## 基本语法：

1. 只有package为**main**的包能包含main函数。（package main）
2. import “fmt” 导入其他包
3. 使用package+别名：当使用第三方包时，包名可能会非常接近或者相同，此时可以使用别名来区分和调用。
4. const PI = 3.14 常量的定义
5. var name = “dadad” 定义全局变量

（1）定义变量的格式：var 变量名字 类型;

（2）Go语言不存在隐式类型转化

（3）化简后的方式： b:=5;

（4）全局变量的声明不能省略var关键字

（5）同时声明几个相同类型的变量并初始化：a, b, c, d := 1, 2, 3, 4

（6）使用空白符号”\_”对赋值进行忽略：a, \_, c, d := 1, 2, 3, 4 // 会忽略2的赋值

（7）在结构体内部，声明结构体的属性，不能使用var

1. 声明别名： type newType int
2. 结构体：
3. 定义结构体： type People struct{ name string}
4. 声明结构体的变量： p := People{name : “yellow”} // {}里面是对结构体属性的赋值
5. 接口

（1）type+接口名字+interface

type iner interface {

}

（2）golang中，类型A如果实现了B接口的方法，那么A实现了B接口

1. import(“dad”, “fmt”) //使用这种方式一次导入多个包
2. 可见性规则：Go语言中，使用大小写来决定常量、变量、类型、接口、结构或函数是否可以被外部所调用：
3. 函数名首字母小写即为private
4. 函数名首字母大写即为public
5. Go语言中private和public的区别在于private在包外面不可见，private在包里面是可见的，public在任何地方可见。
6. 类型转换： A [:]= TypeofA(B);

（1）string(Int) 把整型转化为对应的ASCII文本。

（2）bool不能和其他类型进行转化

1. 常量：const
2. 如果声明常量以后不赋值，并且没有填写类型的时候，并且每一行声明的常量的数量是一样的，系统会给常量赋值为上一行的值。
3. 常量表达式（常量的等号右边的表达式），不允许出现运行时才能计算得到的值，必须要是编译时就能确定的值。常量表达式中必须是常量。
4. const p = "yellow"
5. const实现枚举：

iota：自动赋值，只能用在常量定义中，从常量组第一个常量开始，值为0，然后给下一个常量，值加1。（可以认为是数组下标）

1. 类型判断：

var t = reflect.Type(obj)

## Go基本类型：

1. 复数：complex64/complex128，8/16字节
2. 足够保存指针的32位或者64位整数型：uintptr
3. 浮点型：float32/float64，4/8字节，精确到小数点后7/15位
4. 零值：变量被声明以后，如果没有赋值，系统默认赋予的值。
5. char：golang中没有字符型，可以使用byte进行操作。通过string(byte)可以将整数转化为对应的ASCII字符
6. rune:

（1）rune是int32的别名。用UTF-8进行编码

（2）<http://www.golangtc.com/t/528cc004320b52227200000f>

（3）由于Golang底层是用字节存储字符串的，如果字符串中含有中文，我们获取字符串的长度和设想的会有差距。因此，可以把字符串转化为rune。就可以像正常一样用下标访问对应位置的字符，包括中文。

（4）实例：str := “abc中文” runeStr := []rune(str)

for i:=0; i<len(runeStr); i++ { fmt.Println(string(runeStr[i])) }

// a b c 中 文

1. string：
2. string类型无法修改，要修改字符串，可以先把string类型转化成rune slice，然后修改，再转化成string。

实例：

str:=”abc中文”

runeStr := []rune(str)

runeStr[0] = ‘应’

fmt.Println(string(runeStr)) // 应bc中文

1. string类型实际上是byte[]，Golang的字符串面向字节存储。比如说str:=“abc中文”。在Golang中，字符占1个字节，中文占3个字节。因此，str在内存中的表示为：

a b c 中 文

0 1 2 345 678 //使用string存储

0 1 2 3 4 // 使用rune存储

1. string的索引操作，返回一个byte的值
2. string中的值并不总是UTF-8

## Go数据结构：

1、map：

（1）如果不在初始化的时候赋值，就要通过make(map[xx]xx, n)来使用

（2）map的key类型和value类型都可以设置成interface{}，比如map[interface{}]interface{}，这样子的话，key或value可以接收任何类型的值，包括nil。

