Go学习笔记

这是本人学习Go的学习笔记。在学习的过程中参考了许式伟的《Go语言编程》，郑兆雄的《Go Web编程》以及谢孟军《Go Web编程》。

在你学Go之前，推荐你读一篇文章：我为什么放弃Go语言，希望不要打击到你学Go的激情 :-)。放上来只是希望能客观看待Go这门语言。因为我也是看了这篇文章后，依然选择学习Go，热情不减但不迷信Go。其实语言就是拿来用的，能够解决一部分编程问题的语言就是好语言。没必要从语言中找优越感。

要点提炼

- Go语言的不同总结（与Java/C/C++比较，适合于从其他语言转来的）
    - 小写字母开头的函数只在本包内可见，大写字母开头的函数才能被其他包使用。这个规则也适用于类型和变量的可见性。
    - Go语言不需要分号结束，一行就是一条语句
    - if，switch，for等这些控制语句的条件语句不需要使用圆括号 () 括起来
    - Go的变量声明是，变量类型是放在名字的后面。甚至可以说所以需要指定类型的语句，都放在后面。基本上跟Java的声明顺序逆过来。
    - Go的左花括号 { 不能单独一行，必须函数名，if，switch，for，else，select等语句同一行
    - Go里不能导入了包后不使用，也不能声明了变量，但不使用，否则编译不过。

为什么要学Go

> 学一门技术，遵循是什么-为什么-怎么样的路线是比较好的。首先要了解这门技术是什么，能解决什么问题，怎么解决。更进一步的思考有没有比它" 更优"的解决方案。 by: 沃说德

简单一句话就是Go集齐了最近十几二十年出现的编程语言的“特长”。从Go的身上很明显可以看到C,Java,Python的一些“影子”。**Go希望成为互联网时代的C语言**。Go极力维持语言特性的简洁，力求小而精。

那么,互联网时代的C语言需要考虑哪些关键问题呢? 首先,**并行与分布式支持**。多核化和集群化是互联网时代的典型特征。作为一个互联网时代的C语言,必须要让这门语言操作多核计算机与计算机集群如同操作单机一样容易。 其次,**软件工程支持**。工程规模不断扩大是产业发展的必然趋势。单机时代语言可以只关心问题本身的解决,而互联网时代的C语言还需要考虑软件品质保障和团队协作相关的话题。 最后,**编程哲学的重塑**。计算机软件经历了数十年的发展,形成了面向对象等多种学术流派。什么才是最佳的编程实践?作为互联网时代的C语言,需要回答这个问题。 接下来我们来聊聊Go语言在这些话题上是如何应对的。

- 并发与分布式

  多核化和集群化是互联网时代的典型特征。为了提高并发性，人们“发明”了多进程，但由于多进程“太重”，进一步“细化”，“发明”了多线程，但多线程还是“太重”，于是“发明”了协程(coroutine,也叫轻量级线程)。那为什么要这么“细化”呢？当然是为了提高并发的性能啦（废话），因为人们发现进程,线程的切换都需要从内核态和用户态之间切换。而协程是“尽量”在用户态进行切换，不到迫不得已才进行内核态和用户态之间切换。

  多数语言在语法层面并不直接支持协程，而通过库的方式支持的协程的功能也并不完整,比如仅仅提供协程的创建、销毁与切换等能力。Go语言在语言级别支持协程，叫goroutine。Go语言标准库提供的所有系统调用(syscall)操作，当然也包括所有同步IO操作，都会出让CPU给其他goroutine，这让事情变得非常简单。

并发有2个需要关注的地方，第一个是能创建、销毁、同步与切换并发执行的“执行体”。第二个是能进行通信。“执行体之间的消息传递”在并发编程模型的选择上,有两个流派,一个是共享内存模型,一个是消息传递模型。多数传统语言选择了前者,少数语言选择后者,其中选择“消息传递模型”的最典型代表是Erlang语言。Erlang语言中，当执行体之间需要相互传递消息时,通常需要基于一个消息队列(message queue)或者进程邮箱(process mail box)这样的设施进行通信。

Go语言推荐采用“Erlang风格的并发模型”的编程范式,尽管传统的“共享内存模型”仍然被保留,允许适度地使用。在Go语言中内置了消息队列的支持,只不过它叫通道(channel)。两个goroutine之间可以通过通道来进行交互。

- 软件工程

  多数软件需要一个团队共同去完成,在团队协作的过程中,人们需要建立统一的交互语言来降低沟通的成本。规范化体现在多个层面,如:

  - 代码风格规范
  - 错误处理规范
  - 包管理
  - 契约规范(接口)
  - 单元测试规范
  - 功能开发的流程规范

  Go语言很可能是第一个将代码风格强制统一的语言，例如Go语言要求 public 的变量必须以大写字母开头，private 变量则以小写字母开头，这种做法不仅免除了 public 、 private 关键字，更重要的是统一了命名风格。Go 语言还首创一种错误处理规范。另外，Go语言对 { } 应该怎么写进行了强制,比如以下风格是正确的：

  ```
  if expression {
      .....
  }
  ```

  但下面这个写法就是错误的:

  ```
  if expression
  {
      .....
  }
  ```

- 编程哲学

  C语言是纯过程式的，这和它产生的历史背景有关。Java语言则是激进的面向对象主义推崇者，典型表现是它不能容忍体系里存在孤立的函数。而**Go语言没有去否认任何一方，而是用批判吸收的眼光，将所有编程思想做了一次梳理，融合众家之长，但时刻警惕特性复杂化，极力维持语言特性的简洁，力求小而精**。

  首先，Go语言反对**函数**和**操作符**重载(overload)，而C++、Java和C#都允许出现同名函数或操作符,只要它们的参数列表不同。虽然重载解决了一小部分面向对象编程(OOP)的问题,但同样给这些语言带来了极大的负担。

  其次,Go语言支持类、类成员方法、类的组合,但反对继承,反对虚函数(virtual function)和虚函数重载。确切地说,Go也提供了继承,只不过是采用了组合的文法来提供。

  再次,Go语言也放弃了构造函数(constructor)和析构函数(destructor)。

  在放弃了大量的OOP特性后，Go语言送上了一份非常棒的礼物：接口(interface)。多数语言都提供接口,但它们的接口都不同于Go语言的接口。Go语言中的接口与其他语言最大的一点区别是它的非侵入性。

Go的语言特性

Go语言作为一门全新的静态类型开发语言。Go语言最主要的特性如下：

- 自动垃圾回收

  C和C++语言的指针是一个既爱又恨的“特性”，使用不当造成的内存泄露问题令人抓狂。到目前为止,内存泄露的最佳解决方案是在语言级别引入自动垃圾回收算法(Garbage Collection,简称GC)。所谓垃圾回收,即所有的内存分配动作都会被在运行时记录,同时任何对该内存的使用也都会被记录,然后垃圾回收器会对所有已经分配的内存进行跟踪监测,一旦发现有些内存已经不再被任何人使用,就阶段性地回收这些没人用的内存。

  Go语言也具有自动垃圾回收器。**PS：C/C++没有支持GC的原因之一：支持指针运算使得实现GC很困难**。

- 更丰富的内置类型

  除了几乎所有语言都支持的简单内置类型(比如整型和浮点型等)外,Go语言也内置了一些比较新的语言中内置的高级类型,比如C#和Java中的数组和字符串。

  除此之外,Go语言还内置了一个对于其他静态类型语言通常用库方式支持的字典类型(map)。另外有一个新增的数据类型:数组切片(Slice)。我们可以认为数组切片是一种可动态增长的数组。

- 函数多返回值

  目前的主流语言中除Python外基本都不支持函数的多返回值功能,在不支持多返回值的语言中我们有以下两种做法:要么专门定义一个结构体用于返回,或者以传出参数的方式返回多个结果。

  Go语言革命性地在静态开发语言阵营中率先提供了多返回值功能。这个特性让开发者可以从原来用各种比较别扭的方式返回多个值的痛苦中解脱出来,既不用再区分参数列表中哪几个用于输入,哪几个用于输出,也不用再只为了返回多个值而专门定义一个数据结构。

- 错误处理

  Go语言引入了3个关键字用于标准的错误处理流程,这3个关键字分别为 defer 、 panic 和 recover 。整体上而言与C++和Java等语言中的异常捕获机制相比,Go语言的错误处理机制可以大量减少代码量,让开发者也无需仅仅为了程序安全性而添加大量一层套一层的 try-catch 语句。

- 匿名函数和闭包

  在Go语言中,所有的函数也是值类型,可以作为参数传递。Go语言支持常规的匿名函数和闭包,开发者可以随意对该匿名函数变量进行传递和调用。

- 类型和接口

  Go语言的类型定义非常接近于C语言中的结构(struct),甚至直接沿用了 struct 关键字。相比而言,Go语言并没有直接沿袭C++和Java的传统去设计一个超级复杂的类型系统,不支持继承和重载,而只是支持了最基本的类型组合功能。但又可以较好的表示出类和对象的功能。

  Go语言也不是简单的对面向对象开发语言做减法,它还引入了一个无比强大的“非侵入式”接口的概念,让开发者从以往对C++和Java开发中的接口管理问题中解脱出来。

- 并发编程

  Go语言引入了goroutine概念,它使得并发编程变得非常简单。通过使用goroutine而不是裸用操作系统的并发机制,以及使用消息传递来共享内存而不是使用共享内存来通信,Go语言让并发编程变得更加轻盈和安全。

  通过在函数调用前使用关键字 go ,我们即可让该函数以goroutine方式执行。goroutine是一种比线程更加轻盈、更省资源的协程。Go语言通过系统的线程来多路派遣这些函数的执行,使得每个用 go 关键字执行的函数可以运行成为一个单位协程。当一个协程阻塞的时候,调度器就会自动把其他协程安排到另外的线程中去执行,从而实现了程序无等待并行化运行。而且调度的开销非常小,一颗CPU调度的规模不下于每秒百万次,这使得我们能够创建大量的goroutine,从而可以很轻松地编写高并发程序,达到我们想要的目的。

Go语言实现了CSP(通信顺序进程，Communicating Sequential Process) 模型来作为 goroutine 间的推荐通信方式。在CSP模型中，一个并发系统由若干并行运行的顺序进程组成，每个进程不能对其他进程的变量赋值。进程之间只能通过一对通信原语实现协作。Go语言用 channel(通道) 这个概念来轻巧地实现了CSP模型。channel的使用方式比较接近Unix系统中的管道(pipe)概念，可以方便地进行跨goroutine的通信。

另外,由于一个进程内创建的所有goroutine运行在同一个内存地址空间中,因此如果不同的goroutine不得不去访问共享的内存变量,访问前应该先获取相应的读写锁。Go语言标准库中的sync 包提供了完备的读写锁功能。

- 反射

  反射(reflection)是在Java语言出现后迅速流行起来的一种概念。通过反射,你可以获取对象类型的详细信息,并可动态操作对象。反射是把双刃剑,功能强大但代码可读性并不理想。

  Go语言的反射实现了反射的大部分功能,但没有像Java语言那样内置类型工厂,故而无法做到像Java那样通过类型字符串创建对象实例。在Java中,你可以读取配置并根据类型名称创建对应的类型,这是一种常见的编程手法,但在Go语言中这并不被推荐。

  反射最常见的使用场景是做对象的序列化(serialization,有时候也叫Marshal & Unmarshal)。例如,Go语言标准库的encoding/json、encoding/xml、encoding/gob、encoding/binary等包就大量依赖于反射功能来实现。

- 语言交互性

  由于Go语言与C语言之间的天生联系,Go语言的设计者们自然不会忽略如何重用现有C模块的这个问题,这个功能直接被命名为Cgo。Cgo既是语言特性,同时也是一个工具的名称。

  在Go代码中,可以按Cgo的特定语法混合编写C语言代码,然后Cgo工具可以将这些混合的C代码提取并生成对于C功能的调用包装代码。开发者基本上可以完全忽略这个Go语言和C语言的边界是如何跨越的。与Java中的 JNI 不同, Cgo的用法非常简单。

Go的基础

1. 开始Go

1.1 HelloWorld

程序员学习一门语言的第一个程序：HelloWorld。(先安装Go环境，本人使用的IDE是GoLand)

```
package main  //每个Go源代码文件的开头都是一个 package 声明,表示该Go代码所属的包。必须要有个main包

import "fmt"    // 格式化io包，相当于Java的System，C++的iostream

func main() {   /*main函数*/
    fmt.Println("Hello,World!")
}
```

1.2 Go程序的基本组成

1. Go 程序的基础组成有以下几个部分：
   - 包声明：package 包名
   - 引入包：import 包名
   - 函数
   - 变量
   - 语句 或 表达式
   - 注释：行注释：`//行注释`，多行注释：`/*多行注释*/`。

2. 几点强调：
    - 必须在源文件中非注释的第一行指明这个文件属于哪个包，如：package main。package main表示一个可独立执行的程序。
    - **要生成Go可执行程序，必须要建立一个名字为 main 的包，并且在该包中包含一个叫main() 的函数(该函数是Go可执行程序的执行起点)。Go语言的 main()函数不能带参数，也不能定义返回值**。
    - **不得包含在源代码文件中没有用到的包，和定义不使用的变量，**否则Go编译器会报编译错误。**强制左花括号 { 的放置位置，左花括号 { 不能单独一行**。
    - 在 Go 程序中，一行代表一个语句结束。每个语句不需要像 C 家族中的其它语言一样以分号 ; 结尾，因为这些工作都将由 Go 编译器自动完成。当然，如果你打算将多个语句写在同一行，它们则必须使用 ; 人为区分，但在实际开发中Go并不鼓励这种做法。

1.3 Go程序的执行

1. 实际上，Go可以有两种运行方式(但Go是一门编译型语言)：
    - 解释执行（实际是编译成A.OUT再执行）：go run xxx.go

      ```
      cmh@cmh:~/LearningGo$ go run HelloWorld.go
      Hello,World!
      ```

    - 编译执行（Go的编译是静态编译）：go build xxx.go

      ```
      cmh@cmh:~/LearningGo$ go build HelloWorld.go
      cmh@cmh:~/LearningGo$ ./ HelloWorld
      Hello,World!
      ```

2. Go的调试：使用GDB调试，不用设置什么编译选项，Go语言编译的二进制程序直接支持GDB调试。可以自行参考GDB调试。也可以使用IDE的调试。

1.4 main 函数和 init 函数

1. Go里面有两个保留的函数：init 函数(能够应用于所有的 package)和 main 函数(只能应用于package main)
2. 这两个函数在定义时不能有任何的参数和返回值。虽然一个 package 里面可以写任意多个 init 函数，但这无论是对于可读性还是以后的可维护性来说，我们都强烈建议用户在一个 package 中每个文件只写一个 init 函数。
3. 程序的初始化和执行都起始于 main 包。如果 main 包还导入了其它的包，那么就会在编译时将它们依次导入。有时一个包会被多个包同时导入，那么它只会被导入一次(例如很多包可能都会用到 fmt 包,但它只会被导入一次,因为没有必要导入多次)。当一个包被导入时，如果该包还导入了其它的包，那么会先将其它包导入进来，然后再对这些包中的包级常量和变量进行初始化，接着执行 init 函数(如果有的话)，依次类推。等所有被导入的包都加载完毕了，就会开始对 main 包中的包级常量和变量进行初始化，然后执行 main 包中的 init 函数(如果存在的话)，最后执行 main 函数。

2. 命名与声明

2.1命名

Go语言中的函数名、变量名、常量名、类型名、语句标号和包名等所有的命名,都遵循一个简单的命名规则：一个名字必须以一个字母(Unicode字母)或下划线开头,后面可以跟任意数量的字母、数字或下划线。区分大小写。

2.2声明

> 声明语句定义了程序的各种实体对象以及部分或全部的属性。**Go语言主要有四种类型的声明语句:var、const、type和func,分别对应变量、常量、类型和函数实体对象的声明**。最后还包括函数，包声明。

1. 分组(打包)声明

- 在Go语言中,同时声明多个常量、变量，或者导入多个包时，可采用分组(打包)的方式进行声明
- 例如下面的代码：

```
import "fmt"
import "os"
const i = 100
const pi = 3.1415
const prefix = "Go_"
var i int
var pi float32
var prefix string

// 可以分组写成如下形式：
import(
    "fmt"
    45"os"
)
const(
    i = 100
    pi = 3.1415
    prefix = "Go_"
)
var(
    i int
    pi float32
    prefix string
)
```

3. 包和文件

1. Go语言标准库里的包名字都是小写的，只有与其他语言交互时，才大写。
2. Go语言的所有符号(symbol)里，以大写字母开头的常量在包外可见。小写字母开头为包内私有，不可被其他包访问。
3. 每个 Go 程序都是由包组成的，程序运行的入口是包 `main`。
4. 相对路径和绝对路径：**import支持如下两种方式来加载自己写的模块**。标准库无需这样。
    - 相对路径：`import "./model"` 当前文件同一目录的model目录，但是不建议这种方式来import
    - 绝对路径：`import "shorturl/model"` 加载gopath/src/shorturl/model模块.
5. 特殊的import：
    - 点操作：点操作的含义就是这个包导入之后在你调用这个包的函数时，你可以省略前缀的包名。
    - 别名操作：别名操作顾名思义就是我们可以把包命名成另一个用起来容易记忆的名字。别名操作的话调用包函数时前缀变成了我们的前缀。
    - _ 操作：_ 操作其实是引入该包，而不直接使用包里面的函数，而是只调用执行了该包里面的init函数。

4. 变量, 常量

4.1 变量

> Go语言中的变量使用方式与C语言接近，但变量声明方式与C和C++语言有明显的不同。对于纯粹的变量声明，Go语言引入了关键字 var ，而类型信息放在变量名之后。变量声明语句不需要使用分号作为结束符，当然也可以使用。与C语言相比，Go语言摒弃了语句必须以分号作为语句结束标记的习惯。

1. 变量的声明及初始化：
   - 单个变量声明格式：**其中"类型"或"= 表达式"两个部分可以省略其中的一个**。如果省略的是类型信息，那么将根据初始化表达式来推导变量的类型信息。如果初始化表达式被省略，那么将用零值初始化该变量。

```
var 变量名字[,变量名2] 类型  = 表达式
例如:
var v1, v2, v3 int
var v4 int = 10
var value1, value2, value3, value4 = 10, true, 2.4, "four"    //声明时初始化
var arr [10]int  //数组
var slic []int  //数组切片
var v5 struct {
    f int  //前面的已有var了
}
var v6 *int     //指针
var v7 map[string]int  //map, key为string类型,value为int类型
var v8 func(a int) int //函数指针?
```

   - 多个不同类型的变量一起声明：避免重复写var，例如：

```
var (
    v1 int
    v2 string
)
//或者
var value1, value2, value3, value4 = 10, true, 2.4, "four"
```

   - 简短变量声明：在函数内部,有一种称为简短变量声明语句的形式可用于声明和初始化局部变量。省略了关键字var，变量的类型根据表达式来自动推导。**出现在 := 左侧的变量不应该是已经被声明过的,否则会导致编译错误**

```
格式: 变量名 := 表达式
v1 := 2
v2, v3 := 0, 2.4
var v4 int
v4 := 3     //会出错，因为 v2 已经声明过了。
```

2. 变量的赋值：在Go语法中,变量初始化和变量赋值是两个不同的概念。
   - 普通赋值：跟C/C++/Java一样，直接等号赋值，例如：

```
var v1 int
v1 = 10
```

   - 多重赋值：可以直接交换2个或多个变量，无需引入中间变量，例如：

```
var v1, v2 = 11, 22   //v1=11, v2=22
v1,v2 = v2, v1  //v1=22, v2=11
```

3. 匿名变量：其他语言的在调用函数时为了获取一个值，却因为该函数返回多个值而不得不定义一堆没用的变量。在Go中这种情况可以通过结合使用多重返回和匿名变量来避免这种丑陋的写法，让代码看起来更优雅。所谓的匿名变量就是使用 "_" 来作为占位变量，从而接收想接收的变量。例如：

```
func getName() (firstName, lastName, nickName string) {
    return "chen", "minghai", "jinminghai"
}
_, _, nickName := getName()  //我只想要nickname，其他不要，故用匿名变量来接收。
```

