GO 语言超快速入门

使用哪种 IDE？

目前 Go 最优秀的 IDE 无疑是 GoLand，下载地址：

```
https://www.jetbrains.com/go/download/
```

GoLand 提供 30 天的试用。

你也可以使用免费的 VS Code + Go 插件完成我们的课程：

```
https://www.liwenzhou.com/posts/Go/00_go_in_vscode/
```

你也可以使用纯文本编辑器+go 命令完成我们的课程，但是除了写 helloworld 之外不太推荐，对开发不太友好。

main 包

Go 是由包组成的。 main 包告诉 Go 编译器该程序可以被编译成可执行文件，而不是一个共享的库。它是应用程序的入口。main 包被定义为如下格式：

```
package main
```

go 的工作区

Go 中的工作空间由环境变量「GOPATH」定义。你写的任何代码都将写在工作区内。Go 将搜索 GOPATH 目录和 GOROOT 目录，GOROOT 是 Go 环境的安装路径，一般是不变的。GOPATH 是随着开发的项目不同而变化的。

将 GOPATH 设置为你目前的项目：

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

# cd 到工作区目录、
cd ~/workspace
```

helloworld

使用我们刚刚创建的工作空间文件夹中的以下代码创建文件 main.go

```
package main

import (
 "fmt"
)

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

在上面程序中，fmt 是 Go 中的内置包，它实现了格式化输入输出的功能。

在 Go 中我们导入一个包使用 import 关键字。func main 是代码执行的入口。Println 是 fmt 包中的一个函数，它可以在控制台打印输出。

编译

使用 go build 命令编译：

```
go build main.go
```

生成了一个二进制可执行文件 main：

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

还有一种方法可以直接编译+运行：

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

变量

Go 是一种静态类型的语言，在编码时就应该确定变量类型。一般一个变量的定义如下：

```
var a int
```

此时 a 的初始值为 0，若需要自定义初始值：

```
var a = 1
```

另一种更简单的写法：

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

同一行声明多个同类型变量：

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

基本数据类型

数字、字符串、布尔是 go 中常见的数据类型：

数字类型：int, int8, int16, int32, int64,uint, uint8, uint16, uint32, uint64 等

字符串类型： string

布尔类型：bool

复数类型：complex64，complex128

```
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)
```

数组

数组是相同数据类型的元素序列。 数组在声明中定义要指定长度，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。

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

当数组的值在运行时不能进行更改。 数组也不能直接获取子数组。所以就有了切片这种类型。

切片

切片存储一组元素，可以随时扩展。切片声明类似于数组声明，但是没有定义容量：

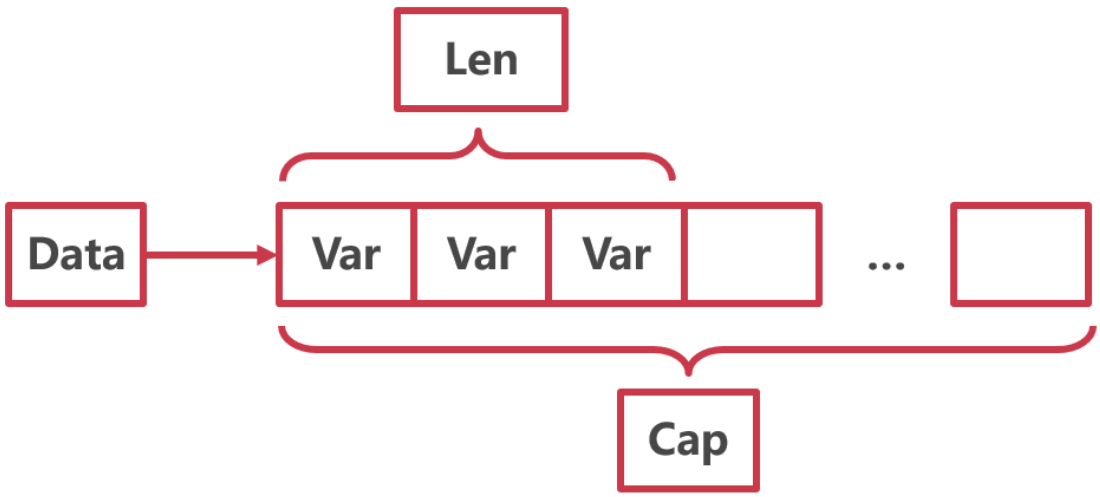
```
var b []int
```

上面的代码将创建一个零容量和零长度的切片。其实切片也可以定义初始的容量和长度：

```
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增加值，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：

