12. entries[i], entries[j] = entries[j], entries[i]
13. }

sort.Interface 接口定义了 3 个方法 Len()、Less() 和 Swap()，它们的函数签名和 Entries 中的同名方法是一样的，这就意味着我们可以使用标准库里的 sort.Sort() 函数来对一个 Entries 进行排序。

1. func SortedIndentedStrings(slice []string) []string {
2. entries := populateEntries(slice)
3. return sortedEntries(entries)
4. }

导出的函数 SortedIndentedStrings() 就做了这个工作，虽然我们已经对它进行了重构，让它把所有东西都传递给辅助函数，函数 populateEntries() 传入一个 []string 并返回一个对应的 Entries（[]Entry 类型）。  
  
而函数 sortedEntries() 需要传入一个 Entries，然后返回一个排过序的 []string（根据缩进的级别进行排序）。

1. func populateEntries(slice []string) Entries {
2. indent, indentSize := computeIndent(slice)
3. entries := make(Entries, 0)
4. for \_, item := range slice {
5. i, level := 0, 0
6. for strings.HasPrefix(item[i:], indent) {
7. i += indentSize
8. level++
9. }
10. key := strings.ToLower(strings.TrimSpace(item))
11. addEntry(level, key, item, &entries)
12. }
13. return entries
14. }

populateEntries() 函数首先以字符串的形式得到给定切片里的一级缩进（如有 4 个空格的字符串）和它占用的字节数，然后创建一个空的 Entries，并遍历切片里的每一个字符串，判断该字符串的缩进级别，再创建一个用于排序的键。  
  
下一步，调用自定义函数 addEntry()，将当前字符串的级别、键、字符串本身，以及指向 entries 的地址作为参数，这样 addEntry() 就能创建一个新的 Entry 并能够正确地将它追加到 entries 里去，最后返回 entries。

1. func computeIndent(slice []string) (string, int) {
2. for \_, item := range slice {
3. if len(item) > 0 && (item[0] == ' ' || item[0] == '\t') {
4. whitespace := rune(item[0])
5. for i, char := range item[1:] {
6. if char != whitespace {
7. i++
8. return strings.Repeat(string(whitespace), i), i
9. }
10. }
11. }
12. }
13. return "", 0
14. }

computeIndent() 函数主要是用来判断缩进使用的是什么字符，例如空格或者缩进符等，以及一个缩进级别占用多少个这样的字符。  
  
因为第一级的字符串可能没有缩进，所以函数必须迭代所有的字符串，一旦它发现某个字符串的行首是空格或者缩进，函数马上返回表示缩进的字符以及一个缩进所占用的字符数。

1. func addEntry(level int, key, value string, entries \*Entries) {
2. if level == 0 {
3. \*entries = append(\*entries, Entry{key, value, make(Entries, 0)})
4. } else {
5. addEntry(level-1, key, value,
6. &((\*entries)[entries.Len()-1].children))
7. }
8. }

addEntry() 是一个递归函数，它创建一个新的 Entry，如果这个 Entry 的 level 是 0，那就直接增加到 entries 里去，否则，就将它作为另一个 Entry 的子集。  
  
我们必须确定这个函数传入的是一个 \*Entries 而不是传递一个 entries 引用（切片的默认行为），因为我们是要将数据追加到 entries 里，追加到一个引用会导致无用的本地副本且原来的数据实际上并没有被修改。  
  
如果 level 是 0，表明这个字符串是顶级项，因此必须将它直接追加到 \*entries，实际上情况要更复杂一些，因为 level 是相对传入的 \*entries 而言的，第一次调用 addEntry() 时，\*entries 是一个第一级的 Entries，但函数进入递归后，\*entries 就可能是某个 Entry 的子集。  
  
我们使用内置的 append() 函数来追加新的 Entry，并使用 \* 操作符获得 entries 指针指向的值，这就保证了任何改变对调用者来说都是可见的，新增的 Entry 包含给定的 key 和 value，以及一个空的子 Entries，这是递归的结束条件。  
  
如果 level 大于 0，则我们必须将它追加到上一级 Entry 的 children 字段里去，这里我们只是简单地递归调用 addEntry() 函数，最后一个参数可能是我们目前为止见到的最复杂的表达式了。  
  
