原文链接

简要介绍下 Go（Golang）语言。Go语言的是由谷歌的工程师 Robert Griesemer、Rob Pike 和 Ken Thompson 创造的一种静态类型编译语言。首个开源版本发布于2012年3月。

> “Go 是一种开源编程语言，可以轻松构建简单、可靠、高效的软件。” — GoLang

在诸多编程语言中，解决给定问题的方法有很多，程序员需要花很多时间思考关于解决问题的最佳方法。 但是，Go 相信更少的功能 - 只有一种正确的方法来解决问题。这节省了开发人员的时间，并使大型代码库易于维护。 Go 中没有 maps 和 filters 等“富有表现力”的功能。

> “当你需要增加功能的丰富性，往往伴随着开销的增加” - Rob Pike

1 入门

Go 程序由包组成。编译器会将 main 包编译成可执行程序，而不是共享库。main 包是整个应用的入口。main 包的定义如下：

```
package main
```

接下来，咱们在 Go 的工作空间中创建一个叫做 main.go 的文件，来实现一个简单的 hello world 例子。

1.1 Workspace

Go 的工作空间可以通过环境变量 GOPATH 定义。你需要在这个工作空间中编写你自己的代码。Go 为搜索 GOPATH 以及 GOROOT 指定的目录中的所有包，GOROOT 变量是在安装 GO 时设置的，也就是 go 语言安装的目录。将 GOPATH 设置到预期目录。现在，增加一个文件夹 ~/workspace

```
# export env
export GOPATH=~/workspace

# go inside the workspace directory
cd ~/workspace
```

接下来在这个工作空间中创建 main.go ，写入下面的代码。

1.2 Hello World!

```
package main

import (
 "fmt"
)

func main(){
  fmt.Println("Hello World!")
}
```

在上面的示例代码中，fmt 是 Go 的一个内置包，主要实现格式化 I/O 的功能。在 Go 语言中，通过 import 关键字引入一个包。func main 是可执行代码的入口。fmt 中的 Println 函数实现 “hello world” 的打印。下面尝试运行这个文件。我们可以通过两种方法运行一个 Go 命令。我们知道 Go 是一个编译性语言，所以，在执行之前我们先来编译它。

```
> go build main.go
```

上面的命令就生成了一个可执行文件 main ，接下来我们运行这个文件：

```
> ./main
 # Hello World!
```

接下来看另外一种更加简单的执行方式。go run 命令将编译步骤抽象。那么，通过下面的命令就可以执行这个程序。

```
go run main.go
 # Hello World!
```

注意：可以使用 https://play.golang.org 尝试上面的代码。

1.3 Variables

Go 的变量必须显式声明。Go 是静态类型的语言，也就是说，在变量声明时要检查变量类型。可以如下声明一个变量：

```
var a int
```

在这种情况下，变量的值会被设为0。也可以通过下面的语法声明变量并初始化为不同的值：

```
var a = 1
```

在这里，变量自动被判断为一个整形变量。我们可以通过简化形式来声明变量：

```
message := "hello world"
```

也可以在一行声明多个变量：

```
var b, c int = 2, 3
```

2.数据类型

和其他编程语言一样，Go 语言也有不同的数据类型。接下来就来看一下：

2.1 Number、String、Boolean

Go可支持数字存储类型有 int, int8, int16, int32, int64, uint, uint8, uint16, uint32, uint64, uintptr...