4.2 常量

> 在Go语言中,常量是指编译期间就已知且不可改变的值。常量可以是数值类型(包括整型, 浮点型和复数类型)、布尔类型、字符串类型等。

1. **Go语言的其他符号(symbol)一样，以大写字母开头的常量在包外可见。小写字母开头为包内私有，不可被其他包访问。**
2. 字面常量：所谓字面常量(literal),是指程序中硬编码的常量，例如：

```
-12
3.14159     //浮点数类型常量
3.2+12i     //复数类型常量
true        //布尔类型常量
"foo"       //字符串常量
```

   在其他语言中,常量通常有特定的类型,比如12在C语言中会认为是一个 int 类型的常量。如果要指定一个值为 -12 的 long 类型常量，需要写成12l,这有点违反人们的直观感觉。**Go语言的字面常量更接近我们自然语言中的常量概念，它是无类型的。只要这个常量在相应类型的值域范围内，就可以作为该类型的常量**，比如上面的常量 -12 ，它可以赋值给 int ,uint ,int32 ,int64 ,float32 ,float64 ,complex64 ,complex128 等类型的变量。

3. 常量定义：通过 const 关键字,你可以给字面常量指定一个友好的名字。Go的常量定义可以限定常量类型,但不是必需的。**如果定义常量时没有指定类型,那么它与字面常量一样,是无类型常量**。常量的赋值是一个编译期行为,所以右值不能出现任何需要运行期才能得出结果的表达式。

```
const PI float64 = 3.14159265358979323846
const zero = 0.0    //无类型浮点数，即既可以是单精度，也可以是双精度浮点数。
const u, v float32 = 0, 3       // u = 0.0, v = 3.0,常量的多重赋值
const a, b, c = 3, 4, "foo"     // a = 3, b = 4, c = "foo", 无类型整型和字符串常量
```

4. 预定义的常量：Go语言预定义了这些常量：true 、 false 和 iota 。**iota** 比较特殊，可以被认为是一个可被编译器修改的常量，**在每一个 const 关键字出现时被重置为0，然后在下一个 const 出现之前，每出现一次 iota ，其所代表的数字会自动增1**。

```
const (
    u = iota * 44   // u=0
    v float64 = iota * 44   // v= 44.0
    w = iota * 44   // w=88
)
const x = iota  // x=0 ,因为iota被重置了。
```

5. 枚举：枚举指一系列相关的常量。Go语言中通常用const 后跟一对圆括号的方式定义一组常量来定义枚举值。**Go语言并不支持众多其他语言明确支持的 enum 关键字**。例如：

```
//如果两个 const 的赋值语句的表达式是一样的,那么可以省略后一个赋值表达式。
//例子中 numberOfDays 为包内私有,其他符号则可被其他包访问。
const (
    Sunday = iota  //Sunday == 0
    Monday  //Sunday == 1,如果两个 const 的赋值语句的表达式一样,那么可以省略后一个赋值表达式。
    Tuesday
    Wednesday
    Thursday
    Friday
    Saturday
    numberOfDays //这个常量没有导出,即只能包内访问。
)
```

5. 基础数据类型, 运算符

5.1 基础数据类型

1. 布尔类型: bool 。
    - Go语言中的布尔类型与其他语言基本一致,关键字也为 bool ,可赋值为预定义的 true 和 false。**布尔类型不能接受其他类型的赋值，*不支持自动或强制的类型转换***。例如：

      ```
      var v1 bool
      v1 = true
      var v2 =true
      v3 := false
      v4 := (1==2)  //v4也会被推导为bool类型
      //错误示范
      var b bool
      b = 1 //编译出错
      b = bool(1)  //编译出错
      ```

2. 整型: int8 、 byte 、 int16 、 int 、 uint 、 uintptr 等。
    - Go语言支持表2-1所示的这些整型类型，一般使用int，uint，uintptr就够了。

表 2-1

类　型	长度（字节）	值　范　围
int8	1	−128 ~ 127
uint8（即byte）	1	0 ~ 255
int16	2	−32 768 ~ 32 767
uint16	2	0 ~ 65 535
int32	4	−2 147 483 648 ~ 2 147 483 647
uint32	4	0 ~ 4 294 967 295
int64	8	−9 223 372 036 854 775 808 ~ 9 223 372 036 854 775 807
uint64	8	0 ~ 18 446 744 073 709 551 615
int	平台相关	平台相关
uint	平台相关	平台相关
uintptr	同指针	在32位平台下为4字节，64位平台下为8字节

3. 浮点类型: float32 、 float64 。
    - Go语言定义了两个类型 float32 和 float64 ，其中 float32 等价于C语言的 float 类型，float64 等价于C语言的 double 类型。**若不明确指定类型，默认浮点数类型是float64**。
4. 复数类型: complex64 、 complex128 。
    - 复数实际上由两个实数(在计算机中用浮点数表示)构成，一个表示实部(real),一个表示虚部(imag)。若不明确指定类型，默认复数类型是complex128。
    - 复数的表示：一种是直接想数学那样使用 i 来表示虚部，或者使用complex(实部, 虚部)。例如：

```
 var c1 complex64
c1 = 3.2 + 12i
c2 := 3.2 -12i   // c2被自动推导为complex128
c3 := complex(3.2, -12)  // c3 === c2
```

    - 获取复数的实部和虚部(计算)：可以通过Go语言内置函数 real(z) 获得该复数的实部，也就是 x ，通过 imag(z) 获得该复数的虚部。关于复数的函数,请查阅 math/cmplx 标准库的文档。
5. 字符串: string 。（Go编译器支持UTF-8的源代码文件格式）
    - 在Go语言中,字符串也是一种基本类型。Go的字符串有点类似与Java的字符串，字符串的内容可以用类似于数组下标的方式获取，但初始化后就不可修改了。也可以直接用 ” + “ 来拼接。例如：

```
 var str string
str = "Hello,World" + "!"
ch := str[0]  //ch = 72，即 H 的ASCII码
length := len(str)  //获取str的长度
str[0] = 'h'  //编译出错
```

    - 字符串操作（参见strings包）
    - 字符串遍历：Go语言支持两种方式遍历字符串。
        - 一种是以**字节数组**的方式遍历（即字符串用字节数组表示）：

```
 str := "Hello,世界"
n := len(str)    //这个字符串长度为12。虽然它只有8个字符，中文字符在UTF-8中占3个字节
for i := 0; i < n; i++ {
    ch := str[i]    // 依据下标取字符串中的字符,类型为byte
    fmt.Println(i, ch)
}
这个例子的输出结果为:
0 72
1 101
....
10 149
11 140
```

- 另一种是以Unicode字符遍历(Unicode字符方式遍历时,每个字符的类型是 rune ， 而不是 byte .)

```
 str := "Hello,世界"
n := len(str)    //这个字符串长度为12。虽然它只有8个字符
for i := 0; i < n; i++ {
    ch := str[i]    // 依据下标取字符串中的字符,类型为byte
    fmt.Println(i, ch)
}
这个例子的输出结果为:
0 72
1 101
2 108
3 108
4 111
5 44
6 19990
9 30028
```

6. 字符类型: rune 。
   - 在Go语言中支持两个字符类型，一个是 byte (实际上是 uint8 的别名)，代表UTF-8字符串的单个字节的值；另一个是 rune ，代表单个Unicode字符。
   - 关于 rune 相关的操作，可查阅Go标准库的 unicode 包。另外 unicode/utf8 包也提供了 UTF8 和Unicode之间的转换。出于简化语言的考虑，Go语言的多数API都假设字符串为UTF-8编码。尽管Unicode字符在标准库中有支持，但实际上较少使用。

5.2 类型转换

1. 类型转换：转换格式：`类型(变量)`。
2. 需要注意的是， int 和 int32 在Go语言里被认为是两种不同的类型,编译器也不会帮你自动做类型转换。需要自己强制类型转换。例如:

```
var v1 int32 =444
v2 := 444    //v2会被自动推导为int型。
v2 = v1  //会编译出错
v2 = int(v1)  //这样才行
```

3. 做强制类型转换时,需要注意数据长度被截短而发生的数据精度损失 (比如将浮点数强制转为整数) 和值溢出 (值超过转换的目标类型的值范围时) 问题。

5.3 运算符

1. 算术运算符：加`+`, 减`-`, 乘`*`, 除`/`, 求余`%`, 自增`++`,自减`--`。跟C/Java的运算效果一样。
2. 关系运算符：相等`==`, 不等`!=`, 大于`>`, 小于`<`, 大于等于`>=`, 小于等于`<=`。跟C/Java的运算效果一样.
3. 逻辑运算符：与`&&`, 或`||`, 非`!`。Go的非是`!`而不是`~`。跟C/Java的运算效果一样。
4. 位运算符： 按位与`&`, 按位或`|`, 按位异或`^`, 左移`<<`, 右移`>>`。
5. 赋值运算符：上面的所有运算符加上`=`就是赋值运算符了，例如：`-=`, `+=`, `&=`, `=`。
6. 其他运算符：取地址运算符`&变量`， 取指针变量指向的值运算符`*变量`。

5.4 作用域

1. 作用域为已声明标识符所表示的常量、类型、变量、函数或包在源代码中的作用范围。
2. Go 语言中变量可以在三个地方声明:
   - 函数内定义的变量称为局部变量，只能到函数内的对于语句块内使用
   - 函数外定义的变量称为全局变量，可以在包内使用
   - 函数定义中的变量称为形式参数，可以在函数内使用

6. 函数

6.1 函数的定义和调用

> 函数构成代码执行的逻辑结构。在Go语言中，函数的基本组成为：关键字 func 、函数名、参数列表、返回值、函数体和返回语句。支持多返回值。可以定义和操作返回值。**在Go里函数和方法的概念是不一样的**。

1. 函数的定义

   > Go语言中函数名字的大小写不仅仅是风格，更直接体现了该函数的可见性，这一点尤其需要注意。小写字母开头的函数只在本包内可见，大写字母开头的函数才能被其他包使用。

   - 格式：其中返回值列表可以是无名，只有类型，且如果返回值只有一个，可以省略圆括号。如果形参或返回值中**若干相邻**的参数类型相同，可以合并类型，即省略前面变量的类型声明。

     ```
     func 函数名( [形式参数列表] ) [(返回值列表)] {
         .....
     }
     ```

   - 例子:

     ```
     func Add(a,b int) (sum int, e error) {
         if a<0 || b<0 || c {
             e = errors.New("Should be non-negative numbers!")    // 操作返回值
             return
         }
         return a+b, nil      // nil相当于NULL
     }
     // 如果不带错误处理
     func Add(a,b int, c bool) int {
         if c {
     ```

```
        return
    }
    return a+b
}
```

2. 函数的调用：
    - 格式：函数名([实参列表])
    - 函数调用非常方便，只要事先导入了该函数所在的包，就可以直接调用。

6.2 不定参数

> 不定参数是指函数传入的参数个数为不定数量。合适地使用不定参数，可以让代码简单易用，尤其是输入输出类函数，比如日志函数等。

1. 不定参数的定义及原理
    - 定义：格式：`名称 ...类型`。形如`...类型`格式的类型**只能作为函数的参数类型存在，并且必须是最后一个参数**。例如：

```
func myfunc(args ...int) {
    for _, v := range args {
        fmt.Println(v)
    }
}
//调用
myfunc(2, 3, 4)
myfunc(2, 3, 4, 5)
```

    - 原理：不定参数是一个语法糖(syntactic sugar)，即这种语法对语言的功能并没有影响,但是更方便程序员使用。通常来说，使用语法糖能够增加程序的可读性，从而减少程序出错的机会。从内部实现机理上来说，名称 ...type 本质上是一个数组切片，也就是 [ ]type ，这也是为什么么上面的参数 args 可以用 for 循环来获得每个传入的参数。
2. 不定参数的传递：例如要将函数的不定参数传递给另一个函数该怎么办呢？格式：`名字...`。不能省略后面的三个点。例如：

```
func myfunc(args ...int) {
    // 按原样传递
    otherfunc(args...)
    // 传递片段,实际上任意的int slice都可以传进去
    anotherfunc(args[1:]...)
}
```

3. 任意类型的不定参数：格式：`名字 ...interface{ }`。前面的不定参数虽然个数不定，但类型要一致。如果还要类型不一致，需要指定不定参数类型为：interface{} 。用 interface{} 传递任意类型数据是Go的惯例用法。使用 interface{} 仍然是类型安全的，这和 Java 的接口或Object不太一样。例如：

```
func MyPrintln(args ...interface{}) {
    for _,arg := agrs {
        switch arg.(type) {
            case int:
```

```
                fmt.Println(arg, " is an int value.")
            case string:
                fmt.Println(arg, " is an string value.")
            case int64:
                fmt.Println(arg, " is an int64 value.")
            default:
                fmt.Println(arg, " is an unknown typr.")
        }
    }
}
```

6.3 多返回值

1. 定义多返回值函数，函数原型如下：

```
func Read(b []byte) (n int, err Error)
```

它可以同时返回读取的字节数和错误信息。如果读取文件成功,则返回值中的 n 为读取的字节数，err 为 nil ； 否则 err 为具体的出错信息。

同样，从上面的方法原型可以看到，我们还可以给返回值命名，就像函数的输入参数一样。返回值被命名之后，它们的值在函数开始的时候被自动初始化为空。**在函数中执行不带任何参数的 return 语句时，会默认返回全部对应的返回值变量的值**。Go语言并不需要强制命名返回值,但是命名后的返回值可以让代码更清晰，可读性更强，同时也可以用于文档。

2. 选择性接收函数返回值：如果调用了一个具有多返回值的方法，但是却不想关心其中的某个返回值，可以简单地用一个下划线 “ _ ” 来跳过这个返回值，比如下面的代码表示调用者在读文件的时候不想关心 Read() 函数返回的错误码：

```
n,_ := f.Read(buffer)
```

6.4 匿名函数与闭包

1. 匿名函数：匿名函数是指不需要定义函数名的一种函数实现方式。
    - 在Go里面，**函数可以像普通变量一样被传递或直接使用，这与C语言的回调函数比较类似**。不同的是，Go语言支持随时在代码里定义匿名函数。
    - 例子：匿名函数可以直接赋值给一个变量或者直接执行：

```
// 匿名函数直接赋值给变量，可以作为参数传递
f := func(a,b int, z float64) bool {
    return a*b < int(z)
}
f(3, 4, 10.5)   //通过变量调用

//定义时直接使用执行
func(a,b int, z float64) bool {
    return a*b < int(z)
} (3, 4, 13.3)   // 函数定义完后，直接在花括号后面加上实参列表即可。
```

2. 闭包：Go的匿名函数就是一个闭包。

- 基本概念

  闭包是可以包含自由 (未绑定到特定对象) 变量的代码块，这些自由变量从未在这个代码块内或者任何全局上下文中定义过，而是在定义代码块的环境中定义。要执行的代码块 (由于自由变量包含在代码块中，所以这些自由变量以及它们引用的对象没有被释放) 为自由变量提供了绑定的计算环境(作用域) 。

- 闭包的价值

  闭包的价值在于可以作为函数对象或者匿名函数，对于类型系统而言，这意味着不仅要表示数据还要表示代码。支持闭包的多数语言都将函数作为第一级对象，就是说这些函数可以存储到变量中作为参数传递给其他函数，最重要的是能够被函数动态创建和返回。

- Go语言中的闭包

  - Go语言中的闭包同样也会引用到函数外的变量。闭包的实现确保了只要闭包还被使用，那么被闭包引用的变量会一直存在。例如：

```
func main() {
    var j int = 5
    a := func()( func(i int) ) {
        var k int = 4  //自由变量
        return func(i int) {
            fmt.Printf("i, j, k: %d, %d, %d\n", i, j, k)
        }
    }
    a()(5)
    j *=2
    a()(10)  // a()返回一个匿名函数，最后的()表示实参列表。
}
执行结果是：
i, j, k: 10, 5, 4
i, j, k: 10, 10, 4
```

7. 流程控制语句

> 程序设计语言的流程控制语句，用于设定计算执行的次序,建立程序的逻辑结构。可以说，流程控制语句是整个程序的骨架。

1. Go语言支持如下的几种流程控制语句:
   - 条件语句，对应的关键字为 if 、 else 和 else if ;
   - 选择语句，对应的关键字为 switch 、 case、 fallthrough 和 select (将在介绍channel的时候细说) ;
   - 循环语句，对应的关键字为 for 和 range 、break 、continue;
   - 跳转语句，对应的关键字为 goto 。

7.1 条件语句

1. 条件语句：关键字为 if 、 else 和 else if
   - 格式：

```
if [变量初始化语句;] 条件 {
    ....
}
//例如
if i:=2; a*i < 5 {
    return -1
} else if a > 10 {
    return 1
} else {
    return i
}
```

- 注意事项：关于条件语句,需要注意以下几点
  - 条件语句不需要使用括号将条件包含起来 () ;
  - 无论语句体内有几条语句，花括号 {} 都是必须存在的 ;
  - 左花括号 { 必须与 if 或者 else 处于同一行;
  - 在 if 之后,条件语句之前，可以添加变量初始化语句，使用 ; 间隔;

7.2 选择语句

1. 选择语句：关键字为 switch,case,fallthrough 和 select。根据传入条件不同，选择语句执行不同的语句。
   - 格式：**case中不需要break，默认执行完后跳出**，除非使用fallthrough，才会接着执行下面的case

```
switch i {
    case 0:
        fmt.Printf("0")  // i=0 时打印完 0 后退出
    case 1:
        fmt.Printf("1")
    case 2:
        fallthrough  // i=2 时，因为使用了fallthrough，故执行下面的case
    case 3:
        fmt.Printf("3")
    case 4, 5, 6:    // case 可以多个条件，用 , 隔开
        fmt.Printf("4, 5, 6")
    default:
        fmt.Printf("Default")
}
```

   - **switch 后面的表达式甚至不是必需的**，例如：

```
Num := 5
switch {  //switch 后面没有表达式
    case 0 <= Num && Num <= 3:
        fmt.Printf("0-3")
    case 4 <= Num && Num <= 6:
        fmt.Printf("4-6")
    case 7 <= Num && Num <= 9:
        fmt.Printf("7-9")
}
```

- 注意事项：在使用 switch 结构时,我们需要注意以下几点
  - 左花括号 { 必须与 switch 处于同一行;
  - case的条件表达式不限制为常量或者整数，可以是字符串，变量等;
  - 一个 case 中，可以出现多个结果选项，用 , 隔开;
  - 与C语言等规则相反,Go语言不需要用 break 来明确退出一个 case ，默认执行完就跳出;
  - 只有在 case 中明确添加 fallthrough 关键字,才会继续执行紧跟的下一个 case ;
  - 可以不设定 switch 之后的条 件表达式 ，在此种情况下,，整个 switch 结构与多个 if...else... 的逻辑作用等同。

7.3 循环语句

1. 循环语句：关键字为 for 和 range 、break 、continue;
   - 与多数语言不同的是，**Go语言中的循环语句只支持 for 关键字，而不支持 while 和 do-while结构**。
   - 格式：跟C/C++/Java一样，只是不需要圆括号()了。

     for 初始化变量; 条件; post { } //等价于：C/C++/Java的for for 条件 { } //等价于：C/C++/Java的while for {} //等价于：C/C++/Java的 for (;;) {} 和 do {} while(1);

     ```
     var sum, multi= 0, 1
     for i, j := 0; i < 10 && j > 0; i, j = i + 1, j - 1 {
         sum += i
         multi *= j
     }
     ```

   - Go语言还进一步考虑到无限循环的场景，让开发者不用写无聊的 for (;;) {} 和 do {} while(1); 而直接简化为如下的写法：

     ```
     sum := 0
     for {
         sum++
         if sum > 100 {
             break
         }
     }
     ```

   - 注意事项：使用循环语句时,需要注意的有以下几点
     - 左花括号 { 必须与 for 处于同一行。
     - Go语言中的 for 循环与C语言一样，都允许在循环条件中定义和初始化变量，唯一的区别是，**Go语言不支持以逗号为间隔的多个赋值语句**，必须使用平行赋值的方式来初始化多个变量。
     - Go语言的 for 循环同样支持 continue 和 break 来控制循环,但是它提供了一个更高级的break ,可以设置标签，跳到那个标签处执行，从而中断循环。跟Java的 continue 和 break 一样。例如：

```
for j := 0; j < 5; j++ {
    for i := 0; i < 10; i++ {
        if i > 5 {
            break JLoop
        }
        fmt.Println(i)
    }
}
JLoop:
// ...
```

7.4 跳转语句

1. 跳转语句：关键字为 goto 。goto 还是会在一些场合下被证明是最合适的。
    - goto 语句的语义非常简单，就是跳转到本函数内的某个标签，如:

```
func myfunc() {
    i := 0

    HERE:
    fmt.Println(i)
    i++
    if i < 10 {
        goto HERE
    }
}
```

