链滴

为 Java 程序员而准备的 Go 指南

作者：ZephyrJung
原文链接：https://ld246.com/article/1483778326912
来源网站：链滴

为Java程序员而准备的Go指南

本文原作者为Stephen Nillson，由ZephyrJung翻译，原文链接如下：

https://www.nada.kth.se/%7Esnilsson/go_for_java_programmers/

鉴于水平有限，未免有很多不合适之处甚至错误，欢迎各位批评指证，Github地址在这里，交流请到客派社区，更多精彩等你发现

本文从Java视角来理解Go，帮助Java程序员们迅速的掌握Go语言。

Hello Stack (example)

为了激发你的兴趣，我们将以一个麻雀虽小五脏俱全的典型案例开始，即Stack.java

Go语言实现如下：

```
// Package collection implements a generic stack.
package collection
// The zero value for Stack is an empty stack ready to use.
type Stack struct {
    data []interface{}
}
// Push adds x to the top of the stack.
func (s *Stack) Push(x interface{}) {
    s.data = append(s.data, x)
}
// Pop removes and returns the top element of the stack.
// It’s a run-time error to call Pop on an empty stack.
func (s *Stack) Pop() interface{} {
    i := len(s.data) - 1
    res := s.data[i]
    s.data[i] = nil  // to avoid memory leak
    s.data = s.data[:i]
    return res
}
// Size returns the number of elements in the stack.
func (s *Stack) Size() int {
    return len(s.data)
}
```

- 在最顶部声明上的注释是文档注释，用纯文本书写
- 声明的名字写在type之后
- struct类似于Java中的class，但struct中的成员不能是方法，只可以是变量
- interface{}类似于Java中的Object。不过它被所有的Go类型实现，不仅仅是引用类型。
- 代码段 (s *Stack)声明了一个方法调用者s，类似于Java中的this
- 操作符 :=同时声明并初始化变量，类型通过初始化表达式的值进行推断

如下是一个Hello World程序，展示如何使用collection.Stack这个抽象数据类型。

```
package collection_test
```

```
import (
    collection "."
    "fmt"
)
func Example() {
    var s collection.Stack
    s.Push("world")
    s.Push("hello, ")
    for s.Size() > 0 {
        fmt.Print(s.Pop())
    }
    fmt.Println()
    // Output: hello, world
}
```

这个cllection_test测试包与collection包的位置在同一目录下。第一个import声明意味着我们将使用前目录(".")下的包，并赋予一个别名collection。第二个声明包含指向标准包路径("fmt")；如果没有定别名，则实际包名fmt将作为默认名称。

概念上的区别

- Go没有带构造器的类，实例方法，继承层次结构，动态方法查找，取而代之的是结构体(struct)和口(interface)。接口也可以用在类似Java的泛型的地方。
- Go提供任何类型的值的指针，不仅仅是对象和数组。对于任何类型 T，都有一个对应的指针类型*T表示该类型值的指针。
- Go允许任意类型上的方法，无需装箱。方法的调用者（类似于Java中的this），可以是一个值或一指针。
- 在Go中，数组属于数值。当数组被用作方法参数时，方法接收到的是一个数组的拷贝，而非指针。而实践中，方法经常使用切片(slice)作为参数。切片是对数组的地址引用。
- 语言内置支持字符串， 一个字符串像一组字节片段, 但它是不可改变的。
- 语言内置支持哈希表，称为映射(map)
- 单独执行线程，goroutines，以及线程之间的通信渠道，channels，都被Go语言内置支持。
- 特定类型（maps，slices，channels）是通过引用而非值传递的。这就是说，向方法传递一个ma类型并不是传递了map的拷贝。如果方法改变了这个map，方法外部调用者也会看到。用Java来讲，以把这个想象成map的引用。
- Go提供了两种访问级别，与Java的public和package private类似。名称首部为大写时代表公有，则为包内私有
- 并没有使用exceptions，Go使用了error类型来代表注入到达文件末尾之类的事件，以及运行时pancs来代表注入试图越界读取数组的运行时错误
- Go不支持隐式类型转换。包含不同类型的表达式需要进行显式转换
- Go不支持方法重载。函数和方法在同一作用域内必须有不同的名称
- Go用 nil代表错误的指针，类似于Java的null

