Go by Example

作者：ZephyrJung

原文链接：https://ld246.com/article/1497146010890

来源网站：链滴

本文收集了Go By Example的示例，并将注释写于代码之上，最后几节参考了其他人的翻译

Github 地址：原版 everyx中文翻译

Go by Example

Hello World

我们第一个程序就是打印经典的“hello world”，下面是完整的代码

```
package main
import "fmt"
func main(){
  fmt.Println("hello world")
}
```

要运行这个程序，将代码保存为hello-world.go，然后使用go run

有时候我们想让程序编译成二进制文件，可以使用go build，然后就可以直接运行了。

Values

Go有多种值的类型，包括string，integer，float，boolean等。如下是几个基本例子。

```
package main
import "fmt"
func main(){
  //string可以使用+连接在一起
  fmt.Println("go"+"lang")
  fmt.Println("1+1=",1+1)
  fmt.Pritnln("7.0/3.0=",7.0/3.0)
  fmt.Println(true&&false)
  fmt.Println(true||false)
  fmt.Pritnln(!true)
}
```

Variables

在Go中，变量被编译器显式的声明和使用，例如检查函数调用类型的正确性。

```
package main
import "fmt"
func main(){
  //声明一个或多个变量
  var a string="initial"
  fmt.Println(a)
  var b,c int = 1,2
  fmt.Println(b,c)
  //Go将推断变量的初始化类型
  var d = true
  fmt.Println(d)
  //声明没有初始值的变量将被初始化为零值，如int的零值是0
```

```
  var e int
  fmt.Println(e)
  //var f string = "short"的简化写法
  f:="short"
  fmt.Println(f)
}
```

Constants

Go支持的常量有字符、字符串、布尔值以及数值

```
package main
import "fmt"
import "math"
const s string = "constant"
func main(){
  fmt.Println(s)
  //const声明可以出现在任何var声明出现的地方
  const n = 50000000
  //常量表达式可以任意精度进行运算
  const d = 3e20 / n
  fmt.Println(d)
  //数值常量没有类型，除非进行了显式转换等类型赋予
  fmt.Println(int64(d))
  //数值在使用环境上下文中会得到类型，如变量赋值或者方法调用，如math.Sin需要的是一个float64
  fmt.Println(math.Sin(n))
}
```

For

for是Go中唯一的循环结构，下面是三种基本的for循环类型。

```
package main
import "fmt"
func main(){
  i := 1
  //最基本的类型，只有一个条件
  for i<=3{
    fmt.Println(i)
    i = i+1
  }
  //经典的 初始化/条件/循环后 for循环
  for j:=7;j<=9;j++{
    fmt.Println(j)
  }
  //没有条件的for语句将无限循环，除非在内部的方法中使用了break或者return
  for{
    fmt.Println("loop")
    break
  }
  //也可以使用continue来直接到下一个循环
  for n:=0;n<=5;n++ {
    if n%2 == 0{
```

```
      continue
    }
    fmt.Println(n)
  }
}
```

If/Else

```
package main
import "fmt"
func main(){
  if 7%2==0{
    fmt.Println("7 is even")
  }else{
    fmt.Println("7 is odd")
  }
  //可以单独使用if
  if 8%4==0{
    fmt.Println("8 is divisible by 4")
  }
  //条件之前可以有语句，声明在该语句中的任何变量可以用于所有分支
  if num:=9;num<0{
    fmt.Println(num,"is negative")
  }else if num<10{
    fmt.Println(num,"has 1 digit")
  }else{
    fmt.Println(num,"has multiple digits")
  }
}
```

注意，Go中条件周围不需要圆括号，但是花括号是必要的。

Go中没有三目if语句，所以对于最基本的条件也需要写完整的if语句。

Switch

```
package main
import "fmt"
import "time"
func main(){
  i:=2
  fmt.Print("write",i," as ")
  switch i{
  case 1: fmt.Println("one")
  case 2: fmt.Println("two")
  case 3: fmt.Println("three")
  }
  //可以在同一个case语句中使用逗号来分隔多个表达式
  switch time.Now().weekday(){
  case time.Saturday,time.Sunday:
    fmt.Println("it's the weekend")
  //这里也使用了default case
  default:
```

```
    fmt.Println("it's a weekday")
  }
  t:=time.Now()
  //没有表达式的switch是另一种实现if/else逻辑的路子，同时这也展示了case表达式可以是非常量
。
  switch{
  case t.Hour()<12:
    fmt.Println("it's before noon")
  default:
    fmt.Println("it's after noon")
  }
  //类型switch比较了类型而非值
  whatAmI := func(i interface{}){
    switch t:=i.(type){
    case bool:
      fmt.Println("I'm a bool")
    case int:
      fmt.Println("I'm an int")
    default:
      fmt.Println("Don't know type %T\n",t)
    }
  }
  whatAmI(true)
  whatAmI(1)
  whatAmI("hey")
}
```

Arrays

在Go中，数组是特定长度元素的编号序列。

```
package main
import "fmt"
func main(){
  //这里创建了一个5个int元素的数组。默认是零值，对于int而言就是0
  var a [5]int
  fmt.Println("emp:",a)
  //可以通过array[index]=value语法设值，或者通过array[index]取值
  a[4]=100
  fmt.Println("set:",a)
  fmt.Println("get:",a[4])
  //内置的len函数返回数组的长度
  fmt.Println("len:",len(a))
  //声明并初始化数组
  b:=[5]int{1,2,3,4,5}
  fmt.Prinln("dc1:",b)
  //数组是一维的，但你可以组合类型来构建多维数组结构
  var twoD [2][3]int
  for i:=0;i<2;i++{
    for j:=0;j<3;j++{
      twoD[i][j]=i+j
    }
  }
  fmt.Println("2d: ",twoD)
```

```
}
```

当使用fmt.Println方法打印时，数组将以[v1 v2 v3 ...]的形式展现

Slices

slice是一个重要的数据类型，对于序列提供了比数组更强大的接口

```
package main
import "fmt"
func main(){
  //与数组不同，切片仅由其包含的元素（不是元素的数量）键入。
  //要创建一个非零长度的空切片，请使用内置的make。 这里我们制作长度为3的字符串（最初为零
) 。
  s:=make([]string,3)
  fmt.Println("emp:",s)
  //可以像数组一样set和get
  s[0]="a"
  s[1]="b"
  s[2]="c"
  fmt.Println("set:",s)
  fmt.Println("get:",s[2])
  //len能够返回slice的长度
  fmt.Println("len:",len(s))
  //作为这些基本操作的补充，slice支持一些令其比数组更丰富的东西。
  //其中一个是append函数，其返回一个包含一个或多个新值的slice
  //注意需要接受append的返回值来获取新的slice值
  s=append(s,"d")
  s=append(s,"e","f")
  fmt.Println("apd:",s)
  //slice可以复制
  c:=make([]string,len(s))
  copy(c,s)
  fmt.Println("cpy:",c)
  //slice支持切片操作，语法为slice[low:high]。如下将得到元素s[2],s[3]和s[4]
  l:=s[2:5]
  fmt.Println("sl1:",l)
  //如下切片截止到（不包含）s[5]
  l=s[:5]
  fmt.Pritnln("sl2:",l)
  //如下切片从s[2]开始（包含）
  l=s[2:]
  fmt.Println("sl3:",l)
  //声明并初始化slice
  t:=[]string{"g","h","i"}
  fmt.Println("dcl:",t)
  //slice也可以组织成一个多为数据结构，内部的slice长度可以变化，这与多维数组不同
  twoD:=make([][]int,3)
  for i:=0;i<3;i++{
    innerLen:=i+1
    twoD[i]=make([]int,innerLen)
    for j:=0;j<innerLen;j++{
      twoD[i][j]=i+j
    }
```

```
  }
  fmt.Println("2d: ",twoD)
}
```

虽然slice与array是不同的类型，但是使用fmt.Println的展示结果很相似

Maps

Map是Go内置的关联数据类型（其他语言可能成为哈希或者字典）

```
package main
import "fmt"
func main(){
  //要创建一个空的map，使用内置make函数：make(map[key-type]val-type)
  m:=make(map[string]int)
  //通过经典的name[key]=val语法来设置key/value对
  m["k1"]=7
  m["k2"]=13
  fmt.Println("map:",m)
  //通过name[key]来获取一个value
  v1:=m["k1"]
  fmt.Println("v1: ",v1)
  //len函数返回map中键值对的个数
  fmt.Println("len:",len(m))
  //内置的delete函数将移除map中的键值对
  delete(m,"k2")
  fmt.Println("map:",m)
  //第一个值是该key的value，但此处不需要，故使用空白占位符“_”忽略
  //第二个可选的返回值表明该键是否在map中，这样可以消除不存在的键，和键值为0或者""的歧义
  _,prs:=m["k2"]
  fmt.Println("prs:",prs)
  //声明并初始化map
  n:=map[string]int{"foo":1,"bar":2}
  fmt.Println("map:",n)
}
```

map将以[k:v k:v]的形式打印

Range

range可以遍历各种数据结构中的元素。

```
package main
import "fmt"
func main(){
  //这里使用range来计算元素的和，数组也是类似的用法
  nums:=[]int{2,3,4}
  sum:=0
  for _,num:=range nums{
    sum+=num
  }
  fmt.Println("sum:",sum)
  //在slice和array上的range均为每个条目提供了索引和值
```

```
  for i,num:=range nums{
    if num==3{
      fmt.Pritnln("index",i)
    }
  }
  //map上的range通过key/value对进行遍历
  kvs:=map[string]string{"a":"apple","b":"banana"}
  for k,v:=range kvs{
    fmt.Printf("%s -> %s\n",k,v)
  }
  //range可以仅通过key进行遍历
  for k:= range kvs{
    fmt.Println("key:",k)
  }
  //range作用在string上将得到unicode code points，第一个值是字符的起始字节索引，第二个值
字符本身
  for i,c:range "go" {
    cmt.Println(i,c)
  }
}
```