8. 错误处理

> 漂亮的错误处理规范是Go语言最大的亮点之一。

8.1 error类型

1. error接口
    - Go语言引入了一个关于错误处理的标准模式，即 error 接口，该接口的定义如下:

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

    - error接口的使用：可以将error接口作为函数的返回值的最后一个，在函数内部判断可能出现的错误，并记录到返回值error中，例如:

```
func Foo(param int) (n int, e error) {
    .....
}
// 调用函数时的代码建议按如下方式处理错误情况：
n, err := Foo(4)
if err
    //错误处理
} else
    //使用返回值 n
}
```

2. 自定义error类型，即自定义异常。
    - 首先，定义一个用于承载错误信息的类型。因为Go接口的灵活性，你根本不需要从error 接口继承或者像Java一样需要使用 implements 来明确指定类型和接口之间的关系。例如：
    - 其次，实现该类型的Error方法，从而让编译器知道 PathError 可以当一个 error 来传递。
    - 例如：

```
type PathError struct {
    Op string
    Path string
    Err error    //使用组合来达到继承
}
func (e *PathError) Error() string {  //实现error接口的方法
    return e.Op + " " + e.Path + ": " + e.Err.Error()
}
```

8.2 defer关键字

1. Go语言引入了defer关键字来确保那些被使用的资源能够释放，例如打开的文件能被关闭。**功能类似与Java的finally**，使用格式：`defer 语句/函数`。例子：

```
func CopyFile(dstName, srcName string) (written int64, err error) {
    src, err := os.Open(srcName)
    if err != nil {
        return
    }
    defer src.Close()

    dst, err := os.Create(dstName)
    if err != nil {
        return
    }
    defer dst.Close()

    defer func() {
        fmt.Println("资源开始关闭")
    }
    return io.Copy(dst, src)
}
```

2. 原理和执行时机
    - Go的defer语句预先设置了一个函数调用（延期的函数），一个作用域中有多个defer则以后进先出（LIFO）的顺行执行，因此该调用在函数执行return语句(出栈)前，立刻运行。功能类似与Java的finally
    - defer的执行时机：函数返回、函数结束或者对应的goroutine发生panic的时候defer就会执行。golang返回时，先把返回值压栈；然后执行defer函数，如果defer函数中有修改栈中的返回值（不过不应该这样做），那么返回值会被修改；之后进行跳转返回。

8.3 错误处理函数

1. panic() 和 recover()函数
    - Go语言引入了两个内置函数 panic() 和 recover() 以报告和处理运行时错误和程序中的错误。
    - 2个函数原型：

```
func panic(interface{})     // 接收任意参数
func recover() interface{}      // 返回任意参数
```

    - `panic() 函数`：用于主动抛出错误。对于不可恢复的错误，Go提供了一个内建的panic函数，它将创建一个运行时错误并使整个程序停止。然后逐层向上执行函数的`defer`语句，然后逐层打印函数调用堆栈，直到被`recover`捕获或运行到最外层函数。
    - `recover()函数`：用于捕获panic抛出的错误。recover用来捕获panic，阻止panic继续向上传递。多个panic只会捕捉最后一个。recover()必须和defer一起使用。
2. 原理：当panic被调用时或者程序产生运行时错误，由运行时检测并退出。程序会从调用panic的函数位置或发生panic的地方立即返回，并开始逐级解开函数堆栈，同时运行所有被defer的函数。如果这种解开达到堆栈的顶端，程序就死亡了。但是，也可以使用recover函数阻止panic继续向上传递，recover函数会捕获该panic，会通知解开堆栈并返回传递给panic的参量，重新获得Go程的控制权并恢复正常的执行。由于在解开期间仅运行defer的函数之内的代码，**recover仅在被延期的函数内部才是有用的**。
3. 例子：

```
func g(i int) {
    if i>1 {
        fmt.Println("Panic!")
        panic(fmt.Sprintf("%v", i))
    }

}
func f() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("Recovered in f", r) //r = panic的参数
        }
    }()
    for i := 0; i < 4; i++ {
        fmt.Println("Calling g with ", i)
        g(i)
        fmt.Println("Returned normally from g.")
     }
}
```

```
func main() {
    f()
    fmt.Println("Returned normally from f.")
}
// 运行结果如下:
Calling g with  0
Returned normally from g.
Calling g with  1
Returned normally from g.
Calling g with  2
Panic!
Recovered in f 2
Returned normally from f.
```

9. 数组与切片

9.1 数组

1. 数组的声明及初始化：[长度] 类型【 = [长度] 类型{...}】，例如：

```
var arr1 [32]byte
var arr2 [10] *float64   //指针数组
var arr [3][4]int       //二维数组，3行，4列。等同于: var arr [4]([3]int)
var [2*4] struct {x,y int32}  //复杂类型数组

var arr [10]int = [10]int{1,2,3,4,5,6,7,8,9,10}
array := [5]int{1,2,3,4,5}  // 定义并初始化一个数组
```

2. 数组的长度是该数组类型的一个内置常量,可以用Go语言的内置函数 len() 来获取。例如：len(arr2)
3. 数组元素的访问及遍历:
    - 使用数组下标来访问数组中的元素。

    ```
    array := [5]int{1,2,3,4,5}  // 定义并初始化一个数组
     for i := 0; i<len(array); i++ {
        fmt.Println("arry[" , i, "]=", array[i])
     }
    ```

    - Go语言提供了一个关键字 range ，用于便捷地遍历容器中的元素。range 具有两个返回值，第一个返回值是元素的数组下标，第二个返回值是元素的值。

    ```
    for i, v := range array {
        fmt.Println("Array element[", i, "]=", v)
    }
    ```

4. 值类型：**在Go语言中数组是一个值类型(value type)。所有的值类型变量在赋值和作为参数传递时都将产生一次复制动作**。如果将数组作为函数的参数类型,则在函数调用时该参数将发生数据复制。若要在函数内操作外部的数据结构，需要使用到切片功能。

9.2 切片

> Go的切片类似与python的切片，当可动态增减。类似C++的vector。切片是对数组的抽象。

1. 数组切片：初看起来，数组切片就像一个指向数组的指针，实际上它拥有自己的数据结构，而不仅仅是个指针。数组切片的数据结构可以抽象为以下3个变量：
   - 一个指向原生数组的指针;
   - 数组切片中的元素个数;
   - 数组切片已分配的存储空间。

   从底层实现的角度来看,数组切片实际上仍然使用数组来管理元素，因此它们之间的关系让C++程序员们很容易联想起STL中 std::vector 和数组的关系。**基于数组，数组切片添加了一系列管理功能，可以随时动态扩充存放空间，并且可以被随意传递而不会导致所管理的元素被重复复制**。
2. 创建数组切片：创建数组切片的方法主要有三种——基于数组，直接创建，基于数组切片创建数组切片，下面我们来简要介绍一下这三种方法。
   - 基于数组创建切片：

     > 数组切片可以基于一个已存在的数组创建。数组切片可以只使用数组的一部分元素或者整个数组来创建,甚至可以创建一个比所基于的数组还要大的数组切片。

     格式：`array[first:last]`，其中 first 和 last 可以任选，指的是索引下标。缺的默认为0/len(slice)，包含first，不包含last，即只取last-first个。例如：

     ```
     package main
     import "fmt"

     func main() {
         var arr [10]int = [10]int{1,2,3,4,5,6,7,8,9,10}
         //基于数组的切片
         var mySlice []int = arr[:5]

         fmt.Println("Print slice of value:")
         for _, v := range mySlice {
             fmt.Println(v, " ")
         }

         fmt.Println("Print array of value:")
         for _, v := range arr {
             fmt.Println(v, " ")
         }
     }
     运行结果为:
     Print slice of value:
     1 2 3 4 5 6 7 8 9 10
     Print array of value:
     1 2 3 4 5
     ```

   - 直接创建创建切片：并非一定要事先准备一个数组才能创建数组切片。Go语言提供的内置函数 make() 可以用于灵活地创建数组切片。

     创建一个初始元素个数为5的数组切片,元素初始值为0：

     ```
     mySlice := make([]int, 5)
     ```

创建一个初始元素个数为5的数组切片,元素初始值为0,并预留10个元素的存储空间:

```
mySlice := make([]int, 5, 10)  //预留10个元素的存储空间
```

直接创建并初始化包含5个元素的数组切片:

```
mySlice3 := []int{1, 2, 3, 4, 5}  //事实上会有一个匿名数组被创建出来
```

- 基于数组切片创建数组切片
  - 类似于数组切片可以基于一个数组创建，数组切片也可以基于另一个数组切片创建。例如:

    ```
    slice1 := []int{1,2,3,4}
    newslice := slice1[:3]  //复制0-2的元素
    ```

  - 基于数组切片创建的数组切片选择 oldSlicef元素的范围可以超过所包含的元素个数。即：上面的可以是：newslice := slice1[:6] ，只要这个选择的范围不超过 oldSlice 存储能力(即 cap() 返回的值),那么这个创建程序就是合法的。 newSlice 中超出oldSlice 元素的部分都会填上0。

3. 切片的元素遍历
   - 操作数组元素的所有方法都适用于数组切片，比如数组切片也可以按下标读写元素，用 len()函数获取元素个数，并支持使用 range 关键字来快速遍历所有元素。
4. 切片动态增减元素

> 数组切片会自动处理存储空间不足的问题。如果追加的内容长度超过当前已分配的存储空间 (即 cap() 调用返回的信息)，数组切片会自动分配一块足够大的内存。这个足够大，不确定，因此不保证只进行一次重新分配内存和搬送内存块。

- 可**动态增减元素是数组切片比数组更为强大的功能**。与数组相比，**数组切片多了一个存储能力的概念，即元素个数和分配的空间(capacity)可以是两个不同的值**。合理地设置存储能力的值，可以大幅降低数组切片内部重新分配内存和搬送内存块的频率，从而大大提高程序性能。
- cap(slice) 函数和 len(slice) 函数：数组切片支持Go语言内置的 cap(slice) 函数和 len(slice) 函数，cap() 函数返回的是数组切片分配的空间大小,而 len() 函数返回的是数组切片中当前所存储的元素个数。
- append(slice, T ...ele) 函数：函数 append() 的第二个参数其实是一个不定参数，我们可以按自己需求添加若干个元素，甚至直接将一个数组切片追加到另一个数组切片的末尾。例如:

  ```
  slice1 := []int {8,9,10}
  slice2 := append(slice2, 5,6,7)
  slice2 := append(slice2, slice1...)  //第二个参数 slice1 后面加了三个点,即一个省略号,如果没有这个省略号的话,会有编译错误.
  ```

  PS：在第二个参数是切片时，必须在后面加了三个点，即一个省略号，如果没有这个省略号的话，会有编译错误，因为按 append() 的语义，从第二个参数起的所有参数都是待附加的元素。因为 slice 中的元素类型为 int ,所以直接传递 mySlice2 是行不通的。加上省略号相当于把 slice 包含的所有元素打散后再传入。

5. 切片的内容复制
   - copy(dest, src)函数：数组切片支持Go语言的另一个内置函数 copy() ，用于将内容从一个数组切片复制到另一个数组切片。复制时，只按dest的大小来覆盖。例如:

```
slice1 := []int{1, 2, 3, 4, 5}
slice2 := []int{5, 4, 3}
copy(slice1, slice2)  //将slice2赋值到slice1,因为slice2只有三个，故覆盖slice1的前三个
copy(slice2, slice1)  //slice2大小为3，故只复制slice1的前三个。
```

10. 指针, map

10.1 指针

1. 什么是指针：我们都知道，变量是一种使用方便的占位符，用于引用计算机内存地址。那有没有直接存储该地址的方式呢？那就是指针了，指针是存放某变量的内存地址的变量。
2. **Go的指针不能进行偏移和运算，即不能进行自增等运算**，因此无法用一个指针指向数组某个元素后，再越界访问。所以把Go的指针当做一个跟基本类型一样的变量即可。
3. Go的普通数组的名字不代表数组的首地址，因此用指针指向数组元素时，需要明确指向第一个数组元素，即：`p := &arr[0]`
4. 声明格式：`var 变量名 *变量类型`，例如：`var p * int`
5. 赋值：使用取地址符`&`，格式为`指针变量名 = &变量名`，例如：`p = &i`
6. 访问指针变量中指向地址的值：`*指针变量`，例如：`i := *p`
7. Go 空指针：当一个指针被定义后没有分配到任何变量时，它的值为`nil`，nil在概念上和其它语言的null、None、nil、NULL一样，都指代零值或空值。
8. 指针数组：每个元素都是指针
    - 声明格式：`var 变量名 [size]*变量类型`。
    - 赋值：**要让指针数组指向一个数组，需要，一个个的对应赋值**。因为指针数组声明后初始化为nil，且Go的指针不支持偏移和运算。
9. 数组指针(也称行指针)：看变量类型跟`数组[]`近，就是数组指针了。是一个指针，指向数组类型。
    - 声明格式：`var 变量名 *[size]变量类型`
    - 赋值：一般可以将数组赋值给数组指针，即：`ptr = &arr`
10. 例子：

```
func main() {
    a := [...]int{1, 2, 3, 4, 5}
    var p *[5]int = &a  // 数组指针
    fmt.Println(*p, a)
    for index, value := range *p {
        fmt.Println(index, value)
    }

    var p2 [5]*int  // 指针数组
    i, j := 10, 20
    p2[0] = &i
    p2[1] = &j
    for index, value := range p2 {
        if value != nil {
            fmt.Println(index, *value)
        } else {
            fmt.Println(index, value)
```

```
        }
    }
}
```

10.2 map

1. map变量声明：map的类型是：map[键类型] 值类型，因此声明格式为：

```
var 名称 map[[键类型]值类型
var myMap map[string] PersonInfo
```

2. map创建：可以使用Go语言内置的函数 make() 来创建一个新 map。也可以选择是否在创建时指定该 map 的初始存储能力。例如：

```
var myMap map[string] PersonInfo
myMap = make(map[string] PersonInfo)
myMap = make(map[string] PersonInfo, 10)
//创建并初始化
myMap = map[string] PersonInfo{
    "1234": PersonInfo{"12345", "Jack", "Room 101"},
}
```

3. map元素的赋值，删除，查找
   - 赋值：`名称[键] = 值`。如：myMap["1234"] = PersonInfo{"1", "Jack", "Room 101"} 。如果该键不存在就新增，存在则修改。
   - 删除：Go语言提供了一个内置函数 delete() ,用于删除容器内的元素。例如：

     ```
     delete(myMap, "1234")
     ```

     PS：上面的代码将从 myMap 中删除键为“1234”的键值对。如果“1234”这个键不存在，那么这个调用将什么都不发生，也不会有什么副作用。
   - 查找：在Go语言中, map 的查找功能设计得比较精巧。格式：

     ```
     value, ok := myMap["1234"]    //第一个变量是键对应的值，ok变量是是否找到，找到为true
     if ok { // 找到了
     // 处理找到的value
     }
     ```

11. 方法, 结构体与接口

11.1 方法

1. 在Go语言中，函数和方法(method)是不一样的。两者的不同之处：
   - 函数是全局的，不依赖于谁，可以单独存在。方法，也叫成员方法，归属与某个结构体或接口。
   - 声明形式不一样，调用方式也不太一样。
   - **函数是基本的代码块，用于执行一个任务。可以通过函数来划分不同功能，逻辑上每个函数执行的是指定的任务。**

- **方法就是一个包含了接受者的函数，接受者可以是命名类型或者结构体类型的一个值或者是一个指针。所有给定类型的方法属于该类型的方法集**。这个接收者实际上相当于Java的this指针，只不过这里可以给this任意命名。

2. 方法的声明

```
func (接收者变量名 接收者类型) 方法名([形式参数]) [(方法返回列表)]{
    /* 通过接收者变量可以访问该对象的成员变量和其他方法*/
   /* 方法体*/
}
```

3. 方法的调用，格式：`对象.方法名([实参])`

```
package main
import "fmt"

/* 定义结构体 */
type Circle struct {
  radius float64
}
//接收者是Circle类型变量，所以该 method 属于 Circle 类型对象中的方法
func (c Circle) getArea() float64 {
  //c.radius 即为 Circle 类型对象中的属性
  return 3.14 * c.radius * c.radius
}

func main() {
  var c1 Circle
  c1.radius = 10.00
  area := c1.getArea()  // 调用方法，c1就是接收者

  fmt.Println("圆的面积 = ", area)
}
```

11.2 结构体(类)

> *Go语言的结构体(struct)和其他语言的类(class)有同等的地位，但Go语言放弃了包括继承在内的大量面向对象特性，只保留了组合(composition)这个最基础的特性。*所有的Go语言类型(指针类型除外)都可以有自己的方法。
> 在这个背景下，Go语言的结构体只是很普通的复合类型，平淡无奇(古天乐)。

1. 结构体的定义及使用，添加方法
   - 定义格式：

```
type 结构体名 struct {    // 一般在里面只定义变量和其他结构体类型
    其他结构体
    ...
    成员变量名 类型
    ...
    成员变量名 类型
}
```

- 使用：Go语言中结构体的使用方式与C语言并没有明显不同，都是：`变量名.成员`，`变量名.方法`。
- 例子：

```
type Rect struct {
    x, y float64
    width, height float64
}
// 然后定义成员方法 Area() 来计算矩形的面积
func (r *Rect) Area() float64 {
    return r.width * r.height
}
```

2. 结构体的创建并初始化
   - 方式一：`new{ 结构体名 }`，如：`rect := new{Rect}`。
   - 方式二：`&结构体名{}`，如：`rect := &Rect{}`。
   - 创建时初始化：在Go语言中，未进行显式初始化的变量都会被初始化为该类型的零值。
     - 按声明顺序初始化：`rect := &Rect{0, 0, 3, 4}`
     - 指定名称初始化：`rect := &Rect{width:3, height:4}`
   - 在Go语言中没有构造函数的概念，对象的创建通常交由一个全局的创建函数来完成，以NewXXX 来命名，表示 “构造函数”，有点类似与工厂模式。例如：

```
func NewRect(x, y, width, height float64) *Rect {
    return &Rect{x, y, width, height}  //返回对象的地址
}
```

3. 匿名组合(继承，组合文法)
   - 确切地说，Go语言也提供了继承，，但是采用了组合的文法，所以我们将其称为匿名组合。在子类中组合其他基类时，**匿名组合类型相当于以其类型名称(去掉包名部分)作为成员变量的名字**。
   - 例子：

```
type Base struct {  // 定义一个基类 Base
    Name string
}
func (base *Base) Foo() { ... }  // 给Base类添加Foo方法
func (base *Base) Bar() { ... }  // 给Base类添加Bar方法
type Foo struct {   // 定义一个子类 Foo
    Base    // 组合基类，不用给名称，匿名组合类型以其类型名称作为成员变量的名字
    ...
}
func (foo *Foo) Bar() { // 改写基类 Base 的Bar方法，相当于重写(Override)
    foo.Base.Bar()  // 调用父类的方法
    ...
}
// 调用
f := new{Foo}
f.Bar() // 输出子类自己的方法，因为重写了
f.Foo() // 还是基类的效果，因为没重写。
```

以上代码定义了一个 Base 类 (实现了 Foo() 和 Bar() 两个成员方法)，然后定义了一个Foo 类，该类从 Base 类 “继承” 并改写了 Bar() 方法 (该方法实现时先调用了基类的 Bar()方法)。在“派生类” Foo 没有改写“基类” Base 的成员方法时，相应的方法就被“继承”，例如在上面的例子中，调用 foo.Foo() 和调用 foo.Base.Foo() 效果一致。

- 例子2：在Go语言中，你还可以以指针方式从一个类型“派生” 。类似C++ 语言中的虚基类。但更简单。下面的Go代码仍然有“派生”的效果，**只是 Foo 创建实例的时候，需要外部提供一个 Base 类实例的指针**。

```
type Foo struct {
    *Base   // 指针类型的匿名组合
    ...
}
```

- 接口组合中的名字冲突问题：组合的类型和被组合的类型，如果出现同名，被组合的类型的会被隐藏。匿名组合类型相当于以其类型名称(去掉包名部分)作为成员变量的名字。因此，如果不能再出现跟被组合类型的类型名称相同的成员。

