

练识课堂 -- Go语言资深工程师培训课程

第一章 Go语言介绍和环境安装

1 Go语言介绍

Go 语言官方介绍

Go 语言是有谷歌推出的一门编程语言。

Go 是一个开源的编程语言，它能让构造简单、可靠且高效的软件变得容易。

Go 语言主要开发者

Go 是从2007年末由Robert Griesemer, Rob Pike, Ken Thompson主持开发，后来还加入了Ian Lance Taylor, Russ Cox等人，并最终于2009年11月开源。



- Rob Pike (UTF-8, Plan 9 from Bell Labs)
- Ken Thompson (UNIX, B, C++, Plan 9)
- Robert Griesemer (HotSpot, V8)

Go语言的几个里程碑

- 2007.09 Go语言设计草稿在白板上诞生
- 2008.01 Ken开发了Go语言编译器把Go代码编译成C代码
- 2009 Go语言诞生
- 2016.01 Go 1.5 版本中实现了自举，不再依赖C编译器
- 2017.02 Go 1.8 版本发布，大幅度提升GC效率

Go 语言特点

- 简洁、快速、安全
- 并行、有趣、开源
- 内存管理、数组安全、编译迅速

Go 语言方向

- 网络编程领域
- 区块链开发领域
- 运维开发领域
- 高性能分布式系统领域

2 环境安装

Go 语言的环境安装:

- Go安装包下载网址(Go语言中文网): <https://studygolang.com/dl>
- 根据操作系统选择对应的安装包 (Mac、Linux、Windows)。
- 不要在安装路径中出现中文。

Go 语言开发工具:

- GoLand
 - GoLand 是 JetBrains 家族的 Go 语言 IDE , 有 30 天的免费试用期。
 - 下载安装网址 : <https://www.jetbrains.com/go/>
 - 根据操作系统选择对应的安装包 (Mac、Linux、Windows) 下载对应的软件。
- LiteIDE
 - LiteIDE 是一款开源、跨平台的轻量级 Go 语言集成开发环境 (IDE)。
 - 下载安装网址 : <https://sourceforge.net/projects/liteide/files/>
 - 根据操作系统选择对应的安装包 (Linux、Windows) 下载对应的软件。
- 其他开发工具 (在IDE安装插件)
 - Eclipse
 - VS Code

第二章 Go语言程序

1 工作区

- 工作区是Go中的一个对应于特定工程的目录, 其包括src, pkg, bin三个目录
 - src: 用于以代码包的形式组织并保存Go源码文件。(比如: .go .c .h .s等)
 - pkg: 用于存放经由go install命令构建安装后的代码包 (包含Go库源码文件) 的“.a”归档文件。
 - bin: 与pkg目录类似, 在通过go install命令完成安装后, 保存由Go命令源码文件生成的可执行文件。
 - 目录src用于包含所有的源代码, 是Go命令行工具一个强制的规则, 而pkg和bin则无需手动创建, 如果必要Go命令行工具在构建过程中会自动创建这些目录。
 - 只有当环境变量 GOPATH 中只包含一个工作区的目录路径时, go install命令才会把命令源码安装到当前工作区的bin目录下。若环境变量 GOPATH 中包含多个工作区的目录路径, 像这样执行go install命令就会失效, 此时必须设置环境变量GOBIN。

```
go install: no install location for .go files listed on command line (GOBIN not set)
```

- 工作区如何设置(linux mac版)
 - 工作区设置路径为环境变量中的GOPATH

```
#mac设置GOPATH
vim .bash_profile
export
GOPATH=/Users/lianshi/Gowork/Public:/Users/lianshi/Gowork/Company
```

```
#linux设置GOPATH
vim .bashrc
export GOROOT=$HOME/go
export PATH=$PATH:$GOROOT/bin
export GOPATH=$HOME/workspace/go
export PATH=$PATH:$GOPATH/bin
```

- 工作区如何设置(windows版)
 - 工作区设置路径为环境变量中的GOPATH

我的电脑 --》 右击属性 --》 高级系统设置 --》 环境变量 --》 系统环境变量 -
-》 添加GOPATH

2 Hello world

下面就用IDE工具，开发第一个GO程序。

Go 语言源文件的扩展是 .go

具体步骤如下：

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, world!")
    fmt.Println("性感法师在线教学~")
}
```

3 编译过程

要执行 Go 语言代码可以使用命令或IDE来完成编译。

命令如下：

- 编译命令：go build hello.go
- 编译并运行命令：go run hello.go

4 常用命令行

go help

- 获取对应命令的帮助文档，可以获取到对应命令的作用以及对应参数

```
go help build
```

go version

- 获取系统安装go语言版本号

go build

- 编译项目，使其打包成可运行程序，配合参数可以进行交叉编译
- 标准格式

- go build [-o output] [-i] [build flags] [packages]
- -o 参数决定了编译后文件名称，例如我们要程序main.go编译后程序名为hello，我们可以执行以下命令

```
go build -o hello main.go
```

- -i install 安装作为目标的依赖关系的包(用于增量编译提速)，一般很少使用。
- 编译参数一般并不会添加，以下列举几个，详细信息可以使用go help build 获取

附加参数	备注
-v	编译时显示包名
-p n	开启并发编译，默认情况下该值为 CPU 逻辑核数
-a	强制重新构建
-n	打印编译时会用到的所有命令，但不真正执行
-x	打印编译时会用到的所有命令

- packages
 - 所编译的包名，如果不填写默认为编译当前路径下的入口文件，文件名称默认为当前文件夹名称

交叉编译

- go语言向下支持C语言，可以在go语言中直接编写C语言代码
- 但是在编译时，必须支持C语言
- Mac上编译Linux可执行二进制文件

```
CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build main.go
```

- Mac上编译Windows可执行二进制文件

```
CGO_ENABLED=0 GOOS=windows GOARCH=amd64 go build main.go
```

- Linux上编译Mac可执行二进制文件

```
CGO_ENABLED=0 GOOS=darwin GOARCH=amd64 go build main.go
```

- Linux上编译Windows可执行二进制文件