Functions

```
package main
import "fmt"
//这个函数接收两个int并以int类型返回他们的和
func plus(a int,b int) int{
  //Go需要显式的return语句，它不会自动返回最后一个表达式的值
  return a+b
}
//如果有多个连续的相同类型的参数，可以忽略前面的类型声明
func plusPlus(a,b,c int) int{
  return a+b+c
}
func main(){
  //如你所想的那样调用函数
  res := plus(1,2)
  fmt.Println("1+2=",res)
  res = plusPlus(1,2,3)
  fmt.Println("1+2+3=",res)
}
```

Multiple Return Values

Go内置支持了返回多个值。这一特点经常用于Go的习惯用法，例如同时返回结果和错误值

```
package main
import "fmt"
//方法签名中的(int,int)表明它将返回两个int值
func vals()(int,int){
  return 3,7
}
func main(){
```

```
  //这里我们通过多重赋值来使用两个不同的返回值
  a,b := vals()
  fmt.Println(a)
  fmt.Println(b)
  //如果只需要返回结果的子集，使用空白占位符_
  _,c := vals()
  fmt.Println(c)
}
```

Variadic Functions

变参函数，可以使用任意数量的参数来进行调用。例如fmt.Println是一个常见的变参函数。

```
package main
import "fmt"
//这是一个接收任意数量的int值的函数
func sum(nums ...int){
  fmt.Print(nums," ")
  total := 0
  for _, num := range nums{
    total += num
  }
  fmt.Println(total)
}
func main(){
  sum(1,2)
  sum(1,2,3)
  //如果你已经在一个slice中定义了多个参数，可以使用func(slice...)来直接应用到变参函数中
  nums :=[]int{1,2,3,4}
  sum(nums...)
}
```

Closures

Go支持匿名函数，它可以形成闭包。当你想要定义一个不记名的内部函数时，匿名函数就很有用了。

```
package main
import "fmt"
//intSeq函数返回另一个函数，它定义在了intSeq函数内部，并且是匿名的。
//返回的函数关闭变量i以形成闭包
func intSeq() func int{
  i := 0
  return func() int{
    i+=1
    return i
  }
}
func main(){
  //调用intSeq，并将结果(一个函数)赋予nextInt
  //这个函数持有一个自己的i值，每次调用nextInt时都会更新
  nextInt:=intSeq()
  //多次调用nextInt函数可以看到闭包的效果
  //zephyr:如非闭包写法，每次函数都会进行初始化变量i，反复调用intSeq不会有这个效果
```

```
  fmt.Println(nextInt())
  fmt.Println(nextInt())
  fmt.Pritnln(nextInt())
  //要确认该状态对该特定函数是唯一的，请创建并测试一个新函数。
  newInts:=intSeq()
  fmt.Println(newInts())
}
```

Recursion

Go支持递归函数。下面是一个经典的斐波那契数列示例

```
package main
import "fmt"
func fact(n int) int {
  if n==0 {
    return 1
  }
  //fact函数调用自身，直到n为0
  return n * fact(n-1)
}
func main(){
  fmt.Println(fact(7))
}
```

Pointers

Go可以使用指针，让你在程序中传递值或记录的引用。

下面通过两种方式的对比来展示指针的使用：zeroval和zeroptr

```
package main
import "fmt"
//zeroval将取得ival的值的拷贝，与调用方不同
func zeroval(ival int){
  ival = 0
}
//zeroptr有一个*int类型的参数，代表它接收的是一个指针
func zeroptr(iptr *int){
  //*iptr解引用，从内存指定地址中获取存放的值
  //对解引用指针的赋值将改变指定地址上的值
  *iptr = 0
}
func main(){
  i:=1
  fmt.Println("initial:",i)
  zeroval(i)
  fmt.Println("zeroval:",i)
  //&i 语法将获得变量i的内存地址，也就是指向变量i的指针
  zeroptr(&i)
  fmt.Println("zeroptr:",i)
  //指针也可以被打印
  fmt.Println("pointer:",&i)
```

```
}
```

zeroval没有改变main函数中i的值，而zeroptr会，因为它拥有指向变量i的内存地址。

Structs

Go的Struct结构是字段类型的集合。对于从记录中将数据组织到一起很有帮助。

```
package main
import "fmt"
type person struct{
  name string
  age int
}
func main(){
  //这个语法创建了一个新的struct
  fmt.Println(person{"Bob",20})
  //在初始化struct时，可以指定字段名
  fmt.Println(person{name:"Alice",age:30})
  //被忽略的字段将会被初始化为零
  fmt.Println(person{name:"Fred"})
  //一个&将产生struct的指针
  fmt.Println(&person{name:"Ann",age:40})
  s:=person{name:"Sean",age:50}
  //通过点号访问结构体中的字段
  fmt.Println(s.name)
  sp:=&s
  fmt.Println(sp.age)
  //对于结构体的指针也可以使用点号操作符，指针将会自动解引用
  sp.age=51
  //结构体是可变的
  fmt.Println(sp.age)
}
```

Methods

Go支持在结构体上定义方法。

```
package main
import "fmt"
type rect struct{
  width,height int
}
//area方法有一个rect指针类型的接收器
func (r *rect) area() int{
  return r.width * r.height
}
//即可以定义指针类型的接收器，也可以定义值类型的接收器
func (r rect) perim() int{
  return 2*r.width+2*r.height
}
func main(){
  r:=rect{width:10,height:5}
```

```
  //调用定义的方法
  fmt.Println("area: ",r.area())
  fmt.Println("perim: ",r.perim())
  //Go为方法调用自动处理了值和引用的转换。使用指针接收器可以避免获得方法调用的拷贝(?)或允
方法修改接收到的struct值
  rp:=&r
  fmt.Println("area: ",rp.area())
  fmt.Println("perim: ",rp.perim())
}
```

Interfaces

接口是方法签名的命名集合。

```
package main
import "fmt"
import "math"
//这是一个geometry的基本接口，本例中将在rect类型和circle类型中实现这个接口
type geometry interface{
  area() float64
  perim() float64
}
type rect struct{
  width,height float64
}
type circle struct{
  radius float64
}
//在Go中，实现一个接口只需要实现其中定义的所有方法即可
func (r rect) area() float64{
  return r.width * r.height
}
func (r rect) perim() float64{
  return 2*r.width+2*r.height
}
func (c circle) area() float64{
  return math.Pi * c.radius * c.radius
}
func (c circle) perim() float64{
  return 2 * math.Pi * c.radius
}
//如果变量是接口类型，那么它可以调用接口内定义的方法
//这是一个通用的measure方法，利用它能够工作在任何geometry上
func measure(g geometry){
  fmt.Println(g)
  fmt.Println(g.area())
  fmt.Println(g.perim())
}
func main(){
  r:=rect{width:3,height:4}
  c:=circle{radius:5}
  //rect和circle均实现了geometry接口，所以可以作为measure的参数
  measure(r)
  measure(c)
```

```
}
```

Errors

在Go中，传递错误的惯用法是通过明确的，分离的返回值。这和Java活Ruby中的exception以及C中载使用单个的结果/错误值不同。Go的方法使得很容易看出哪些函数返回错误，并使用与任何其他非误任务相同的语言结构来处理它们。

```
package main
import "errors"
import "fmt"
//按照惯例，错误是最后一个返回值，类型为error，一个内置的接口。
func f1(arg int)(int,error){
  if arg==42{
    //errors.New使用给定的错误信息构建了一个基本的error值
    return -1,errors.New("can't work with 42")
  }
  //在error位置上放nil值代表没有错误
  return arg+3,nil
}
//通过实现Error()方法，可以自定义错误类型。这里有一个上例的变种
//使用一个自定义类型来显式地展示一个参数错误
type argError struct{
  arg int
  prob string
}
type argError struct{
  arg int
  prob string
}
func (e *argError) Error() string{
  return fmt.Sprintf("%d - %s",e.arg,e.prob)
}
func f2(arg int)(int,error){
  if arg==42{
    //这里我们使用&argError语法来创建一个新的结构体，提供了arg和prob两个域的值
    return -1,&argError{arg,"can't work with it"}
  }
  return arg+3,nil
}
func main(){
  //下面的两个循环测试了每个返回error的函数
  //注意在if中的错误检查是Go代码中的习惯用法
  for _,i:=range []int{7,42}{
    if r,e:=f1(i);e!=nil{
      fmt.Println("f1 failed:",e)
    }else{
      fmt.Println("f1 worked:",r)
    }
  }
  for _,i := range []int{7,42}{
    if r,e :=f2(i);e!=nil{
      fmt.Println("f1 failed:",e)
    }else{
```

```
    fmt.Println("f1 worked:",r)
   }
  }
  //如果要以编程方式使用自定义错误中的数据，
  //则需要通过类型断言将错误作为自定义错误类型的实例获取。
  _,e:=f2(42)
  if ae,ok := e.(*argError);ok{
   fmt.Println(ae.arg)
   fmt.Println(ae.prob)
  }
}
```