字符串以字节序列的形式存储，用string关键字表示和声明。布尔型变量的关键字是 bool 。 Go 还支持以 complex64 和 complex128 声明的复杂数字类型。

```
var a bool = true
var b int = 1
var c string = 'hello world'
var d float32 = 1.222
var x complex128 = cmplx.Sqrt(-5 + 12i)
```

2.2 Arrays、Slices、Maps

数组是相同数据类型的元素序列。数组具有在声明中定义的固定长度，因此不能进行扩展。数组声明为：

```
var a [5]int
```

数组也可以是多维的。我们可以使用以下格式创建它们：

```
var multiD [2][3]int
```

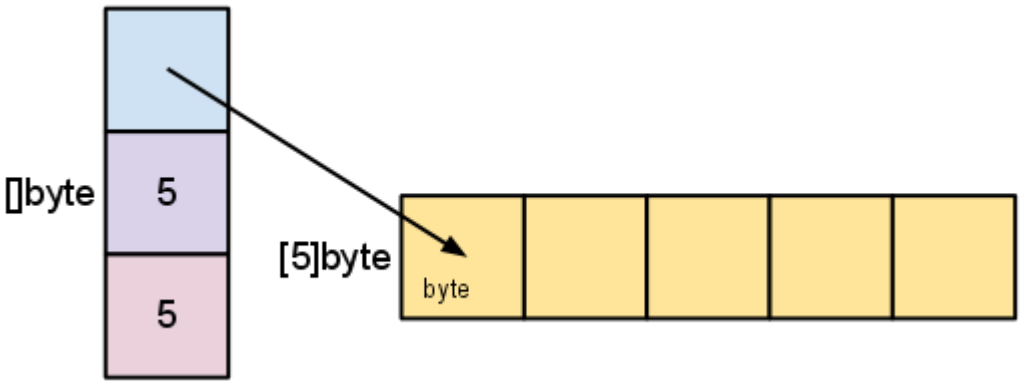
当数组的值在运行时更改时，数组限制了这种情况。 数组也不提供获取子数组的能力。 为此，Go 有一个名为 slices 的数据类型。 切片存储一系列元素，可以随时扩展。 切片声明类似于数组声明但它没有定义容量：

```
var b []int
```

这将创建一个零容量和零长度的切片。切片也可以定义容量和长度。我们可以使用以下语法：

```
numbers := make([]int,5,10)
```

这里，切片的初始长度为5，容量为10。切片是数组的抽象。切片使用数组作为底层结构。切片包含三个组件：容量，长度和指向底层数组的指针，如下图所示：



通过使用append或copy函数可以增加切片的容量。 append函数可以为数组的末尾添加值，并在需要时增加容量。

```
numbers = append(numbers, 1, 2, 3, 4)
```

增加切片容量的另一种方法是使用复制功能。只需创建另一个具有更大容量的切片，并将原始切片复制到新创建的切片：

```
// create a new slice
number2 := make([]int, 15)
// copy the original slice to new slice
copy(number2, number)
```

我们可以创建切片的子切片。这可以使用以下命令完成：

```
// initialize a slice with 4 len and values
number2 = []int{1,2,3,4}
fmt.Println(numbers) // -> [1 2 3 4]
// create sub slices
slice1 := number2[2:]
fmt.Println(slice1) // -> [3 4]
slice2 := number2[:3]
fmt.Println(slice2) // -> [1 2 3]
slice3 := number2[1:4]
fmt.Println(slice3) // -> [2 3 4]
```

Map是Go中的数据类型，它将键映射到值。我们可以使用以下命令定义映射：

```
var m map[string]int
```

这里m是新的集合变量，它的键为字符串，值为整数。我们可以轻松地将键和值添加到地图中：

```
// adding key/value
m['clearity'] = 2
m['simplicity'] = 3
// printing the values
fmt.Println(m['clearity']) // -> 2
fmt.Println(m['simplicity']) // -> 3
```

2.3 数据类型转换

一种数据类型可以通过类型转换得到另一种数据类型。我们来看一个简单的例子：

```
a := 1.1
b := int(a)
fmt.Println(b)
//-> 1
```

不是所有的数据类型都可以转换为别的数据类型。必须确保两种数据类型之间的兼容性。

3.条件语句

3.1 if else

对于条件判断，我们可以像如下的例子那样使用 if-else 语句。要确保大括号和条件语句在同一行。

```
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")
}
```