```
CGO_ENABLED=0 GOOS=windows GOARCH=amd64 go build main.go
```

- Windows上编译Mac可执行二进制文件

```
SET CGO_ENABLED=0 SET GOOS=darwin SET GOARCH=amd64 go build main.go
```

- Windows上编译Linux可执行二进制文件

```
SET CGO_ENABLED=0 SET GOOS=linux SET GOARCH=amd64 go build main.go
```

- 交叉编译参数含义

- CGO_ENABLED 是否使用cgo编译，0为不使用，1为使用，使用cgo进行交叉编译时需要编译机器安装对应的cgo程序
- GOOS 目标操作系统标识，windows对应Windows操作系统exe可执行文件，darwin对应Mac可执行文件，linux对应Linux可执行文件，freebsd对应UNIX系统
- GOARCH 目标可执行程序操作系统构架，包括 386，amd64，arm
- go build 后接所编译程序的入口文件

go install

- 编译并安装项目
- 标准格式
 - go install [-i] [build flags] [packages]
 - 参数与用法与go build类似

go doc

- 获取go函数帮助文档
- 命令行形式获取某个包的介绍以及包下所有可用的公共方法列表

```
go doc strconv
```

- 命令行形式获取某个方法的文档

```
go doc strconv.Itoa
```

- 使用网页形式查看帮助文档

```
godoc -http=localhost:6060
```

- 在浏览器中输入：<http://localhost:6060/> 可以查看对应的文档信息

go env

- 查看当前系统内go相关的环境变量信息

go test

- Go语言自带的测试工具，会自动读取源码目录下面名为 *_test.go 的文件，生成并运行测试用的可执行文件
- 原则
 - 文件名必须是 _test.go 结尾的，这样在执行 go test 的时候才会执行到相应的代码
 - 必须 import testing 这个包
- 可执行测试
 - 原则
 - 所有的测试用例函数必须是 Test 开头
 - 测试用例会按照源代码中写的顺序依次执行
 - 测试函数 TestXxx() 的参数是 testing.T，我们可以使用该类型来记录错误或者是测试状态

- 测试格式: func TestXxx (t *testing.T), Xxx 部分可以为任意的字母数字的组合, 但是首字母不能是小写字母[a-z]
 - 函数中通过调用 testing.T 的 Error, Errorf, FailNow, Fatal, Fatalf 方法, 说明测试不通过, 调用 Log 方法用来记录测试的信息。
- 创建测试文件class_test.go

```
package main

import (
    "testing"
    "time"
)

func TestHelloWorld(t *testing.T) {
    timestamp := time.Now().Unix()
    t.Log(timestamp)
}
```

- 执行命令行查看测试结果

```
go test -v class_test.go
```

- 结果

```
=== RUN   TestHelloWorld
--- PASS: TestHelloWorld (0.00s)
    class_test.go:10: 1571237717
PASS
ok      command-line-arguments  0.005s
```

- === RUN TestHelloWorld 表示开始运行名叫 TestHelloWorld 的测试用例
- --- PASS: TestHelloWorld (0.00s) 表示已经运行完 TestHelloWorld 的测试用例, PASS 表示当前方法测试成功, 如果是FAIL 表示当前方法测试失败, 时间表示这个测试用例所使用的时间
- ok command-line-arguments 0.005s 表示整体测试结果, ok 表示所有被测试方法测试通过, 如果是FAIL则表示测试失败, command-line-arguments 是测试用例需要用到的一个包名, 0.005s 表示测试花费的时间。

第三章 关键字

1 Go语言关键字

关键字	作用	一级分类	二级分类	三级分类
var	变量声明	基本结构	变量与常量	-
const	常量声明	基本结构	变量与常量	-
package	包声明	基本结构	包管理	-
import	包引用	基本结构	包管理	-
func	函数声明	基本组件	函数	-
return	函数返回	基本组件	函数	-
interface	接口	基本组件	自定义类型	-
struct	结构体	基本组件	自定义类型	-
type	定义类型	基本组件	自定义类型	-
map		基本组件	引用类型	-
range		基本组件	引用类型	-
go		流程控制	并发	-
select		流程控制	并发	-
chan		流程控制	并发	-
if		流程控制	单任务流程控制	单分支流程
else		流程控制	单任务流程控制	单分支流程
switch		流程控制	单任务流程控制	多分支流程
case		流程控制	单任务流程控制	多分支流程
default		流程控制	单任务流程控制	多分支流程
fallthrough		流程控制	单任务流程控制	多分支流程
for		流程控制	单任务流程控制	循环流程
break		流程控制	单任务流程控制	循环流程
continue		流程控制	单任务流程控制	循环流程
goto		流程控制	单任务流程控制	-
defer		流程控制	延时流程控制	-

2 关键字使用方式

- var
定义变量

```
//用于声明变量，声明方式包括以下几种：
var name1 int64
var name2 = 15 //int 类型
//在func内可以使用简写等价，但是简写不可用于声明全局变量，即不能用于func外
name3 := 15
name4,name5,name6 := "a","b","c" //同时赋值给多个变量
```

- const

定义常量或常量集

```
//用于声明常量，可以不用声明类型，对于int类型的常量可以用于int，int64等计算
package main

import "fmt"
//全局常量
const a = 14

func main() {
var b = 16 //int
var c = int64(20) //int64
d := a + b
e := a + c
fmt.Println("d",d)
fmt.Println("e",e)
}
```

- package

用于包声明，在Go中包的概念一般指同一文件夹下的文件，与其它部分语言每个文件可以自己就是一个包不同。

包名可以与文件夹名称不同，但是一般建议相同（如果文件夹带有版本号情况可以忽略版本号部分）

需要写在程序的可执行文件的第一行。

```
//这里是注释
package main
```

- import

用于包的引用，引用路径为工作区下的相对路径

如果程序内引用了两个不同路径下相同的包名，可以通过设定别名的方式进行区分

如果程序内需要用到引用包的初始化或者接口实现，但是没有显示调用，则需要使用_来进行区分