语法

声明

声明语法与Java正好相反，将类型写在名称的后面。类型声明从左往右读可能更容易点儿。

Go	Approximate Java equivalent
var v1 int	int v1 = 0;
var v2 *int	Integer v2 = null;
var v3 string	String v3 = "";
var v4 [10]int	int[] v4 = new int[10]; // v4 is a value in Go.
var v5 []int	int[] v5 = null;
var v6 *struct { a int }	C v6 = null; // Given: class C { int a; }
var v7 map[string]int	HashMap v7 = null;
var v8 func(a int) int	F v8 = null; // interface F { int f(int a); }

声明通常是以关键字后面跟随着被声明的对象名称的形式，关键字可能是const，type，var，或func你可以用一个关键字把一系列的声明写在括号内。

```
var(
  n int
  x float64
)
```

当声明一个方法时，你必须要么为所有参数提供名称，要么一个名称也不写，不能忽略了一些命名，又提供了另一部分的名称。可以对相同类型的名称进行分组：

```
func f(i,j,k int,s,t string)
```

变量可以在声明的时候初始化。如果初始化了，仍然可以进行类型定义但是没有必要，它将默认位初化表达式的值类型。

```
var v9=*v2
```

如果变量没有被显式初始化，那么必须要指定类型。防止它被隐式初始化为零值（0，nil，""，等）。o里没有未初始化的变量。

简单声明

在方法内部，可以使用简化的声明语法:=。

v10:=v1等价于var v10=v1

方法类型

在Go中，方法是一等公民。Go的方法类型表示一系列相同参数和返回结果类型的方法。

```
type binOp func(int,int) int
var op binOp
add :=func(i,j int) int{return i+j}
```

```
op=add
n=op(100,200)
```

多重赋值

Go允许多重复值，右边的表达式将在对操作符左边进行赋值前求值。

```
i,j=j,i //替换i和j
```

方法可能有多个返回值，由括号内的列表指出。返回值可以赋给变量列表。

```
func f() (i int,pj *int){...}
v1,v2=f()
```

空白标识

空白标识（用下划线符号表示），提供了一个可以忽略多重返回值表达值的方法：

```
v1,_=f() //忽略f()函数返回的第二个值
```

分号与格式

与其担心分号与格式问题，不如使用gofmt来创建一个标准的Go风格程序。或许这个风格一开始看起有点古怪，但它和其他的风格没有什么大的不同，并且随着熟悉而逐渐变得顺眼。

实践中Go代码几乎没有分号。技术上讲，所有的Go语句都是以分号为结尾的，只不过Go为每个非空行的末尾隐式的添加了一个，除非语句明显没有结束。这导致了一个而结果就是在某些情况下Go不许换行，如下代码是不允许的：

```
func g()
{
  //非法; "{" 应该在上面那一行
}
```

在g()后面会加分号，因为它是一个方法的声明而非定义。同理，也不能这样写：

```
if n==0 {
}
else {    //错误；"else {"应该在上一行
  ...
}
```

else上面的}会加上分号，导致语法错误。

条件语句

在Go中，if条件、for表达式、switch的选择值部分没有括号，另一反面，它要求if或者for语句体必须花括号。

```
if a<b {f()}
if(a<b){f()} //括号没有必要
if(a<b) f() //错误
```

```
for i=0;i<10;i++{}
for(i=0;i<10;i++){} //错误
```

此外，if和switch可以接受一个可选的初始化语句，一般用于创建一个局部变量。

```
if err:=file.Chmod(0664);err!=nil{
  log.Print(err)
  return err
}
```

For语句

Go既没有while语句也没有do-while语句。for语句可以用单个条件，如此等价于while语句。如果缺整个条件将产生一个无限循环。

一个for语句可以包含一个range子句来遍历字符串、数组、切片、maps或者channels，而非写

```
for i:=0;i<len(a);i++ {...}
```

想要循环a的所有元素，还可以这样写

```
for i,v := range a {...}
```

这为i赋予索引值，v赋予连续的数组，切片或字符串元素。对于字符串而言，i代表一个位的索引，v表Unicode的code point，类型为rune(rune是int32的别名)。映射(map)的迭代器将产生key-value，而通道(channel)只有一个迭代值。