3.2 switch case

Switch cases有助于组织多个条件语句。以下示例显示了一个简单的switch case语句：

```
i := 2
switch i {
case 1:
 fmt.Println("one")
case 2:
 fmt.Println("two")
default:
 fmt.Println("none")
}
```

4 循环语句

Go有一个循环关键字。 单个for循环命令有助于实现不同类型的循环：

```
i := 0
sum := 0
for i < 10 {
 sum += 1
  i++
}
fmt.Println(sum)
```

上面的示例类似于C中的while循环。对于for循环，也可以使用常见的for语句

```
sum := 0
for i := 0; i < 10; i++ {
  sum += i
}
fmt.Println(sum)
```

Go中的无限循环：

```
for {
}
```

5 指针

Go提供指针。指针是保存值的地址的地方。指针由*定义。根据数据类型定义指针。例：

```
ar ap *int
```

这里ap是指向整数类型的指针。 &运算符可用于获取变量的地址。

```
a := 12
ap = &a
```

可以使用*运算符访问指针的值指针：

```
fmt.Println(*ap)
// => 12
```

在将结构作为参数传递时或在为已定义类型声明方法时，通常首选指针

1.传递值时实际上复制了值,这意味着更多的内存通过指针传递

2.函数更改的值将反映在方法/函数调用者中。

例：

```
func increment(i *int) {
  *i++
}
func main() {
  i := 10
  increment(&i)
  fmt.Println(i)
}
//=> 11
```

注意：尝试本文示例代码时，不要忘记将其包含在main包中，并在需要时导入fmt或其他包，如上面第一个main.go示例所示。

6 函数

主函数中定义的主要功能是执行程序的入口点，可以定义和使用更多功能。让我们看一个简单的例子：

```
func add(a int, b int) int {
  c := a + b
  return c
}
func main() {
  fmt.Println(add(2, 1))
}
//=> 3
```

正如我们在上面的例子中所看到的，使用 func 关键字后跟函数名来定义 Go 函数。函数所需的参数需要根据其数据类型定义，最后是返回的数据类型。函数的返回也可以在函数中预定义：

```
func add(a int, b int) (c int) {
  c = a + b
  return
}
func main() {
  fmt.Println(add(2, 1))
}
//=> 3
```

这里c被定义为返回变量。 因此，定义的变量c将自动返回，而无需在结尾的return语句中定义。
还可以从使用逗号分隔返回值的单个函数返回多个值。

```
func add(a int, b int) (int, string) {
  c := a + b
  return c, "successfully added"
}
func main() {
  sum, message := add(2, 1)
  fmt.Println(message)
  fmt.Println(sum)
}
```

7 方法，结构和接口

Go不是一个完全面向对象的语言，但是对于结构，接口和方法，它有很多面向对象的支持和感觉。

7.1 结构

结构是不同字段的类型集合。 结构用于将数据分组在一起。 例如，如果我们想要对Person类型的数据进行分组，我们会定义一个人的属性，其中可能包括姓名，年龄，性别。 可以使用以下语法定义结构：

```
type person struct {
  name string
  age int
  gender string
}
```

在定义了人类型结构的情况下，现在让我们创建一个人：

```
//way 1: specifying attribute and value
p = person{name: "Bob", age: 42, gender: "Male"}
//way 2: specifying only value
person{"Bob", 42, "Male"}
```

我们可以用点（.）轻松访问这些数据

```
p.name
//=> Bob
p.age
//=> 42
p.gender
//=> Male
```

您还可以使用其指针直接访问结构的属性

```
pp = &person{name: "Bob", age: 42, gender: "Male"}
pp.name
//=> Bob
```