（3）map中值得注意的地方：

A、给map添加键值对的时候，m[“key”]=value，有可能是插入键值对，也有可能是修改已存在的键值对

B、map是非线程安全的，由多协程同时读写，会出现报错：concurrent map read and map write。解决办法：自定义struct来封装map，编写read和write函数，在函数中加锁，以解决对map的并发读写问题。

C、判断map中是否存在键对应的值。 if \_,ok:=m[“key”], ok {…} // 注意不能通过零值来检查，不一定准确，比如value是bool类型，不能通过判断value是false，就认为没有这个键值对。

2、数组：

（1）声明数组： arr := [3]int{1, 2, 3}

（2）声明slice： arr := []int{1, 2, 3}

（3）Golang中，想函数传递数组，是传递数组的拷贝，因此不能修改原值。解决方法可以使用指针，比如arr \*[]int， 也可以使用slice。

## Go常用包及函数：

1. strconv:

(1)strconv.Itoa(i int); //把i转化为对应的字符串

(2) strconv.Atoi(b string); // 把转化为字符串转化为对应的整数，返回类型为int、error两个

1. sort：

（1）Ints(arr []int); // 对数组排序

1. append(集合, 元素)：向集合里面添加一个元素
2. delete(集合, key)：向集合删除key对应的键值对
3. sync包

## Go常用运算符：

1、& 两个位都为1，结果为1

2、| 两个位有一个为1，结果为1

3、^ 两个位相同为0，不同为1

4、&^ 两个位，上面为1，下面为0，结果才为1，其他为0

0110

1011

-------

0100

5、&& 左右两个条件，如果左边的条件不满足了，不会执行后面的条件

6、& 左右两个条件，如果左边的条件不满足了，会执行后面的条件

7、++、-- 不能作为表达式，只能作为单独的一条语句

## Go语句：

1. if：
2. 条件表达式没有括号
3. for：
4. for {

} // 死循环

1. switch：
2. 不需要写break语句，一旦满足条件，自动终止。
3. 如果使用希望继续执行下一个case（此时无视下一个case的判断），需要使用fallthrough语句。

利用switch根据对象类型采取对应的操作。(下面的 接口interface有更新具体的实例)

func Just(obj interface{}) {

    switch obj.(type) {

    case int:

        fmt.Println("I am int")

    case string:

        fmt.Println("i am string")

    }

}

1. goto、break配合标签使用的异同：
2. 如果goto跳转到标签处，从标签处开始执行
3. break是跳出与标签相对应的循环，然后继续完后执行。

5、Select：

1. select的功能：监听I/O操作，当I/O操作发生时，触发相应的动作。
2. select的代码形式和switch非常相似，不过select的case里的操作语句只能是I/O操作
3. <http://blog.csdn.net/zhonglinzhang/article/details/45913443>
4. 实例1：

for {

            select {

            case <-time.After(1 \* time.Hour):

                this.\_load()

            }

        }

1. 表示多少时间之后，但是在取出channel内容之前不阻塞，后续程序可以继续执行。参见：<http://studygolang.com/articles/4331>

由于select等待time.After的I/O操作，因此相当于阻塞了。因此达到1小时以后执行this.\_load()语句的效果。

## 数组：

1. 定义：var arr[n] 类型，如var arr [4]int = [4]int{1, 2, 3, 4}
2. 指定数组中其中一个值：var arr = [3]int{2: 3} //把2下标对应的元素设置为3
3. 使用new申请数组空间：var p = new([5]\*int) //申请5个整型指针元素的数组，这里的p是指向数组的指针（数组指针）
4. 定义二维数组：

var arr = [2][3]int{

{1, 2, 3},

{4, 5, 6},

}

1. 去索引a到b，arr[a:b+1]