```
// 创建切片
number2 := make([]int, 15)
// 将原始切片复制到新切片
copy(number2, numbers)
```

切片还可以创建子切片：

```
// 初始化长度为 4，以及赋值
number2 := []int{1,2,3,4}
fmt.Println(numbers) // -> [1 2 3 4]
// 创建子切片
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

map 是 go 的一种 Key-Value 类型的数据结构，我们可以通过下面的命令声明一个 map ：

```
m := make(map[string]int)
```

m 是 一个 Key 类型为 string、Value 类型为 int 的 map 类型的变量。我们可以很容易地添加键值对到 map 中：

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

类型转化

通过类型转化，能将一种类型转为另一种类型。让我们来看一个简单的例子：

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

并不是所有类型都可以转为另一种类型。需要确保数据类型是可以转化的。

流程控制 - if else

对于流程控制，我们可以使用 if-else 语句，如下例所示。 确保花括号与条件位于同一行。

```
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")
}
```

流程控制 - 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")
}
```

循环

Go 为循环设置了一个关键字：for

```
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，可以使用相同的 for 语句

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

Go 中的无限循环：

```
for {
}
```

指针

Go 支持指针。指针是保存值的地址的地方。 一个指针用 * 定义 。根据数据类型定义指针。 例：

```
var ap *int
```

上面的 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 * 运算符访问指针指向的值：

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

在将结构体作为参数传递或者为已定义类型声明方法时，通常首选指针。

- 传递值时，实际复制的值意味着更多的内存
- 传递指针后，函数更改的值将反映在方法 / 函数调用者中。

例子：

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

函数

main 函数 定义在 main 包中，是程序执行的入口：

```
func add(a int, b int) int {
```

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

上面的例子中可以看到，使用 func 关键字后面跟函数名，可以直接定义 Go 的函数

函数的返回值 c 也可以在函数中预先定义：

```
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 语句中再次定义。

Go 函数可以返回多个返回值，将多个返回值用逗号分隔开即可：

```
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)
}
```

结构体

Go 不是绝对的面向对象的语言，但是使用结构体接口和方法它有很多面向对象的风格以及对面向对象的支持。

结构体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 person 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 person 结构体的情况下，现在让我们创建一个 person 实例 p：

```
//方式 1：指定属性和值
p := person{name: "Bob", age: 42, gender: "Male"}
//方式 2：指定值
person{"Bob", 42, "Male"}
```

用英文的点号（.）可以访问结构体的成员（字段）

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

你还可以使用其指针直接访问结构体里面的属性：

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

方法

方法是一个特殊类型的带有返回值的函数。返回值既可以是值，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 person 结构体类型的返回值：

```
package main
import "fmt"

//定义结构体
type person struct {
  name   string
  age    int
  gender string
}

// 方法定义
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
}
```

方法可以使用点号（.）直接调用，就像 pp.describe 这样。请注意，这个方法的返回值是指针类型。指针类型的返回值不会创建对象的新副本，从而节省了内存。

因为方法 setName 是返回值是值类型，而 setAge 方法的返回值是类型指针。所以上面的示例中，age 的值已更改，而 name 的值不会改变。

接口

Go 的接口是一系列方法的集合。接口有助于将类型的属性组合在一起。下面，我们以接口 animal 为例：

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

这里的 animal 是一个接口。现在，我们用两个不同的实例来实现 animal 这个接口：

```
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 animal 接口类型的变量 a。我们为 animal 接口指定了 snake 和 cat 两个实例对象，并使用 Println 方法打印 a.description 。