7.2 方法

方法是具有接收器的特殊类型的功能。接收器既可以是值，也可以是指针。让我们创建一个名为describe的方法，它具有我们在上面的例子中创建的接收器类型：

```
package main
import "fmt"

// struct defination
type person struct {
  name   string
  age    int
  gender string
}

// method defination
func (p *person) describe() {
  fmt.Printf("%v is %v years old.", p.name, p.age)
}
func (p *person) setAge(age int) {
  p.age = age
}

func (p person) setName(name string) {
  p.name = name
}

func main() {
  pp := &person{name: "Bob", age: 42, gender: "Male"}
  pp.describe()
  // => Bob is 42 years old
  pp.setAge(45)
  fmt.Println(pp.age)
  //=> 45
  pp.setName("Hari")
  fmt.Println(pp.name)
  //=> Bob
}
```

正如我们在上面的例子中所看到的，现在可以使用点运算符作为pp.describe来调用该方法。 请注意，接收器是指针。 使用指针，我们传递对值的引用，因此如果我们对方法进行任何更改，它将反映在接收器pp中。它也不会创建对象的新副本，这样可以节省内存。

请注意，在上面的示例中，age的值已更改，而name的值未更改，因为方法setName属于receiver类型，而setAge属于pointer类型。

7.3 接口

Go接口是方法的集合。接口有助于将类型的属性组合在一起。我们以接口动物为例：

```
type animal interface {
  description() string
}
```

这里的动物是一种接口类型。现在让我们创建两种不同类型的动物来实现动物接口类型：

```
package main

import (
  "fmt"
)

type animal interface {
  description() string
}

type cat struct {
  Type  string
  Sound string
}

type snake struct {
  Type  string
  Poisonous bool
}

func (s snake) description() string {
  return fmt.Sprintf("Poisonous: %v", s.Poisonous)
}

func (c cat) description() string {
  return fmt.Sprintf("Sound: %v", c.Sound)
}

func main() {
  var a animal
  a = snake{Poisonous: true}
  fmt.Println(a.description())
  a = cat{Sound: "Meow!!!"}
  fmt.Println(a.description())
}
```

```
//=> Poisonous: true
//=> Sound: Meow!!!
```

在main函数中，我们创建了一个动物类型的变量a。 我们为动物分配蛇和猫类型，并使用Println打印a.description。 由于我们以不同的方式实现了两种类型（猫和蛇）中描述的方法，我们可以打印出动物的描述。

8 包 （Package）

我们在Go中编写所有代码。主程序包是程序执行的入口点。 Go中有很多内置包。我们一直使用的最着名的是fmt包。

> “在主要机制中使用程序包进行大规模编程，并且可以将大型项目分成更小的部分。” - 罗伯特格里塞默

8.1 安装包

```
go get <package-url-github>
// example
go get github.com/satori/go.uuid
```

我们安装的软件包保存在GOPATH env中，这是我们的工作目录。您可以通过cd $GOPATH/pkg进入我们的工作目录中的pkg包管理文件夹。

8.2 自定义一个

让我们开始创建一个自定义包文件目录：

```
> mkdir custom_package
> cd custom_package
```

要创建自定义包，我们需要首先使用我们需要的包名创建一个文件夹。假设我们正在建立一个包person。为此，我们在custom_package文件夹中创建一个名为person的文件夹：

```
> mkdir person
> cd person
```

现在让我们在这个文件夹中创建一个文件person.go。

```
package person
func Description(name string) string {
  return "The person name is: " + name
}
func secretName(name string) string {
  return "Do not share"
}
```

我们现在需要安装包，以便可以导入和使用它。所以让我们安装它：

```
> go install
```

现在让我们回到custom_package文件夹并创建一个main.go文件

```
package main
import(
  "custom_package/person"
  "fmt"
)
func main(){
  p := person.Description("Milap")
  fmt.Println(p)
}
// => The person name is: Milap
```

现在，我们可以导入我们创建的包person并使用函数Description。请注意，我们在包中创建的函数secretName将无法访问。在Go中，以大写字母开头的方法名称将是私有的。