## 函数function：

1. Go函数不支持嵌套、重载和默认函数
2. 闭包

（1）概念：函数和函数特定环境的组合

（2）使用示例：

func Person(base int) func(int) int {

    return func(i int) int {

        base += i

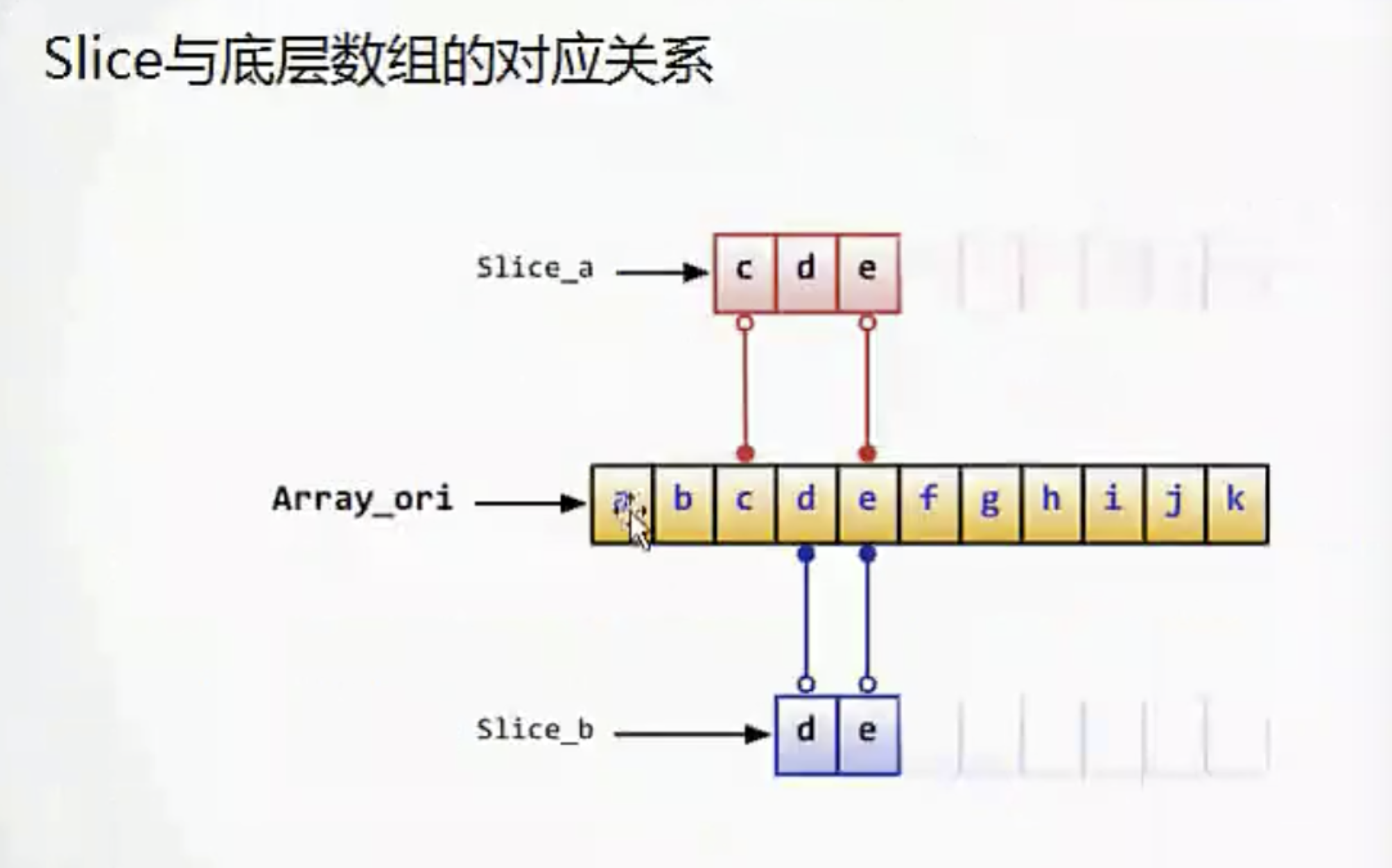
        return base // 该base的值，会在下一次继续使用