Break和Continue

如Java，Go也允许break和continue指定一个标记，但标记必须是指向一个for，switch或select语。

Switch语句

在switch语句中，case标记并不会默认的一路到底，不过你可以通过在case后面添加fallthrough语来做到这点。

```
switch n{
case 0: //empty case body
case 1:
  f() //当n为0的时候，f并不会被调用
}
```

不过case可以包含多个值

```
switch n{
case 0,1:
  f() //当n为0或1的时候，f将会被调用
}
```

case后面的值可以是任何支持相等比较操作的类型，比如字符串或指针。如果switch表达式后面忽略则与表达式为true等价：

```
switch {
case n<0:
  f1()
case n==0:
  f2()
default:
  f3()
}
```

++和--语句

++和--只可以用于后缀操作符，并且只可以用于语句而不能写在表达式里。例如，不可以写n=i++。

defer语句

一个defer语句将激活一个方法，它的执行将被递延到包含该语句的方法返回的那一刻。被延迟执行方法将在方法返回前执行，无论包含其的方法以什么路线到达返回语句。

```
f,err:=os.Open("filename")
defer f.Close() //当这个方法结束时f将执行close方法
```

常量

Go的常量可能是无类型的，这是用于数值常量，只用了无类型常量的表达式以及const声明中没有给类型，初始化表达式也是非类型化的。一个无类型常量衍生出来的值将在上下文中通过给定类型的值类型化。这使得常量能够相对自由的运用，即便Go并没有隐式类型转换。

```
var a unit
f(a+1) //无类型的数值常量被类型化为unit
f(a+1e3) //1e3也被类型化为uint
```

语言并没有对无类型的数值常量强加以任何大小限制，只有当常量用于需要指定类型的地方时会得到应的限制。

```
const huge = 1 << 100
var n int = huge >> 98
```

如果在变量声明中不包含类型，相应的表达式语句是无类型的数值常量，那常量将会按序转换为rune，nt，float64或complex128，与数值是字符、整数、浮点数或是复数常量有关。

```
c := 'å' //rune (alias for int32)
n := 1 + 2 //int
x := 2.7 //float64
z := 1+2i //complex128
```

Go没有枚举类型，然而，你可以在一个const声明中使用特殊名字iota来得到一列增长的值。如果缺初始化语句，则会沿用上面的规则。

```
const(
  red = iota //red==0
  blue       //blue==1
  green        //green==2
)
```

结构体

结构体类似于Java中的类，但是它不允许包含方法成员，只可以是变量。一个指向接哦固体的指针类于Java中的引用变量。与Java的类相反，Go还允许你定义直接值。无论那种情况，可以使用.来访问构体的成员

```
type MyStruct struct {
  s string
  n int64
}
var x MyStruct //x初始化为MyStruct{"",0}
var px *MyStruct //px初始化为nil
px=new(MyStruct) //px指向一个新的struct MyStruct{"",0}.
x.s="Foo"
px.s="Bar"
```

在Go中，任何命名的类型都可以关联方法，不仅仅是结构体，参见下面关于方法和接口的讨论。

指针

假设要为整型或结构体或数组赋予对象内容的副本，为了达到这个效果，Java中使用引用变量，而G使用指针。对于任何类型T，都有一个相应的指针类型*T，表示类型T值的指针。

为了分配指针变量的内存，可以使用内置方法new，它将产生一个类型，并返回分配内存的指针。这空间将会初始化为该类型的零值。例如，new(int)将分配一个新的int类型的内存，初始化为0，并且回它的地址，类型为*int。

对于Java代码T p=new T()，T是一个有两个类型为int的实例变量a和b的类，如：

```
type T struct {a,b int}
var p *T=new(T)
```

或更常用的是：

```
p := new(T)
```

var v T，声明了一个变量来保存类型T的值，与Java一点也不同。也可以通过一个复合写法来创建并始化值，例如：

```
v := T{1,2}
```

等价于

```
var v T
v.a=1
v.b=2
```

对于T类型的操作数x，取地址操作符&x返回x的地址，一个*T类型的值，例如：

```
p := &T{1,2} //p的类型为*T
```

对于指针类型的操作数x，解指针操作*x表示x所指对象的值。解指针很少用到，像Java一样，Go可自动获取变量的地址：

```
p := new(T)
p.a=1 //等价于(*p).a=1
```