4. 可见性
    - Go语言对关键字的增加非常吝啬，其中没有 private 、 protected 、 public 这样的关键字。要使某个符号对其他包 (package) 可见 (即可以访问) ，需要将该符号定义为以大写字母开头。Go语言中符号的可访问性是包一级的而不是类型一级的，即包内没有访问控制。即大写字母开头的符号类似与Java的public，小写字符开头类似与Java的默认。

11.3 接口

1. 其他语言的接口
    - 其他语言的接口主要作为不同组件之间的契约存在。对契约的实现是强制的，必须明确声明你实现了该接口。为了实现一个接口，你需要从该接口继承，并实现接口的方法。即使另外有一个接口IFoo2与 IFoo 完全一样，甚至名字也叫 IFoo 只不过位于不同的名字空间下，编译器也会认为下面的类 Foo 只实现了 IFoo 而没有实现 IFoo2 接口。例如：

```
interface IFoo {
    void Bar();
}
class Foo implements IFoo {      // 必须明确声明实现的接口，使用继承来声明。
    void Bar() {
        ...
    }
}
```

    - 这类语言的接口称为侵入式接口。“侵入式”的主要表现在于实现类需要明确声明自己实现了某个接口。这时就要思考2个问题了：
        - 问题1：我提供哪些接口好呢？例如接口要拆的多细，放那些方法。因为一旦实现了某个接口，修改该接口是灾难性的。
        - 问题2：如果两个类实现了相同的接口，应该把接口放到哪个包好呢？接口如何划分使得类库的继承树图更好。
2. Go语言的接口概念如何避免这几个困扰了无数开发人员的传统难题呢？
    - Go 语言提供了另外一种数据类型即接口，它把所有的具有共性的方法定义在一起，任何其他类型只要实现了这些方法就是实现了这个接口。**接口类型是由一组方法定义的集合**。

- 在Go语言中，一个类(即结构体)只需要实现了接口要求的所有函数，我们就说这个类实现了该接口，就可以使用多态功能了。不需要明确指定实现了那个接口，即不需要通过继承来实现接口。
- Go语言的非侵入式接口，看似只是做了很小的文法调整，实则影响深远。好处如下：
  - Go语言的标准库，再也不需要绘制类库的继承树图。在Go中，类的继承树并无意义，你只需要知道这个类实现了哪些方法，每个方法是啥含义。
  - 实现类的时候，只需要关心自己应该提供哪些方法，不用再纠结接口需要拆得多细才合理。接口由使用方按需定义，而不用事前规划。
  - 不用为了实现一个接口而导入一个包,因为多引用一个外部的包,就意味着更多的耦合。接口由使用方按自身需求来定义,使用方无需关心是否有其他模块定义过类似的接口。

3. Go语言的接口的定义，实现：
   - 定义格式：

```
type 名称 interface {
    方法名([形式参数]) [(return_type)]
}
// 例如
type Phone interface {
    call()
    shut() error
}
```

   - 实现接口：在Go中，一个类(结构体)只需要实现了接口定义的所有函数，我们就说这个类实现了该接口

```
/* 定义接口 */
type 接口名 interface {
   方法名 [return_type]
   ...
   方法名 [return_type]
}

/* 定义结构体 */
type struct_name struct {
   /* variables */
}

/* 实现接口方法 (变量名 struct_name) 相当于Java的this*/
func (变量名 struct_name) 方法名([形参]) [( [return_type] )] {
   /* 方法实现 */
}
...
// 如果需要修改调用者，就需要使用 指针
func (变量名 * struct_name) 方法名([形参]) [return_type] {
   /* 方法实现*/
}

/*多态*/
var 变量名 接口名 = new{结构体名}
变量名.方法名(..)
```

   - 例子：

```
// 这里我们定义了一个 File 类,并实现有 Read() 、Write() 、Seek() 、Close() 等方法。
type File struct {
    // ...
}
func (f *File) Read(buf []byte) (n int, err error)
func (f *File) Write(buf []byte) (n int, err error)
func (f *File) Seek(off int64, whence int) (pos int64, err error)
func (f *File) Close() error

//设想我们有如下接口:
type IFile interface {
    Read(buf []byte) (n int, err error)
    Write(buf []byte) (n int, err error)
    Seek(off int64, whence int) (pos int64, err error)
    Close() error
}
type IReader interface {
    Read(buf []byte) (n int, err error)
}
type IWriter interface {
    Write(buf []byte) (n int, err error)
}
type ICloser interface {
    Close() error
}

// 尽管File类没有继承这些接口,甚至可以不知道这些接口的存在
// 但是 File 类还是实现了这些接口，因为File实现了他们的方法，因此可以进行赋值，拥有多态
var file1 IFile = new{File}
file1.Read(..)  // 多态，执行的是File的效果
var file2 IReader = new{File}
var file3 IWriter = new{File}
var file4 ICloser = new{File}
```

4. 接口赋值(多态)
   - 接口赋值在Go语言中分为如下两种情况:
     - 将对象实例赋值给接口，格式: `var 变量名 = &对象实例名` ，或`var 变量名 = 对象实例名`
     - 将一个接口赋值给另一个接口，赋值格式: `var 变量名 = 另一个接口变量`
   - 将对象实例赋值给接口：要求该对象实例实现了接口要求的所有方法。最好在赋值时，使用&+对象实例,为什么呢？例子:

```
type Integer int    // 定义一个Intger类
func (a Integer) Less(b Integer) bool { // 给Intger类添加Less方法
    return a < b
}
func (a *Integer) Add(b Integer) {  // 如果需要修改调用者，就需要使用 指针
    *a += b
}

// 相应地,我们定义接口 LessAdder
type LessAdder interface {
```

```
    Less(b Integer) bool
    Add(b Integer)
}
// 现在有个问题:假设我们定义一个 Integer 类型的对象实例,怎么将其赋值给 LessAdder接口呢?
var a Integer = 1
var b LessAdder = &a      (1)     // 正确
var b LessAdder = a       (2)     // 编译会出错

// 为什么(2)会编译出错，原因在于,Go语言可以根据下面的函数：
func (a Integer) Less(b Integer) bool    自动生成一个新的 Less() 方法：
func (a *Integer) Less(b Integer) bool {
    return (*a).Less(b)
}
这样,类型 *Integer 就既存在 Less() 方法,也存在 Add() 方法,满足 LessAdder 接口。
从另一方面来说,根据     func (a *Integer) Add(b Integer) 函数，无法自动生成以下这个成员方法：
func (a Integer) Add(b Integer) {
    (&a).Add(b)
}
因为 (&a).Add() 改变的只是函数参数 a ,对外部实际要操作的对象并无影响,这不符合用户的预期。所以,Go语言不会自动为其生成该函数。因此,类型 Integer 只存在 Less() 方法,缺少 Add() 方法,不满足 LessAdder 接口,故此上面的语句(2)不能赋值。
```

- 将一个接口赋值给另一个接口：只要两个接口拥有相同的方法列表(次序不同不要紧)，那么它们就是等同的，可以相互赋值。当然，接口赋值并不要求两个接口必须等价。如果接口A的方法列表是接口B的方法列表的子集，那么接口B可以赋值给接口A，但反过来不行。例如：

```
type Writer interface {
    Write(buf []byte) (n int, err error)
}
type IStream interface {
    Write(buf []byte) (n int, err error)
    Read(buf []byte) (n int, err error)
}
var f1 IStream = new{File}  //File是前面的例子，它实现了Write,Read方法的结构体(类)
//接口的赋值，Write是IStream的子集，所以可以接收IStream，但f1=f2会出错。
var f2 Writer = f1
```

5. 接口查询：类似于Java的 instanceof ，判断接口指向的对象实例是不是实现了这个接口。有没有将Writer 接口转换为 IStream 接口的方法呢？这就需要使用接口查询了。
   - 例如：下面例子中， if 语句检查 file1 接口指向的对象实例是否实现了 IStream 接口，如果实现了，则执行特定的代码。

```
var file1 Writer = ...
if file5, ok := file1.(IStream); ok {
    ...
}
```

   - 接口查询是否成功，要在运行期才能够确定。它不像接口赋值，编译器只需要通过静态类型检查即可判断赋值是否可行。

- 查询接口所指向的对象是否为某个类型的这种用法可以认为只是接口查询的一个特例。接口是对一组类型的公共特性的抽象，所以查询接口与查询具体类型的区别好比是下面这两句问话的区别，第一句问话查的是一个群体，是查询接口；而第二句问话已经到了具体的个体，是查询具体类型。

  > 1. 你是医生吗? 是。
  > 2. 你是某某某? 是。

6. 类型查询

- 在Go语言中，还可以更加直截了当地询问接口指向的对象实例的类型，例如：

```
var v1 interface{} = ...
switch v := v1.(type) {
    case int:
    // 现在v的类型是int
    case string: // 现在v的类型是string
    ...
}
```

- 就像现实生活中物种多得数不清一样, 语言中的类型也多得数不清, 所以类型查询并不经常使用。它更多是个补充,需要配合接口查询使用，例如:

```
type Stringer interface {
String() string
}
func Println(args ...interface{}) {
    for _, arg := range args {
        switch v := v1.(type) {
            case int:       // 现在v的类型是int
            case string:        // 现在v的类型是string
            default:
                if v, ok := arg.(Stringer); ok { // 现在v的类型是Stringer
                val := v.String()
                    // ...
                } else {
                    // ...
                }
        }
    }
}
```

7. 接口组合

- 像之前介绍的类型组合一样, Go语言同样支持接口组合。可以认为接口组合是类型匿名组合的一个特定场景，只不过接口只包含方法,而不包含任何成员变量。例如：

```
// ReadWriter接口将基本的Read和Write方法组合起来
type ReadWriter interface {
    Reader  // 直接就使用接口类型了，不用给它命名，故是匿名接口
    Writer
}

// 上面的等同于
type ReadWriter interface {
    Read(p []byte) (n int, err error)
    Write(p []byte) (n int, err error)
}
```

8. Any类型
    - 由于Go语言中任何对象实例都满足空接口 interface{} ，所以 interface{} 看起来像是可以指向任何对象的 Any 类型。类似于Java的Object，C的void * 。例如：

```
var v1 interface{} = 1  // 将int类型赋值给interface{}
var v2 interface{} = "abc"  // 将string类型赋值给interface{}
var v3 interface{} = &v2        // 将*interface{}类型赋值给interface{}
var v4 interface{} = struct{ X int }{1}
var v5 interface{} = &struct{ X int }{1}
```

1.4 接口的实现机制

类型赋值接口

1. 一个例子：

```
package main
import "fmt"

type ISpeaker interface {
    Speak()
}

type SimpleSpeaker struct {
    Message string
}

func (speaker *SimpleSpeaker) Speak() {
    fmt.Println("I am speaking? ", speaker.Message)
}

func main() {
    var speaker ISpeaker
    speaker = &SimpleSpeaker{"Hell"}    // 一个类型实例赋值接口
    speaker.Speak()
}
```

2. 一个核心的问题：从机器的角度如何判断一个 SimpleSpeaker 类型实现了 ISpeaker 接口的所有方法?

- 一个简单的逻辑就是需要获取这个类型的所有方法集合(集合A)，并获取该接口包含的所有方法集合(集合B)，然后判断列表B是否为列表A的子集。是则意味着 SimpleSpeaker类型实现了 ISpeaker 接口。而这些工作Go语言可以在编译期获取足够多的信息来进行判断，并进行代码的优化。

3. 判断类型赋值接口的示例：

- Go语言的一个类型赋值接口例子：

```
package main
import "fmt"
type IReadWriter interface {
    Read(buf *byte, cb int) int
    Write(buf *byte, cb int) int
}
type A struct {
    a int
}
func NewA(params int) *A {
    fmt.Println("NewA:", params);
    return &A{params}
}
func (this *A) Read(buf *byte, cb int) int {
    fmt.Println("A_Read:", this.a)
    return cb
}
func (this *A) Write(buf *byte, cb int) int {
    fmt.Println("A_Write:", this.a)
    return cb
}
type B struct {
    A
}
func NewB(params int) *B {
    fmt.Println("NewB:", params);
    return &B{A{params}}
}
func (this *B) Write(buf *byte, cb int) int {
    fmt.Println("B_Write:", this.a)
    return cb
}
func (this *B) Foo() {
    fmt.Println("B_Foo:", this.a)
}
func main() {
    var p IReadWriter = NewB(8)
    p.Read(nil, 10)
    p.Write(nil, 10)
}
```

- 对应的C语言版本的实现代码：

```
#include <stdio.h>
#include <stdlib.h>
// ----------------------------------------------------------------
typedef struct _TypeInfo {
    // 用于运行时取得类型信息, 比如反射机制
} TypeInfo;
typedef struct _InterfaceInfo {
    // 用于运行时取得接口信息
} InterfaceInfo;
// ----------------------------------------------------------------
typedef struct _IReadWriterTbl {
    InterfaceInfo* inter;
    TypeInfo* type;
    int (*Read)(void* this, char* buf, int cb);
    int (*Write)(void* this, char* buf, int cb);
} IReadWriterTbl;
typedef struct _IReadWriter {
    IReadWriterTbl* tab;
    void* data;
} IReadWriter;
InterfaceInfo g_InterfaceInfo_IReadWriter = {
    // ...
};
// ----------------------------------------------------------------
typedef struct _A {
    int a;
} A;
int A_Read(A* this, char* buf, int cb) {
    printf("A_Read: %d\n", this->a);
    return cb;
}
int A_Write(A* this, char* buf, int cb) {
    printf("A_Write: %d\n", this->a);
    return cb;
}
TypeInfo g_TypeInfo_A = {
    // ...
};

A* NewA(int params) {
    printf("NewA: %d\n", params);
    A* this = (A*)malloc(sizeof(A));
    this->a = params;
    return this;
}
// ----------------------------------------------------------------
typedef struct _B {
    A base;
} B;
int B_Write(B* this, char* buf, int cb) {
    printf("B_Write: %d\n", this->base.a);
    return cb;
}
```

```
void B_Foo(B* this) {
    printf("B_Foo: %d\n", this->base.a);
}
TypeInfo g_TypeInfo_B = {
    // ...
};
B* NewB(int params) {
    printf("NewB: %d\n", params);
    B* this = (B*)malloc(sizeof(B));
    this->base.a = params;
    return this;
}
// ------------------------------------------------------------
IReadWriterTbl g_Itbl_IReadWriter_B = {
    &g_InterfaceInfo_IReadWriter,
    &g_TypeInfo_B,
    (int (*)(void* this, char* buf, int cb))A_Read,
    (int (*)(void* this, char* buf, int cb))B_Write
};
int main() {
    B* unnamed = NewB(8);
    IReadWriter p = {
        &g_Itbl_IReadWriter_B,
        unnamed
    };
    p.tab->Read(p.data, NULL, 10);
    p.tab->Write(p.data, NULL, 10);
    return 0;
}
// ------------------------------------------------------------
```

接口查询

1. 接口查询是一个在软件开发中非常常见的使用场景，比如一个拿着 IReader 接口的开发者，在某些时候会需要知道 IReader 所对应的类型是否也实现了 IReadWriter 接口，这样它可以切换到 IReadWriter 接口,然后调用该接口的 Write() 方法写入数据。在Go语言的使用中,这个过程非常简单:

```
var reader IReader = NewReader()
if writer, ok := reader.(IReadWriter); ok {
    writer.Write()
}
```

2. 那么到底接口查询是如何被支持的呢?
    - 按Go语言的定义，接口查询其实是在做接口方法查询，只要该类型实现了某个接口的所有方法，就可以认为该类型实现了此接口。相比类型赋值给接口时可以做的编译期优化，运行期接口查询就只能老老实实地做一次接口匹配了。并且整个过程其实跟发起查询的那个源接口毫无关系，真正的查询是针对源接口所指向的具体类型以及目标接口。

接口赋值

1. 与接口查询相比，其实还有另外一种简单一些的场景，叫接口赋值，就是将一个接口直接赋值给另外一个接口。比如:

```
var rw IReadWriter = ...
var r IReader = rw
```

   这种赋值是否可以通过编译的判断依据是：源接口和目标接口是否存在方法集合的包含关系。

2. 接口赋值初看起来和我们描述的接口查询过程有些像，但因为接口赋值过程在编译期就可以确定，所以没必要动用消耗比较大的动态接口查询流程。我们可以认为接口赋值是接口查询的一种优化。在编译期,编译器就能判断是否可进行接口转换。

1.5 make、new操作

1. make 用于内建类型(map 、 slice 和 channel)的内存分配。 new 用于各种类型的内存分配。
2. new： 内建函数 new 本质上说跟其它语言中的同名函数功能一样：new(T) 分配了零值填充的 T 类型的内存空间，并且返回其地址，即一个 *T 类型的值。用Go的术语说，它返回了一个指针，指向新分配的类型 T 的零值。有一点非常重要：**new 返回指针**。
3. make： 内建函数 make(T, args) 与 new(T) 有着不同的功能，make只能创建 slice 、map 和 channel ，并且返回一个有初始值(非零)的 T 类型，而不是 *T 。

Go的进阶

1. 并发编程

1.1 并发的基础知识

1. 互联网时代的应用一个显著的特征就是高并发。现今，并发包含以下几种主流的实现模型：
   - 多进程：多进程是在操作系统层面进行并发的基本模式。同时也是开销最大的模式。比如某个Web服务器，它会有专门的进程负责网络端口的监听和链接管理，还会有专门的进程负责事务和运算。这种方法的好处在于简单、进程间互不影响，坏处在于系统开销大，因为所有的进程都是由内核管理的。
   - 多线程：多线程在大部分操作系统上都属于系统层面的并发模式，也是我们使用最多的最有效的一种模式。目前，我们所见的几乎所有工具链都会使用这种模式。它比多进程的开销小很多，但是其开销依旧比较大，且在高并发模式下，效率会有影响。在很多高并发服务器开发实践中，使用多线程模式会很快耗尽服务器的内存和CPU资源。
   - 基于回调的非阻塞/异步IO：这种架构的诞生实际上来源于多线程模式的危机。在很多高并发服务器开发实践中，使用多线程模式会很快耗尽服务器的内存和CPU资源。而这种模式通过事件驱动的方式使用异步IO，使服务器持续运转，且尽可能地少用线程，降低开销，它目前在Node.js中得到了很好的实践。但是使用这种模式，编程比多线程要复杂，因为它把流程做了分割，对于问题本身的反应不够自然。
   - 协程：协程(Coroutine)本质上是一种用户态线程，不需要操作系统来进行抢占式调度，且在真正的实现中寄存于线程中，因此，系统开销极小，可以有效提高线程的任务并发性，而避免多线程的缺点。使用协程的优点是编程简单，结构清晰；缺点是需要语言的支持，如果不支持，则需要用户在程序中自行实现调度器。目前，原生支持协程的语言还很少。
2. 传统并发模型的缺陷：即多进程，多线程，异步IO的不足
   - 多进程，多线程，异步IO，这些“线程类型”的并发模式在原先的确定性中引入了不确定性，这种不确定性给程序的行为带来了意外和危害，也让程序变得不可控。
   - 线程之间通信只能采用共享内存的方式。为了保证共享内存的有效性，需要采取了很多措施，比如加锁等，来避免死锁或资源竞争。实践证明，我们很难面面俱到，往往会在工程中遇到各种奇怪的故障和问题。