子表达式 entries.Len() - 1 产生一个 int 型整数，表示 \*entries 指向的 Entries 值的最后一个条目的索引位置（注意 Entries.Len() 传入的是一个 Entries 值而不是 \*Entries 指针，不过Go语言也可以自动对 entries 指针进行解引用并调用相应的方法）。  
  
完整的表达式（&(...) 除外）访问了 Entries 最后一个 Entry 的 children 字段（这也是一个 Entries 类型），所以如果把这个表达式作为一个整体，实际上我们是将 Entries 里最后一个 Entry 的 children 字段的内存地址作为递归调用的参数，因为 addEntry() 最后一个参数是 \*Entries 类型的。  
  
为了帮助大家弄清楚到底发生了什么，下面的代码和上述代码中 else 代码块中的那个调用是一样的。

theEntries := \*entries  
lastEntry := &theEntries[theEntries.Len()-1]  
addEntry(level-1, key, value, &lastEntry.children)

首先，我们创建 theEntries 变量用来保存 \*entries 指针指向的值，这里没有什么开销因为不会产生复制，实际上 theEntries 相当于一个指向 Entries 值的别名。  
  
然后我们取得最后一项的内存地址（即一个指针），如果不取地址的话就会取到最后一项的副本，最后递归调用 addEntry() 函数，并将最后一项的 children 字段的地址作为参数传递给它。

1. func sortedEntries(entries Entries) []string {
2. var indentedSlice []string
3. sort.Sort(entries)
4. for \_, entry := range entries {
5. populateIndentedStrings(entry, &indentedSlice)
6. }
7. return indentedSlice
8. }

当调用 sortedEntries() 函数的时候，Entries 显示的结构和原先程序输出的字符串是一样的，每一个缩进的字符串都是上一级缩进的子级，而且还可能有下一级的缩进，依次类推。  
  
创建了 Entries 之后，SortedIndentedStrings() 函数调用上面这个函数去生成一个排好序的字符串切片 []string，这个函数首先创建一个空的 []string 用来保存最后的结果，然后对 entries 进行排序。  
  
Entries 实现了 sort.Interface 接口，因此我们可以直接使用 sort.Sort() 函数根据 Entry 的 key 字段来对 Entries 进行排序（这是 Entries.Less() 的实现方式），这个排序只是作用于第一级的 Entry，对其他未排序的子集是没有任何影响的。  
  
为了能够对 children 字段以及 children 的 children 等进行递归排序，函数遍历第一级的每一个项并调用 populateIndentedStrings() 函数，传入这个 Entry 类型的项和一个指向 []string 切片的指针。  
  
切片可以传递给函数并由函数更新内容（如替换切片里的某些项），但是这里需要往切片里新增一些数据，所以这里将一个指向切片的指针（也就是指针的指针）作为参数传进去，并将指针指向的内容设置为 append() 函数的返回结果，可能是一个新的切片，也可能是原先的切片。  
  
另一种办法就是传入切片的值，然后返回 append() 之后的切片，但是必须将返回的结果赋值给原来的切片变量（例如 slice = function(slice)），不过这么做的话，很难正确地使用递归函数。

1. func populateIndentedStrings(entry Entry, indentedSlice \*[]string) {
2. \*indentedSlice = append(\*indentedSlice, entry.value)
3. sort.Sort(entry.children)
4. for \_, child := range entry.children {
5. populateIndentedStrings(child, indentedSlice)
6. }
7. }

populateIndentedStrings() 函数将顶级项追加到创建的切片，然后对顶级项的子项进行排序，并递归调用自身对每一个子项做同样的处理，这就相当于对每一项的子项以及子项的子项等都做了排序，所以整个字符串切片就是已经排好序的了。

# Go语言实现二分查找算法

< [示例：缩进排序](http://c.biancheng.net/view/vip_7311.html)[示例：冒泡排序](http://c.biancheng.net/view/vip_7313.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

二分查找也称折半查找（Binary Search），它是一种效率较高的查找方法。但是，二分查找算法的前提是传入的序列是有序的（降序或升序），并且有一个目标值。  
  
二分查找的核心思想是将 n 个元素分成大致相等的两部分，取中间值 a[n/2] 与 x 做比较，如果 x=a[n/2]，则找到 x，算法中止，如果 x<a[n/2]，则只要在数组 a 的左半部分继续搜索 x，如果 x>a[n/2]，则只要在数组 a 的右半部搜索 x。  
  