切片(slice)

一个切片理论上是一个包含三个字段的结构体：数组的指针，长度以及容量。切片支持[]操作符来访底层数组的元素。内置的len方法返回切片的长度。内置的cap方法返回容量。

当有一个数组，或一个切片时，通过a[i:j]可以创建一个新的切片。这创建了一个指向a的切片，以索值i为开始，在索引值j之前结束。它的长度为j-i，如果i被忽略，那便从0开始。如果j被忽略，则默认为en(a)。a所指向的切片和数组是同一个，也即说，当通过切片进行修改时，数组元素跟着改变。切片容量是a减去i。数组的容量等于数组的长度。

```
var s []int
var a [10]int
s = a[:] //s = a[0:len(a)]的简写
```

如果你创建了一个[100]byte类型的值（一个拥有100位的数组，比如一个缓冲区），并想将它传递给个方法而不是拷贝，可以声明这个方法的参数为[]byte，并且传递给它数组的一个切片。切片也可以过使用make方法创建，见下描述。

切片以及它的内置方法append提供了与Java中ArrayList大同小异的功能。

```
s0 := []int{1,2}
s1 := append(s0,3)
s2 := append(s1,4,5)
s3 := append(s2,s0...) //???
```

切片的语法也可以用于字符串。它将返回一个新的字符串，其值为原字符串的子串。

值创建

映射(map)以及通道(channel)的值必须通过内置方法make来分配。例如，调用make(map[string]int)回一个新创建的map[string]int类型的值。与new相反，make返回的是实际的对象，而不是一个地址这与map和channel是引用类型的事实相一致。

对于map而言，make使用隐含的容量作为一个可选的第二个参数。对于channel，有一个可选的参来设置缓冲区的大小，默认位0（不做缓冲）。

make方法可以用于分配切片。这种情况下它为底层数组分配内存并返回一个指向它的切片。只需要个必须的参数，即这个切片元素的个数。第二个可选的参数是切片的容量。

```
m := make([]int,10,20) //与new([20]int)[:10]等价
```

函数和接口

函数(Method)

一个函数看起来和方法定义一样，除了它包含一个调用者。这个调用者类似于Java实例方法的this引。

```
type MyType struct {i int}
```

```
func (p *MyType) Get() int {
  return p.i
}
var pm=new(MyType)
var n=pm.Get()
```

这里声明了一个与MyType关联的Get方法。名为p的调用者在方法的内部。

函数定义在命名的类型上。如果你将值转换为了另一个类型，那它将有那个新类型的函数，而非原来型的。

你还可以为内置类型声明方法，只要为内置类型定义一个别名，这个类型将与原内置类型有所区别。

```
//这个例子本人没有看懂……
type MyInt int
func (p MyInt) Get() int {
  return int(p) //这个转换式有必要的
}
func f(i int){}
var v MyInt
v = v * v
f(int(v))
f(v)
```

接口

Go接口与Java的接口类似，但是，任何类型只要提供了Go接口所定义的方法，就可以被认为是这个口的实现，而无需特别的声明。

如下这个接口

```
type MyInterface interface {
  Get() int
  Set(i int)
}
```

上面的MyType已经定义了Get方法，我们再加如下方法来满足这个接口

```
func (p *MyType) Set(i int){
  p.i=i
}
```

这样，任何接受MyInterface参数的方法，都可以接受*MyType类型了。

```
func GetAndSet(x MyInterfaces){}
func f1(){
  var p MyTYpe
  GetAndSet(&p)
}
```

用Java的术语来讲，为*MyType定义了Set和Get方法，使得*MyType自动实现了MyInterface接口一个类型可以满足多个接口。这是 duck typing的一种形式.