Goroutines

goroutine是一个轻量级的执行线程。

```
package main
import "fmt"
func f(from string){
  for i:=0;i<3;i++{
    fmt.Println(from,";",i)
  }
}
func main(){
  //假设我们有个f(s)的函数调用。这里我们通过一般方法调用，令其同步执行
  f("direct")
  //要想让这个函数在goroutine中触发，使用go f(s)。这个新的goroutine将会与调用它的并行执行
  go f("goroutine")
  //我们也可以启动一个调用匿名函数的goroutine
  go func(msg string){
    fmt.Println(msg)
  }("going")
  //现在，这两个方法调用在独立的goroutine中异步执行了，故方法执行直接落到了这里
  //Scanln代码需要在程序退出前按下一个键
  var input string
  fmt.Scanln(&input)
  fmt.Println("done")
}
```

当我们运行这个程序的时候，我们将首先看到阻塞调用，然后是两个goroutine的交错输出。这个交反应了goroutine在Go运行时是并发执行的。

Channels

Channel是连接并发执行的goroutine的管道。你可以从一个goroutine传递值到channel中，再在另个goroutine接收它。

```
package main
import "fmt"
func main(){
  //通过make(chan val-type)创建新的channel
  //channel的类型依赖于它们要传递的值
  messages:=make(chan string)
```

```
  //向channel传递值使用channel <- 语法。在这里我们从一个新的goroutine中发送了一个"ping"
message通道中
  go func(){messages<-"ping"}()
  //<-channel语法从channel中获取值。这里我们接收了上面发送的"ping"信息并打印
  msg:=<-messages
  fmt.Println(msg)
}
```

当我们运行这个程序时，"ping"信息成功的通过我们的channel从一个goroutine传递到了另一个。

默认情况下，发送和接收在发送者和接受者都准备好之前阻塞。这个特性允许我们在程序结尾等待"pig"信息而无需使用其他的同步手段

Channel Buffering

默认下channel没有缓冲区，这意味着他们将只有在响应的接收者(<-chan)准备好时，才能允许发送(can<-)。具有缓冲区的channel，接受有限个数的值，而无需相应的接收者。

```
package main
import "fmt"
func main(){
  //这里我们创建了一个能够缓冲2个字符串值的channel
  messages:=make(chan string,2)
  //由于channel带有缓冲区，我们可以发送值，无需响应的并发接收
  messages<-"buffered"
  messages<-"channel"
  //稍后，我们像往常一样，接收了这两个值
  fmt.Println(<-messages)
  fmt.Println(<-messages)
}
```

Channel Synchronization

我们可以使用channel来跨goroutine同步执行。这里是一个使用阻塞接收来等待goroutine结束的例。

```
package main
import "fmt"
import "time"
//这个方法将在一个goroutine中运行。
//done channel用来通知其他的goroutine这个方法执行完毕
func worker(done chan bool){
  fmt.Print("working...")
  time.Sleep(time.Second)
  fmt.Println("done")
  //发送一个值来通知这里已经做完
  done<-true
}
func main(){
  启动一个worker goroutine，赋予它用以通知的channel
  done:=make(chan bool,1)
  go worker(done)
  //在channel接收到来自worker的通知前，保持阻塞
```

```
  <-done
}
```

如果你移除了<-done行，这个程序可能会在worker开始前就结束。

Channel Directions

当把channel用作函数的参数时，你可以指定一个channel是否只发送或者只接收数据。这种特异性加了程序的类型安全性。

```
package main
import "fmt"
//ping函数只接受一个发送数据的channel，如果试图在其上获取数据，将会引发编译时异常
func ping(pings chan<- string,msg string){
  pings<-msg
}
//pong函数接受一个通道用于接收(pings)，另一个用于发送(pongs)
func pong(pings <-chan string,pongs chan<- string){
  msg:=<-pings
  pongs<-msg
}
func main(){
  pings:=make(chan string,1)
  pongs:=make(chan string,1)
  ping(pings,"passed message")
  pong(pings,pongs)
  fmt.Println(<-pongs)
}
```

Select

Go的select让你能够等待多个channel操作。通过select结合goroutine和channel是Go的重要特色。

```
package main
import "time"
import "fmt"
func main(){
  //本例中我们将在两个通道中进行选择
  c1:=make(chan string)
  c2:=make(chan string)
  //每个通道都会在一定时间后接收到一个值，在并发的goroutine中模拟阻塞RPC操作执行
  go func(){
    time.Sleep(time.Second*1)
    c1<-"one"
  }()
  go func(){
    time.Sleep(time.Second*2)
    c2<-"two"
  }()
  //我们将使用select来同时等待这两个值，当它们到达时打印。
  for i:=0;i<2;i++{
    select{
    case msg1:=<-c1:
```

```
      fmt.Println("received",msg1)
    case msg2:=<-c2:
      fmt.Println("received",msg2)
    }
  }
}
```

按照预期，我们将接收到"one"，然后"two"。

注意，整个执行只需要大约2秒的时间，因为1秒和2秒的沉睡是并发执行的。

Timeouts

超时对于连接到外部资源的程序或其他需要绑定执行时间的程序很重要。在Go中，可以通过channel select轻松而优雅的实现超时。

```
package main
import "time"
import "fmt"
func main(){
  //在本例中，假设我们执行了一个外部调用，它在两秒后将结果返回到通道c1上
  c1:=make(chan string,1)
  go func(){
    time.Sleep(time.Second*2)
    c1<-"result 1"
  }()
  //这里是用select实现超时。res:=<-c1等待一个结果，而<-Time.After等待超时1秒后发送一个值。
  //由于select将在有第一个准备就绪的接收时继续，我们会在操作超过允许的1秒时进入超时事件。
  select{
  case res:=<-c1:
    fmt.Println(res)
  case <-time.After(time.Second*1):
      fmt.Println("timeout 1")
  }
  //如果我们允许一个更长的超时时间3秒，则将能成功得到c2的值，并打印
  c2:=make(chan string,1)
  go func(){
    time.Sleep(time.Second*2)
    c2<-"result 2"
  }()
  select{
  case res:=<-c2:
    fmt.Println(res)
  case <-time.After(time.Second*3):
    fmt.Printnln("timeout 2")
  }
}
```

运行这个程序，将显示第一个操作超时了，第二个则成功。

使用select超时模式需要在通道上进行结果通讯。一般情况下这是很好的主意，因为其他的重要Go特基于通道和选择。我们将在之后看到有关的两个例子：timer和ticker

Non-Blocking Channel Operations

channel上简单的发送和接收是阻塞的。然而，我们可以使用select和default子句来实现非阻塞发送接收甚至非阻塞的多路选择。

```
package main
import "fmt"
func main(){
  messages:=make(chan string)
  signals:=make(chan bool)
  //这是一个非阻塞的接收。如果message的值可以获取，select将随值进入<-message子句
  //否则将立刻进入default事件
  select{
  case msg:=<-messages:
    fmt.Println("received message",msg)
  default:
    fmt.Pritln("no message received")
  }
  msg:="hi"
  //类似的有非阻塞发送
  select{
  case messages<-msg:
    cmt.Println("sent message",msg)
  default:
    fmt.Println("no message sent")
  }
  //我们可以在default上使用多个事件来实现多路非阻塞select。
  //这里我们试图在message和signal上均进行非阻塞接收
  select{
  case msg:=<-messages:
    fmt.Println("received message",msg)
  case sig:=<-signals:
    fmt.Println("received signal",sig)
  default:
    fmt.Println("no activity")
  }
}
```

Closing Channels

关闭通道意味着不会再有值在其上发送。这对于通道的接收方通讯完成很有帮助

```
package main
import "fmt"
//在这个例子中，我们将使用一个作业通道将工作从主函数goroutine传递给工人 goroutine。当没更多作业给工人时，我们将关闭工作渠道。
func main(){
  jobs:=make(chan int,5)
  done:=make(chan bool)
  //下面是工人goroutine。它通过j,more:=<-jobs反复获取作业
  //在这个2返回值的接收中，如果作业关闭，所有值都已接收，more会变为false
  //我们用其在完成所有作业时进行已完成通知
  go func(){
    for{
      j,more:=<-jobs
      if more{
```

```
        fmt.Println("received job",j)
      }else{
        fmt.Println("received all jobs")
        done<-true
        return
      }
    }
  }()
  //此处向工人发送了3个作业，然后关闭它
  for j:=1;j<=3;j++{
    jobs<-j
    fmt.Println("sent job",j)
  }
  close(jobs)
  fmt.Println("sent all jobs")
  //使用前面见过的同步机制来等待工人、
  <-done
}
```

Range over Channels

在前面的例子中，我们看到如何使用for和range来遍历基本的数据结构。我们同样可以使用这个语法遍历通道上接收的值。

```
package main
import "fmt"
func main(){
  //遍历queue通道上的2个值
  queue:=make(chan string,2)
  queue<-"one"
  queue<-"two"
  close(queue)
  //range遍历通道上接收的每个值。由于在上面关闭了通道，这个遍历将在接收2个值后结束。
  for elem:=range queue{
    fmt.Println(elem)
  }
}
```