- 多进程，多线程，异步IO，这些“线程类型”的并发模式都是“重量级”，创建，切换，通信都极其消耗资源，切换时都需要进行内核态和用户态之间的切换。

3. 并发编程的难度在于协调，而协调就要通过交流。从这个角度看来，并发单元间的通信是最大的问题。
4. 在工程上,有两种最常见的并发通信模型：共享数据和消息
   - 共享数据：共享数据是指多个并发单元分别保持对同一个数据的引用，实现对该数据的共享。被共享的数据可能有多种形式，比如内存数据块、磁盘文件、网络数据等。在实际工程应用中最常见的无疑是内存了，也就是常说的共享内存。线程之间通信只能采用共享内存的方式。进程，协程两中都可以使用。共享内存，最大的一个问题是，在访问共享资源时，要进行加锁，操作完后再解锁。如果共享资源太多，那同步会是一个很复杂的问题。
   - 消息：消息机制认为每个并发单元是自包含的、独立的个体，并且都有自己的变量，但在不同并发单元间这些变量不共享。**每个并发单元的输入和输出只有一种，那就是消息**。这有点类似于进程的概念,每个进程不会被其他进程打扰，它只做好自己的工作就可以了。不同进程间靠消息来通信，它们不会共享内存。缺点相较于共享内存系统，速度是有点慢。但线程中的状态管理工作通常会更为简单。
5. 协程
   - 执行体：执行体是个抽象的概念，在操作系统层面有多个概念与之对应，比如操作系统自己掌管的进程(process)、进程内的线程 (thread) 以及进程内的协程 (coroutine,也叫轻量级线程) 。
   - 协程的优势：
     - 协程(Coroutine)本质上是一种用户态线程，不需要操作系统来进行抢占式调度，且在真正的实现中寄存于线程中，因此，系统开销极小，可以有效提高线程的任务并发性，而避免多线程的缺点。
     - 与传统的系统级线程和进程相比,协程的最大优势在于其“轻量级”,可以轻松创建上百万个而不会导致系统资源衰竭,而线程和进程通常最多也不能超过1万个。这也是协程也叫轻量级线程的原因。
     - 使用协程的优点是编程简单，结构清晰。
   - 对协程的支持：
     - 多数语言在语法层面并不直接支持协程，而是通过库的方式支持。但用库的方式支持的功能也并不完整，比如仅仅提供轻量级线程的创建、销毁与切换等能力。如果在这样的轻量级线程中调用一个同步IO 操作，比如网络通信、本地文件读写，都会阻塞其他的并发执行轻量级线程，从而无法真正达到轻量级线程本身期望达到的目标。
     - Go语言的支持：Go 语言在语言级别支持轻量级线程，叫goroutine。Go 语言标准库提供的所有系统调用操作 (当然也包括所有同步 IO 操作)，都会出让 CPU 给其他goroutine。这让事情变得非常简单,让轻量级线程的切换管理不依赖于系统的线程和进程,也不依赖于CPU的核心数量。

1.2 goroutine

> goroutine是Go语言中的轻量级线程实现，由Go运行时(runtime)管理。

1. 启动goroutine：并发执行
   - 格式：go+执行体，执行体一般是函数形式。即：`go 函数调用` 或 `go 匿名函数` 。例如：

```
func Add(x, y int) {
    z := x + y
    fmt.Println(z)
}
go Add(2, 4)     //启动一个协程，go类似Java的Thread.start(), Add(2,4)类似Java的run(){}
```

   - 在一个函数调用前加上 go 关键字，这次调用就会在一个新的goroutine中并发执行。当被调用的函数返回时，这个goroutine也自动结束了。需要注意的是，如果这个函数有返回值，那么这个返回值会被丢弃。

2. Go语言的程序执行机制：Go程序从初始化 main package 并执行 main() 函数开始，当 main() 函数返回时，程序退出，且程序并不等待其他goroutine(非主goroutine)结束。因此，如果要让主函数等待所有goroutine退出后再返回呢？那就需要知道是否其他goroutine都退出了，这就引出了多个goroutine之间通信的问题。
3. goroutine机理(协程机理)
    - 从根本上来说goroutine就是一种Go语言版本的协程(coroutine)。协程，也有人称之为轻量级线程,具备以下几个特点：
        - 能够在单一的系统线程中模拟多个任务的并发执行
        - 在一个特定的时间，只有一个任务在运行，即并非真正地并行
        - 被动的任务调度方式，即没有任务主动抢占时间片的说法。当一个任务正在执行时，外部没有办法中止它。要进行任务切换，只能通过由该任务自身调用 yield() 来主动出让CPU使用权 或者发生了IO，导致执行阻塞(其实也是主动让出)。
        - 每个协程都有自己的堆栈和局部变量
    - 协程的状态：每个协程都包含3种运行状态：挂起、运行和停止。停止通常表示该协程已经执行完成(包括遇到问题后明确退出执行的情况)，挂起则表示该协程尚未执行完成，但出让了时间片，以后有机会时会由调度器继续执行。
    - Go语言协程与C语言的协程
        - Go语言的作者之一拉斯·考克斯(Russ Cox)在Go语言出世之前设计实现了一个轻量级协程库 libtask，有足够充分的理由相信goroutine和用于goroutine之间通信的channel都是参照 libtask 库实现的(甚至可能直接使用这个库)。至于 go 关键字等Go语言特性，可以将其认为只是为了便于开发者使用而设计的语糖。
    - 详细了解可查看libtask 库：包括任务的结构，任务的调度，任务上文切换，以及通信机制。

1.3 channel (通道)

> Go语言的通信模型：以消息机制作为通信方式。Go语言提供的消息通信机制被称为channel (通道)。

1. channel是Go语言在语言级别提供的goroutine间的通信方式。我们可以使用channel在两个或多个goroutine之间传递消息。**channel是进程内的通信方式**，因此通过channel传递对象的过程和调用函数时的参数传递行为比较一致，比如也可以传递指针等。
2. channel是类型相关的。也就是说，**一个channel只能传递一种类型的值，这个类型需要在声明channel时指定**。如果对Unix/Linux的管道有所了解的话，就不难理解channel，可以将其认为是一种类型安全的管道。
3. 基本语法：
    - 声明：格式为：`var 名称 chan 类型`。与一般的变量声明不同的地方仅仅是在类型之前加了 chan 关键字。
    - 创建(初始化)：格式为：`通道变量名 = make(chan 通道变量类型, [size])`。使用内置的函数 make(),默认创建的channel是阻塞的，如果不指定大小，默认为1。
    - 写入：格式为：`通道变量 <- 值`。将一个数据写入(发送)至channel。**向channel写入数据通常会导致程序阻塞，直到有其他goroutine从这个channel中读取数据**。
    - 读出：格式为：`value[,isClose] := <-通道变量`。**从channel中读取数据，会返回2个值，一个是内容，一个是被关闭了没(也可判断还有没有内容)，为false则是关闭了(没内容了)。如果channel之前没有写入数据,那么从channel中读取数据也会导致程序阻塞，直到channel中被写入数据为止**。
    - 关闭：格式为：`close(通道变量)`。
    - 例子：

```
var ch chan string  // 声明一个字符串类型的channel
ch = make(chan string)  // 创建(初始化)一个channel
ch1 := make(chan string)     // 上面两步合在一起。
ch <- "hello,world"      //写入一个字符串到channel中
str := <-ch      // 从channel中读出一个字符串给str
```

4. select：多个Channel的select，可以实现超时机制。
    - 早在Unix时代，select 机制就已经被引入。通过调用 select() 函数来监控一系列的文件句柄。一旦某个文件句柄发生了IO动作，该 select() 调用就会被返回。后来该机制也被用于实现高并发的Socket服务器程序。Go语言直接在语言级别支持 select 关键字，用于处理异步IO问题。
    - select的用法：select 的用法与 switch 语言非常类似，由 select 开始一个新的选择块(channel通道)，每个选择条件由case 语句来描述。与 switch 语句可以选择任何可使用相等比较的条件相比，select 有比较多的限制，其中最大的一条限制就是**每个 case 语句里必须是一个IO操作**，大致的结构如下:

      ```
      select {
          case <-chan1:
          // 如果chan1成功读到数据,则进行该case处理语句
          case chan2 <- 1:
          // 如果成功向chan2写入数据,则进行该case处理语句
          default:
          // 如果上面都没有成功,则进入default处理流程
      }
      ```

      可以看出， select 不像 switch ，后面并不带判断条件，而是直接去查看 case 语句。每个case 语句都必须是一个面向channel的操作。**如果没有default，select会一直阻塞的。除非使用了超时机制**
    - 有趣的程序:

      ```
      ch := make(chan int, 1)
      for {
          select {
              case ch <- 0:
              case ch <- 1:
          }
          i := <-ch
          // 这里可以加一些break，sleep等语句，从而实现等待一段时间，结束channel的阻塞。
          fmt.Println("Value received:", i)
      }
      ```

      这个程序实现了一个不断的随机向 ch 中写入一个0或者1的过程。当然，这是个死循环。
5. 缓冲机制：即创建一个指定大小的channel。如：`c := make(chan int, 1024)`
    - 缓冲机制对于需要持续传输大量数据的场景很有用，因为即使没有读取方，写入方也可以一直往channel里写入，在缓冲区被填完之前都不会阻塞。
    - 从带缓冲的channel中读取数据可以使用与常规非缓冲channel完全一致的方法，但我们也可以使用 range 关键来实现更为简便的循环读取:

```
c := make(chan int, 1024)
for i := range c {
    fmt.Println("Received:", i)
}
```

6. 超时机制：

- 在并发编程的通信过程中，最需要处理的就是超时问题，即向channel写数据时发现channel已满，或者从channel试图读取数据时发现channel为空。如果不正确处理这些情况，很可能会导致整个goroutine锁死。
- Go语言没有提供直接的超时处理机制，但我们可以利用 select 机制。虽然 select 机制不是专为超时而设计的，却能很方便地解决超时问题。因为 **select 的特点是只要其中一个 case 已经完成，程序就会继续往下执行，而不会考虑其他 case 的情况**。
- 为channel实现超时机制：

```
// 首先,我们实现并执行一个匿名的超时等待函数
timeout := make(chan bool, 1)
go func() {
    time.Sleep(1e9) // 等待1秒钟
    timeout <- true
}()
// 然后我们把timeout这个channel利用起来
select {
    case <-ch:
    // 从ch中读取到数据
    case <-timeout:
    // 一直没有从ch中读取到数据,但从timeout中读取到了数据
}
// 或者，更简单的方式:
select {
    case msg1 := <-c1:
        fmt.Println("msg1 received", msg1)
    case msg2 := <-c2:
        fmt.Println("msg2 received", msg2)
    case  <-time.After(time.Second * 30):   //用来让select返回的,注意time.After事件
        fmt.Println("Time Out")
        timout_cnt++
}
```

这样使用 select 机制可以避免永久等待的问题，因为程序会在 timeout 中获取到一个数据后继续执行，无论对 ch 的读取是否还处于等待状态，从而达成1秒超时的效果。

7. channel的传递：

> 利用channel的可传递特性，可以实现非常强大、灵活的系统架构。比之下,在C++、Java、C#中，要达成这样的效果，通常就意味着要设计一系列接口。

- 在Go语言中channel本身也是一个原生类型，与 map 之类的类型地位一样，因此channel本身在定义后也可以通过channel来传递。

- 我们可以使用channel可被传递的特性来实现类unix上非常常见的管道(pipe)特性。管道也是使用非常广泛的一种设计模式，比如在处理数据时，我们可以采用管道设计，这样可以比较容易以插件的方式增加数据的处理流程。
- 例子：利用channel可被传递性实现管道

```
type PipeData struct {
    //假设在管道中传递的数据只是一个整型数,在实际的应用场景中这通常会是一个数据块。
    value int
    handler func(int) int   //类似于函数指针
    next chan int
}
// 一个常规的处理函数。
func handle(queue chan *PipeData) {
    for data := range queue {
        data.next <- data.handler(data.value)
    }
}
// 之后只要定义一系列 PipeData 的数据结构并一起传递给这个函数,就可以达到流式处理数据的目的:
```

8. 单向channel：所谓的单向channel概念，其实只是对channel的一种使用限制。
   - 顾名思义，单向channel只能用于发送或者接收数据。当然，channel本身必须是同时支持读写的，否则根本没法用。因为一个channel真的只能读，那么肯定只会是空的，因为你没机会往里面写数据。反之亦然。所谓的单向channel概念，其实只是对channel的一种使用限制。
   - 我们在将一个channel变量传递到一个函数时，可以通过将其指定为单向channel变量，从而限制该 函数中可以对此 channel 的操 作 , 比如只能往这个 channel 写 , 或者只能从这个 channel读。
   - 声明单向channel格式：单向写入：`var 名称 chan <- 类型`，单向读出：`var 名称 <-chan 类型`。
   - 创建单向channel格式：只能通过将双向channel进行**类型转换**为单向channel。例如：

```
ch1 := make(chan int)
ch2 := <-chan int(ch1) // ch2就是一个单向的读取channel
ch3 := chan<- int(ch1) // ch3 是一个单向的写入channel
```

   - channel是一个原生类型，因此不仅支持被传递，还支持类型转换。只有在介绍了单向channel的概念后，读者才会明白类型转换对于channel的意义：就是在单向channel和双向channel之间进行转换。
   - 最小权限原则：为什么要对channel做这样的单向限制呢?
     - 从设计的角度考虑,所有的代码应该都遵循 “最小权限原则”，从而避免没必要地使用泛滥问题，进而导致程序失控。写过C++程序的读者肯定就会联想起 const指针的用法。非 const 指针具备 const 指针的所有功能，将一个指针设定为 const 就是明确告诉函数实现者不要试图对该指针进行修改。**单向channel也是起到这样的一种契约作用**。
   - 单向channel的用法:

```
func Parse(ch <-chan int) {
    for value := range ch {
        fmt.Println("Parsing value", value)
    }
}
```

除非这个函数的实现者无耻地使用了类型转换，否则这个函数就不会因为各种原因而对 ch进行写，避免在 ch 中出现非期望的数据，从而很好地实践最小权限原则。

1.4 多核并行化, 控制goroutine

1. 现在的计算机都是多核计算机，在执行一些昂贵的计算任务时，我们希望能够尽量利用现代服务器普遍具备的多核特性来尽量将任务并行化，从而达到降低总计算时间的目的。**此时我们需要了解CPU核心的数量,并针对性地分解计算任务到多个goroutine中去并行运行**。
2. Go语言当前版本的Go编译器还不能很智能地去发现和利用多核的优势。即使我们创建了多个goroutine，但这些goroutine都运行在同一个CPU核心上，在一个goroutine得到时间片执行的时候，其他goroutine 都会处于等待状态。
3. 在 Go语言升级到默认支持多 CPU 的某个版本之前 ，我们可以先通过设置环境变量GOMAXPROCS 的值 来控制使用多少个 CPU 核心 来运行我们的程序。
4. 具体操作方法是在main方法前通过直接设置环境变量GOMAXPROCS 的值，或者在代码中启动goroutine之前先调用以下这个语句以设置使用16个CPU核心：

```
import "runtime"
...
ncpu := runtime.NumCPU()     //获取本机的cpu核心数
runtime.GOMAXPROCS(ncpu)     //设置多个 CPU 核心 来运行我们的程序
```

5. 多核并行化一般用来解决网络io等慢设备访问的等待问题的。
6. 出让时间片：runtime.Gosched() 。类似于Java的Thread.yield()方法。在那个goroutine里调用，就是它出让。类似与静态方法（Java的Thread.yield()也是）。
    - 我们可以在每个goroutine中控制何时主动出让时间片给其他goroutine，这可以使用 runtime包中的 Gosched() 函数实现。例如：

```
...
runtime.Gosched()
.....
```

7. 实际上，如果要比较精细地控制goroutine的行为，就必须比较深入地了解Go语言开发包中runtime 包所提供的具体功能。

1.5 同步：锁

1. 即使成功地用channel来作为通信手段，还是避免不了多个goroutine之间共享数据的问题，Go语言的设计者虽然对channel有极高的期望，但也提供了妥善的资源锁方案。
2. 同步锁：Go语言包中的 sync 包提供了两种锁类型：sync.Mutex 和 sync.RWMutex 。
    - 声明即创建：`lock := &sync.Mutex{}` 或 `var lock sync.Mutex` 。之后直接就可以使用了
    - Mutex：互斥锁，最简单的一种锁类型，同时也比较暴力，当一个goroutine获得了 Mutex 后，其他goroutine就只能乖乖等到这个goroutine释放该 Mutex ，才能重新尝试获取。
    - RWMutex：读写锁，相对友好些，是经典的读多写少模型。在读锁占用的情况下，会阻止写，但不阻止其他读，也就是多个goroutine可同时获取读锁(调用 RLock() 方法)。而写锁(调用 Lock() 方法)会阻止任何其他goroutine (无论读和写)进来，整个锁相当于由该goroutine独占。
        - 从 RWMutex 的实现看， RWMutex 类型其实组合了 Mutex :

```
type RWMutex struct {
    w Mutex
    writerSem uint32
    readerSem
    uint32
    readerCount int32
    readerWait int32
}
```

- 对于这两种锁类型，任何一个 Lock() 或 RLock() 均需要保证对应有 Unlock() 或 RUnlock()调用与之对应，否则可能导致等待该锁的所有goroutine处于饥饿状态,甚至可能导致死锁。锁的典型使用模式如下:

```
var lock sync.Mutex
func foo() {
    lock.Lock()
    //...
    defer lock.Unlock()
}
```

3. 互斥锁与读写锁例子:

```
import "sync"
// 互斥锁
lock := &sync.Mutex{}   // lock := &sync.Mutex{}
....
lock.Lock() // 上锁
...do something
lock.Unlock() // 上锁

// 读写锁
rwlock := &sync.RWMutex{}
// 写协程
for i := 0; i < 2; i++ {
    go func(b map[int]int) {
        rwLock.Lock()
        b[8] = rand.Intn(100)
        defer rwLock.Unlock()    //一定要写在defer里，避免饥饿
    }(a)
}
// 读协程
for i := 0; i < 100; i++ {
    go func(b map[int]int) {
        rwLock.RLock() //读锁
        fmt.Println(a)
        defer rwLock.RUnlock()
    }(a)
}
```

1.6 全局唯一性操作与原子操作

1. 全局唯一性操作：在程序里就只运行一次

- 对于从全局的角度只需要运行一次的代码，比如全局初始化操作，Go语言提供了一个 Once类型来保证全局的唯一性操作。Once类型变量的 Do(functionName) 方法可以保证在全局范围内只调用指定的函数一次，而且所有其他goroutine在调用到此语句时，将会先被阻塞,直至全局唯一的once.Do() 调用结束后才继续。例如：

```
var a string
var once sync.Once  // 声明并创建一个Once类型变量
func setup() {
    a = "hello, world"
}
func doprint() {
    once.Do(setup)  //指定某函数，全局范围内只执行一次
    print(a)
}
func twoprint() {
    go doprint()
    go doprint()
}
```

- 如果没有 once.Do() ，我们很可能只能添加一个全局的 bool 变量，在函数 setup() 的最后一行将该 bool 变量设置为 true 。在对 setup() 的所有调用之前，需要先判断该 bool 变量是否已经被设置为 true ，如果该值仍然是 false ，则调用一次 setup() ，否则应跳过该语句。并且还要保证setup函数是一个原子性操作，否则还是有可能被调用多次(只有bool值没被执行的)。例如：

```
var done bool = false
func setup() {
    a = "hello, world"  //这句和下面一句还要保证原子性。
    done = true
}
func doprint() {
    if !done {
        setup()
    }
    print(a)
}
```

2. 原子操作：为了更好地控制并行中的原子性操作，sync 包中还包含一个 atomic 子包，它提供了对于一些基础数据类型的原子操作函数。类似于Java的Atomic包。
    - 在atomic包中对几种基础类型提供了原子操作，包括int32, int64, uint32, uint64, uintptr, unsafe.Pointer。对于每一种类型，提供了五类原子操作分别是：
        - Add+首字母大写的基本类型，增加和减少。例如：

          `func AddInt32(src *int32, dest int32) (new int32) // 将计算结构修改到src`

          使用：atomic.AddInt32(&a, 10) // 返回修改后的a
        - CompareAndSwap+首字母大写的基本类型，比较并交换。
        - Swap+首字母大写的基本类型，交换。
        - Load+首字母大写的基本类型，读取。
        - Store+首字母大写的基本类型，存储。

- 例子：

```
package main
import (
    "sync/atomic"
    "fmt"
)

func main() {
    var a int32
    a += 10
    atomic.AddInt32(&a, 10)
    fmt.Println(a == 20)    // true

    var b uint32
    b += 20
    atomic.AddUint32(&b, ^uint32(10-1))     // 等价于 b -= 10
    fmt.Println(b == 10)     // true
}
```

2. 网络编程

2.1 Socket编程

1. 传统Socket编程的步骤：
   - 建立Socket：使用 socket() 函数。
   - 绑定Socket：使用 bind() 函数。
   - 监听：使用 listen() 函数。或者连接:使用 connect() 函数。
   - 接受连接：使用 accept() 函数。
   - 接收：使用 receive() 函数。或者发送:使用 send() 函数。
2. **Go语言标准库对上述过程进行了抽象和封装。无论我们期望使用什么协议建立什么形式的连接，都只需要调用 net.Dial() 即可**。
   - Dial() 函数：
     - Dial() 函数的原型：`func Dial(net, addr string) (Conn, error)`
     - 其中 net 参数是网络协议的名字，addr 参数是IP地址或域名，**而端口号以“:”的形式跟随在地址或域名的后面，端口号可选**。如果连接成功，返回连接对象，否则返回 error 。
     - 几种常见协议的调用方式：
       - TCP链接：`conn, err := net.Dial("tcp", "192.168.0.10:2100")`
       - UDP链接：`conn, err := net.Dial("udp", "192.168.0.12:975")`
       - ICMP链接：
         - 使用协议名称：`conn, err := net.Dial("ip4:icmp", "www.baidu.com")`
         - 使用协议编号：`conn, err := net.Dial("ip4:1", "10.0.0.3")`
     - 目前，Dial() 函数支持如下几种网络协议:：“tcp” 、 “tcp4” (仅限IPv4)、 “tcp6” (仅限IPv6)、 “udp” 、 “udp4” (仅限IPv4)、 “udp6” (仅限IPv6)、 “ip” 、 “ip4” (仅限IPv4)和 “ip6”(仅限IPv6)。