```

package main

import (
    "context"
    echoContext "echo/context"
    _ "notuse/context"
)

func main() {
    context.Background()
    echoContext.text()
}

```

- func /return

定义函数接收返回值

- func

函数方法体声明

返回参数如果为一个且不带参数名，可以不用写括号

返回参数超过一个或者带参数名则需要用括号扩起

- return

函数方法体返回

如果函数返回参数带有参数名称则返回时不需要将每个参数显示返回，否则需要显示返回

```

package main

//无入参，无出参
func main() {
    demo1("hello")
    demo2("hello", 1, 2, 3)
    intArr := []int64{1, 2, 3}
    demo2("hello", intArr...)
    demo3()
    demo4()
    demo5()
    demo6()
}

//有固定入参，无出参数
func demo1(name string) {
    return
}

//无固定入参，无出参数
func demo2(name string, params ...int64) {
    return
}

//无入参，单参数无名称
func demo3() int64 {
    return 0
}

//无入参，单参数有名称

```

```

func demo4() (age int64) {
    return
}

//无入参，多参数无名称
func demo5() (int64,string) {
    return 0, ""
}

//无入参，多参数有名称
func demo6() (age int64,name string) {
    return
}

```

- type / interface / struct
- type
结构体或者接口的声明
- interface
接口，可以存放任意格式数据，也可以定义接口方法
- struct
结构体

```

package main

import "fmt"

type demoI interface {
    funDemo1(string)int
    funDemo2(int64)string
}

//定义结构体实现接口方法
type demoS1 struct {}

func (d *demoS1) funDemo1(string)int {
    return 0
}

func (d *demoS1) funDemo2(int64)string {
    return ""
}

func main() {
    s := &demoS1{}
    demo3(s)
}

//接受参数为接口类型
func demo3(d demoI) {
    v,t := d.(*demoS1)
    fmt.Println(v)
    fmt.Println(t)
}

```

一个 interface 被多种类型实现时，需要区分 interface 的变量究竟存储哪种类型的值。

go 可以使用 comma, ok 的形式做区分 value, ok := em.(T) : em 是 interface 类型的变量。

T代表要断言的类型，value 是 interface 变量存储的值，ok 是 bool 类型表示是否为该断言的类型 T

如果是按 pointer 调用，go 会自动进行转换，因为有了指针总是能得到指针指向的值

如果是 value 调用，go 将无从得知 value 的原始值是什么，因为 value 是份拷贝。go 会把指针进行隐式转换得到 value，但反过来则不行。

- map

map 是 Go 内置关联数据类型（在一些其他的语言中称为"哈希"或者"字典"）

```
package main

func main() {
    //仅仅进行了类型声明，并没有分配内存空间，直接调用则会报错
    var m0 map[string]int
    //声明的同时进行了内存空间的申请
    m1 := make(map[string]int)
    //读取数据参数，返回参数可选，一个参数时为返回值，两个参数时第二个参数表示是否存在
    //如果数据不存在则会返回数据类型的默认数据，但数据为指针类型时则返回nil，不做判断直接使用的话会panic
    value, h := m1["key"]
    m0 = make(map[string]int, 100) //对变量进行初始化，并且预先分配存储空间大小
    _, _, _ = m0, value, h
}
```

- range

与for配合，用于遍历，可遍历数组，map，string（string底层存储为byte数组）

- go

goroutine异步携程,但是不建议携程处理大文件，可以携程来写日志和处理高IO操作，如果是cpu密集型运算，则不建议使用

- select

- Go 中的一个控制结构，类似于用于通信的 switch 语句。每个 case 必须是一个通信操作，要么是发送要么是接收。
- 如果有同时多个case去处理,比如同时有多个channel可以接收数据，那么Go会伪随机的选择一个case处理(pseudo-random)。如果没有case需要处理，则会选择default去处理，如果default case存在的情况下。如果没有default case，则select语句会阻塞，直到某个case需要处理。

- chan

- channel可以理解为一个先进先出的消息队列。
- channel里面的value buffer的容量也就是channel的容量。channel的容量为零表示这是一个阻塞型通道，非零表示缓冲型通道[非阻塞型通道]。

```
package main

var aChan chan int64 //nil chan

var bChan = make(chan int64) //无缓冲

var cChan = make(chan int64, 100) //带缓冲区
```

- if / else / switch / case / default / fallthrough

流程控制语句

- for / break / continue

循环控制语句

- goto

goto语句可以无条件地转移到过程中指定的行。

通常与条件语句配合使用。可用来实现条件转移，构成循环，跳出循环体等功能。

在结构化程序设计中一般不主张使用goto语句，以免造成程序流程的混乱 goto 对应(标签)

既可以定义在for循环前面，也可以定义在for循环后面，当跳转到标签地方时，继续执行标签下面的代码。

```
Loop:
//代码块
goto Loop
```

- defer
- defer后面必须是函数调用语句，不能是其他语句，否则编译器会出错。
- defer后面的函数在defer语句所在的函数执行结束的时候会被调用。