二分查找虽然性能比较优秀，但应用场景也比较有限，底层必须依赖数组，并且还要求数据是有序的，对于较小规模的数据查找，我们直接使用顺序遍历就可以了，二分查找的优势并不明显，二分查找更适合处理静态数据，也就是没有频繁插入、删除操作的数据。

#### 思路：

1) 先找到中间的下标 middle = (leftIndex + RightIndex) /2 ，然后用中间的下标值和需要查找的值（ FindVal）比较。

a：如果 arr[middle] > FindVal，那么就向 LeftIndex ~ (midlle-1) 区间找  
b：如果 arr[middle] < FindVal，那么就向 middle+1 ~ RightIndex 区间找  
c：如果 arr[middle] == FindVal，那么直接返回

2) 从第一步的 a、b、c 递归执行，直到找到位置。  
  
3) 如果 LeftIndex > RightIndex，则表示找不到，退出。

#### 代码/举例：

定义一个包含（1, 2, 5, 7, 15, 25, 30, 36, 39, 51, 67, 78, 80, 82, 85, 91, 92, 97）等值的数组，假设说要查找 30 这个值，如果按照循环的查找方法，找到 30 这个值要执行 7 次，那么如果是按照二分查找呢？二分查找的过程如下：

left = 1, right = 18; mid = (1+18)/2 = 9; 51 > 30  
left = 1, right = mid - 1 = 8; mid = (1+8)/2 = 4; 15 < 30  
left = mid + 1 = 5, right = 8; mid = (5+8)/2 = 6; 30 = 30 查找完毕

如上所示只需要执行 3 次，大大减少了执行时间，具体代码实现如下所示：

1. package main
2. import (
3. "fmt"
4. )
5. //二分查找函数 //假设有序数组的顺序是从小到大（很关键，决定左右方向）
6. func BinaryFind(arr \*[]int, leftIndex int, rightIndex int, findVal int) {
7. //判断 leftIndex是否大于rightIndex
8. if leftIndex > rightIndex {
9. fmt.Println("找不到")
10. return
11. }
12. //先找到 中间的下标
13. middle := (leftIndex + rightIndex) / 2
14. if (\*arr)[middle] > findVal {
15. //要查找的数，范围应该在 leftIndex 到 middle+1
16. BinaryFind(arr, leftIndex, middle-1, findVal)
17. } else if (\*arr)[middle] < findVal {
18. //要查找的数，范围应该在 middle+1 到 rightIndex
19. BinaryFind(arr, middle+1, rightIndex, findVal)
20. } else {
21. fmt.Printf("找到了，下标为：%v \n", middle)
22. }
23. }
24. func main() {
25. //定义一个数组
26. arr := []int{1, 2, 5, 7, 15, 25, 30, 36, 39, 51, 67, 78, 80, 82, 85, 91, 92, 97}
27. BinaryFind(&arr, 0, len(arr) - 1, 30)
28. fmt.Println("main arr=",arr)
29. }

执行结果如下所示：

D:\code>go run main.go  
找到了，下标为：6  
main arr= [1 2 5 7 15 25 30 36 39 51 67 78 80 82 85 91 92 97]

# Go语言冒泡排序

< [示例：二分查找算法](http://c.biancheng.net/view/vip_7312.html)[Go语言分布式id生成器](http://c.biancheng.net/view/vip_7314.html) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

冒泡排序法是一种最简单的交换类排序方法，它是通过相邻数据的交换逐步将无序列表排列为有序列表。  
  
冒泡排序的基本原理是重复地循环遍历要排序的元素列，依次比较两个相邻的元素，如果顺序（如从小到大或者首字母从 Z 到 A）错误就把两个元素的位置交换过来，直到没有相邻元素需要交换，也就是说该元素列已经排序完成。  
  