- 在成功建立连接后，我们就可以进行数据的发送和接收。发送数据时，使用 conn 的 Write()成员方法，接收数据时使用 Read() 方法。
- 例子：建立TCP链接来实现初步的HTTP协议，通过向网络主机发送HTTP Head 请求，读取网络主机返回的信息。

```
package main

import (
    "net"
    "os"
    "bytes"
    "fmt"
)

func main() {
    if len(os.Args) != 2 {
        fmt.Fprintf(os.Stderr, "Usage: %s host:port", os.Args[0])
        os.Exit(1)
    }
    service := os.Args[1]

    conn, err := net.Dial("tcp", service)
    checkError(err)
    _, err = conn.Write([]byte("HEAD / HTTP/1.0\r\n\r\n"))
    checkError(err)

    result, err := readFully(conn)
    checkError(err)
    fmt.Println(string(result))

    os.Exit(0)
}

func checkError(err error) {
    if err != nil {
        fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
        os.Exit(1)
    }
}

func readFully(conn net.Conn) ([]byte, error) {
    defer conn.Close()
    result := bytes.NewBuffer(nil)
    var buf [512]byte
    for {
        n, err := conn.Read(buf[0:])
        result.Write(buf[0:n])
        if err != nil {
            if err == io.EOF {
                break
            }
            return nil, err
```

```
        }
    }
    return result.Bytes(), nil
}
```

- 更丰富的网络通信
  - 实际上，Dial() 函数是对 DialTCP()、 DialUDP()、 DialIP() 和 DialUnix() 的封装。我们也可以直接调用这些函数，它们的功能是一致的。这些函数的原型如下:
    - `func DialTCP(net string, laddr, raddr *TCPAddr) (c *TCPConn, err error)`
    - `func DialUDP(net string, laddr, raddr *UDPAddr) (c *UDPConn, err error)`
    - `func DialIP(netProto string, laddr, raddr *IPAddr) (*IPConn, error)`
    - `func DialUnix(net string, laddr, raddr *UnixAddr) (c *UnixConn, err error)`
  - net 包中包含了一系列的工具函数，合理地使用这些函数可以更好地保障程序的质量。例如:
    - 解析地址和端口号：net.ResolveTCPAddr()，例如:
      - `tcpAddr, err := net.ResolveTCPAddr("tcp4", service)`
    - 验证IP地址有效性：`func net.ParseIP()`，例如:
    - 创建子网掩码：`func IPv4Mask(a, b, c, d byte) IPMask`，例如:
    - 获取默认子网掩码：`func (ip IP) DefaultMask() IPMask`，例如:
    - 根据域名查找IP:
      - `func ResolveIPAddr(net, addr string) (*IPAddr, error)`
      - `func LookupHost(name string) (cname string, addrs []string, err error)`

2.2 HTTP编程

1. Go语言标准库内建提供了 net/http 包，涵盖了HTTP客户端和服务端的具体实现。使用net/http 包，可以很方便地编写HTTP客户端或服务端的程序。一个简单的Http服务端:

```
package main
import (
    "io"
    "log"
    "net/http"
)

func helloHandler(w http.ResponseWriter, r *http.Request) {
    io.WriteString(w, "Hello, world!")
}
func main() {
    http.HandleFunc("/hello", helloHandler) //用于分发请求,设置回调函数，会自动注册到监听器
    err := http.ListenAndServe(":8080", nil)
    if err != nil {
        log.Fatal("ListenAndServe: ", err.Error())
    }
}
```

2. HTTP客户端

- Go内置的 net/http 包提供了最简洁的HTTP客户端实现，我们无需借助第三方网络通信库(例如Java使用HttpClient，linux下c使用libcurl)，就可以直接使用HTTP中用得最多的GET和POST方式请求数据。
- 基本方法：net/http 包的 Client 类型提供了如下几个方法：
  - `func (c *Client) Get(url string) (r *Response, err error)`
    - 要请求一个资源，只需调用 http.Get() 方法
    - 例如：请求一个网站首页，并将其网页内容打印到标准输出流中。

```
resp, err := http.Get("http://baidu.com/")
if err != nil {
    // 处理错误 ...
    return
}
defer resp.Body.close()
io.Copy(os.Stdout, resp.Body)
```

  - `func (c *Client) Post(url string, bodyType string, body io.Reader) (r *Response, err error)`
    - 要以POST的方式发送数据，只需要传递三个参数：请求的目标 URL，将要 POST 数据的资源类型(MIMEType)，数据的比特流([]byte 形式)。
    - 例如：上传一张图片

```
imageDataBuf := ....    // 读取一直读片，放入imageDataBuf
resp, err := http.Post("http://xxx.com/upload", "image/jpeg",
&imageDataBuf)
if err != nil {
    // 处理错误
    return
}
if resp.StatusCode != http.StatusOK {
    // 处理错误
    return
}
// ...
```

  - `func (c *Client) PostForm(url string, data url.Values) (r *Response, err error)`
    - http.PostForm() 方法实现了标准编码格式为 application/x-www-form-urlencoded的表单提交。data url.Values一般是Json数据。
    - 例如：模拟HTML表单提交一篇新文章

```
resp, err := http.PostForm("http://example.com/posts",
                           url.Values{ "title":{"题目"}, "content": {"内容"}
})
if err != nil {
    // 处理错误
    return
}
// ...
```

- `func (c *Client) Head(url string) (r *Response, err error)`
  - HTTP 中的 Head 请求方式表明只请求目标 URL 的头部信息，不需要返回 HTTP Body。Go 内置的 net/http 包同样也提供了 http.Head() 方法，该方法同 http.Get() 方法一样，只需传入目标 URL 一个参数即可。
  - 例如：请求一个网站首页的 HTTP Header信息

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

- `func (c *Client) Do(req *Request) (resp *Response, err error)`
  - 在多数情况下， http.Get() 和 http.PostForm() 就可以满足需求，但是**如果我们发起的HTTP 请求需要更多的定制信息，我们希望设定一些自定义的 Http Header 字段**，比如：设定自定义的 "User-Agent" ，而不是默认的 "Go http package"；传递 Cookie等。此时可以使用 net/http 包 http.Client 对象的 Do() 方法来实现。
  - 例如：

    ```
    req, err := http.NewRequest("GET", "http://example.com", nil) //新建请求
    // ...
    req.Header.Add("User-Agent", "Gobook Custom User-Agent")
    // ...
    client := &http.Client{ //... } // 自定义client，也可以使用默认client，即{}
    resp, err := client.Do(req)
    // ...
    ```

- 高级封装
  - 除了上面介绍的基本HTTP操作，Go语言标准库也暴露了比较底层的HTTP相关库，让开发者可以基于这些库灵活定制HTTP服务器和使用HTTP服务。
  - 自定义 http.Client：使用http.Client 对象的 Do() 方法 + 一些自定义的http.Client。
    - http.DefaultClient：Go语言中默认的Http Client实现。前面使用的http.Get()，http.Post()，http.PostForm() 和 http.Head() 方法其实都是在http.DefaultClient 的基础上进行调用的，比如 http.Get() 等价于 http.Default-Client.Get()，依次类推。
    - 既然存在默认的 Client，那么HTTP Client 也是可以自定义的。在 net/http 包中提供了 Client 类型，http.Client 类型的结构为：

```
type Client struct {
    // Transport用于确定HTTP请求的创建机制。如果为空,将会使用DefaultTransport
    Transport RoundTripper
    /*CheckRedirect定义重定向策略。如果CheckRedirect不为空,客户端将在
     *跟踪HTTP重定向前调用该函数。两个参数req和via分别为即将发起的请求和
     *已经发起的所有请求,最早的已发起请求在最前面。如果CheckRedirect返回
     *错误,客户端将直接返回错误,不会再发起该请求。如果CheckRedirect为空,
     *Client将采用一种确认策略,将在10个连续请求后终止
     */
    CheckRedirect func(req *Request, via []*Request) error
    // 如果Jar为空,Cookie将不会在请求中发送,并会在响应中被忽略
    Jar CookieJar
}
```

其中 Transport 类型必须实现 http.RoundTripper 接口。 Transport 指定了执行一个HTTP 请求的运行机制，倘若不指定具体的 Transport ，默认会使用 http.DefaultTransport ，这意味着 http.Transport 也是可以自定义的。 net/http 包中的 http.Transport 类型实现了http.RoundTripper 接口。

CheckRedirect 函数指定处理重定向的策略。当使用 HTTP Client 的 Get() 或者是 Head()方法发送 HTTP 请求时，若响应返回的状态码为 30x (比如 301 / 302 / 303 / 307)，HTTP Client 会在遵循跳转规则之前先调用这个 CheckRedirect 函数。

Jar 可用于在 HTTP Client 中设定 Cookie，Jar 的类型必须实现了 http.CookieJar 接口，该接口预定义了 SetCookies() 和 Cookies() 两个方法。如果 HTTP Client 中没有设定 Jar，Cookie将被忽略而不会发送到客户端。实际上，我们一般都用 http.SetCookie() 方法来设定Cookie。

- 自定义 http.Transport
  - 在 http.Client 类型的结构定义中，我们看到的第一个数据成员就是一个 http.Transport对象,该对象指定执行一个 HTTP 请求时的运行规则。
  - http.Transport 类型的具体结构：

```
type Transport struct {
    /* Proxy指定用于针对特定请求返回代理的函数。如果该函数返回一个非空的错误,
     *请求将终止并返回该错误。如果Proxy为空或者返回一个空的URL指针,将不使用代理
     */
    Proxy func(*Request) (*url.URL, error)
    // Dial指定用于创建TCP连接的dail()函数。如果Dial为空,将默认使用net.Dial()
函数
    Dial func(net, addr string) (c net.Conn, err error)
    // TLSClientConfig指定用于tls.Client的TLS配置。如果为空则使用默认配置
    TLSClientConfig *tls.Config
    // 是否取消长连接,默认值为 false ,即启用长连接。
    DisableKeepAlives bool
    // 是否取消压缩(GZip),默认值为 false ,即启用压缩。
    DisableCompression bool
    /* 指定与每个请求的目标主机之间的最大非活跃连接(keep-alive)数量。
     *如果MaxIdleConnsPerHost为非零值,则是控制每个host所需要保持的最大空闲
     *连接数。如果该值为空,则使用DefaultMaxIdleConnsPerHost的常量值
     */
    MaxIdleConnsPerHost int
```

```
    // ...
}
```

- http.Transport 类型除了上面定义的公开数据成员以外,它同时还提供了几个公开的成员方法。
  - `func(t *Transport) CloseIdleConnections()`：该方法用于关闭所有非活跃的连接。
  - `func(t *Transport) RegisterProtocol(scheme string, rt RoundTripper)`：该方法可用于注册并启用一个新的传输协议，比如 WebSocket 的传输协议标准(ws)，或者 FTP、File 协议等。
  - `func(t *Transport) RoundTrip(req *Request) (resp *Response, err error)`：用于实现 http.RoundTripper 接口。
- 自定义 http.Transport 也很简单,如下列代码所示:

```
tr := &http.Transport{
    TLSClientConfig:
    &tls.Config{RootCAs: pool},
    DisableCompression: true,
}
client := &http.Client{Transport: tr}
resp, err := client.Get("https://example.com")
```

Client 和 Transport 在执行多个 goroutine 的并发过程中都是安全的，但出于性能考虑，应当创建一次后反复使用。

- 灵活的 http.RoundTripper 接口
  - http.Client 定义的第一个公开成员就是一个 http.Transport 类型的实例，且该成员所对应的类型必须实现 http.RoundTripper 接口。http.RoundTripper 接口的具体定义:

```
type RoundTripper interface {
    /*RoundTrip执行一个单一的HTTP事务,返回相应的响应信息。RoundTrip函数
     *的实现不应试图去理解响应的内容。如果RoundTrip得到一个响应,无论该响应
     *的HTTP状态码如何,都应将返回的err设置为nil。非空的err只意味着没有成功
     *获取到响应。类似地,RoundTrip也不应试图处理更高级别的协议,比如重定向、
     *认证和Cookie等。非必要情况下,RoundTrip不应修改请求内容， 除非了是为了
     *理解Body内容。每一个请求的URL和Header域必须在传入 RoundTrip() 之前就
     *已组织好并完成初始化。
     */
    RoundTrip(*Request) (*Response, error)
}
```

  - 通常，可以在默认的 http.Transport 之上包一层 Transport 并实现 RoundTrip()方法:

```
package main
import(
"net/http"
)

type OurCustomTransport struct {
    Transport http.RoundTripper
```

```
}

func (t *OurCustomTransport) transport() http.RoundTripper {
    if t.Transport != nil {
        return t.Transport
    }
    return http.DefaultTransport
}

func (t *OurCustomTransport) RoundTrip(req *http.Request)
(*http.Response, error) {
    // 处理一些事情 ...
    // 发起HTTP请求
    // 添加一些域到req.Header中
    return t.transport().RoundTrip(req)
}

func (t *OurCustomTransport) Client() *http.Client {
    return &http.Client{Transport: t}
}

func main() {
    t := &OurCustomTransport{...}
    c := t.Client()
    resp, err := c.Get("http://example.com")
    // ...
}
```

因为实现了 http.RoundTripper 接口的代码通常需要在多个 goroutine中并发执行，因此我们必须确保实现代码的线程安全性。

- 设计优雅的 HTTP Client：
  - Go语言标准库提供的 HTTP Client 是相当优雅的。一方面提供了极其简单的使用方式，另一方面又具备极大的灵活性。
  - Go语言标准库提供的HTTP Client 被设计成上下两层结构。一层是上述提到的 http.Client类及其封装的基础方法，我们不妨将其称为“业务层”。因为调用方通常只需要关心请求的业务逻辑本身,而无需关心非业务相关的技术细节，如：HTTP 底层传输细节，HTTP 代理，gzip 压缩等。
  - HTTP Client 在 底 层 抽 象 了http.RoundTripper 接口，而 http.Transport 实现了该接口，从而能够处理更多的细节, 我们不妨将其称为“传输层”。HTTP Client 在业务层初始化 HTTP Method、目标URL、请求参数、请求内容等重要信息后,经过“传输层”，“传输层”在业务层处理的基础上补充其他细节，然后再发起 HTTP 请求,接收服务端返回的 HTTP 响应。

3. HTTP服务端：HTTP服务端技术：如何处理HTTP/HTTPS请求。
   - 处理HTTP请求： http.ListenAndServe() 方法
     - 使用 net/http 包提供的 http.ListenAndServe() 方法，可以在指定的地址进行监听，开启一个HTTP, 服务端该方法的原型是：`func ListenAndServe(addr string, handler Handler) error`
     - 该方法有两个参数：第一个参数 addr 即监听地址；第二个参数表示服务端处理程序，通常为空，即服务端默认调用 http.DefaultServeMux 进行处理，而服务端编写的业务逻辑处理程序 http.Handle() 或 http.HandleFunc() 默认注入 http.DefaultServeMux 中。故，只需将业务逻辑写在http.Handle() 或 http.HandleFunc() 里即可。**这2个方法方法用于分发请求，即针对某一路径请求将其映射到指定的业务逻辑处理方法中**。例如：

```
http.Handle("/foo", fooHandler)
http.HandleFunc("/bar", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello, %q", html.EscapeString(r.URL.Path))
})  // 参数是url，回调函数。指明某个函数处理这个路径的请求
http.ListenAndServe(":8080", nil)
```

- 如果想更多地控制服务端的行为，可以自定义 http.Server ，代码如下：

```
s := &http.Server{
    Addr:    ":8080",
    Handler:     你的Handle,
    ReadTimeout:    10 * time.Second,
    WriteTimeout:   10 * time.Second,
    MaxHeaderBytes: 1 << 20,
}
s.ListenAndServe()
```

- 处理HTTPS请求：http.ListenAndServeTLS() 方法
  - net/http 包还提供 http.ListenAndServeTLS() 方法，用于处理 HTTPS 连接请求，函数原型为：

    `func ListenAndServeTLS(addr string, certFile string, keyFile string, handler Handler) error`
  - ListenAndServeTLS() 和 ListenAndServe() 的行为一致，区别在于只处理HTTPS请求。只是certFile 对应SSL证书文件存放路径，keyFile 对应证书私钥文件路径。
  - 开启 SSL 监听服务也很简单，如下列代码所示：

```
http.Handle("/foo", fooHandler) // 可以多个，分别对于不同路径，对应不同的业务函数
http.HandleFunc("/bar", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello, %q", html.EscapeString(r.URL.Path))
})
http.ListenAndServeTLS(":10443", "cert.pem", "key.pem", nil)
// 想更多地控制服务端的行为，也可以自定义 http.Server
ss := &http.Server{
    Addr:    ":10443",
    Handler:     myHandler,
    ReadTimeout:    10 * time.Second,
    WriteTimeout:   10 * time.Second,
    MaxHeaderBytes: 1 << 20,
}
ss.ListenAndServeTLS("cert.pem", "key.pem")
```

2.3 RPC编程

1. RPC(Remote Procedure Call, 远程过程调用)是一种通过网络对远程计算机的程序进行调用，而不需要了解底层网络细节的应用程序通信协议。RPC协议构建于TCP或UDP，或者是 HTTP之上。
2. Go语言中的RPC支持与处理

- Go的标准库提供的 net/rpc 包实现了 RPC 协议需要的相关细节，开发者可以很方便地使用该包编写 RPC 的服务端和客户端程序，这使得用 Go 语言开发的多个进程之间的通信变得非常简单。
- net/rpc 包允许 RPC 客户端程序通过网络或是其他 I/O 连接调用一个远端对象的公开方法(必须是大写字母开头、可外部调用的)。在 RPC 服务端，可将一个对象注册为可访问的服务，之后该对象的公开方法就能够以远程的方式提供访问。一个 RPC 服务端可以注册多个不同类型的对象，但不允许注册同一类型的多个对象。
- 一个对象中只有满足如下这些条件的方法，才能被 RPC 服务端设置为可供远程访问：
  - 必须是在对象外部可公开调用的方法(首字母大写) ；
  - 必须有两个参数，且参数的类型都必须是包外部可以访问的类型或者是Go内建支持的类型
  - 第二个参数必须是一个指针
  - 方法必须返回一个 error 类型的值

  以上4个条件，可以简单地用如下一行代码表示：

  `func (t *T) MethodName(argType T1, replyType *T2) error`

  在这行代码中，类型 T 、T1 和 T2 默认会使用 Go 内置的 encoding/gob 包进行编码解码。该方法(MethodName)的第一个参数表示由 RPC 客户端传入的参数，第二个参数表示要返回给RPC客户端的结果，该方法最后返回一个 error 类型的值。
- RPC 服务端：通过调用 rpc.ServeConn 处理单个连接请求。多数情况下，通过 TCP 或是 HTTP 在某个网络地址上进行监听来创建该服务是个不错的选择。
- RPC 客户端：
  - 建立与RPC服务端的连接：Go 的 net/rpc 包提供了便利的 **rpc.Dial() 和 rpc.DialHTTP() 方法**来与指定的 RPC 服务端建立连接。
  - 远程调用并接收结果：在建立连接之后，Go 的 net/rpc 包允许我们使用同步或者异步的方式接收 RPC 服务端的处理结果。
    - 同步接收：调用 RPC 客户端的 **Call() 方法**则进行同步处理。这时客户端程序按顺序执行，只有接收完 RPC 服务端的处理结果之后才可以继续执行后面的程序。
    - 异步接收：调用 RPC 客户端的 **Go() 方法**则进行异步处理。这时RPC 客户端程序无需等待服务端的结果即可执行后面的程序，而当接收到 RPC 服务端的处理结果时，再对其进行相应的处理。
  - 无论是调用 RPC 客户端的 Call() 或者是 Go() 方法，都必须指定要调用的服务及其方法名称，以及一个客户端传入参数的引用，还有一个用于接收处理结果参数的指针。如果没有明确指定 RPC 传输过程中使用何种编码解码器,默认将使用 Go 标准库提供的encoding/gob 包进行数据传输。
- RPC 服务端和客户端交互的示例程序：
  - RPC服务端示例代码：

```
package server
import " net/rpc"
type Args struct {
    A, B int
}

type Quotient struct {
    Quo, Rem int
}

type Arith int
```

```
func (t *Arith) Multiply(args *Args, reply *int) error {
    *reply = args.A * args.B
    return nil
}
func (t *Arith) Divide(args *Args, quo *Quotient) error {
    if args.B == 0 {
        return errors.New("divide by zero")
    }
    quo.Quo = args.A / args.B
    quo.Rem = args.A % args.B
    return nil
}

func main() {   // 注册服务对象并开启该 RPC 服务
    arith := new(Arith)
    // 注册了一个 Arith 类型的对象及其公开方法 Arith.Multiply()和Arith.Divide()
    // 供 RPC 客户端调用。
    rpc.Register(arith)
    rpc.HandleHTTP()    // 通过 HTTP 在网络地址上进行监听来创建该服务
    l, e := net.Listen("tcp", ":1234")
    if e != nil {
    log.Fatal("listen error:", e)
    }
    go http.Serve(l, nil)
}
```