```
package main

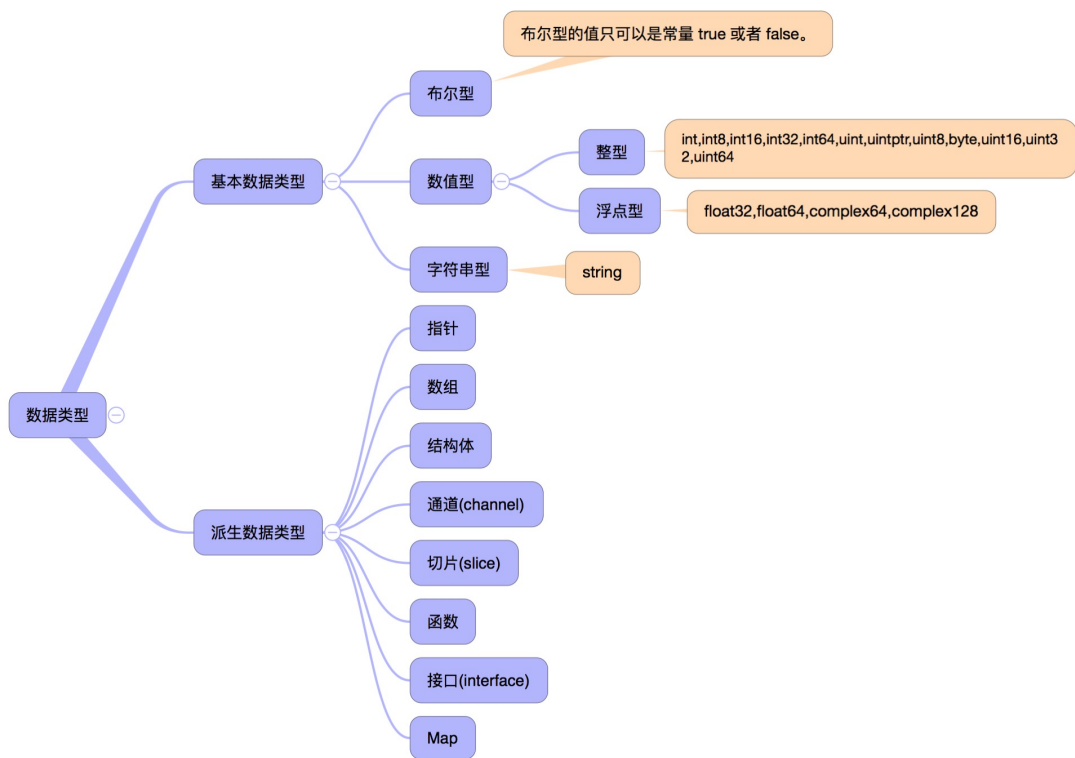
import "fmt"

func main() {
    defer fmt.Println("性感法师")
    defer fmt.Println("在线教学")
    defer fmt.Println("日薪越亿")
    defer fmt.Println("轻松就业")
}
```

第四章 数据类型

数据类型表示数据在内存中开辟的空间大小的别称。

在Go语言中数据类型可以分为：**基本数据类型**和**派生数据类型**



1 Go语言命名规则

命名规则

- 允许使用字母、数字、下划线
- 不允许使用Go语言关键字
- 不允许使用数字开头
- 区分大小写
- 见名知义

驼峰命名法

- 小驼峰式命名法 (lower camel case) :
第一个单词以小写字母开始，第二个单词的首字母大写，例如：myName、aDog
- 大驼峰式命名法 (upper camel case) :
每一个单字的首字母都采用大写字母，例如：FirstName、LastName

2 基本数据类型

Go语言数据类型分为五种：

- 布尔类型
- 整型类型
- 浮点类型
- 字符类型
- 字符串类型

类型	名称	长度	零值	说明
bool	布尔类型	1	false	其值不为真即为假，不可以用数字代表true或false
byte	字节型	1	0	uint8别名

类型	名称	长度	零值	说明
uint	整型		0	根据操作系统设定数据的值。
int8	整型	1	0	-128 ~ 127
uint8	整型	1	0	0 ~ 255
int16	整型	2	0	-32768 ~ 32767
uint16	整型	2	0	0 ~ 65535
int32	整型	4	0	-2147483648 ~ 2147483647
uint32	整型	4	0	0 ~ 4294967295(42亿)
int64	整型	8	0	-9223372036854775808 ~ 9223372036854775807
uint64	整型	8	0	0 ~ 18446744073709551615(1844京)
float32	浮点型	4	0.0	小数位精确到7位
float64	浮点型	8	0.0	小数位精确到15位
string	字符串	-	""	utf-8字符串

布尔型

- 布尔类型也叫做bool类型，bool类型数据只允许取值true或false
- bool类型占1个字节
- bool类型适用于逻辑运算，一般用于流程控制

整型

- 整型数据分为两类，有符号和无符号两种类型
 - 有符号: int, int8, int16, int32, int64
 - 无符号: uint, uint8, uint16, uint32, uint64, byte
 - 不同位数的整型区别在于能保存整型数字范围的大小
 - 有符号类型可以存储任何整数，无符号类型只能存储自然数
 - int和uint的大小和系统有关，32位系统表示int32和uint32，如果是64位系统则表示int64和uint64

```
fmt.Printf("%T", var_name) //输出变量类型
unsafe.Sizeof(var_name)   //查看变量占用字节
```

- 十进制整数，使用0-9的数字表示且不以0开头
- 八进制整数，以0开头，0-7的数字表示
- 十六进制整数，以0X或者是0x开头，0-9|A-F|a-f组成

浮点型

- 浮点数由整数部分、小数点和小数部分组成，整数部分和小数部分可以隐藏其中一种。也可以使用科学计数法表示
- 尾数部分可能丢失，造成精度损失
- float64的精度要比float32的要准确

字符

- Golang中没有专门的字符类型，如果要存储单个字符(字母)，一般使用byte来保存
- 字符只能被单引号包裹，不能用双引号，双引号包裹的是字符串
- 字符使用UTF-8编码，英文字母占一个字符，汉字占三个字符
- 可以直接给某个变量赋一个数字，然后按格式化输出时%c，会输出该数字对应的unicode字符
- 字符类型是可以运算的，相当于一个整数，因为它们都有对应的unicode码

```
package main

import "fmt"

func main() {
    //字符只能被单引号包裹，不能用双引号，双引号包裹的是字符串
    var c1 byte = 'z'
    var c2 byte = '5'
    //当我们直接输出type值时，就是输出了对应字符的ASCII码值
    fmt.Println(c1, "--", c2)
    //如果我们希望输出对应字符，需要使用格式化输出
    fmt.Printf("c2 = %c c2 = %c ,The results of %d", c1, c2, c1 - c2)
}
```

- 但是如果我们保存的字符大于255，比如存储汉字，这时byte类型就无法保存，可以使用rune类型保存

字符串

- 字符串就是一串固定长度的字符连接起来的字符序列。Go的字符串是由单个字节连接起来的。Go语言的字符串的字节使用UTF-8编码标识Unicode文本
- 字符串一旦赋值了，就不能修改了:在Go中字符串是不可变的
- 字符串的两种标识形式
 - 双引号，会识别转义字符