Timers

我们常想在未来的某一刻执行Go代码，或者在某一时间内重复执行。Go内置的timer和ticker使这些务十分简易。首先我们看看timer，然后再看ticker。

```
package main
import "time"
import "fmt"
func main(){
  //timer代表未来的一个单独事件。你要告诉它要等待多久，它提供一个通道，在指定时间发出通知
下面这个timer将等待2秒钟
  timer1:=time.NewTimer(time.Second*2)
  //定时器通道由于操作<-timer1.c发生阻塞，直到它发送了一个值来表明定时器到时
  <-timer1.C
  fmt.Println("Timer 1 expired")
```

```
  //如果你仅仅想等待一段时间，可以用time.Sleep，使用timer的一个原因是，你可以在计时结束前消，如下例：
  timer2:=time.NewTimer(time.Second)
  go func(){
    <-timer2.C
    fmt.Println("Timer 2 expired")
  }()
  stop2:=timer2.Stop()
  if stop2{
    fmt.Println("Timer 2 stopped")
  }
}
```

第一个timer将在启动程序后大约2秒到时，但第二个应会在其有机会到时前先行停止。

Tickers

timer用来在将来的某一时间做某事一次。而ticker会在一个指定时间间隔重复做某事。这里是一个ticer的例子，它会在我们停止之前定期触发。

```
package main
import "time"
import "fmt"
func main(){
  //ticker与timer的机制相似，都是一个发送值的通道
  //这里我们使用channel内置的range来遍历每500ms到达的值
  ticker:=time.NewTicker(time.Millisecond*500)
  go func(){
    for t:=range ticker.C{
      fmt.Prinltn("Tick at ",t)
    }
  }()
  //ticker可以像timer一样停止。一旦ticker停止，将不会在其通道上接收到任何信息。我们将在1600ms后结束。
  time.Sleep(time.Millisecond*1600)
  ticker.Stop()
  fmt.Println("Ticker stopped")
}
```

运行这个程序，在结束之前，应该会tick三次。

Worker Pools

本例中我们将看到如何使用goroutine和channel来实现一个工人池

```
package main
import "fmt"
import "time"
//这里是我们将要运行多个并发实例的工人。他们将从jobs通道获取任务，并将相应的结果返回到results上。我们将在每个作业沉睡1秒，来模拟一个复杂的任务。
func worker(id int,jobs <-chan int,results chan<- int){
  for j:=range jobs{
    fmt.Println("worker",id,"started job",j)
```

```
    time.Sleep(time.Second)
    fmt.Println("worker",id,"finished job",j)
    results <- j*2
  }
}
func main(){
  //为了使用工人池，我们需要派发任务并且回收结果。我们使用两个通道来做这件事
  jobs:=make(chan int,100)
  results:=make(chan int,100)
  //这里启动了3个工人，初始时阻塞，因为当前没有作业
  for w:=1;w<=3;w++{
    go worker(w,jobs,results)
  }
  //添加5个作业，然后关闭通道来表明这就是所有的工作
  for j:=1;j<=5;j++{
    jobs<-j
  }
  close(jobs)

  for a:=1;a<=5;a++{
    <-results
  }
}
```

运行的项目展示了有5个作业得以被不同的工人执行。尽管总共有5秒钟的时间，这个程序只需要2秒，因为有3名工作人员同时进行操作。

同时启动了3个worker，来监听通道是否有作业发出，无作业时worker不会进入循环体，为空操作。而形成工人池

Rate Limiting

速率限制是控制资源利用和维护服务质量的重要机制。Go通过goroutine，channel和ticker可以优的支持速率控制。

```
package main
import "time"
import "fmt"
func main(){
  //首先，我们看下基本的速率控制。
  //假设我们想要控制处理的输入请求，我们通过同一个通道来为这些请求提供服务
  requests:=make(chan int,5)
  for i:=1;i<=5;i++{
    request<-i
  }
  close(requests)
  //limiter通道将每过200毫秒接收一次数据。这是速率控制策略中的调节器
  limiter:=time.Tick(time.Millisecond*200)
  //在服务每个请求之前，通过limiter通道阻塞接收，我们将自己限制在每200ms处理1个请求上。
  for req:=range request{
    <-limiter
    fmt.Println("request",req,time.now())
  }
  //我们可能希望在我们的速率限制方案中允许短时间的请求，同时保留整体速率限制。
```

```
  //我们可以通过缓冲限制器通道来实现这一点。
  //这个burstyLimiter通道将允许多达3个事件的突发。
  burstyLimiter:=make(chan time.Time,3)
  //填充通道，来展示可允许的突发
  for i:=0;i<3;i++{
    burstyLimiter<-time.Now()
  }
  //每过200毫秒将试图添加一个新值到burstyLimiter，最多3个
  go func(){
    for t:=range time.Tick(time.Millisecond*200){
      burstyLimiter<-t
    }
  }()
  //模拟5个输入请求。前3个将受益于burstyLimiter的突发能力
  burstyRequests:=make(chan int,5)
  for i:=1;i<=5;i++{
    burstyRequest<-i
  }
  close(burstyRequests)
  for req:=range burstyRequests{
    <-burstyLimiter
    fmt.Println("request",req,time.Now())
  }
}
```

运行我们的程序，我们看到第一批请求根据需要每200毫秒处理一次。

对于第二批请求，由于可突发速率限制，我们立即为前3个服务，然后以约200ms的延迟提供剩余的2
。

Atomic Counters

Go中管理状态的主要机制是通过渠道进行沟通。 我们以工人池为例。 还有一些管理状态的其他选项。 这里我们来看一下使用sync / atomic包来进行多个goroutines访问的原子计数器。

```
package main
import "fmt"
import "time"
import "sync/atomic"

func main(){
  //我们使用无符号整数来代表我们的计数器
  var ops unit64=0
  //为了模拟并发更新操作，我们将启动50个goroutine，每个都会在每过1毫秒时增长一次计数器
  for i:=0;i<50;i++{
    go func(){
      for{
        //为了原子地增加计数器，我们使用AddUint64，使用＆语法给它我们的ops计数器的内存地址

        atomic.AddUnit64(&ops,1)
        //在两个增长之间稍作等待
        time.Sleep(time.Millisecond)
      }
    }()
```

```
  }
  //等一下让一些操作积累起来。
  time.Sleep(time.Second)
  //为了安全地使用计数器，当它仍被其他goroutine更新时，我们通过LoadUint64将当前值的副本取到opsFinal中。
  //如上所述，我们需要给出这个函数来获取值的内存地址和操作。
  opsFinal:=atomic.LoadUnit64(&ops)
  fmt.Println("ops:",opsFinal)
}
```

运行程序可以看到我们执行了大约40,000次操作。

Mutexes

在前面的例子中我们看到如何使用原子操作管理简单的计数器状态。为了处理更复杂的状态，我们可使用一个mutext来安全的访问不同gorotine之间的数据。

```
package main
import(
  "fmt"
  "math/rand"
  "sync"
  "sync/atomic"
  "time"
)
func main(){
  //这个例子中，状态是一个map
  var state=make(map[int]int)
  //这个mutext将同步访问状态值
  var mutex=&sync.Mutex{}
  //我们将记录做了多少读写操作
  var readOps unit64=0
  var writeOps unit64=0
  //这里我们启动了100个goroutine来重复读取状态值
  //每毫秒在每个goroutine执行一次
  for r:=0;r<100;r++{
    go func(){
      total:=0
      for{
        //每次读取时我们将获得一个进入的钥匙
        //Lock()住mutex来保证独家访问状态值
        //在选中的钥匙上读取值，Unlock()掉mutext
        //对readOps计数加1
        key:=rand.Intn(5)
        mutex.Lock()
        total+=state[key]
        mutex.Unlock()
        atomic.AddUnit64(&readOps,1)
        //每次操作等待一下
        time.Sleep(time.Millisecond)
      }
    }()
  }
  //再启动10个模拟写的goroutine，与读取类似的模式
```

```
  for w:=0;w<10;w++{
    go func(){
      for{
        key:=rand.Intn(5)
        val:=rand.Intn(100)
        mutex.Lock()
        state[key]=val
        mutex.Unlock()
        atomic.AddUnit64(&writeOps,1)
        time.Sleep(time.Millisecond)
      }
    }()
  }
  //让这10个go协程在state和mutext上工作一会儿
  time.Sleep(time.Second)
  //获取并且报告最终操作的个数。
  readOpsFinal:=atomic.LoadUnit64(&readOps)
  fmt.Println("readOps:",readOpsFinal)
  writeOpsFinal:=atomic.LoadUnit64(&writeOps)
  fmt.Println("writeOps:",writeOpsFinal)
  //随着最后一次锁住状态，展示它是如何结束的
  mutex.Lock()
  fmt.Println("state:",state)
  mutex.Unlock()
}
```