- RPC客户端：

```
// 与 RPC 服务端建立连接
client, err := rpc.DialHTTP("tcp", serverAddress + ":1234")
if err != nil {
    log.Fatal("dialing:", err)
}
// 同步调用
args := &server.Args{7,8}
var reply int
err = client.Call("Arith.Multiply", args, &reply)
if err != nil {
    log.Fatal("arith error:", err)
}
fmt.Printf("Arith: %d*%d=%d", args.A, args.B, reply)

// 异步方式进行调用
quotient := new(Quotient)
divCall := client.Go("Arith.Divide", args, &quotient, nil)
replyCall := <-divCall.Done
```

3. Go语言的序列化支持

- 在RPC调用和返回过程中，对象需要进行序列化来传输，减轻传输量和错误。

- Gob简介
  - Gob 是 针对 Go 的数据结构进行编码和解码的专用序列化工具，在 Go 标准库中内置 encoding/gob 包以供使用。这也意味着 Gob 无法跨语言使用。在 Go 的 net/rpc 包中，传输数据所需要用到的编码解码器,默认就是 Gob。
  - 与 JSON 或 XML 这种基于文本描述的数据交换语言不同，Gob 是二进制编码的数据流，并且 Gob 流是可以自解释的，它在保证高效率的同时，也具备完整的表达能力。

4. 设计优雅的RPC接口：可以自定义编码解码器
   - Go 的 net/rpc 很灵活，它在数据传输前后实现了编码解码器的接口定义。这意味着，开发者可以自定义数据的传输方式以及 RPC 服务端和客户端之间的交互行为。
   - RPC 提供的编码解码器接口如下：
     - 客户端接口

       ```
       type ClientCodec interface {
           WriteRequest(*Request, interface{}) error
           ReadResponseHeader(*Response) error
           ReadResponseBody(interface{}) error
           Close() error
       }
       ```

       接口 ClientCodec 定义了 RPC 客户端如何在一个 RPC 会话中发送请求和读取响应。客户端程序通过 WriteRequest() 方法将一个请求写入到 RPC 连接中，并通过 ReadResponseHeader()和 ReadResponseBody() 读取服务端的响应信息。当整个过程执行完毕后，再通过 Close() 方法来关闭该连接。

     - 服务端接口

       ```
       type ServerCodec interface {
           ReadRequestHeader(*Request) error
           ReadRequestBody(interface{}) error
           WriteResponse(*Response, interface{}) error
           Close() error
       }
       ```

       接口 ServerCodec 定义了 RPC 服务端如何在一个 RPC 会话中接收请求并发送响应。服务端程序通过 ReadRequestHeader() 和 ReadRequestBody() 方法从一个 RPC 连接中读取请求信息，然后再通过 WriteResponse() 方法向该连接中的 RPC 客户端发送响应。当完成该过程后，通过 Close() 方法来关闭连接。
   - 通过实现上述接口,我们可以自定义数据传输前后的编码解码方式,而不仅仅局限于 Gob。同样，可以自定义RPC 服务端和客户端的交互行为。实际上，Go 标准库提供的 net/rpc/json包，就是一套实现了 rpc.ClientCodec 和 rpc.ServerCodec 接口的 JSON-RPC 模块。

2.4 JSON处理

1. Go语言内建对JSON的支持。使用Go语言内置的 encoding/json 标准库，开发者可以轻松使用Go程序生成和解析JSON格式的数据。
2. 编码为JSON格式：json.Marshal() 函数
   - 使用 json.Marshal() 函数可以对一组数据进行JSON格式的编码。 json.Marshal() 函数的声明如下：

`func Marshal(v interface{}) ([]byte, error)`

- Go语言的大多数数据类型都可以转化为有效的JSON文本，但channel、 complex和函数这几种类型外。如果转化前的数据结构中出现指针，那么将会转化指针所指向的值，如果指针指向的是零值，那么 null 将作为转化后的结果输出。
- 在Go中,JSON转化前后的数据类型映射如下：
  - 布尔值转化为JSON后还是布尔类型
  - 浮点数和整型会被转化为JSON里边的常规数字
  - 字符串将以UTF-8编码转化输出为Unicode字符集的字符串,特殊字符比如 < 将会被转义为\u003c
  - 数组和切片会转化为JSON里边的数组，但 []byte 类型的值将会被转化为 Base64 编码后的字符串，slice 类型的零值会被转化为 null 。
  - 结构体会转化为JSON对象，并且只**有结构体里边以大写字母开头的可被导出的字段才会被转化输出,**而这些可导出的字段会作为JSON对象的字符串索引。
  - 转化一个 map 类型的数据结构时，该数据的类型必须是 “map[string] T” (T可以是encoding/json包支持的任意数据类型)。

3. 解码JSON数据：json.Unmarshal() 函数
   - 可以使用 json.Unmarshal() 函数将JSON格式的文本解码为Go里边预期的数据结构。json.Unmarshal() 函数的原型如下：`func Unmarshal(data []byte, v interface{}) error` 。该函数的第一个参数是输入，即JSON格式的文本(比特序列)，第二个参数表示目标输出容器，用于存放解码后的值。
   - 在解码Json数据时，用于接收Json的值的容器，**容器里的变量名只有跟Json的key相同才被赋值，而且这些字段的名字必须是大写字符开头**。
   - 在解码JSON数据的过程中，JSON数据里边的元素类型将做如下转换：
     - JSON中的布尔值将会转换为Go中的 bool 类型
     - 数值会被转换为Go中的 float64 类型
     - 字符串转换后还是 string 类型
     - JSON数组会转换为 []interface{} 类型
     - JSON对象会转换为 map[string]interface{} 类型
     - null 值会转换为 nil
4. 解码未知结构的JSON数据：即不知道Json是什么结构，有什么key的Json数据
   - 在Go语言里,接口是一组预定义方法的组合，任何一个类型均可通过实现接口预定义的方法来实现，且无需显示声明，所以没有任何方法的空接口可以代表任何类型。换句话说，每一个类型其实都至少实现了一个空接口。
   - **如果要解码一段未知结构的JSON，只需将这段JSON数据解码输出到一个空接口即可。最终该空接口将会是一个键值对的 map[string]interface{} 结构 或者 []interface{} 或者两者结合**。
5. Json处理例子：

```
type Book struct {
    Title string
    Authors []string
    Publisher string
    IsPublished bool
    Price float
    code int    //不会被转到json，Json的数据也转不到这里
}
gobook := Book{"Go语言编程",["XuShiwei", "HughLv", "Pandaman", "GuaguaSong", "HanTuo",
"BertYuan","XuDaoli"],"ituring.com.cn",true,9.99,100000}
// 将gobook转为json
```

```
b, err := json.Marshal(gobook)  // b 是一个byte数组

// 将json转为对象
var book Book
err := json.Unmarshal(b, &book)

/* 最终 r 将会是一个键值对的 map[string]interface{} 结构
    map[string]interface{}{
        "Title": "Go语言编程",
        "Authors": ["XuShiwei", "HughLv", "Pandaman", "GuaguaSong", "HanTuo",
"BertYuan",
        "XuDaoli"],
        "Publisher": "ituring.com.cn",
        "IsPublished": true,
        "Price": 9.99,
        "Sales": 1000000
    }
*/
var r interface{}  // 如果不知道Json的结构，则用空接口来作为容器
err := json.Unmarshal(b, &r)
```

6. JSON的流式读写：
   - Go内建的 encoding/json 包还提供 Decoder 和 Encoder 两个类型，用于支持JSON数据的流式读写，并提供 NewDecoder() 和 NewEncoder() 两个函数来便于具体实现：
     - `func NewDecoder(r io.Reader) *Decoder`
     - `func NewEncoder(w io.Writer) *Encoder`
   - 使用 Decoder 和 Encoder 对数据流进行处理可以应用到其他地方，比如读写 HTTP 连接、WebSocket 或文件等，Go的标准库 net/rpc/jsonrpc 就是一个应用了 Decoder 和 Encoder 的实际例子。
   - 例子：

```
package main
import (
    "encoding/json"
    "log"
    "os"
)
func main() {
    dec := json.NewDecoder(os.Stdin)
    enc := json.NewEncoder(os.Stdout)
    for {
        var v map[string]interface{}
        if err := dec.Decode(&v); err != nil {
            log.Println(err)
            return
        }
        for k := range v {
            if k != "Title" {
                v[k] = nil, false
            }
        }
        if err := enc.Encode(&v); err != nil {
```

```
            log.Println(err)
        }
    }
}
```

3. 安全编程

3.1 数据的加密

1. 对称加密：采用单密钥的加密算法，整个系统由如下几部分构成：需要加密的明文、加密算法和密钥。在加密和解密中，使用的密钥只有一个。常见的单密钥加密算法有DES、AES、RC4等。
2. 非对称加密：采用双密钥的加密算法，整个系统由如下几个部分构成：需要加密的明文、加密算法、私钥和公钥。在该系统中,私钥和公钥都可以被用作加密或者解密，但是用私钥加密的明文，必须要用对应的公钥解密,用公钥加密的明文，必须用对应的私钥解密。常见的双密钥加密算法有RSA等。
3. 信息摘要：防止篡改和明文可见，只需要加密，形成一个密文，而不需要也很难解密的算法，哈希算法是一种从任意数据中创建固定长度摘要信息的办法。一般我们要求，对于不同的数据，要求产生的摘要信息也是唯一的。常见的哈希算法包括MD5、SHA-1等。

3.2 数字签名和数字证书

1. 数字签名：是指用于标记数字文件拥有者、创造者、分发者身份的字符串。数字签名拥有标记文件身份、分发的不可抵赖性等作用。常用的数字签名采用了非对称加密。
2. 数字证书：为了标记公钥的真实性。数字证书中包含了银行的公钥，有了公钥之后，就可以用公钥加密我们的信息，这样只有用对应的私钥得到我们的信息，确保安全。
3. PKI 体系：PKI,全称公钥基础设施，是使用非对称加密理论，提供数字签名、加密、数字证书等服务的体系，一般包括权威认证机构(CA)、数字证书库、密钥备份及恢复系统、证书作废系统、应用接口(API)等。

3.3 Go 语言的哈希函数

1. Go提供了MD5、SHA-1等几种哈希函数，可以用于对信息进行摘要，防止篡改和明文可见。
2. 使用例子：

```
package main
import(
    "fmt"
    "crypto/sha1"
    "crypto/md5"
)
func main(){
    TestString:="Hello, Word!"
    // MD5
    Md5Inst:=md5.New()
    Md5Inst.Write([]byte(TestString))
    Result:=Md5Inst.Sum([]byte(""))
    fmt.Printf("%x\n\n",Result)
    // SHA-1
    Sha1Inst:=sha1.New()
    Sha1Inst.Write([]byte(TestString))
    Result=Sha1Inst.Sum([]byte(""))
```

```
    fmt.Printf("%x\n\n",Result)
}
```

3.4 加密通信

> 一般的HTTPS是基于SSL(Secure Sockets Layer)协议。SSL是网景公司开发的位于TCP与HTTP之间的透明安全协议，通过SSL，可以把HTTP包数据以非对称加密的形式往返于浏览器和站点之间，从而避免被第三方非法获取。由IETF(Internet Engineering Task Force)实现的TLS(Transport Layer Security)是建立于SSL v3.0之上的兼容协议,它们主要的区别在于所支持的加密算法。

1. 加密通信流程
    - 以一个用户访问`https://baidu.com`为例，展现一下SSL/TLS的基本工作方式：
        - 在浏览器中输入HTTPS协议的网址
        - 服务器向浏览器返回证书，浏览器检查该证书的合法性
        - 浏览器验证合法性后，浏览器使用证书中的公钥加密一个随机对称密钥，并将加密后的密钥和使用密钥加密后的请求URL一起发送到服务器。
        - 服务器用私钥解密随机对称密钥，并用获取的密钥解密加密的请求URL
        - 服务器把用户请求的网页用密钥加密，并返回给用户
        - 用户浏览器用密钥解密服务器发来的网页数据,并将其显示出来
2. Go语言目前实现了TLS协议的部分功能,已经可以提供最基础的安全层服务。
    - 例子：实现一个支持HTTPS的Web服务器

```
package main
import (
    "fmt"
    "net/http"
)
const SERVER_PORT = 8080
const SERVER_DOMAIN = "localhost"
const RESPONSE_TEMPLATE = "hello"

func rootHandler(w http.ResponseWriter, req *http.Request) {
    w.Header().Set("Content-Type", "text/html")
    w.Header().Set("Content-Length", fmt.Sprint(len(RESPONSE_TEMPLATE)))
    w.Write([]byte(RESPONSE_TEMPLATE))
}

func main() {
    http.HandleFunc(fmt.Sprintf("%s:%d/", SERVER_DOMAIN, SERVER_PORT),
rootHandler)
    http.ListenAndServeTLS(fmt.Sprintf(":%d", SERVER_PORT), "rui.crt", "rui.key",
nil)
}
```

    - 例子2：基于SSL/TLS的ECHO程序
        - 服务端

```
package main
import (
```

```
    "crypto/rand"
    "crypto/tls"
    "io"
    "log"
    "net"
    "time"
)

func main() {
    cert, err := tls.LoadX509KeyPair("rui.crt", "rui.key")
    if err != nil {
        log.Fatalf("server: loadkeys: %s", err)
    }
    config := tls.Config{Certificates:[]tls.Certificate{cert}}
    config.Time = time.Now
    config.Rand = rand.Reader

    service := "127.0.0.1:8000"

    listener, err := tls.Listen("tcp", service, &config)
    if err != nil {
        log.Fatalf("server: listen: %s", err)
    }

    log.Print("server: listening")
    for {
        conn, err := listener.Accept()
        if err != nil {
            log.Printf("server: accept: %s", err)
            break
        }
        log.Printf("server: accepted from %s", conn.RemoteAddr())

        go handleClient(conn)
    }
}

func handleClient(conn net.Conn) {
    defer conn.Close()

    buf := make([]byte, 512)
    for {
        log.Print("server: conn: waiting")
        n, err := conn.Read(buf)
        if err != nil {
            if err != io.EOF {
                log.Printf("server: conn: read: %s", err)
            }
            break
        }
        log.Printf("server: conn: echo %q\n", string(buf[:n]))
        n, err = conn.Write(buf[:n])
        log.Printf("server: conn: wrote %d bytes", n)
```

```
        if err != nil {
            log.Printf("server: write: %s", err)
            break
        }
    }
    log.Println("server: conn: closed")
}
```

- 客户端

```
package main
import (
    "crypto/tls"
    "io"
    "log"
)

func main() {
    conn, err := tls.Dial("tcp", "127.0.0.1:8000", nil)
    if err != nil {
        log.Fatalf("client: dial: %s", err)
    }
    defer conn.Close()
    log.Println("client: connected to: ", conn.RemoteAddr())

    state := conn.ConnectionState()
    log.Println("client: handshake: ", state.HandshakeComplete)
    log.Println("client: mutual: ", state.NegotiatedProtocolIsMutual)

    message := "Hello\n"
    n, err := io.WriteString(conn, message)
    if err != nil {
        log.Fatalf("client: write: %s", err)
    }
    log.Printf("client: wrote %q (%d bytes)", message, n)

    reply := make([]byte, 256)
    n, err = conn.Read(reply)
    log.Printf("client: read %q (%d bytes)", string(reply[:n]), n)
    log.Print("client: exiting")
}
```

4. 反射

4.1 基础内容

1. 反射(reflection)是在Java出现后迅速流行起来的一种概念。通过反射，你可以获取丰富的类型信息，并可以利用这些类型信息做非常灵活的工作。反射是把双刃剑，功能强大但代码可读性并不理想。

2. Go语言的反射实现了反射的大部分功能，但没有像Java语言那样内置类型工厂，**故而无法做到像Java那样通过类型字符串创建对象实例**。
3. 基本概念
    - Go语言中 reflect 包里最重要的两个类型 `Type` 和 `Value` 。对所有接口进行反射,都可以得到一个包含 Type 和 Value 的信息结构。**Type 主要表达的是被反射的这个变量本身的类型信息，而 Value 则为该变量实例本身的信息，Type 和 Value 都包含了大量的方法**。例如：

      ```
      var reader io.Reader
      reader = &MyReader{"a.txt"} //MyReader实现了 io.Reader 接口的所有方法
      ```

      如果我们对上面的 reader 变量进行反射，将得到一个 Type 和 Value , Type 为 io.Reader , Value 为 MyReader{"a.txt"} 。

4. 基本用法：通过使用 Type 和 Value ,我们可以对一个类型进行各项灵活的操作。
    - 获取类型信息：`reflect.TypeOf(变量)` 返回Type类型(*reflect.rtype)指针变量

      ```
      package main
      import (
       "fmt"
          "reflect"
      )
      func main() {
          var x float64 = 3.4
          fmt.Println("type:", reflect.TypeOf(x))
      }
      输出的结果如下：
      type: float64
      ```

    - 获取值类型：`reflect.ValueOf(变量)` 返回Value类型(reflect.Value)变量

      ```
      var x float64 = 3.4
      v := reflect.ValueOf(x)
       fmt.Println("type:", v.Type())
      fmt.Println("kind is float64:", v.Kind() == reflect.Float64)
      fmt.Println("value:", v.Float())
      结果如下：
      type: float64
      kind is float64: true
       value: 3.4
      ```

    - 用反射修改变量的值：必须获取的是变量的指针的值类型，即：`reflect.ValueOf(&变量名)`
        - Go语言中所有的类型都是值类型，即这些变量在传递给函数的时候将发生一次复制，操作的是副本，对原变量无影响。为了在反射过程中判断能不能修改，Go语言引入了可设属性这个概念(Settability). 如果 CanSet() 返回 false ，表示你不应该调用 Set() 和 SetXxx() 方法，否则会收到这样的错误：

          `panic: reflect.Value.SetFloat using unaddressable value`

- 为了可以成功地用反射的方式修改了变量 x 的值：只需要，获取值类型时，传入变量的地址即可。
- 例子：

```
   package main
import (
           "fmt"
           "reflect"
)
   func main() {
        var x float64 = 3.14
           p := reflect.ValueOf(&x)    // 注意:得到x的地址
        fmt.Println("type of p:",p.Type())
           fmt.Println("settability of p:",p.CanSet())
        v := p.Elem()
           fmt.Println("settability of v:" , v.CanSet())
           v.SetFloat(7.1)
           fmt.Println(v.Interface())
           fmt.Println(x)
   }
   运行结果:
   type of p: *float64
   settability of p: false
   settability of v: true
   7.1
   7.1
```

5. 对结构体的反射操作：用 Field() 方法来按索引获取对应的成员。
   - 例子：获取一个结构中所有成员的值

```
type T struct {
    A int
    B string
}
t := T{203, "mh203"}
s := reflect.ValueOf(&t).Elem()
typeOfT := s.Type()
for i := 0; i < s.NumField(); i++ {
    f := s.Field(i)
    fmt.Printf("%d: %s %s = %v\n", i, typeOfT.Field(i).Name, f.Type(),
f.Interface())
}
运行结果:
0: A int = 203
1: B string = mh203
```