```
package main

import "fmt"

func main() {
    var str = "123\nabc" //输出时会换行
    fmt.Println(str)
}
```

- 反引号，以字符串的原生形式输出，包括换行和特殊字符，可以实现防止攻击、输出源代码等效果

```
package main

import "fmt"

func main() {
    str := `123\nabc` //输出时原样输出，不会转义
    fmt.Println(str)
}
```

- 字符串拼接方式"+"

```
package main

import "fmt"

func main() {
    var str = "hello " + "world"
    str += "!"
    fmt.Println(str)
}
```

- 当一行字符串太长时，需要使用到多行字符串，可以使用如下处理，需要注意 + 必须在当前行的结尾

```
package main

import "fmt"

func main() {
    var str = "hello " +
        "world"
    str += "!"
    fmt.Println(str)
}
```

3 类型转换

数据有不同的类型，不同类型数据之间进行混合运算时必然涉及到类型的转换问题。

两种不同的类型在计算时，Go语言要求必须进行类型转换。

类型转换用于将一种数据类型的变量转换为另外一种类型的变量。

Go 语言类型转换基本格式如下：

数据类型(变量)	//将变量转成指定的类型
数据类型(表达式)	//将表达式转成指定的类型

4 派生数据类型

数组

- var array [5] int // 全部为 0
- var array [5] *int // 指针类型

- `var array := [5] int {1,2,3,4,5} // 初始化`
- `var array := [5] {1:1, 4:5} // 初始化 1, 5`
- `var array := [...] {1,2,3,4,5} // 长度根据初始化确定`
- `var array [5][2] int` 二维数组
- 数组特点
 - 长度固定，不能修改
 - 赋值和函数传递过程是值复制，涉及到内存 copy，性能低下

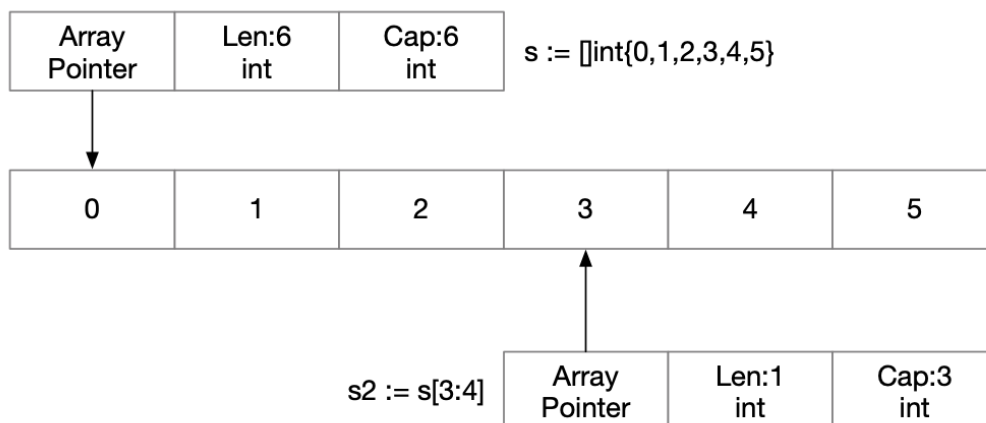
切片

- slice声明方式

```
slice := make([]int, 5)
slice2 := make([]int, 0, 5)
```

- slice数据结构

```
type slice struct {
    array unsafe.Pointer
    len    int //当前存储长度
    cap    int //可用长度
}
```



- 计算方法
 - `s3 := s1[i:j]`
 - `len : j - i`
 - `cap : k - i`, `k`为`s1`的长度
- 切片追加

```
slice = append(slice, 1, 2, 3)
slice = append(slice, slice2...)
```

map

- 底层基于 Hash 实现，基于 Key-Value，无序的数据集合
- `dict := make(map[T_KEY]T_VALUE)` // key 的类型需要具备 `==` 和 `!=` 操作
- 函数类型、字典类型和切片类型不能作为 key，不支持的操作类型会导致 panic

- 检测值是否存在

```
m := make(map[string]bool)
_, h := m["hello"]
if !h {

}
if _, h := m["hello"]; h {

}
```

- var m map[string]int // nil 类型，添加和修改会导致 panic
- nil: len/map[key]/delete(m, key) // 可以正常工作
- map 默认并发不安全，多个 goroutine 写同一个 map，引发竞态错误，go run -race 或者 go build -race
- map 对象即使删除了全部的 key，但不会缩容空间

指针

只要将数据存储在内存中都会为其分配内存地址。内存地址使用十六进数据表示。

内存为每一个字节分配一个32位或64位的编号（与32位或者64位处理器相关）。

可以使用运算符 &（取地址运算符）来获取数据的内存地址。

```
//定义变量
var i int = 10
//使用格式化打印变量的内存地址
//%p是一个占位符 输出一个十六进制地址格式
fmt.Printf("%p\n", &i)
```

如果想将获取的地址进行保存，应该怎样做呢？

可以通过指针变量来存储，所谓的指针变量：就是用来存储任何一个值的内存地址。

```
//定义指针变量
var 指针变量名 //默认初始值为nil 指向内存地址编号为0的空间
var 指针变量名 *数据类型 = &变量
```

```
func main() {
    var i int = 10
    //指针类型变量
    //指针变量也是变量 指针变量指向了变量的内存地址
    //对变量取地址 将结果赋值给指针变量
    var p *int = &i
    //打印指针变量p的值 同时也是i的地址
    fmt.Println(p)
}
```

指针变量，除了有正确指向，还可以通过new()函数来指向。

具体的应用方式如下：

```
//根据数据类型 创建内存空间 返回值为数据类型对应的指针  
new(数据类型)
```

new()函数的作用就是动态分配空间，不需要关心该空间的释放，Go语言会自动释放。