运行程序可以看到我们在mutex同步状态上执行了将近90,000操作。

Stateful Goroutines

在上个例子我们使用mutex显式锁定多个goroutine要同步访问的共享状态。另一个选择是使用goroutine和channel内置的同步功能来达到相同的结果。这种基于渠道的方法与Go通过通信和拥有完全一个oroutine的每个数据来共享内存的想法相一致。

```
package main
import(
  "fmt"
  "math/rand"
  "sync/atomic"
  "time"
)
//在这个例子中，状态值将被一个单独的goroutine拥有
//这保证了数据不会受到并发访问的影响
//为了读或写状态值，其它goroutine将向拥有它的goroutine发送一个消息，然后接收其回复。
//readOp和writeOp结构封装了这些请求和响应
type readOp struct{
  key int
  resp chan int
}
type writeOp struct{
  key int
  val int
  resp chan bool
}
```

```
func main() {
  //像之前一样我们记录执行了多少次操作
  var readOp unit64=0
  var writeOps unit64=0
  //reads和writes通道将被用于其它goroutine分别发送读写请求
  reads:=make(chan *readOp)
  writes:=make(chan *writeOp)
  //这里便是拥有状态值的goroutine，与之前一样是个map，但被私有化
  //这个goroutine反复选择reads和writes通道，响应到达的请求。
  //首先执行所请求的操作然后在响应通道上发送值来表示成功执行响应
  //(或者reads期望的数据)
  go func(){
    var state=make(map[int]int)
    for{
      select{
      case read:=<-reads:
        read.resp<-state[read.key]
      case write:=<-writes:
        state[write.key]=write.val
        write.resp<-true
      }
    }
  }()
  //启动100个goroutine，通过读取通道来读取有状态的goroutine
  //每次读取需要构建一个readOp，通过reads发送给它
  //再通过所提供的resp通道获取结果
  for r:=0;r<100;r++{
    go func(){
      for {
        read:=&readOp{
          key:rand.Intn(5),
          resp:make(chan int)
        }
        reads<-read
        <-read.resp
        atomic.AddUnit64(&readOps,1)
        time.Sleep(time.Millisecond)
      }
    }()
  }
  //启动10个写操作
  for w:=0;w<10;w++{
    go func(){
      for {
        write:=&writeOp{
          key:rand.Intn(5),
          val:rand.Intn(100),
          resp:make(chan bool)
        }
      }
    }()
  }
  time.Sleep(time.Second)
  readOpsFinal:=atomic.LoadUnit64(&readOps)
```

```
  fmt.Println("readOps:",readOpsFinal)
  writeOpsFinal:=atomic.LoadUint64(&writeOps)
  fmt.Println("writeOps:",writeOpsFinal)
}
```

运行项目显示基于gouroutine的状态管理完成了大约80,000操作。

Sorting

sort包实现了内置和自定义类型的排序。首先看看内置类型的排序。

```
package main
import "fmt"
import "sort"
func main(){
  //sort改变了给定的slice，而不是返回一个新的
  strs:=[]string{"c","a","b"}
  sort.Strings(strs)
  fmt.Println("Strings:",strs)
  ints:=[]int{7,2,4}
  sort.Ints(ints)
  fmt.Println("Ints:",ints)
  //可以使用sort检查一个slice是不是已经排好序了
  s:=sort.IntsAreSorted(ints)
  fmt.Println("Sorted:",s)
}
```

Sorting by Functions

有时候我们想要对一个集合进行非自然顺序的排序。例如，我们想要把字符串根据长度而非字典顺序序，下面是一个定制排序的例子。

```
package main
import "fmt"
import "sort"
//为了根据自定义函数排序，我们需要相应的类型
//这里我们创建了一个ByLength类型
//这就是一个[]string类型的别名
type ByLength []string
//我们在ByLength上实现了sort接口的Len、Less和Swap方法
//这里我们想要按照字符串长度的增序排列
func (s ByLength) Len() int{
  return len(s)
}
func (s ByLength) Swap(i,j int){
  s[i],s[j] = s[j],s[i]
}
func (s ByLength) Less(i,j int) bool {
  return len(s[i])<len(s[j])
}
//通过将原有的fruits片段转换为ByLength
//就可以使用sort.Sort来进行自定义排序了。
func main(){
```

```
  fruits:=[]string{"peach","banana","kiwi"}
  sort.Sort(ByLength(fruits))
  fmt.Println(fruits)
}
```

通过类似的模式创建自定义类型，实现三个接口方法，然后调用sort，我们可以对集合进行任意的排
。

Panic

panic通常指发生了未曾预料的错误。大多数情况下，我们使用它来将不应当在正常操作中发生的东
快速失败，或者不准备妥善处理。

```
package main
import "os"
func main(){
  //我们将在整个网站使用panic来检查意外的错误。
  //这是该网站上唯一旨在panic的程序。
  panic("a problem")
  //panic的一个常见作用是终止一个函数返回了一个不知道如何处理或者不想处理的错误。
  //这里是一个panic的例子，在创建一个新文件时发生意外错误
  _,err:=os.Create("/tmp/file")
  if err!=nil{
    panic(err)
  }
}
```

运行这个程序将引起一个panic，打印错误信息和goroutine踪迹，并以非0状态退出。

注意，不像一些用异常来处理大多错误的语言，在Go的习惯用法中，尽可能使用错误指示的返回值。

Defer

defer用于确保一个函数调用在程序执行中延迟作用，经常用于清理目的。defer常用语其他语言的ens
re和finnaly用的地方。

```
packgae main
import "fmt"
import "os"
//假设我们想要创建并写一个文件，在完成后关闭它。
func main(){
  //在获取一个文件对象后立即使用defer并关闭这个文件
  //这个将在main函数末尾关闭的时候执行，在writeFile完成后
  f:=createFile("/tmp/defer.txt")
  defer closeFile(f)
  writeFile(f)
}
func createFile(p string) *os.File{
  fmt.Println("creating")
  f,err:=os.Create(p)
  if err!=nil{
    panic(err)
  }
```

```
  return f
}
func writeFile(f *os.File){
  fmt.Println("writing")
  fmt.Println(f,"data")
}
func closeFile(f *os.File){
  fmt.Println("closing")
  f.Close()
}
```

运行程序，确认这个文件的确在写之后再关闭的。

Collection Functions

我们经常需要我们的程序对数据集合执行操作，例如选择满足给定谓词的所有项目或将所有项目映射具有自定义函数的新集合。

一些语言通常的惯用法是使用泛型数据结构和算法。 Go不支持泛型; 在Go中，通常在程序和数据类特别需要时提供集合功能。

以下是一些用于字符串切片的示例集合函数。 您可以使用这些示例来构建自己的函数。 请注意，在些情况下，直接内联集合操作代码可能是最为清晰的，而不是创建和调用帮助函数。

```
package main
import "strings"
import "fmt"
// 返回目标字符串t的第一个索引，如果没有找到则返回-1
func Index(vs []string,t string) int{
  for i,v:=range vs{
    if v==t{
      return i
    }
  }
  return -1
}
//如果字符串在切片中，则返回true
func Include(vs []string,t string) bool{
  return Index(vs,t)>=0
}
//如果有一个字符串满足期望的f则返回true
func Any(vs []string,f func(string) bool) bool{
  for _,v:=range vs{
    if f(v){
      return true
    }
  }
  return false
}
//所有的字符串满足期望的f则返回true
func All(vs []string,f func(string) bool) bool{
  for _,v:=range vs{
    if !f(v){
      return false
```

```
    }
  }
  return true
}
//返回一个满足给定方法的所有字符串
func Filter(vs []string,f func(string) bool) []string{
  vsf :=make([]string,0)
  for _,v:=range vs{
    if f(v){
      vsf=append(vsf,v)
    }
  }
  return vsf
}
//返回一个包含将f函数应用于原始切片中每个字符串的结果的新切片。
func Map(vs []string,f func(string) string)[] string{
  vsm := make([]string,len(vs))
  for i,v:=range vs{
    vsm[i]=f(v)
  }
  return vsm
}
func main(){
  var strs=[]string{"peach","apple","pear","plum"}
  fmt.Println(Index(strs,"pear"))
  fmt.Println(Include(strs,"grape"))
  fmt.Println(Any(strs,func(v string) bool{
    return strings.HasPrefix(v,"p")
  }))
  fmt.Println(All(strs,func(v string) bool{
    return strings.HasPrefix(v,"p")
  }))
  fmt.Println(Filter(strs,func(v string) bool{
    return strings.Contains(v,"e")
  }))
  //以上示例全部使用匿名函数，但也可以使用正确类型的命名函数。
  fmt.Println(Map(strs,strings.ToUpper))
}
```

String Functions

标准库的String包提供了许多有用的字符串相关的函数。这里有些示例让你对这个包有个初步的认识。

```
package main
import s "strings"
import "fmt"
//鉴于后文大量用到，我们给字符串输出函数起一个别名
var p=fmt.Println
func main(){
  //这里是字符串可以用的函数示例
  //由于这些函数是包中的而不是字符串对象本身的方法，我们需要给他们传递一个字符串参数
  //你可以在strings包下的doc找到更多的函数
  p("Contains: ",s.Contains("test","es"))
  p("Count: ",s.Count("test","t"))
```

```
  p("HasPrefix: ",s.HasPrefix("test","te"))
  p("HasSuffix: ",s.HasSuffix("test","st"))
  p("Index: ",s.Index("test","e"))
  p("Join: ",s.Join([]string{"a","b"},"-"))
  p("Repeat: ",s.Repeat("a",5))
  p("Replace: ",s.Replace("foo","o","0",-1))
  p("Replace ",s.Replace("foo","o","0",1))
  p("Split: ",s.Split("a-b-c-d-e","-"))
  p("ToLower: ",s.ToLower("TEST"))
  p("ToUpper: ",s.ToUpper("test"))
  p()
  //并非strings包的一部分，但是这里值得一提
  //即获取字符串长度以及字符串中的某个字符
  p("Len: ",len("hello"))
  p("Char: ","hello"[1])
}
```