8.3 包文档

Go内置了对包文档的支持。运行以下命令以生成文档：

```
godoc person Description
```

这将为我们的包person生成Description函数的文档。要查看文档，请使用以下命令运行Web服务器：

```
godoc -http=":8080"
```

现在转到URL http://localhost:6060/pkg /并查看我们刚刚创建的包的文档。

8.4 Go中的一些内置包

fmt

该包实现了格式化的I/O功能。我们已经使用该包打印到stdout。

JSON

Go中另一个有用的包是json包。这有助于编码/解码JSON。让我们举一个例子来编码/解码一些json：

编码

```
package main

import (
  "fmt"
  "encoding/json"

func main(){
  mapA := map[string]int{"apple": 5, "lettuce": 7}
  mapB, _ := json.Marshal(mapA)
  fmt.Println(string(mapB))
}
```

解码

```
package main

import (
  "fmt"
  "encoding/json"
)

type response struct {
  PageNumber int `json:"page"`
  Fruits []string `json:"fruits"`
}

func main(){
  str := `{"page": 1, "fruits": ["apple", "peach"]}`
  res := response{}
  json.Unmarshal([]byte(str), &res)
  fmt.Println(res.PageNumber)
}
//=> 1
```

在使用unmarshal解码json字节时，第一个参数是json字节，第二个参数是我们希望json映射到的响应类型struct的地址。 请注意，json:“page”将页面键映射到struct中的PageNumber键。

9 错误处理

错误是程序的意外和意外结果。 假设我们正在对外部服务进行API调用。 此API调用可能成功或可能失败。 当存在错误类型时，可以识别Go程序中的错误。 我们来看看这个例子：

```
resp, err := http.Get("http://example.com/")
```

这里对错误对象的API调用可能通过或可能失败。我们可以检查错误是否为nil，并相应地处理响应：

```
package main

import (
  "fmt"
  "net/http"
)

func main(){
  resp, err := http.Get("http://example.com/")
  if err != nil {
    fmt.Println(err)
    return
  }
  fmt.Println(resp)
}
```

9.1 从函数返回自定义错误

当我们编写自己的函数时，有些情况下我们会遇到错误。可以在错误对象的帮助下返回这些错误：

```
func Increment(n int) (int, error) {
  if n < 0 {
// return error object
return nil, errors.New("math: cannot process negative number")
  }
  return (n + 1), nil
}
func main() {
  num := 5

  if inc, err := Increment(num); err != nil {
fmt.Printf("Failed Number: %v, error message: %v", num, err)
  }else {
fmt.Printf("Incremented Number: %v", inc)
  }
}
```

使用Go构建的大多数软件包或我们使用的外部软件包都有一种错误处理机制。 所以我们调用的任何函数都可能存在错误。 这些错误永远不应该被忽略，并且总是在我们称之为函数的地方优雅地处理，就像我们在上面的例子中所做的那样。

9.2 Panic

Panic是一种未经处理的异常，在程序执行期间突然遇到。 在Go中，Panic不是处理程序中异常的理想方式。 建议使用错误对象。 发生Panic时，程序执行停止。 Panic之后执行的事情就是defer。

9.3 Defer

Defer总是在函数结束时执行。

```
//Go
package main

import "fmt"

func main() {
f()
fmt.Println("Returned normally from f.")
}

func f() {
defer func() {
if r := recover(); r != nil {
fmt.Println("Recovered in f", r)
}
}()
fmt.Println("Calling g.")
g(0)
```

```
fmt.Println("Returned normally from g.")
}

func g(i int) {
if i > 3 {
fmt.Println("Panicking!")
panic(fmt.Sprintf("%v", i))
}
defer fmt.Println("Defer in g", i)
fmt.Println("Printing in g", i)
g(i + 1)
}
```

在上面的例子中，我们使用panic()执行程序。 正如您所注意到的，有一个defer语句，它将使程序在程序执行结束时执行该行。 当我们需要在函数结束时执行某些操作时，也可以使用Defer，例如关闭文件。