```
func main() {  
    var p *int  
    //创建一个int大小的内存空间 返回值为*int  
    p = new(int)  
    *p = 123  
    //打印值  
    fmt.Println(*p)  
    //打印地址  
    fmt.Println(p)  
}
```

第五章 流程控制

1 选择结构语句

if语句

在编程中实现选择判断结构就是用 if

if结构语法

```
if 条件判断 {  
    //代码语句  
}
```

条件判断如果为真（true），那么就执行大括号中的语句；如果为假（false），就不执行大括号中的语句。

if else结构语法

```
if 条件判断 {  
    //代码语句1  
}else{  
    //代码语句2  
}
```

条件判断如果为真（true），那么就执行if大括号中的语句。

条件判断如果为假（false），那么就执行else大括号中的语句。

if代码块或else代码块，必须有一块被执行。

if else if结构语法

```
if 条件判断1 {  
    //代码语句1  
}else if 条件判断2{  
    //代码语句2  
}else if 条件判断3{  
    //代码语句3  
}else{  
    //代码语句4  
}
```

从上到下依次判断条件，如果结果为真，就执行 {} 内的代码。

switch语句

```
switch 变量或者表达式的值{  
    case 值1:  
        //代码语句1  
    case 值2:  
        //代码语句2  
    case 值3:  
        //代码语句3  
    default:  
        //代码语句4  
}
```

执行流程：

程序执行到switch处，首先将变量或者表达式的值计算出来，然后拿着这个值依次跟每个case后面所带的值进行匹配，一旦匹配成功，则执行该case所带的代码，执行完成后，跳出switch-case结构。

如果，跟每个case所带的值都不匹配。就看当前这个switch-case结构中是否存在default，如果有default，则执行default中的语句，如果没有default，则该switch-case结构什么都不做。

注意：

某个case 后面跟着的代码执行完毕后，不会再执行后面的case，而是跳出整个switch结构，相当于每个case后面都跟着break(终止)。

但是如果想执行完成某个case后，强制执行后面的case，可以使用fallthrough。

2 循环结构语句

for语句

语法结构如下：

```
for 表达式1;表达式2;表达式3 {  
    //循环体  
}
```

表达式1：定义一个循环的变量，记录循环的次数。

表达式2：一般为循环条件，循环多少次。

表达式3：一般为改变循环条件的代码，使循环条件终有不再成立。

循环体：重复要做的事情。

示例1：

```
package main
import "fmt"

func main() {
    sum := 0
    //计算1-100的和
    for i := 1; i <= 100; i++ {
        sum += i
    }
    fmt.Println(sum)
}
```

示例2：

```
package main
import "fmt"

func main() {
    //水仙花数
    //一个三位数 各个位数的立方和等于本身的数
    for i := 100; i <= 999; i++ {
        a := i / 100      //百位
        b := i / 10 % 10  //十位
        c := i % 10       //个位
        if a*a*a+b*b*b+c*c*c == i {
            fmt.Println(i)
        }
    }
}
```

嵌套循环

循环语句之间可以相互嵌套：

```
for 循环条件{
    for 循环条件{
        //执行代码
    }
}
```

跳出语句

break语句

在循环语句中可以使用break跳出语句：

- 当它出现在循环语句中，作用是跳出当前内循环语句，执行后面的代码。
- 当它出现在嵌套循环语句中，跳出最近的内循环语句，执行后面的代码。

continue语句

在循环语句中，如果希望立即终止本次循环，并执行下一次循环，此时就需要使用continue语句。

第六章 函数和闭包

1 函数执行流程

在程序编译完后，函数会以计算机指令方式存储在代码区，在定义函数时指定的形参，在未出现函数调用时，它们并不占内存中的存储单元，函数的参数以及局部变量会在调用时加载到内存中。

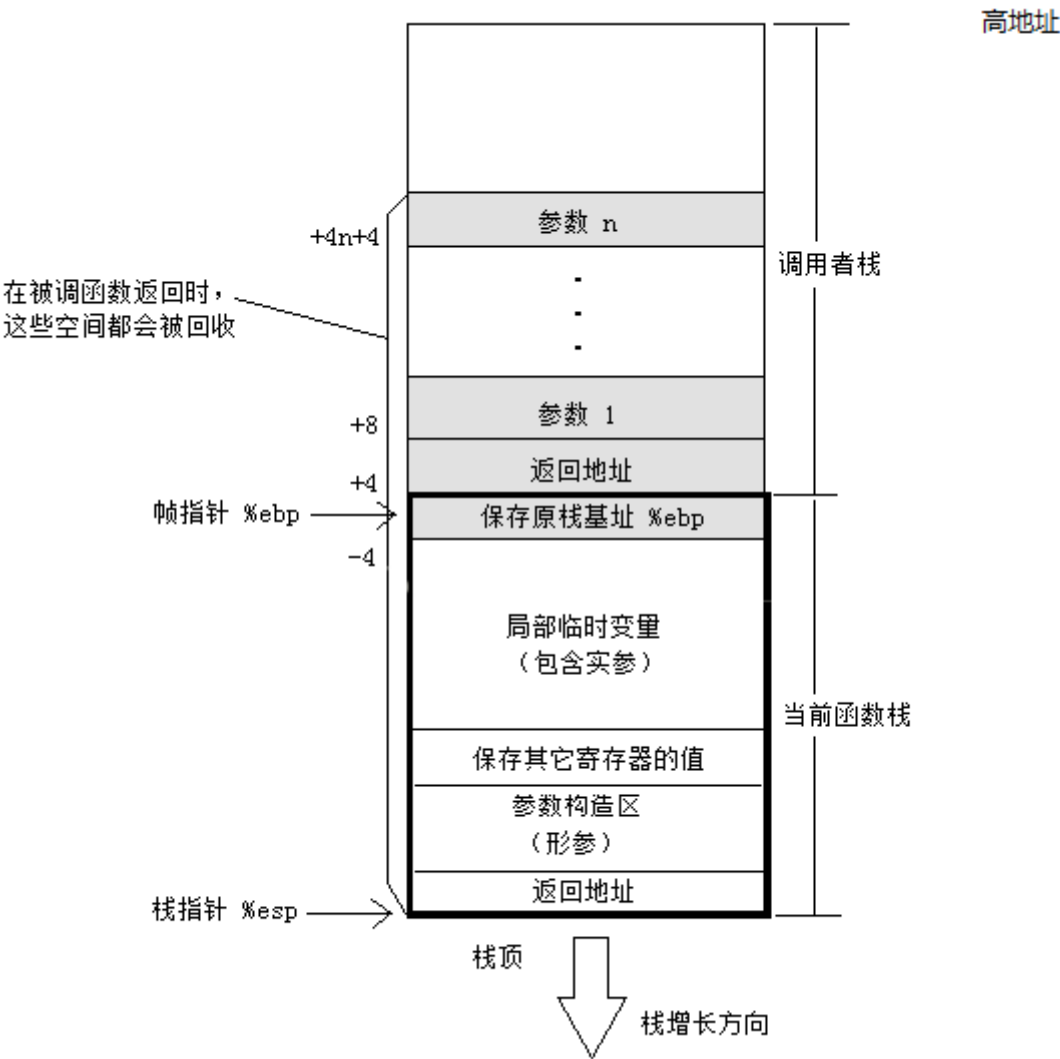
栈(stack)是现代计算机程序里最为重要的概念之一，几乎每一个程序都使用了栈，没有栈就没有函数，没有局部变量。

栈被定义为一个特殊的容器，用户可以将数据压入栈中(入栈，push)，也可以将压入栈中的数据弹出(出栈，pop)，但是栈容器必须遵循一条规则：先入栈的数据最后出栈(First In LAST Out,FIFO)。

在经典的操作系统中，栈总是向下增长的。压栈的操作使得栈顶的地址减小，弹出操作使得栈顶地址增大。

栈在程序运行中具有极其重要的地位。最重要的，栈保存一个函数调用所需要维护的信息，这通常被称为堆栈帧(Stack Frame)或者活动记录(Activate Record).一个函数调用过程所需要的信息一般包括以下几个方面：

- 函数的返回地址；
- 函数的参数；
- 保存的上下文：包括在函数调用前后需要保持不变的寄存器。



保存的上下文：包括在函数调用前后需要保持不变的寄存器。

2 defer延迟调用

defer语句被用于预定对一个函数的调用。可以把这类被defer语句调用的函数称为延迟函数。

defer作用：

- 释放占用的资源
- 捕捉处理异常
- 输出日志