注意，获取长度和索引字符是工作在字节级别上的。Go使用UTF-8编码字符串。

String Formatting

Go在经典的printf上提供了优秀的字符串格式化支持。这里有一些常见格式化的例子。

```
package main
import "fmt"
import "os"
type point struct{
  x,y int
}
func main(){
  //Go提供了一些修饰符来格式化一般的Go数值
  //例如，这里输出了一个point结构体的实例
  p:=point{1,2}
  fmt.Printf("%v\n",p)
  //如果值是结构体，%+v变量将输出结构体中的域名称
  fmt.Printf("%+v\n",p)
  //%#v将打印Go语法形式
  fmt.Printf("%#v\n",p)
  //要想打印值的类型，使用%T
  fmt.Printf("%T\n",p)
  //格式化输出布尔值
  fmt.Printf("%t\n",true)
  //整数有许多格式化选项，使用%d来进行标准十进制格式化
  fmt.Printf("%d\n",123)
  //这样打印了二进制形式
  fmt.Printf("%b\n",14)
  //这里打印了十进制数字对应的字符
  fmt.Printf("%c\n",33)
  //%x提供了十六进制编码
  fmt.Printf("%x\n",456)
  //浮点数也有许多格式化选项，标准十进制使用%f
  fmt.Printf("%f\n",78.9)
  //%e和%E格式化浮点数为科学计数法
  fmt.Printf("%e\n",123400000.0)
```

```
  fmt.Printf("%E\n",123400000.0)
  //基本的字符串输出，使用%s
  fmt.Pirntf("%s\n","\"string\"")
  //将Go代码中的字符串加引号输出(等同参数形式)
  fmt.Printf("%q\n","\"string\"")
  //将字符串十六进制编码化
  fmt.Printf("%x\n","hex this")
  //打印一个指针
  fmt.Printf("%p\n",&p)
  //控制数字输出的宽度和精度，在%后面跟上数字可以控制宽度,.后面跟上数字控制精度
  //默认靠右显示，用空格填充
  fmt.Printf("|%6d|%6d|\n",12,345)
  fmt.Printf("|%6.2f|%6.2f|\n",1.2,3.45)
  //靠左对齐使用-
  fmt.Printf("|%-6.2f|%-6.2f|\n",1.2,3.45)
  //你可能也需要控制字符串输出的格式，特别是要以表格形式输出的时候
  fmt.Printf("|%6s|%6s|\n","foo","b")
  fmt.Printf("|%-6s|%-6s|\n","foo","b")
  //Printf将格式化的字符串打印到标准输出上
  //Sprintf格式化并返回这个字符串而不打印
  s:=fmt.Sprintf("a %s","string")
  fmt.Println(s)
  //你也可以格式化并打印到io.Writers而非os.StdOut
  fmt.Fprintf(os.Stderr,"an %s\n","error")
}
```

Regular Expressions

Go为正则表达式提供了内置的支持。这里是一些常用的正则相关的任务。

```
package main
import "bytes"
import "fmt"
import "regexp"
func main(){
  //测试模式是否符合字符串
  math,_:=regexp.MatchString("p([a-z]+)ch","peach")
  fmt.Printf(match)
  //上面我们直接使用了字符串模式。但是对于其他正则任务，你需要先编译一个正则表达式结构体
  r,_:=regexp.Compile("p([a-z]+)ch")
  //这种结构体上有许多方法。这里是一个如同上面的匹配测试
  fmt.Println(r.MatchString("peach"))
  //这里找到一个匹配
  fmt.Println(r.FindString("peach punch"))
  //这里也是寻找第一个匹配，但返回起止的索引而非字符串
  fmt.Println(r.FindStringIndex("peach punch"))
  //submatch包含整串匹配，也包含内部的匹配。
  fmt.Println(r.FindStringSubmatch("peach punch"))
  //与上面类似，返回整串匹配和内部匹配的索引信息
  fmt.Println(r.FindStringSubmatchIndex("peach punch"))
  //这些函数的All修饰将返回输入中所有匹配的，不仅仅是第一个。例如找到所有匹配正则表达式的
  fmt.Println(r.FindAllString("peach punch pinch",-1))
  fmt.Println(r.FindAllStringSubmatchIndex("peach punch pinch",-1))
  //第二个参数如果是非负数，则将限制最多匹配的个数
```

```
  fmt.Println(r.FindAllString("peach punch pinch",2))
  //例子中都是字符串参数，并且用了类似于MatchString这样的名字
  //我们可以提供[]byte参数，并将函数名中的String去掉
  fmt.Println(r.Match([]byte("peach")))
  //当用正则表达式创建一个常量的时候你可以使用MustCompile。
  //一个纯Compile不能用于常量，因为它有2个返回值。
  r=regexp.MustComplie("p([a-z]+)ch")
  fmt.Println(r)
  //regexp包也能用于使用其他值替换字符串的子集
  fmt.Println(r.ReplaceAllString("a peach","<fruit>"))
  //Func修饰允许你使用一个给定的函数修改匹配的字符串
  in:=[]byte("a peach")
  out:=r.ReplaceAllFunc(in,bytes.ToUpper)
  fmt.Println(string(out))
}
```

JSON

Go内置提供了JSON的编码和解码是，包括内置和自定义的数据类型。

```
package main
import "encoding/json"
import "fmt"
import "os"
//我们使用这两个结构体来展示自定义类型的编码和解码
type Response1 struct{
  Page int
  Fruits []string
}
type Response2 struct{
  Page int `json:"page"`
  Fruits []string `json:"fruits"`
}
func main(){
  //首先我们来看编码基本数据类型到JSON字符串。
  //这里有一些原子类型的示例
  bolB,_:=json.Marshal(true)
  fmt.Println(string(bolB))
  intB,_:=json.Marshal(1)
  fmt.Println(string(intB))
  fltB,_:=json.Marshal(2.34)
  fmt.Println(string(fltB))
  strB,_:=json.Marshal("gopher")
  fmt.Println(string(strB))
  //这里有些slice和map的例子，他们将按需编码成JSON数组和对象。
  slcD:=[]string{"apple","peach","pear"}
  slcB,_:=json.Marshal(slcD)
  fmt.Println(string(slcB))
  mapD:=map[string]int{"apple":5,"lettuce":7}
  mapB,_:=json.Marshal(mapD)
  fmt.Println(string(mapB))
  //JSON包将自动编码你的自定义数据类型
  //他将在编码后的输出中只包含对外的域并且用名字作为JSON键
  res1D:=&Response1{
```

```
    Page:1,
    Fruits:[]string{"apple","peach","pear"}
  }
  res1B,_:=json.Marshal(res1D)
  fmt.Println(string(res1B))
  //你可以在结构体的域声明上定制编码后的JSON键名。
  //可以看上面Reponse2的定义
  res2D:=&Reponse2{
    Page:1,
    Fruits:[]string{"apple","peach","pear"}
  }
  res2B,_:=json.Marshal(res2D)
  fmt.Println(string(res2B))
  //我们需要提供一个变量来放JSON包编码的值。
  //map[string]interface{}将承载一个字符串数据类型的map
  byt:=[]byte(`{"num":6.13,"strs":["a","b"]}`)
  var dat map[string]interface{}
  //这里是相应的解码，以及相关错误的检查
  if err:=json.Unmarshal(byt,&dat);err!=nil{
    panic(err)
  }
  fmt.Println(dat)
  //为了使用解码的map的值，我们需要将它们转换为合适的类型
  //例如这里我们将num中饿值转变为float64类型
  num:dat["num"].(float64)
  fmt.Println(num)
  //访问内部数据需要一系列的转换
  strs:=dat["strs"].([]interface{})
  str1:=strs[0].(string)
  fmt.Println(str1)
  //我们也可以将JSON解码到定制的数据类型
  //这样为程序添加了额外的类型安全，并不再需要在访问数据的时候进行类型断言
  str:=`{"page":1,"fruits":["apple","peach"]}`
  res:=Response2{}
  json.Unmarshal([]byte(str),&res)
  fmt.Println(res)
  fmt.Println(res.Fruits[0])
  //在上面的例子中我们经常使用bytes和字符串在标准输出上来进行数据和JSON形式的交互
  //我们也可以将JSON编码流入到os.Writers甚至HTTP响应体
  enc:json.NewEncoder(os.Stdout)
  d:=map[string]int{"apple":5,"lettuce":7}
  enc.Encode(d)
}
```