    }

}

1. defer：
2. 当一个函数调用前有关键字**defer**时,那么这个函数的执行会推迟到包含这个 defer 语句的函数即将返回前才执行.
3. 如果函数体内某个变量作为defer时匿名函数的参数，则在定义defer时即已经获得了变量拷贝，否则则是引用某个变量的地址。
4. defer调用的函数，必须放在return之前，才能被调用

## 切片slice：



1. 概念：slice是一个引用类型，它指向一个底层的数组。是可变长的。
2. var s1 []int // 中括号里面既没有数字也没有3个点，表示这个是slice数组.
3. Slice本质是指向底层数组的位置。
4. Reslice:

（0）概念：切片重组

（1）实例: slice1 = slice1[start:end] // end是新的末尾索引, start是新的起始索引

（2）Reslice时索引以被slice的切片为准

（3）索引越界不会导致底层数组的重新分配而是引发错误。（Slice索引会导致底层数组重新分配）

（4）索引不可以超过声明slice时的cap（）值(容量)

1. **Append**：
2. 可以再slice尾部追加元素
3. 可以将一个slice追加在另一个slice尾部
4. 如果最终长度未超过追加到slice的容量则返回原始slice
5. 如果超过slice的容量则将重新分配数组并拷贝原始数据
6. **Copy**：第二个slice拷贝到第一个slice中，第一个slice容量不够的话，不会扩充容量

## Range函数：

range函数是个神奇而有趣的内置函数，你可以使用它来遍历数组，切片和字典。

当用于遍历数组和切片的时候，range函数返回索引和元素；

当用于遍历字典的时候，range函数返回字典的键和值

## struct结构体：

1. 定义匿名结构体：

p := &struct {

// 这个大括号内定义结构体

name string

age int

}{

// 这个大括号内赋值

name: "abc",

age: 12,

}

1. 定义普通结构体：

type person struct {

name string

age int

}

func main() {

p := &person{

// 这个大括号内赋值

name: "abc",

age: 12,

}

fmt.Println(p)

}

## 方法method:

1. Go 中虽没有class，但依旧有method
2. 通过显式说明receiver来实现与某个类型的组合
3. 只能为同一个包中的类型定义方法
4. Receiver 可以是类型的值或者指针
5. 不存在方法重载
6. 可以使用值或指针来调用方法，编译器会自动完成转换
7. 从某种意义上来说，方法是函数的语法糖，因为receiver其实就是

方法所接收的第1个参数（Method Value vs. Method Expression）

1. 如果外部结构和嵌入结构存在同名方法，则优先调用外部结构的方法
2. 类型别名不会拥有底层类型所附带的方法
3. 方法可以调用结构中的非公开字段
4. 格式：

func (t \*TF) incre() {

(\*t) += 100

}

// 通过(t \*TF)和结构TF或者基本类型TF绑定在一起

## 接口interface：

1、package main

import (

"fmt"

)

type USB interface {

name() string

connect()

}

type sUSB struct {

n string

}

func disconnect(usb USB) { // 注意，这里使用接口类型来接收对象

// if \_, ok := usb.(sUSB); ok {

// fmt.Println("disconnect")

// }

switch v := usb.(type) {

case USB:

fmt.Println(v.name() + " " + "disconnect")

default:

}

}

func (s sUSB) name() string { // 注意，实现接口的方法，不能使用指针，需要(s sUSB)

return s.n

}

func (s sUSB) connect() {

fmt.Println("connect")

}

func main() {

var s USB = sUSB{

n: "sUSB"}

s.connect()

fmt.Println(s.name())

disconnect(s)

}

2、接口是一个或多个方法签名的集合

3、只要某个类型拥有该接口的所有方法签名，即算实现该接口，无需显式

声明实现了哪个接口，这称为 Structural Typing

1. 接口只有方法声明，没有实现，没有数据字段
2. 接口可以匿名嵌入其它接口，或嵌入到结构中
3. 将对象赋值给接口时，会发生拷贝，而接口内部存储的是指向这个

复制品的指针，既无法修改复制品的状态，也无法获取指针

1. 只有当接口存储的类型和对象都为nil时，接口变量才等于nil
2. 接口调用不会做receiver的自动转换
3. 接口同样支持匿名字段方法
4. 接口也可实现类似OOP中的多态性
5. 空接口可以作为任何类型数据的容器
6. 类型判断：

switch v := usb.(type) {

case USB:

fmt.Println(v.name() + " " + "disconnect")

default:

}

## 反射：

1. 如果结构体的字段是私有的，则反射机制无法无法反射出其字段，会报错。cannot return value obtained from unexported field or method
2. 获取类型，查看是否是结构体：

if k := t.Kind(); k != reflect.Struct {

return

}

1. t := reflect.TypeOf(s) // 相当于获取java的Class对象，java的Class对象代表.class文件
2. f := t.Field(i) // 获取第一个字段
3. v := reflect.ValueOf(s); val := v.Field(i).Interface() // 获取第i个字段的值
4. 取出结构中的匿名字段：通过序号

(1)t.FieldByIndex([]int{0: 1}) // 通过传入一个slice，0是相对最外层的结构的字段顺序，1是相对内层的字段序号

(2) v.FieldByName("Name") // 通过名称，推荐这种方法

7、 通过指针改变变量的值。

// x := 123

v := reflect.ValueOf(&x)

fmt.Printf("%v -- %p", v, &x)

v.Elem().SetInt(999)

8、调用方法: u := User{1, "OK", 12}

v := reflect.ValueOf(u)

mv := v.MethodByName("Hello")

args := []reflect.Value{reflect.ValueOf("jk")}

mv.Call(args)

## 并发：

1. goroutine：
2. goroutine :协程（轻量级线程，创建,销毁的代价很小）,协程在逻辑上上顺序执行
3. 每个协程实例占用4-5KB 的栈内存。由于实现机制而大幅减少的创建和销毁开销，是制造 Go 号称的高并发的根本原因。另外，goroutine 的简单易用，也在语言层面上给予了开发者巨大的便利
4. 并发主要由切换时间片来实现“同时”运行，在并行则是直接利用

多核实现多线程的运行，但 Go 可以设置使用核数，以发挥多核计算机

的能力

## 同步：

1、Chanel：

1. Chanel是goroutine沟通的桥梁，大都是阻塞同步的。
2. 通过make创建，close关闭
3. Chanel是引用类型
4. 可以使用单向或双向通道
5. 有缓存和无缓存的区别：无缓冲channel在读和写是都会阻塞，而有缓冲channel在向channel中存入数据没有达到channel缓存总数时，可以一直向里面存，直到缓存已满，再继续存，就会发生阻塞。
6. 使用实例1：

c := make(chan bool)

go func() {

fmt.Println("GO")

c <- true //匿名函数结束后，写入一个值

}()

<-c //main函数在这里阻塞，直到可以从chanel中读出值

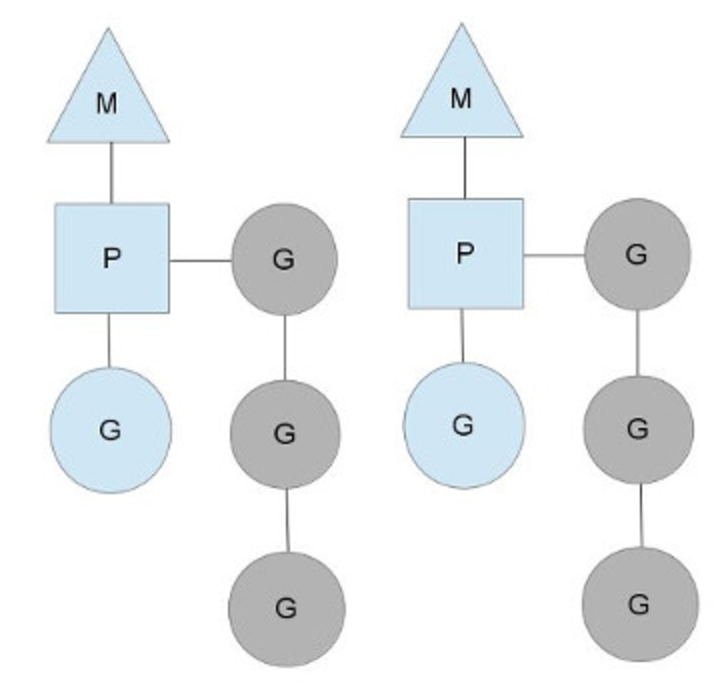
（7）关闭channel：需要在生产者线程（向channel里面存入数据的线程），调用close方法，关闭以后，可以继续读取通道，但是不能往通道里面写入数据。