```
func Demo(){
    defer fmt.Println("性感法师")
    defer fmt.Println("在线教学")
    defer fmt.Println("日薪越亿")
    defer fmt.Println("轻松就业")
}
func main() {
    Demo()
}
```

结果

```
轻松就业
日薪越亿
在线教学
性感法师
```

如果一个函数中有多个defer语句，它们会以LIFO（后进先出）的顺序执行。

3 recover错误拦截

运行时panic异常一旦被引发就会导致程序崩溃。

Go语言提供了专用于“拦截”运行时panic的内建函数“recover”。它可以是当前的程序从运行时panic的状态中恢复并重新获得流程控制权。

```
func recover interface{}
```

注意：recover只有在defer调用的函数中有效。

示例代码

```
package main

import "fmt"

func Demo(i int) {
    //定义10个元素的数组
    var arr [10]int
    //错误拦截要在产生错误前设置
    defer func() {
        //设置recover拦截错误信息
        err := recover()
        //产生panic异常 打印错误信息
    }()
}
```

```

        if err != nil {
            fmt.Println(err)
        }
    }()
    //根据函数参数为数组元素赋值
    //如果i的值超过数组下标 会报错误：数组下标越界
    arr[i] = 10
}

func main() {

    Demo(10)
    //产生错误后 程序继续
    fmt.Println("程序继续执行...")
}

```

结果

```

runtime error: index out of range
程序继续执行...

```

如果程序没有异常，不会打印错误信息。

4 闭包

Go 语言支持匿名函数，可作为闭包。匿名函数是一个"内联"语句或表达式。匿名函数的优越性在于可以直接使用函数内的变量，不必申明。

闭包是将函数内部和函数外部连接起来的桥梁。

它的最大用处有两个，

- 可以读取函数内部的变量
- 变量的值始终保持在内存中

以下实例中，创建了函数 `getSequence()`，返回另外一个函数。该函数的目的是在闭包中递增 `i` 变量，代码如下：

```

package main
import "fmt"

func getSequence() func() int {
    i:=0
    return func() int {
        i+=1
        return i
    }
}

func main(){
    /* nextNumber 为一个函数，函数 i 为 0 */
    f1 := getSequence()

    /* 调用 f1 函数，i 变量自增 1 并返回 */
    fmt.Println(f1())
    fmt.Println(f1())
}

```



```

fmt.Println(f1())

/* 创建新的函数 f2，并查看结果 */
f2 := getSequence()
fmt.Println(f2())
fmt.Println(f2())
}

```

结果：

```

1
2
3
1
2

```

注意：

由于闭包会使得函数中的变量都被保存在内存中，内存消耗很大，所以不能滥用闭包，否则会造成网页的性能问题，在IE中可能导致内存泄露。**解决方法是，在退出函数之前，将不使用的局部变量全部删除。**

闭包会在父函数外部，改变父函数内部变量的值。所以，如果你把父函数当作对象（object）使用，把闭包当作它的公用方法（Public Method），把内部变量当作它的私有属性（private value），这时一定要小心，不要随便改变父函数内部变量的值。

第七章 结构体和匿名字段

Go 语言中数组切片可以存储同一类型的数据，但在结构体中可以为不同项定义不同的数据类型。

结构体是由一系列具有不同类型的数据构成的数据集合。