当我看到一只鸟，像鸭子一样走路，像鸭子一样游泳，像鸭子一样嘎嘎叫，我称那个鸟为鸭子

匿名字段

一个匿名字段可以用来实现类似于Java中的子类的东西。

```
type MySubType struct{
  MyType
  j int
}
func (p *MySubType) Get() int{
  p.j++
  return p.MyType.Get()
}
```

这有效的为MyType实现了一个子类MySubType

```
func f2(){
  var p MySubType
  GetAndSet(&p)
}
```

Set方法继承于MyType，因为与匿名字段关联的函数将升级为封装类型的函数。此时，因为MySubType有一个MyType类型的匿名字段，MyType的函数也成为了MySubType的函数，Get方法被重写，St方法则被继承。

这与Java中的子类不完全一样。当一个匿名字段的函数被调用时，它的调用者是这个匿名字段，而非含它的结构体。换句话说，在匿名字段上的函数没有自动分配。如果你想实现Java那样的动态函数查，用接口。

```
func f3(){
  var v MyInterface
  v = new(MyType)
  v.Get() //调用*MyType的Get函数
  v = new(MySubType)
  v.Get() //调用*MySubType的Get函数
}
```

类型断言

一个接口类型的变量可能使用类型断言转换为不同的接口类型。这是在运行期间动态实现的。不同于Java，两个接口之前并不需要声明任何关系。

```
type Printer interface{
  Print()
}
func f4(x MyInterface){
  x.(Printer).Print()
}
```

转换为Printer是完全动态的。只要动态类型x（存储在x中的实际类型）定义了Print函数，它便将工。

泛型

Go没有泛型类型，但通过结合匿名字段和类型断言，它可以近似达到Java的参数化类型。

```
type StringStack struct{
  Stack
}
func (s *StringStack) Push(n string){s.Stack.Push(n)}
func (s *StringStack) Pop() string {return s.Stack.Pop().(string)}
```

StringStack个性化了Hello stack例子使得它可以像只能用于string元素的Stack，正如Java中一样。意Size方法继承于Stack。

错误

对于Java通常使用的异常，Go有两种不同的机制。大多数方法返回错误(error)，只有在不可挽回的况下，比如索引越界，才会产生运行时异常。

Go能返回多个值，使得它很容易的返回错误的详细信息以及返回值。作为约定，这个信息的类型为error，一个简单的内置接口。

```
type error interface{
  Error() string
}
```

比如，os.Open方法在打开文件失败时会返回一个non-nilerror

```
func Open(name string)(file *File,err error)
```

下面的代码使用os.Open打开文件。如果error发生，则调用log.Fatal来打印错误信息，并停止。

```
f, err := os.Open("filename.ext")
if err != nil{
  log.Fatal(err)
}
// do something with the open *File f
```

error接口仅需要一个Error方法，然而具体的error实现经常包含很多其他的函数，来使得调用者可以入调查错误的细节。

Panic和恢复

panic是运行时错误，它将释放goroutine栈，并运行所有等待的延迟方法，然后停止程序。panic与Ja a的异常类似，但只针对运行时错误，例如空指针或数组越界。如上文所示，为了表示文件末尾，Go用内置的error类型。

内置的recover方法可以用来重新获得一个有问题的goroutine的控制权，并恢复正常的执行。调用reover方法停止释放并返回传递给panic的参数。由于释放的代码位于延迟方法中，recover只在延迟方内部有用。如果goroutine没有发生问题，recover返回nil。

Goroutine和通道

Goroutines

Go可以使用go语句启动一个新的执行线程，goroutine，它运行在方法内部，以一个不同的，全新建的goroutine。一个程序中的所有goroutine共享同一个地址空间。

goroutine是轻量级的，消耗比分配大不了多少的栈空间。这块栈在需要时分配和释放堆容量。本质，goroutine和corotine的行为类似，在多线程操作系统中多路复用。你无需担心这些细节。

```
go list.Sort() //平行执行list.Sort()方法，不用等它
```

Go有方法字面量，可以表现的像闭包一样，在处理go语句的时候会很有用。

```
//经过指定时间后提交打印的文字到标准输出
func Publish(text string,delay time.Duration){
  go func(){
    time.Sleep(delay)
    fmt.Println(text)
  }() //注意括号，我们必须调用这个方法
}
```

变量text和delay与所在方法和方法字面量所共享，只要他们能被访问到，就会存活。

通道

通道(channel)通过传递一个特定元素类型的值为两个gorotine之间提供了一个同步执行和通信的机。<-操作符说明了通道的方向，发送或者接收。如果没有指明方向，则是双向通信。

```
chan Sushi //可以用来发送和接收Sushi类型的值
chan<- float64 //只能用来发出float64类型的值
<-chan int //只能用来接收ints的值
```