Time

Go为时间和持续时间提供了额外支持。这里是一些例子。

```
package main
import "fmt"
import "time"
func main(){
  p:=fmt.Pritnln
  //首先获取当前时间
```

```
  now:=time.Now()
  p(now)
  //你可以创建一个时间结构体，提供年月日等。
  //时间经常与地区，例如时区关联
  then:=time.Date(2009,11,17,20,34,58,651387237,time.UTC)
  p(then)
  //可以按照需要抽取时间变量中的不同部分
  p(then.Year())
  p(then.Month())
  p(then.Day())
  p(then.Hour())
  p(then.Minute())
  p(then.Second())
  p(then.Nanosecond())
  p(then.Location())
  p(then.Weekday())
  p(then.Before(now))
  p(then.After(now))
  p(then.Equal(now))
  //sub方法返回两个时间中的时间长度
  diff:=now.Sub(then)
  p(diff)
  //我们可以计算不同单位的持续时间长度
  p(diff.Hours())
  p(diff.Minutes())
  p(diff.Seconds())
  p(diff.Nanoseconds())
  //可以使用Add来向前推进一个给定的时间
  //或者使用减号来倒退
  p(then.Add(diff))
  p(then.Add(-diff))
}
```

Epoch

程序中的一个常见需求是获取秒、毫秒或微秒，自Unix时代，这里是Go的做法。

```
package main
import "fmt"
import "time"
func main(){
  now:=time.Now()
  secs:=now.Unix()
  nanos:=now.UnixNano()
  fmt.Println(now)
  millis:=nanos/1000000
  fmt.Println(secs)
  fmt.Println(millis)
  fmt.Println(nanos)
  fmt.Println(time.Unix(secs,0))
  fmt.Println(time.Unix(0,nanos))
}
```

Time Formatting / Parsing

Go支持时间格式化和解析，根据基于模式的布局。

```
package main
import "fmt"
import "time"
func main(){
  p:=fmt.Println
  //这里是一个根据RFC3339基本的格式化时间的例子，使用响应的布局常量
  t:=time.Now()
  p(t.Format(time.RFC3339))
  //时间解析使用格式化相同的布局值
  t1,e:=time.Parse(
    time.RFC3339,
    "2012-11-01T22:08:41+00:00"
  )
  p(t1)
  //格式化和解析使用基于示例的布局
  //通常你使用常量在进行布局。但你也可以提供自定义的格式。
  //但你必须使用Mon Jan 2 15:04:05 MST 2006来作为示例
  p(t.Format("3:04PM"))
  p(t.Format("Mon Jan _2 15:04:05 2006"))
  p(t.Format("2006-01-02T15:04:05.999999-07:00"))
  form:="3 04 PM"
  t2,e:=time.Parse(form,"8 41 PM")
  p(t2)
  //纯数字展示你可以使用标准的字符串格式化及响应部分的时间值
  fmt.Printf("%d-%02d-%02dT%02d:%02d:%02d-00:00\n",
    t.Year(),t.Month(),t.Day(),
    t.Hour(),t.Minute(),t.Second()
  )
  //解析返回一个错误来说明是什么问题
  ansic:="Mon Jan _2 15:04:05 2006"
  _,e=time.Parse(ansic,"8:41PM")
  p(e)
}
```

Random Numbers

Go的math/rand包提供产生伪随机数。

```
package main
import "time"
import "fmt"
import "math/rand"
func main(){
  //例如，rand.Intn产生一个随机的整数n，0<=n<100
  fmt.Print(rand.Intn(100),",")
  fmt.Print(rand.Intn(100))
  fmt.Println()
  //rand.Float64返回一个随机的浮点数，0.0<=f<1.0
  fmt.Println(rand.Float64())
```

```
  //这个可以用于产生其他区间的浮点数，如5.0<=f<10.0
  fmt.Print((rand.Float64()*5)+5,",")
  fmt.Print((rand.Float64()*5)+5)
  fmt.Println()
  //默认的数字产生器是deterministic，故它每次善生的数字序列是固定的
  //为了产生不同的序列，赋予它一个变化的种子
  //注意，随即数字用来加密并不安全，用crypto/rand来做
  s1:=rand.NewSource(time.Now().UnixNano())
  r1:=rand.New(s1)
  //调用产生的rand.Rand
  fmt.Print(r1.Intn(100),",")
  fmt.Print(r1.Intn(100))
  fmt.Println()
  //如果你给Source设置了相同的数字种子，他将产生相同的随即数字序列。
  s2:=rand.NewSrouce(42)
  r2:=rand.New(s2)
  fmt.Print(r2.Intn(100),",")
  fmt.Print(r2.Intn(100))
  fmt.Println()
  s3:=rand.NewSource(42)
  r3:=rand.New(s3)
  fmt.Print(r3.Intn(100),",")
  fmt.Print(r3.Intn(100))
}
```

Number Parsing

从字符串中解析数字是一个常见的任务，这里是Go的做法。

```
package main
//内置的strconv包提供了数字解析
import "strconv"
import "fmt"
func main(){
  //64代表要解析的浮点数精度
  f,_:=strconv.ParseFloat("1.234",64)
  fmt.Println(f)
  //0代表根据字符串推断基数，64要求结果要适应64位
  i,_:=strconv.ParseInt("123",0,64)
  fmt.Println(i)
  //ParseInt可以识别十六进制
  d,_:=strconv,Parseint("0x1c8",0,64)
  fmt.Println(d)
  u,_:=strconv.ParseUint("789",0,64)
  fmt.Println(u)
  //atoi是十进制整数解析的简便函数
  k,_:=strconv.Atoi("135")
  fmt.Println(k)
  //不合法的输入将导致解析函数返回一个错误
  _,e:=strconv.Atoi("wat")
  fmt.Println(e)
}
```

URL Parsing

```
package main
import "fmt"
import "net"
import "net/url"
func main(){
  //我们解析这个示例URL，包含一个协议，授权信息，地址，端口，路径，查询参数以及查询拆分
  s:="postgres://user:pass@host.com:5432/path?k=v#f"
  //解析这个URL并且保证没有错误
  u,err:=url.Parse(s)
  if err!=nil{
    panic(err)
  }
  //可以直接访问协议
  fmt.Println(u.Scheme)
  //User包含所有授权信息，调用Username和Password可以得到单独的值
  fmt.Println(u.User)
  fmt.Println(u.User.Username())
  p,_:=u.User.Password()
  fmt.Println(p)
  //Host包含地址和端口。使用SplitHostPort来抽取他们
  fmt.Println(u.Host)
  host,port,_:=net.SplitHostPort(u.Host)
  fmt.Println(host)
  fmt.Println(port)
  fmt.Println(u.Path)
  fmt.Println(u.Fragment)
  //为了以k=v的格式得到查询参数，使用RawQuery
  //你也可以将查询参数解析到一个map中
  //解析的查询参数是从字符串到字符串的片段，故索引0可以只得到第一个值
  fmt.Println(u.RawQuery)
  m,_:=url.ParseQuery(u.RawQuery)
  fmt.Println(m)
  fmt.Println(m["k"][0])
}
```

SHA1 Hashes

SHA1哈希经常用于计算二进制或者文本块的短标识。例如，git版本控制系统使用SHA1来标示文本目录。这里是Go如何计算SHA1哈希值。

```
package main
//Go实现了多种哈希函数，在crypto包下
import "crypto/sha1"
import "fmt"
func main(){
  s:="sha1 this string"
  //产生一个哈希的模式是sha1.New()，sha1.Write(bytes)然后sha1.Sum([]bytes{})
  h:=sha1.New()
  h.Write([]byte(s))
  //这里得到最终的哈希结果字节片段值。参数用于向已存在的字节片段追加，通常不需要
  bs:=h.Sum(nil)
```

```
  //SHA1值经常用于打印成十六进制，如git提交时。使用%x格式参数来转换为十六进制
  fmt.Println(s)
  fmt.Printf("%x\n",bs)
}
```

Base64 Encoding

```
package main
//这里为引入的包起了名称，来节省下面代码的空间
import b64 "encoding/base64"
import "fmt"
func main(){
  data:="abc123!?$*&()'-=@~"
  sEnc:=b64.StdEncoding.EncodeToString([]byte(data))
  fmt.Println(sEnc)
  sDec,_:=b64.StdEncoding.DecodeString(sEnc)
  fmt.Println(string(sDec))
  fmt.Println()
  uEnc:=b64.URLEncoding.EncodeToString([]byte(data))
  fmt.Println(uEnc)
  uDec,_:=b64.URLEncoding.DecodeString(uEnc)
  fmt.Println(string(uDec))
}
```