```

type 结构体名 struct {
    结构体成员名 数据类型    `结构体标签`
    结构体成员名 数据类型
    结构体成员名 数据类型
}

```

注意：结构体成员在定义时不能赋值。

1 空结构体

go语言中的struct是一种很重要的语法，里面可以存放各种字段，当然它里面也可以为空。这样可以用来单纯的做控制信息。

```

a := struct{}{}
fmt.Println(unsafe.Sizeof(a))
// 结果：0

```

由于空结构体占用0字节，那么空结构体也不需要填充字节。所以空结构体组成的组合数据类型也不会占用内存空间。

```

type S struct {
    A struct{}
    B struct{}
}
var s S
fmt.Println(unsafe.Sizeof(s))
// 结果: 0

```

有一种特殊的struct{}类型的channel，它不能被写入任何数据，只有通过close()函数进行关闭操作，才能进行输出操作。struct{}类型的channel不占用任何内存。
定义：

```

var sig = make(chan struct{})

```

使用空 struct 是对内存更友好的开发方式，在 go 源代码中针对 空struct 类数据内存申请部分，返回地址都是一个固定的地址。那么就避免了可能的内存滥用。

2 结构体标签

结构体的tag设置

```

type People struct {
    Name string `json:"name"`
    Age  uint  `json:"age"`
}

```

tag 是类型的一部分，主要用于通过反射获取字段的相关 tag 设置

```

typeOfCat := reflect.TypeOf(People{})
for i := 0; i < typeOfCat.NumField(); i++ {
    field := typeOfCat.Field(i)
    tag := field.Tag.Get("json")
    fmt.Println(tag)
}

```

3 匿名字段

可以通过匿名字段实现“继承关系”，所谓的匿名字段就是将结构体名称作为另外结构体的成员。

```

type People struct {
    Name string
    Age  uint
}
type Student struct {
    //将结构体名称作为另外结构体的成员 匿名字段
    People
    Score int
}

```

继承后，子类可以直接使用父类的结构体成员。

第八章 方法

1 方法定义

值对象接收者

```
func (p People) GetName() string {  
    return p.Name  
}
```

- 无须修改状态的小对象或者固定值
- 引用类型、字符串或者函数等
- 调用时会按照其一个副本来执行调用

指针对象接收者

```
func (p *People) GetAge() uint {  
    return p.Age  
}
```

- 修改实例状态
- 大对象
- 调用时会按照实际值来执行调用

结构体同名字段

```
type People struct {  
    Name string `json:"name"`  
    Age  uint  `json:"age"`  
}  
  
type Admin struct {  
    People  
    Name string //会覆盖People中的Name  
    Level uint  
}
```

调用方法

```
a := Admin{  
    People: People{  
        Name: "liLei",  
        Age: 30,  
    },  
    Name: "HanJie",  
    Level: 3,  
}  
fmt.Println(a.Name) //调用覆盖后的，打印值HanJie  
fmt.Println(a.People.Name) //调用的子结构体内覆盖前的liLei
```

2 方法继承和重写

继承

在结构体操作时，存在继承关系，子类可以使用父类的方法

```
type Person10 struct {
```

```

    name string
    age  int
    sex  string
}
type Student struct {
    Person10
    Score int
}

//父类方法 只能使用父类的结构体属性信息
func (p *Person) Print() {
    fmt.Println(*p)
}

func main() {
    stu := Student{Person10{"xx", 25, "男"}, 100}

    //子类对象调用父类方法
    stu.Print()
}

```

重写

如果子类(结构体)中的方法名与父类(结构体)中的方法名同名，在调用的时候是先调用子类(结构体)中的方法，这就方法的重写。

所谓的重写：就是子类(结构体)中的方法，将父类中的相同名称的方法的功能重新给改写。

```

type Person10 struct {
    name string
    age  int
    sex  string
}
type Student struct {
    Person10
    Score int
}

//父类方法 只能使用父类的结构体属性信息
func (p *Person) Print() {
    fmt.Println(*p)
}

//子类方法
func (s *Student) Print() {
    fmt.Println(*s)
}

func main() {
    stu := Student{Person10{"xx", 25, "男"}, 100}

    //子类对象调用子类方法
    stu.Print()
}

```

第九章 接口

- 接口是什么
 - 接口是一组方法签名。当一个类型为接口中的所有方法提供定义时，它被称为实现该接口
 - Interface{} 可以理解成空接口，可以接收任意类型的参数
- 接口的实现

```
type Animal interface {
    GetName() string
}
```

- 只要是实现了方法体GetName的结构体都是实现了Animal这个接口
- 接口接收参数的类型判断
 - 接口可能为nil，所以需要做nil判断

```
func E(x interface{}) {
    if x == nil {
        fmt.Println("empty interface")
        return
    }
    fmt.Println("non-empty interface")
}

func main() {
    var x *int = nil
    E(x)
}
```

1 接口方法集

```
type Animal interface {
    GetName() string
}

type People struct {
    Name string `json:"name"`
    Age  uint  `json:"age"`
}

func (p *People) GetName() string {
    return p.Name
}
```

- 调用情况

```
var n Animal
p1 := People{}
p2 := &People{}
n = p1 //无法编译
n = p2
```

2 接口方法集调用关系

仅影响接口，与实例或者指针调用无关;接口方法集定义了接口接受的规则规范里面描述的方法集

值	方法接受者
T	(t T)
*T	(t T) && (t *T)
T 包含匿名字段 S	(t T) && (s S)
T 包含匿名字段 *S	(t T) && (s S) && (s *S)
*T 包含匿名字段 *S	(t T) && (t *T) && (s S) && (s *S)

- 判断某个值是不是某个接口实现

```
var o interface{} = &People{Name:"LiLei"}
if n,ok := o.(Animal);ok {
    fmt.Println(n.GetName())
}
```

```
var o interface{} = &People{Name:"LiLei"}
switch o.(type) {
case People:
    fmt.Println("is people")
case *People:
    fmt.Println("is people *")
case Animal:
    fmt.Println("is animal")
}
```

- 输出为is people *
- 如果没有选项 case *People 情况下，会输出is animal，因为People实现了接口 Animal
- People和*People是不同的类型，需要注意