10 并发

Go是建立在并发性的基础上的。 Go中的并发可以通过轻量级线程的Go例程来实现。

10.1 Go routine

Go routine是可以与另一个函数并行或同时运行的函数。 创建Go routine非常简单。 只需在函数前添加关键字Go，我们就可以使它并行执行。 Go routine非常轻量级，因此我们可以创建数千个例程。 让我们看一个简单的例子：

```
package main
import (
  "fmt"
  "time"
)
func main() {
  go c()
  fmt.Println("I am main")
  time.Sleep(time.Second * 2)
}
func c() {
  time.Sleep(time.Second * 2)
  fmt.Println("I am concurrent")
}
//=> I am main
//=> I am concurrent
```

正如您在上面的示例中所看到的，函数c是一个Go routine ，它与主Go线程并行执行。 有时我们想要在多个线程之间共享资源。 Go更喜欢不与另一个线程共享变量，因为这会增加死锁和资源等待的可能性。 还有另一种在Go routine 之间共享资源的方法：via go channels。

10.2 通道

我们可以使用通道在两个Go例程之间传递数据。在创建频道时，必须指定频道接收的数据类型。让我们创建一个字符串类型的简单通道，如下所示：

```
c := make(chan string)
```

使用此通道，我们可以发送字符串类型数据。我们都可以在此频道中发送和接收数据：

```
package main

import "fmt"

func main(){
  c := make(chan string)
  go func(){ c <- "hello" }()
  msg := <-c
  fmt.Println(msg)
}
//=>"hello"
```

接收方通道等待发送方向通道发送数据。

10.3 单向通道

在某些情况下，我们希望 Go routine 通过通道接收数据但不发送数据，反之亦然。为此，我们还可以创建单向通道。让我们看一个简单的示例：

```
package main

import (
 "fmt"
)

func main() {
 ch := make(chan string)

 go sc(ch)
 fmt.Println(<-ch)
}

func sc(ch chan<- string) {
 ch <- "hello"
}
```

在上面的例子中，sc 是一个 Go routine ，它只能向通道发送消息但不能接收消息。

10.4 使用select为Go例程组织多个通道

函数可能有多个通道正在等待。为此，我们可以使用select语句。让我们看一个更清晰的例子：

```
package main

import (
 "fmt"
 "time"
```

```
)

func main() {
 c1 := make(chan string)
 c2 := make(chan string)
 go speed1(c1)
 go speed2(c2)
 fmt.Println("The first to arrive is:")
 select {
 case s1 := <-c1:
  fmt.Println(s1)
 case s2 := <-c2:
  fmt.Println(s2)
 }
}

func speed1(ch chan string) {
 time.Sleep(2 * time.Second)
 ch <- "speed 1"
}

func speed2(ch chan string) {
 time.Sleep(1 * time.Second)
 ch <- "speed 2"
}
```

在上面的例子中，main正在等待两个通道c1和c2。使用select case语句打印主函数，消息从通道发送，无论它先收到哪个。

10.5 缓冲通道

有些情况下我们需要向通道发送多个数据。您可以为此创建缓冲通道。使用缓冲通道，接收器在缓冲区已满之前不会收到消息。我们来看看这个例子：

```
package main

import "fmt"

func main(){
  ch := make(chan string, 2)
  ch <- "hello"
  ch <- "world"
  fmt.Println(<-ch)
}
```

为什么Golang成功了？

> 简单...—Rob-pike

好极了！我们已经学习了 Go 语言的一些主要组件及功能。

1. 变量，数据类型
2. 数组分片及 map

3. 函数
4. 循环及条件语句
5. 指针
6. 包
7. 方法，结构和接口
8. 错误处理
9. 并发 —— Go routine 及通道

恭喜，你对 Go 已经有不错的理解了。

> 最富有成效的一天是丢弃1000行代码。— Ken Thompson

不要止步于此。继续前进。构思一个小应用，然后开始构建之。