2、sync.WaitGroup:

（1）可以创建任务goroutine（每创建一个goroutine，就调用一次Add(1)方法），利用Wait方法阻塞当前线程，直到所有goroutine都执行完毕了（每一个goroutine调用Done方法），才接触阻塞。

## Goroutine:

1. Goroutine的实现：



（1）M：物理线程 P：调度器 G：Goroutine

（2）一个M，负责多个G的运行，每次只有一个G可以运行，其他属于这个M的G处于就绪状态（未运行的G在runqueue队列中等待，P维护着这个队列）。

（3）P的数量代表真正的并发度，有多少个P，就代表有多少个G可以同时运行。

（4）当P对应的M1阻塞时，P会寻找一条未阻塞的线程M2继续执行G（M2的获取可以是从线程缓存中获得，也可以是新创建的）

（5）当阻塞的线程M1返回时，他会试图从其他线程那里获得一个P，以便继续执行自己的G。如果没有获得P，M1就会把自己的G放到global queue（全局的goroutine队列）中，P会周期性检查global queue，把其中的G拿出来执行。

（6）如果一个P维护的runqueue上没有G可以运行了，那么P会尝试从global queue中取得G，如果global queue也没有了，那么P会试图从其他p那里获得G。

## GO数据库编程：

（1）除了查询操作，其他操作需要用事务。

（2）rows.Scan

这个函数的参数个数必须和数据库表字段的个数一样

1. 别的表对于本表的依赖删除了，本表的记录才能删除

res, err := stmt.Exec(employee.Name, employee.Sex, employee.Email, employee.Phone)

    beego.Info(res.LastInsertId())

当Id为自动增长的，就可以通过lastId, err := res.LastInsertId()

获取插入后的Id

1. mysql不同的表的字段名要不一样，防止
2. 字段类型是字符串的话，默认要default '' 否则，可能插入的时候没有值，拿出来就是null的，会报错
3. 事务：

事务是原子的，在每一次Exec执行过程中，如果出错，则可以回退，最后都没有出错，才进行提交到数据库（commit）

事物步骤：

1. 开始事务：
2. 执行操作a,b,c每次执行都用同一个错误对象err接收错误结果
3. 执行错误检查，有错误就回滚，没有错误就把刚刚修改的东西提交到数据库(如图)

defer func() {

        if err == nil {

            tx.Commit()

        } else {

            tx.Rollback()

        }

    }()

1. 善用主键对应的记录存在时，更新记录，否则，插入记录语句：

insert into agent\_customer (AgentId,CustomerId) values (?,?) on duplicate key update AgentId = ?

比如，这条语句中，CustomerId是主键，当CustomerId这个主键的记录存在时，更新update后面的数据，否则，插入一条新的纪录。

（9）Scan的值为null的问题？ 如果存入数据库的数据为null，然后Scan接收的字段不为指针类型，就不能接收该null。因此解决的方法就是把字段弄成指针类型。

## 模板：

## 注意事项：

1. 在Golang中，一个中文的Unicode编码占3字节，字符占1字节。
2. 不用像c++，Object-C一样，Golang只要直接编译main文件，就可以生成可执行文件了。
3. vscode的launch.json中，${workspaceRoot}从环境变量gopath中获取路径。
4. 包的架构规范：

如果要新建一个工程，一般就在src下面建立一个工程文件夹。比如图中的TcpClient，然后在这个文件夹下面直接生成main.go文件，作为工程的起点。如果有子模块，就要在TcpClient这个根文件夹下面新建子模块的文件夹，比如图中client。代码不能不能直接在TcpClient中。这也是包架构规范。

1. Golang中，主线程结束，整个程序结束。（Java的话，只要还有用户线程，就会继续执行）