4.2 Type类型的成员方法

1. 类型 reflect.Type 中有很多成员函数，如下：

```
// 通用方法
func (t *rtype) String() string // 获取 t 类型的字符串描述，不要通过 String 来判断两种类型是否一
```

```
致。
func (t *rtype) Name() string // 获取 t 类型在其包中定义的名称，未命名类型则返回空字符串。
func (t *rtype) PkgPath() string // 获取 t 类型所在包的名称，未命名类型则返回空字符串。
func (t *rtype) Kind() reflect.Kind // 获取 t 类型的类别。
func (t *rtype) Size() uintptr // 获取 t 类型的值在分配内存时的大小，功能和 unsafe.SizeOf 一
样。
func (t *rtype) Align() int  // 获取 t 类型的值在分配内存时的字节对齐值。
func (t *rtype) FieldAlign() int  // 获取 t 类型的值作为结构体字段时的字节对齐值。
func (t *rtype) NumMethod() int  // 获取 t 类型的方法数量。

/*根据索引获取 t 类型的方法,如果方法不存在,则panic.
 *如果 t 是一个实际的类型，则返回值的 Type 和 Func 字段会列出接收者。
 *如果 t 只是一个接口，则返回值的 Type 不列出接收者，Func 为空值。
 */
func (t *rtype) Method() reflect.Method
func (t *rtype) MethodByName(string) (reflect.Method, bool) // 根据名称获取 t 类型的方法。
func (t *rtype) Implements(u reflect.Type) bool // 判断 t 类型是否实现了 u 接口。
func (t *rtype) ConvertibleTo(u reflect.Type) bool // 判断 t 类型的值可否转换为 u 类型。
func (t *rtype) AssignableTo(u reflect.Type) bool // 判断 t 类型的值可否赋值给 u 类型。
func (t *rtype) Comparable() bool // 判断 t 类型的值可否进行比较操作

// 注意对于：数组、切片、映射、通道、指针、接口这些引用类型返回的是指针。
func (t *rtype) Elem() reflect.Type // 获取元素类型、获取指针所指对象类型，获取接口的动态类型

-------------------针对特定类型的其他方法-----------------------------
// 数值
func (t *rtype) Bits() int  // 获取数值类型的位宽，t 必须是整型、浮点型、复数型

// 数组
func (t *rtype) Len() int  // 获取数组的元素个数

// 映射
func (t *rtype) Key() reflect.Type // 获取映射的键类型

// 通道
func (t *rtype) ChanDir() reflect.ChanDir // 获取通道的方向

// 结构体
func (t *rtype) NumField() int  // 获取字段数量
func (t *rtype) Field(int) reflect.StructField  // 根据索引获取字段
func (t *rtype) FieldByName(string) (reflect.StructField, bool)  // 根据名称获取字段
// 根据指定的匹配函数 math 获取字段
func (t *rtype) FieldByNameFunc(match func(string) bool) (reflect.StructField, bool)
func (t *rtype) FieldByIndex(index []int) reflect.StructField  // 根据索引链获取嵌套字段

// 函数
func (t *rtype) NumIn() int // 获取函数的参数数量
func (t *rtype) In(int) reflect.Type // 根据索引获取函数的参数信息
func (t *rtype) NumOut() int // 获取函数的返回值数量
func (t *rtype) Out(int) reflect.Type // 根据索引获取函数的返回值信息
// 判断函数是否具有可变参数。如果有可变参数，则 t.In(t.NumIn()-1) 将返回一个切片。
func (t *rtype) IsVariadic() bool
```

2. 示例

```
package main

import (
    "fmt"
    "reflect"
)

type Skills interface {
    reading()
    running()
}

type Student struct {
    Name string
    Age   int

}

func (self Student) runing(){
    fmt.Printf("%s is running\n",self.Name)
}
func (self Student) reading(){
    fmt.Printf("%s is reading\n" ,self.Name)
}
func main() {
    // 通用方法
    stu1 := Student{Name:"wd",Age:22}
    inf := new(Skills)
    stu_type := reflect.TypeOf(stu1)
    // 特别说明，引用类型需要用Elem()获取指针所指的对象类型
    inf_type := reflect.TypeOf(inf).Elem()
    fmt.Println(stu_type.String())  //main.Student
    fmt.Println(stu_type.Name()) //Student
    fmt.Println(stu_type.PkgPath()) //main
    fmt.Println(stu_type.Kind()) //struct
    fmt.Println(stu_type.Size())  //24
    fmt.Println(inf_type.NumMethod())  //2
    // {reading main func() <invalid Value> 0} reading
    fmt.Println(inf_type.Method(0),inf_type.Method(0).Name)
    //{reading main func() <invalid Value> 0} true
    fmt.Println(inf_type.MethodByName("reading"))

    // 针对特定类型的方法
    stu1 := Student{Name:"wd",Age:22}
    stu_type := reflect.TypeOf(stu1)
    fmt.Println(stu_type.NumField())  //2
    fmt.Println(stu_type.Field(0))  //{Name  string  0 [0] false}
    fmt.Println(stu_type.FieldByName("Age"))  //{{Age  int  16 [1] false} true
}
```

4.3 Value类型的成员方法

1. 不是所有go类型值的Value表示都能使用所有方法。请参见每个方法的文档获取使用限制。在调用有分类限定的方法时，应先使用Kind方法获知该值的分类。调用该分类不支持的方法会导致运行时的panic。
    - reflect.Value.Kind()：获取变量类别，返回常量，如：Uint 、 Float64 等。
    - 例子

```
package main
import (
"reflect"
    "fmt"
)
func main() {
    str := "wd"
    val := reflect.ValueOf(str).Kind()
    fmt.Println(val)//string
}
```

2. reflect.Value类型的成员方法
    - 返回对应的值：Value 类型包含了一系列获取对应类型的值方法，比如 Int() ， Float()等用于返回对应的值。
    - 设置对应的值：如SetInt(x int64) ， SetString(x string)等设置方法。
    - 其他方法

```
##########结构体相关:
func (v Value) NumField() int // 获取结构体字段（成员）数量

func (v Value) Field(i int) reflect.Value  //根据索引获取结构体字段

func (v Value) FieldByIndex(index []int) reflect.Value // 根据索引链获取结构体嵌套字段

func (v Value) FieldByName(string) reflect.Value // 根据名称获取结构体的字段，不存在返回reflect.ValueOf(nil)

func (v Value) FieldByNameFunc(match func(string) bool) Value // 根据匹配函数 match 获取字段,如果没有匹配的字段，则返回零值（reflect.ValueOf(nil)）


########通道相关:
func (v Value) Send(x reflect.Value)// 发送数据（会阻塞），v 值必须是可写通道。

func (v Value) Recv() (x reflect.Value, ok bool) // 接收数据（会阻塞），v 值必须是可读通道。

func (v Value) TrySend(x reflect.Value) bool // 尝试发送数据（不会阻塞），v 值必须是可写通道。

func (v Value) TryRecv() (x reflect.Value, ok bool) // 尝试接收数据（不会阻塞），v 值必
```

```
须是可读通道。

func (v Value) Close() // 关闭通道


########函数相关
func (v Value) Call(in []Value) (r []Value) // 通过参数列表 in 调用 v 值所代表的函数（或方法）。函数的返回值存入 r 中返回。
// 要传入多少参数就在 in 中存入多少元素。
// Call 即可以调用定参函数（参数数量固定），也可以调用变参函数（参数数量可变）。

func (v Value) CallSlice(in []Value) []Value // 调用变参函数
```

3. 例子
   - 示例一：获取和设置普通类型的值

```
package main

import (
    "reflect"
    "fmt"
)

func main() {
    str := "wd"
    age := 11
    fmt.Println(reflect.ValueOf(str).String()) //获取str的值，结果wd
    fmt.Println(reflect.ValueOf(age).Int())   //获取age的值，结果age
    str2 := reflect.ValueOf(&str)         //获取Value类型
    str2.Elem().SetString("jack")      //设置值
    fmt.Println(str2.Elem(),age) //jack 11
}
```

   - 示例二：简单结构体操作

```
package main

import (
    "fmt"
    "reflect"
)

type Skills interface {
    reading()
    running()
}

type Student struct {
    Name string
    Age   int
```

```
}

func (self Student) runing(){
    fmt.Printf("%s is running\n",self.Name)
}
func (self Student) reading(){
    fmt.Printf("%s is reading\n" ,self.Name)
}
func main() {
    stu1 := Student{Name:"wd",Age:22}
    stu_val := reflect.ValueOf(stu1) //获取Value类型
    fmt.Println(stu_val.NumField()) //2
    fmt.Println(stu_val.Field(0),stu_val.Field(1)) //wd 22
    fmt.Println(stu_val.FieldByName("Age")) //22
    stu_val2 := reflect.ValueOf(&stu1).Elem()
    stu_val2.FieldByName("Age").SetInt(33)  //设置字段值 , 结果33
    fmt.Println(stu1.Age)

}
```

- 示例三：通过反射调用结构体中的方法，通过reflect.Value.Method(i int).Call()或reflect.Value.MethodByName(name string).Call()实现

```
package main

import (
    "fmt"
    "reflect"
)

type Student struct {
    Name string
    Age int
}

func (this *Student) SetName(name string) {
    this.Name = name
    fmt.Printf("set name %s\n",this.Name )
}

func (this *Student) SetAge(age int) {
    this.Age = age
    fmt.Printf("set age %d\n",age )
}

func (this *Student) String() string {
    fmt.Printf("this is %s\n",this.Name)
    return this.Name
}

func main() {
    stu1 := &Student{Name:"wd",Age:22}
```

```
    val := reflect.ValueOf(stu1) //获取Value类型,使用reflect.ValueOf(&stu1).Elem()亦
可
    val.MethodByName("String").Call(nil)  //调用String方法

    params := make([]reflect.Value, 1)
    params[0] = reflect.ValueOf(18)
    val.MethodByName("SetAge").Call(params)  //通过名称调用方法

    params[0] = reflect.ValueOf("jack")
    val.Method(1).Call(params)    //通过方法索引调用

    fmt.Println(stu1.Name,stu1.Age)
}
```

运行结果

```
//this is wd
//set age 18
//set name jack
//jack 18
```

5. 工程管理

我们知道(假装你知道)，C语言中要编译一个大型项目，必须使用makefile工具来编写工程文件(就是linux中常看到的makefile文件)，进行编译安装。但是，Go语言发布的强大命令行工具彻底消除了工程文件的概念，**完全用目录结构和包名来推导工程结构和构建顺序**。我们不需要写makefile，因为这个工具会替我们分析，知道目标代码的编译结果应该是一个包还是一个可执行文件，并**分析 import 语句以了解包的依赖关系，从而在编译mian包之前先把依赖的编译打包好**。Go命令行程序制定的目录结构规则让代码管理变得非常简单。

5.1 Go命令行工具

1. Go命令行工具的用法：即以go开头的命令使用方法
    - 查看Go语言的版本：go version
    - 解释执行：go run xxx.go
    - 编译执行：go build xxx.go
    - 帮助：go help 或者 go help 某条命令
    - 格式化代码：go fmt xxx.go 或者格式化当前目录下全部：go fmt
2. Go命令行工具可以帮你完成以下这几类工作：
    - 代码格式化
    - 代码质量分析和修复
    - 单元测试与性能测试
    - 工程构建
    - 代码文档的提取和展示
    - 依赖包管理
    - 执行其他的包含指令，比如 6g 等
3. 代码风格

> Go语言很可能是第一个将代码风格强制统一的语言。Go有两类编码规范：由Go编译器进行强制的编码规范，以及由Go命令行工具推行的非强制性编码风格建议。

    - 强制性编码规范

        - 命名：命名规则涉及变量、常量、全局函数、结构、接口、方法等的命名。Go语言从语法层面进行了以下限定：**任何需要对外暴露的名字必须以大写字母开头，不需要对外暴露的则应该以小写字母开头.**
        - 排列：Go语言甚至对代码的排列方式也进行了语法级别的检查，约定了代码块中花括号的明确摆放位置：左花括号不能单独一行；else 甚至都必须紧跟在之前的右花括号 } 。
    - 非强制性编码风格建议：使用go fmt 来格式化代码。
        - 以制表符缩进代码
        - 运算符左右添加空格
        - .....
4. 远程 import 支持：
    - Go可以像导入本地包一样导入远程的包，例如：

```
package main
import (
    "fmt"
    "github.com/myteam/exp/crc32"    //域名的 . 换成 /
)
```

    - 在执行 go build 或者 go install 之前，只需要先执行这么一句：go get github.com/myteam/exp/crc32 就会自动下载到本地。
5. 跨平台开发：在Android手机上可以运行Go程序。
6. 单元测试：
    - Go本身提供了一套轻量级的测试框架。符合规则的测试代码会在运行测试时被自动识别并执行。单元测试源文件的命名规则如下：在需要测试的包下面创建以“_test”结尾的go文件，形如`[^.]*_test.go`。
    - Go的单元测试函数分为两类：功能测试函数和性能测试函数，分别为以 `Test` 和 `Benchmark` 为函数名前缀并以 `*testing\.T` 为单一参数的函数。下面是测试函数声明的例子：
        - `func TestAdd1(t *testing.T)`
        - `func BenchmarkAdd1(t *testing.T)`
    - 执行功能单元测试非常简单 , 直接执行`go test 包名`命令即可 。

6. 语言交互性

1. 自C语言诞生以来，程序员们已经积累了无数的代码库，且有无数的代码库还很偏执地只提供了C语言版本。
2. 作为一门直接传承于C的语言，Go当然应该将与C语言的交互作为首要任务之一。Go确实也提供了这一功能,称为Cgo。
3. 与C语言交互的例子：
    - 在Go中用C获取一个随机数

```
package main
import "fmt"
/*
#include <stdlib.h>
*/
import "C"  // 不是一个包，是一个信号：启动Cgo。上面的注释必须紧跟着这句的前面，不能没有
func Random() int {
    return int(C.random())
```

```
}
func Seed(i int) {
    C.srandom(C.uint(i))
}
func main() {
    Seed(100)
    fmt.Println("Random:", Random())
}
```

- **事实上，根本就不存在一个名为 C 的包。`import "C"`语句其实就是一个信号，告诉Cgo它应该开始工作了。做什么事情呢？就是对应这条 import 语句之前的块注释中的C源代码自动生成包装性质的Go代码。**
- **注意到 import 语句前紧跟的注释了。这个注释的内容是有意义的，而不是传统意义上的注释作用。这个例子里用的是一个块注释，实际上用行注释也是没问题的，只要是紧贴在 import 语句之前即可。并且在这个块注释中，可以写任意合法的C源代码**，例如：

```
// #include <stdio.h>
// #include <stdlib.h>
/*
void hello() {
printf("Hello, Cgo! -- From C world.\n");
}
*/
import "C"
....
C.hello()
```

- 函数调用从汇编的角度看，就是一个将参数按顺序压栈(push)，然后进行函数调用(call)的过程。Cgo生成的代码只不过是帮你封装了这个压栈和调用的过程，从外面看起来就是一个普通的Go函数调用。

4. 类型映射，对象生命周期和内存管理：
    - 在跨语言交互中，比较复杂的问题有两个：类型映射以及跨越调用边界传递指针所带来的对象生命周期和内存管理的问题。
    - C语言与Go语言之间的类型映射
        - 对于C语言的原生类型，Cgo都会将其映射为Go语言中的类型：
            - C.char 和 C.schar (对应于C语言中的 signed char)
            - C.uchar (对应于C语言中的 unsigned char)
            - C.short 和 C.ushort(对应于 unsigned short)
            - C.int 和 C.uint (对应于 unsigned int)
            - C.long 和 C.ulong(对应于 unsigned long)
            - C.longlong (对应于C语言中的 long long)
            - C.ulonglong (对应于C语言中的 unsigned long long 类型)以及 C.float 和 C.double
            - C语言中的 void* 指针类型在Go语言中则用特殊的 unsafe.Pointer 类型来对应
            - C语言中的 struct、union 和 enum 类型，对应到Go语言中都会变成带这样前缀的类型名称：struct_ 、 union_ 和 enum_ 。比如一个在C语言中叫做 person 的 struct 会被 Cgo 翻译为C.struct_person 。
            - 如果C语言中的类型名称或变量名称与Go语言的关键字相同，Cgo会自动给这些名字加上下划线前缀。
    - 对象生命周期和内存管理：

- 因为Go语言中有明确的 string 原生类型，而C语言中用字符数组表示，要保证映射到C语言的对象的生命周期足够长，以避免在C语言执行过程中该对象就已经被垃圾回收。
- Cgo 提供了一系列函数来将Go的字符串转为C的字符串：C.CString, C.GoString和C.GoStringN 。
- 由于 C.CString 的内存管理方式与Go语言自身的内存管理方式不兼容，因此，我们在使用完后必须显式释放调用 C.CString 所生成的内存块，否则将导致严重的内存泄露。
- 例子：

```
var gostr string = "Hello,World!"
cstr := C.CString(gostr)
defer C.free(unsafe.Pointer(cstr))  // 调用C的free函数释放内存
// 接下来大胆地使用cstr吧，因为保证可以被释放掉了
C.sprintf(cstr, "content is: %d", 123)
```

5. 调用C语言的函数
   - 对于常规返回了一个值的函数，调用者可以用以下的方式顺便得到错误码：
     - `n, err := C.atoi("a234")`
   - 在传递数组类型的参数时需要注意，在Go语言中将第一个元素的地址作为整个数组的起始地址传入，这一点就不如C语言本身直接传入数组名字那么方便了。例如
     - `n, err := C.f(&array[0]) // 需要显示指定第一个元素的地址`

7. Go语言标准库

1. Go标准库可以大致按其中库的功能进行以下分类：
   - 输入输出：这个分类包括二进制以及文本格式在屏幕、键盘、文件以及其他设备上的输入输出等。比如二进制文件的读写。此分类的包有 bufio 、 fmt 、 io 、 log 和 flag等，其中 flag 用于处理命令行参数。
   - 文本处理：这个分类包括字符串和文本内容的处理，比如字符编码转换等。此分类的包有 encoding 、 bytes 、strings 、 strconv 、 text 、 mime 、 unicode 、 regexp 、index 和 path 等。其中 path 用于处理路径字符串。
   - 网络：这个分类包括开发网络程序所需要的包,比如Socket编程和网站开发等。此分类的包有: net 、 http 和 expvar 等。
   - 系统：这个分类包含对系统功能的封装,比如对操作系统的交互以及原子性操作等。此分类的包有 os 、 syscall 、 sync 、 time 和 unsafe 等。
   - 数据结构与算法：此分类的包有 math 、sort 、container 、 crypto 、 hash 、archive 、compress 和 image 等。因为 image 包里提供的图像编解码都是算法，所以也归入此类。
   - 运行时：对应于此分类的包有 runtime 、 reflect 和 go 等。
2. 常用包介绍：
   - fmt：它实现了格式化的输入输出操作，其中的 fmt.Printf() 和 fmt.Println() 是开发者使用最为频繁的函数。
   - io：它实现了一系列非平台相关的IO相关接口和实现，比如提供了对 os 中系统相关的IO功能的封装。我们在进行流式读写(比如读写文件)时，通常会用到该包。
   - bufio：它在 io 的基础上提供了缓存功能。在具备了缓存功能后，bufio 可以比较方便地提供 ReadLine 之类的操作。
   - strconv：本包提供字符串与基本数据类型互转的能力。

- os：本包提供了对操作系统功能的非平台相关访问接口，接口为Unix风格。提供的功能包括文件操作、进程管理、信号和用户账号等。
- sync：它提供了基本的同步原语。在多个goroutine访问共享资源的时候需要使用 sync中提供的锁机制。
- flag：它提供命令行参数的规则定义和传入参数解析的功能。绝大部分的命令行程序都需要用到这个包。
- encoding/json：JSON目前广泛用做网络程序中的通信格式。本包提供了对JSON的基本支持，比如从一个对象序列化为JSON字符串，或者从JSON字符串反序列化出一个具体的对象等。
- http：它是一个强大而易用的包，也是Golang语言是一门“互联网语言”的最好佐证。通过 http 包，只需要数行代码，即可实现一个爬虫或者一个Web服务器，这在传统语言中是无法想象的。

Go的Web编程

> Web编程方面，主要包括数据库，MVC，Session等话题。参照了郑兆雄的《Go Web编程》以及谢孟军《Go Web编程》。使用Beego框架。