通道是引用类型，通过make来分配

```
ic := make(chan int) //无缓存的int类型channel
wc := make(chan *Work, 10) //有缓存的Work指针channel
```

像一个通道上发送值，可以使用<-作为二元操作符。为了接收通道上的值，用它作为一元操作符。

```
ic <- 3 //发送3到通道上
work := <-wc //从通道上接收一个Work指针
```

如果通道是无缓冲的，发送者将在接收者收到值前阻塞。如果有缓冲，则只有在放进缓冲之前会阻塞当缓冲满时，将等待接收者读取一个值。接收者在有值读取之前会阻塞。

close方法记录在通道上没有信息。当调用了close，并且在上一次发送数据被接收到后，接收操作将回一个零值而不阻塞。多返回值的接收方法会额外返回一个表明通道是否关闭的标识。

```
ch := make(chan string)
go func(){
  ch <- "Hello!"
  close(ch)
}()
fmt.Println(<-ch) //打印Hello!
fmt.Println(<-ch) //无阻塞打印零值
fmt.Println(<-ch) //再次打印零值
v,ok:=<-ch //v是零值""，ok值为false
```

下一个例子中我们让Publish方法返回一个通道，用来广播被打印文字的信息。

```
//Publish在给定时间后打印文本到标准输出
//当文本被提交后关闭等待的通道
func Publish(text string,delay time.Duration)(wait <-chan struct{}){
  ch := make(chan struct{})
  go func(){
    time.Sleep(delay)
    fmt.Println(text)
    close(ch)
  }()
  return ch
}
```

这是你对Publish可能的用法。

```
wait:=Publish("important news",2*time.Minute)
//Do some more work
<-wait //当文本提交前阻塞
```

选择语句

选择(select)语句是Go并发工具集中最后的工具。它选择可能发生通信的集合。如果有哪些通讯可可继续，其中一个将被选中，相应的语句得以执行。否则，如果没有默认情况，语句将阻塞，知道其中个通讯可以完成。

下面是一个玩具示例来展现选择语句是如何用来实现一个随机数产生器。

```
rand := make(chan int)
for {
  select { //向rand发送随机序列位
  case rand <-0: //注：无语句
  case rand <-1:
  }
}
```

稍实际一点，下面是一个选择语句，可以用来向接收操作设置一个时间限制

```
select {
case news := <-AFP:
  fmt.Println(news)
case <-time.After(time.Minute):
  fmt.Println("Time out: no news in one minute.")
}
```

time.After方法是标准库中的一部分；它等待经过指定的时间并且发送当前的时间到返回的通道上。

并发示例

我们用一个小而完整的例子来展示这些片段如何糅合在一起。这段草稿代码从通道上接收Work的请。每个请求服务在不同的goroutine上。Work自身包含一个可以用来返回结果的通道。

```
//server.go
package server
```

```
import "log"
func New()(req chan<- *Work){
  wc:=make(chan *Work)
  go serve(wc)
  return wc
}
type Work struct{
  Op func(int,int) int
  A,B int
  Reply chan int
}
func serve(wc <-chan *Work){
  for w:=range wc{
    go safelyDo(w)
  }
}
func safelyDo(w *Work){
  defer func(){
    if err:=recover();err!=nil{
      log.Println("work failed: ",err)
    }
  }()
  do(w)
}
func do(w *Work){
  w.Reply <- w.Op(w.A,w.B)
}
```

下面是使用这个的例子

```
//example_test.go
package server_test
import (
  server "."
  "fmt"
  "time"
)
func main(){
  s:=server.New()
  divedByZero:=&server.Work{
    Op:func(a,b int) int {return a/b},
    A:100,
    B:0,
    Reply:make(chan int)
  }
  s<-divideByZero
  select{
  case res:=<-divideByZero.Reply:
    fmt.Println(res)
  case <-time.After(time.Second):
    fmt.Println("No result in one second.")
  }
}
```

并发编程是一个庞大的主题，而且Go的方式与Java有相当的区别。这里有两个文章涵盖了基础知识。

- Fundamentals of concurrent programming 是一个对并发短小精悍的介绍，包含了一些Go的例。
- Share Memory by Communicating 是一个有大量示例的代码走廊。