冒泡排序的名字由来是因为越小的元素会经由交换慢慢“浮”到数列的顶端（升序或降序排列），就如同气泡最终会上浮到顶端一样，故名“冒泡排序”。  
  
下面通过一个实例来演示一下冒泡排序，完整代码如下：

1. package main
2. import (
3. "fmt"
4. )
5. /\*\*
6. 冒泡排序
7. \*/
8. func main() {
9. arr := [...]int{21,32,12,33,34,34,87,24}
10. var n = len(arr)
11. fmt.Println("--------没排序前--------\n",arr)
12. for i := 0; i <= n-1; i++ {
13. fmt.Println("--------第",i+1,"次冒泡--------")
14. for j := i; j <= n-1; j++ {
15. if arr[i] > arr[j] {
16. t := arr[i]
17. arr[i] = arr[j]
18. arr[j] = t
19. }
20. fmt.Println(arr)
21. }
22. }
23. fmt.Println("--------最终结果--------\n",arr)
24. }

执行结果如下所示：

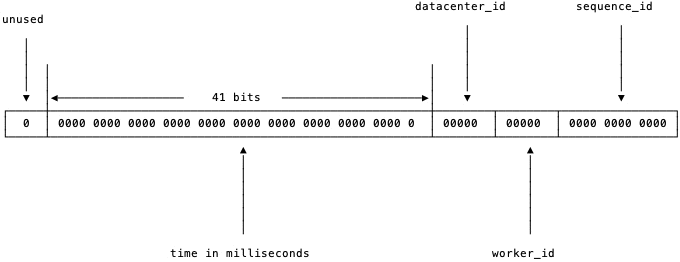
D:\code>go run main.go  
--------没排序前--------  
[21 32 12 33 34 34 87 24]  
--------第 1 次冒泡--------  
[21 32 12 33 34 34 87 24]  
[21 32 12 33 34 34 87 24]  
[12 32 21 33 34 34 87 24]  
[12 32 21 33 34 34 87 24]  
[12 32 21 33 34 34 87 24]  
[12 32 21 33 34 34 87 24]  
[12 32 21 33 34 34 87 24]  
[12 32 21 33 34 34 87 24]  
--------第 2 次冒泡--------  
[12 32 21 33 34 34 87 24]  
[12 21 32 33 34 34 87 24]  
[12 21 32 33 34 34 87 24]  
[12 21 32 33 34 34 87 24]  
[12 21 32 33 34 34 87 24]  
[12 21 32 33 34 34 87 24]  
[12 21 32 33 34 34 87 24]  
--------第 3 次冒泡--------  
[12 21 32 33 34 34 87 24]  
[12 21 32 33 34 34 87 24]  
[12 21 32 33 34 34 87 24]  
[12 21 32 33 34 34 87 24]  
[12 21 32 33 34 34 87 24]  
[12 21 24 33 34 34 87 32]  
--------第 4 次冒泡--------  
[12 21 24 33 34 34 87 32]  
[12 21 24 33 34 34 87 32]  
[12 21 24 33 34 34 87 32]  
[12 21 24 33 34 34 87 32]  
[12 21 24 32 34 34 87 33]  
--------第 5 次冒泡--------  
[12 21 24 32 34 34 87 33]  
[12 21 24 32 34 34 87 33]  
[12 21 24 32 34 34 87 33]  
[12 21 24 32 33 34 87 34]  
--------第 6 次冒泡--------  
[12 21 24 32 33 34 87 34]  
[12 21 24 32 33 34 87 34]  
[12 21 24 32 33 34 87 34]  
--------第 7 次冒泡--------  
[12 21 24 32 33 34 87 34]  
[12 21 24 32 33 34 34 87]  
--------第 8 次冒泡--------  
[12 21 24 32 33 34 34 87]  
--------最终结果--------  
[12 21 24 32 33 34 34 87]

# Go语言分布式id生成器

< [示例：冒泡排序](http://c.biancheng.net/view/vip_7313.html)[Go语言函数](http://c.biancheng.net/golang/func/) >

[C语言中文网推出辅导班啦，包括「C语言辅导班、C++辅导班、算法/数据结构辅导班」，全部都是一对一教学：一对一辅导 + 一对一答疑 + 布置作业 + 项目实践 + 永久学习。QQ在线，随时响应！](http://fudao.biancheng.net/" \t "http://c.biancheng.net/view/_blank)