Reading Files

读取和写入文件是许多Go程序的基本任务需求。首先我们来看读取文件。

```
package main
import(
  "bufio"
  "fmt"
  "io"
  "io/ioutil"
  "os"
)
//读取文件需要检查大多数调用是否错误，这个helper可以流水化错误检查
func check(e error){
  if e!=nil{
    panic(e)
  }
}
func main(){
  //最基本的一个文件读取任务是将所有内容放入内存中
  dat,err:=ioutil.ReadFile("/tmp/dat")
  check(err)
  fmt.Print(string(dat))
  //如果你想对文件的那部分如何进行读取有更多的控制
  //首先你需要打开它
  f,err:=os.Open("/tmp/dat")
  check(err)
  //从文件的开头读取一些字节。允许到5，同时也是实际读取了的字节。
  b1:=make([]byte,5)
```

```
  n1,err:=f.Read(b1)
  check(err)
  fmt.Printf("%d bytes: %s\n",n1,string(b1))
  //你也可以找到一个已知的位置并从那里开始读取
  o2,err:=f.Seek(6,0)
  check(err)
  b2:=make([]byte,2)
  n2,err:=f.Read(b2)
  check(err)
  fmt.Printf("%d bytes @ %d: %s\n",n2,o2,string(b2))
  //io包提供了一些函数，对于文件读取可能很有帮助
  //例如上面的继续读取的例子，可以使用ReadAtLeast更稳定的实现
  o3,err:=f.Seek(6,0)
  check(err)
  b3:=make([]byte,2)
  n3,err:=io.ReadAtLeast(f,b3,2)
  check(err)
  fmt.Printf("%d bytes @ %d: %s\n",n3,o3,string(b3))
  //没有内置的退回，但是Seek(0,0)完成了这个事情
  _,err=f.Seek(0,0)
  check(err)
  //bufio包实现了一个带缓冲区的读取，它对于一些小的读取以及由于它所提供的额外方法会很有帮助
  r4:=bufio.NewReader(f)
  b4,err:=r4.Peek(5)
  check(err)
  fmt.Printf("5 bytes: %s\n",string(b4))
  //在完成时关闭文件(通常会在打开时通过defer计划执行)
  f.Close()
}
```

Writing Files

写文件与读取的模式类似。

```
package main
import(
  "bufio"
  "fmt"
  "io/ioutil"
  "os"
)
func check(e error){
  if e!=nil{
    panic(e)
  }
}
func main(){
  //这里将一个字符串(或者仅仅是字节)写入到一个文件。
  d1:=[]byte("hello\ngo\n")
  err:=ioutil.WriteFile("/tmp/dat1",d1,0644)
  check(err)
  //打开一个文件以供写入
  f,err:=os.Create("/tmp/dat2")
  check(err)
```

```
  //这是一个惯用法，在打开时立刻通过defer关闭
  defer f.Close()
  d2:=[]byte{115,111,109,101,10}
  n2,err:=f.Write(d2)
  check(err)
  fmt.Printf("wrote %d bytes\n",n2)
  n3,err:=f.WriteString("writes\n")
  fmt.Printf("wrote %d bytes\n",n3)
  f.Sync()
  w:=bufio.NewWriter(f)
  n4,err:=w.WriteString("buffered\n")
  fmt.Printf("wrote %d bytes\n",n4)
  w.Flush()
}
```

Line Filters

一个行过滤器经常见于读取标准输入流的输入，处理然后输出到标准输出的程序中。grep和sed是常的行过滤器。

```
//下面这个行过滤器示例将所有输入的文字转换为大写的版本
package main
import (
  "bufio"
  "fmt"
  "os"
  "strings"
)
func main(){
  //使用一个带缓冲的scanner可以方便的使用Scan方法来直接读取一行
  //每次调用该方法可以让scanner读取下一行
  scanner:=bufio.NewScanner(os.Stdin)
  //Text方法返回当前的token，现在是输入的下一行
  for scanner.Scan(){
    ucl:=strings.ToUpper(scanner.Text())
    //输出大写的行
    fmt.Println(ucl)
  }
  //检查scanner的错误，文件结束符不会被当作是一个错误
  if err:=scanner.Err();err!=nil{
    fmt.Fprintln(os.Stderr,"error:",err)
    os.Exit(1)
  }
}
```

可以使用如下命令来试验这个行过滤器：

```
$ echo 'hello' > /tmp/lines
$ echo 'filter' >> /tmp/lines
$ cat /tmp/lines | go run line-filters.go
```

Command-Line Arguments

```
package main
import "os"
import "fmt"
func main(){
  //os.Args提供原始命令行参数访问功能
  //切片的第一个值是程序的路径
  argsWithProg:=os.Args
  //os.Args[1:]保存程序的所有参数
  argsWithoutProg:=os.Args[1:]
  //你可以通过自然索引获取到每个单独的参数
  arg:=os.Args[3]
  fmt.Println(argsWithProg)
  fmt.Println(argsWithoutProg)
  fmt.Println(arg)
}
```

本例应当先go build，然后再运行并指定参数

Command-Line Flags

命令行标志是一个指定特殊选项的常用方法。例如，在wc -l的-l就是一个命令行标志。

```
package main
//flag包支持基本的命令行标志解析
import "flag"
import "fmt"
func main(){
  //基本的标志声明仅支持字符串、整数和布尔值选项
  //这里声明了一个默认值为foo的字符串标志word，并带有一个简短的描述。flag.String返回一个
符串指针。
  wordPtr:=flag.String("word","foo","a string")
  //类似声明一个整数和布尔值标志。
  numbPtr:=flag.Int("numb",42,"an int")
  boolPtr:=flag.Bool("fork",false,"a bool")
  //可以使用程序中已有的参数声明一个标志，声明时需要指定该参数的指针
  var svar string
  flag.StringVar(&svar,"svar","bar","a string var")
  //所有标志声明完成后，调用flag.Parse()来执行命令行解析
  flag.Parse()
  //通过对指针解引用来获取选项的实际值
  fmt.Println("word:",*wordPtr)
  fmt.Println("numb:",*numbPtr)
  fmt.Println("fork:",*boolPtr)
  fmt.Println("svar:",svar)
  fmt.Println("tail:",flag.Args())
}
```

测试用例：

```
$ go build command-line-flags.go
# 省略的标志将自动设定为默认值
$ ./command-line-flags -word=opt
# 位置参数可以出现在任何标志后面
$ ./command-line-flags -word=opt a1 a2 a3
```

```
# flag包需要的所有标志出现在位置参数之前，否则标志将会被解析为位置参数
$ ./command-line-flags -word=opt a1 a2 a3 -numb=7
# 使用-h或者--help标志来得到自动生成的命令行帮助文本
$ ./command-line-flags -h
# 如果提供了一个没有用flag包指定的标志，将会得到错误信息和帮助文档
$ ./command-line-flags -wat
```

Environment Variables

```
package main
import "os"
import "strings"
import "fmt"
func main(){
  //使用os.Setenv来设置一个键值对
  //使用os.Getenv来获取一个环境变量，如果不存在，返回空字符串
  os.Setenv("FOO","1")
  fmt.Println("FOO:",os.Getenv("FOO"))
  fmt.Println("BAR:",os.Getenv("BAR"))
  fmt.Println()
  //使用os.Environ来列出所有环境变量键值对
  for _,e:=range os.Environ(){
    pair:=strings.Split(e,"=")
    fmt.Prinln(pair[0])
  }
}
```

Spawning Processes

```
package main
import "fmt"
import "io/iouitl"
import "os/exec"
func main(){
  //exec.Command函数帮助我们创建一个表示这个外部进程的对象
  dateCmd:=exec.Command("date")
  //Output等待命令运行完成，并收集命令的输出
  dateOut,err:=dateCmd.Output()
  if err!=nil{
    panic(err)
  }
  fmt.Println("> date")
  fmt.Println(string(dateOut))
  grepCmd:=exec.Command("grep","hello")
  //获取输入输出管道
  grepIn,_:=grepCmd.StdinPipe()
  grepOut,_:=grepCmd.StdoutPipe()
  //运行进程，写入输入信息，读取输出结果，等待程序运行结束
  grepCmd.Start()
  grepIn.Write([]byte("hello grep\ngoodbye grep"))
  grepIn.Close()
  grepByte,_:=ioutil.ReadAll(grepOut)
  grepCmd.Wait()
```

```
  fmt.Println("> grep hello")
  fmt.Println(string(grepBytes))
  //通过bash命令的-c选项来执行一个字符串包含的完整命令
  lsCmd:=exec.Command("bash","-c","ls -a -l -h")
  lsOut,err:=lsCmd.Output()
  if err!=nil{
    panic(err)
  }
  fmt.Println("> ls -a -l -h")
  fmt.Pritnln(string(lsOut))
}
```

Exec'ing Processes

```
package main
import "syscall"
import "os"
import "os/exec"
func main(){
  //通过LookPath得到需要执行的可执行文件的绝对路径
  binary,lookErr:=exec.LookPath("ls")
  if lookErr!=nil{
    panic(lookErr)
  }
  //Exec需要的参数是切片形式的，第一个参数为执行程序名
  args:=[]string{"ls","-a","-l","-h"}
  env:=os.Environ()
  execErr:=syscall.Exec(binary,args,env)
  if execErr!=nil{
    panic(execErr)
  }
}
```

Signals

```
package main
import "fmt"
import "os"
import "os/signal"
import "syscall"
func main(){
  //Go通过向一个通道发送os.Signal值来进行信号通知
  sigs:=make(chan os.Signal,1)
  //同时创建一个用于在程序可以结束时进行通知的通道
  done:=make(chan bool,1)
  //注册给定通道用于接收特定信号
  signal.Notify(sigs,syscall.SIGINT,syscall.SIGTERM)
  //Go协程执行一个阻塞的信号接收操作，当它得到一个值时，打印并通知程序可以退出
  go func(){
    sig:=<-sigs
    fmt.Println()
    fmt.Println(sig)
    done<-true
```

```
  }()
  fmt.Println("awaiting signal")
  <-done
  fmt.Println("exiting")
}
```

运行，使用ctrl-c发送信号

Exit

```
package main
import "fmt"
import "os"
func main(){
  //当使用os.Exit时，defer将不会执行
  defer fmt.Println("!")
  //退出，并且状态为3
  os.Exit(3)
}
```