有时我们需要能够⽣成类似 MySQL 自增 ID 这样不断增大，同时又不会重复的 ID。以支持业务中的⾼并发场景。比较典型的是电商促销时短时间内会有大量的订单涌入到系统，比如每秒 10w+。明星出轨时会有大量热情的粉丝发微博以表心意，同样会在短时间内产生大量的消息。  
  
在插入数据库之前，我们需要给这些消息、订单先打上一个 ID，然后再插⼊到我们的数据库。对这个 ID 的要求是希望其中能带有一些时间信息，这样即使我们后端的系统对消息进行了分库分表，也能够以时间顺序对这些消息进⾏排序。  
  
Twitter 的 snowflake 算法是这种场景下的一个典型解法。先来看看 snowflake 是怎么回事，如下图所示：

  
图：snowflake 中的比特位分布

⾸先确定我们的数值是 64 位的 int64 类型，被划分为了四部分，不含开头的第一个 bit，因为这个 bit 是符号位。用 41 位来表示收到请求时的时间戳，单位为毫秒，然后五位来表示数据中心的 ID，然后再五位来表示机器的实例 ID，最后是 12 位的循环自增 ID（到达 1111,1111,1111 后会归 0）。  
  
这样的机制可以⽀持我们在同一台机器上，同一毫秒内产⽣ 2 ^ 12 = 4096 条消息。一秒共 409.6 万条消息。从值域上来讲完全够⽤了。  
  
数据中心加上实例 ID 共有 10 位，可以⽀持我们每数据中心部署 32 台机器，所有数据中心共 1024 台实例。  
  
表示 timestamp 的 41 位，可以⽀持我们使用 69 年。当然，我们的时间毫秒计数不会真的从 1970 年开始记，那样我们的系统跑到 2039/9/7 23:47:35 就不能用了，所以这里的 timestamp 实际上只是相对于某个时间的增量，比如我们的系统上线是 2018-08-01，那么我们可以把这个 timestamp 当作是从 2018-08-01 00:00:00.000 的偏移量。

## worker\_id 分配

timestamp、datacenter\_id、worker\_id 和 sequence\_id 这四个字段中，timestamp 和 sequence\_id 是由程序在运⾏期⽣成的。但 datacenter\_id 和 worker\_id 需要我们在部署阶段就能够获取得到，并且一旦程序启动之后，就是不可更改的了（想想，如果可以随意更改，可能被不慎修改，造成最终生成的 ID 有冲突）。  
  
一般不同数据中⼼的机器，会提供对应的获取数据中心 ID 的 API，所以 datacenter\_id 我们可以在部署阶段轻松地获取到。而 worker\_id 是我们逻辑上给机器分配的一个 ID，这个要怎么办呢？比较简单的想法是由能够提供这种自增 ID 功能的工具来支持，比如 MySQL:

mysql> insert into a (ip) values("10.1.2.101");  
Query OK, 1 row affected (0.00 sec)  
mysql> select last\_insert\_id();  
+------------------+  
| last\_insert\_id()     |  
+------------------+  
| 2                          |  
+------------------+  
1 row in set (0.00 sec)

从 MySQL 中获取到 worker\_id 之后，就把这个 worker\_id 直接持久化到本地，以避免每次上线时都需要获取新的 worker\_id，让单实例的 worker\_id 可以始终保持不变。  
  
当然，使用 MySQL 相当于给我们简单的 id 生成服务增加了一个外部依赖，依赖越多，我们的服务的可运维性就越差。  
  
考虑到集群中即使有单个 ID 生成服务的实例挂了，也就是损失一段时间的一部分 ID，所以我们也可以更简单暴力一些，把 worker\_id 直接写在 worker 的配置中，上线时由部署脚本完成 worker\_id 字段替换。

## 标准 snowflake 实现

github.com/bwmarrin/snowflake 是一个相当轻量化的 snowflake 的 Go 实现。其文档对各位使用的定义如下图所示。

IMG_257  
图：snowflake库

和标准的 snowflake 完全一致。使用上比较简单：

1. package main
2. import (
3. "fmt"
4. "os"
5. "github.com/bwmarrin/snowflake"
6. )
7. func main() {
8. n, err := snowflake.NewNode(1)
9. if err != nil {
10. println(err)
11. os.Exit(1)
12. }
13. for i := 0; i < 3; i++ {
14. id := n.Generate()
15. fmt.Println("id", id)
16. fmt.Println(
17. "node: ", id.Node(),
18. "step: ", id.Step(),
19. "time: ", id.Time(),
20. "\n",
21. )
22. }
23. }

运行结果如下：

go run main.go  
id 1218796327232606208  
node:  1 step:  0 time:  1579418663210  
  
id 1218796327232606209  
node:  1 step:  1 time:  1579418663210  
  
id 1218796327232606210  
node:  1 step:  2 time:  1579418663210

当然，这个库也给我们留好了定制的后路，其中预留了一些可定制字段：

// Epoch is set to the twitter snowflake epoch of Nov 04 2010 01:42:54 UTC  
// You may customize this to set a different epoch for your application.  
Epoch int64 = 1288834974657  
  
// Number of bits to use for Node  
// Remember, you have a total 22 bits to share between Node/Step  
NodeBits uint8 = 10  
  
// Number of bits to use for Step  
// Remember, you have a total 22 bits to share between Node/Step  
StepBits uint8 = 12

Epoch 就是本节开头讲的起始时间，NodeBits 指的是机器编号的位⻓，StepBits 指的是自增序列的位⻓。

## sonyflake

sonyflake 是 Sony 公司的一个开源项目，基本思路和 snowflake 差不多，不过位分配上稍有不同，如下图所示：

IMG_258  
图：sonyflake

这⾥的时间只用了 39 个 bit，但时间的单位变成了 10ms，所以理论上比 41 位表示的时间还要久（174 年）。  
  
Sequence ID 和之前的定义一致，Machine ID 其实就是节点 ID。sonyflake 与众不同的地方在于其在启动阶段的参数配置：

func NewSonyflake(st Settings) \*Sonyflake

Settings 数据结构如下：

type Settings struct {  
    StartTime time.Time  
    MachineID func() (uint16, error)  
    CheckMachineID func(uint16) bool  
}

StartTime 选项和我们之前的 Epoch 差不多，如果不设置的话，默认是从 2014-09-01 00:00:00 +0000 UTC 开始。  
  
MachineID 可以由用户自定义的函数，如果用户不定义的话，会默认将本机 IP 的低 16 位作为 machineid。  
  
CheckMachineID 是由用户提供的检查 MachineID 是否冲突的函数。这里的设计还是比较巧秒的，如果有另外的中心化存储并支持检查重复的存储，那我们就可以按照自己的想法随意定制这个检查 MachineID 是否冲突的逻辑。如果公司有现成的 Redis 集群，那么我们可以很轻松地用 Redis 的集合类型来检查冲突。

redis 127.0.0.1:6379> SADD base64\_encoding\_of\_last16bits MzI0Mgo=  
(integer) 1  
redis 127.0.0.1:6379> SADD base64\_encoding\_of\_last16bits MzI0Mgo=  
(integer) 0

使用起来也比较简单，这里省略了一些逻辑简单的函数：

[纯文本复制](http://c.biancheng.net/view/vip_7314.html)

1. package main
2. import (
3. "fmt"
4. "os"
5. "time"
6. "github.com/sony/sonyflake"
7. )
8. func getMachineID() (uint16, error) {
9. var machineID uint16
10. var err error
11. machineID = readMachineIDFromLocalFile()
12. if machineID == 0 {
13. machineID, err = generateMachineID()
14. if err != nil {
15. return 0, err
16. }
17. }
18. return machineID, nil
19. }
20. func checkMachineID(machineID uint16) bool {
21. saddResult, err := saddMachineIDToRedisSet()
22. if err != nil || saddResult == 0 {
23. return true
24. }
25. err := saveMachineIDToLocalFile(machineID)
26. if err != nil {
27. return true
28. }
29. return false
30. }
31. func main() {
32. t, \_ := time.Parse("2006-01-02", "2018-01-01")
33. settings := sonyflake.Settings{
34. StartTime: t,
35. MachineID: getMachineID,
36. CheckMachineID: checkMachineID,
37. }
38. sf := sonyflake.NewSonyflake(settings)
39. id, err := sf.NextID()
40. if err != nil {
41. fmt.Println(err)
42. os.Exit(1)
43. }
44. fmt.Println(id)
45. }