实验一：Go语言基础&区块链中的典型密码算法

一、 实验概述

Go（又称golang）是Google开发的一种静态强类型、编译型、并发型，并具有垃圾回收功能的编程语言。由于实验环境是在go环境下开发，因此需要预先对go的语法规则有一个基本的了解。

二、 实验准备

实验系统：Win10，Linux，Mac OS均可。

要求环境：Go 1.12.9或更高版本

三、 配置Go语言环境

本节将首先对Go语言的编译环境进行配置，已安装的同学可以跳过这一步。

3.1 安装go

Win10：运行go1.12.9.windows-amd64.msi，将安装至C:\Go

Linux：解压压缩包go1.12.9.linux-amd64.tar.gz至/usr/local，可能需要sudo权限

```
1. $ tar -C /usr/local -xzf go1.12.9.linux-amd64.tar.gz
```

同时将go的路径添加至环境变量，在$HOME/.profile文件最后添加如下代码

```
1. $ export PATH=$PATH:/usr/local/go/bin
2. $ export GOPATH=$HOME/go
```

然后重启Terminal或者执行source $HOME/.profile

Mac OS：

运行go1.12.9.darwin-amd64.pkg，将安装至/usr/local/go

3.2 测试是否安装成功

Win10：进入C:\Users*你的用户名*\go\src\hello（没有就创建一个）

Linux and Mac OS：Terminal下，进入$HOME/go/src/hello（没有就创建一个）然后创建hello.go文件，输入

```
1. package main
```

```
import "fmt"

func main() {
fmt.Printf("hello, world\n")
}
```

在当前路径的命令行下运行

```
$ go build
$ ./hello
```

如果能够成功看到输出hello world说明配置完成。

四、 Go语言入门

针对未接触过Go语言的同学，本节将对Go语言语法进行简单的入门介绍。Go语言是近年开始活跃的一门编程语言，在保持简洁、快速、安全的情况下提供了对海量并发的支持，这也使其成为一门适合Web 服务器，存储集群或类似用途的编程语言。

```
//语言结构
package main                //声明该 go 文件属于 main 包
import (                  //导入包语法
    "fmt"                //包含格式化 I/O 函数，如 Printf,Scanf 等
  "database/sql"
)

func main(){                //"{"不能单独写在一行
 ...
        }
```

```
//变量与常量
   var a string = "hello"   //声明 string 类型变量 a，并赋值“hello”；Go 语言不以分号结尾。
   b := "world"          //将"world"赋值给变量 b，并自动判断类型；b 必须为新变量。
   var c bool           //声明变量 c 并赋予“零值”。
   const d uint32 = 1      //定义常量
```

```
//控制语句（Go 不以缩进来区分代码层次）
//for 循环，初始化语句和后置语句都是非必须的。
   sum := 0
   for i := 0; i < 10; i++ {
     sum += i
   }
```

```
7.
8. //if...else 语句，if 语句可以在条件表达式前执行一个简单的语句。
9.    if v := math.Pow(x, n); v < lim {
10.           return v
11.         } else {
12.           fmt.Printf("%g >= %g\n", v, lim)
13.         }
14.
15.       //switch 语句，与其他语言的区别在于，case 可以不为常量；执行完匹配 case 后会自动停止（相当于加了 break）
16.         fmt.Print("Go runs on ")
17.         switch os := runtime.GOOS; os {
18.         case "darwin":
19.           fmt.Println("OS X.")
20.         case "linux":
21.           fmt.Println("Linux.")
22.         default:
23.           // freebsd, openbsd,
24.           // plan9, windows...
25.           fmt.Printf("%s.\n", os)
26.         }
27.
28.       //defer 语句，会将函数推迟到外层函数返回之后执行。如本例程将在 main 函数执行完输出 hello 后，再输出 world。
29.       func main() {
30.         defer fmt.Println("world")
31.         fmt.Println("hello")
32.       }
```

```
1. //数据结构
2. //指针,与 C 语言类似
3. i := 10
4. p = &i
5. *p = 11
6.
7. //结构体
8. type LAB struct{
9.  number int
10.         date string
11.         done bool
12.       }
13.       lab1 := LAB{1,"2020-10-01",false}
14.
15.       //数组与切片
16.       var balance = [5]float32{1000.0, 2.0, 3.4, 7.0, 50.0} //声明并赋值了长度 5，类型 float32 的数组
17.       var s []float = balance[1:4] //[]T 表示切片类型，其*引用*了数组 balance 的 1 至 3 号元素
18.       s1 := []int{1,2,3}       //也可直接创建一个切片
19.
20.       //映射,元素为键值对
21.       var m = map[string]int{
```

```
    "store1" : 100
    "store2" : 90
  }
  m["store1"] = 80    //修改元素
  delete(m,"store1")   //删除元素

  //函数值和闭包
  //在 Go 语言中，函数可以作为值被传递，也可以作为其他函数的参数或返回值。
  //函数作为值被赋给 hypot，而 hypot 可作为参数被其他函数调用。
    hypot := func(x, y float64) float64 {
      return math.Sqrt(x*x + y*y)
    }
    fmt.Println(hypot(5, 12))
  //go 函数可以是闭包，闭包是一个函数值，它引用了其函数体之外的变量，该函数可以访问并赋予其引用的变量的值。
  func adder() func(int) int {
    sum := 0
    return func(x int) int {    //该闭包与其外部的 sum 相绑定
      sum += x
      return sum
    }
  }

  func main() {
    pos = adder()
    for i := 0; i < 10; i++ {
    fmt.Println(pos(1))       //由于 pos 与 sum 绑定，每调用一次 pos(1) 都会执行 sum+1，并维持 sum 的值。
    }
```

```
//方法和接口
//Go 语言没有类，但是通过结构体和方法实现了相关功能。
type Vertex struct {        //定义结构体，其成员相当于类的成员变量
  X, Y float64
}

func (v Vertex) Abs() float64 {  //定义方法,(v Vertex)表示接收者，该方法类似于类的成员方法
  return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

  //接口是对所有的具有共性的方法的一种抽象，任何其他类型只要实现了这些方法就是实现了这个接口。
  type Car interface {        //所有汽车都能驾驶，因此抽象 drive()方法
    drive()
  }
  type TeslaCar struct {       //其他汽车类型，只要实现了 call()方法，就是是实现了 Car 接口
  }
  func (tc TeslaCar) drive() {
    fmt.Println("I am tesla!")
```

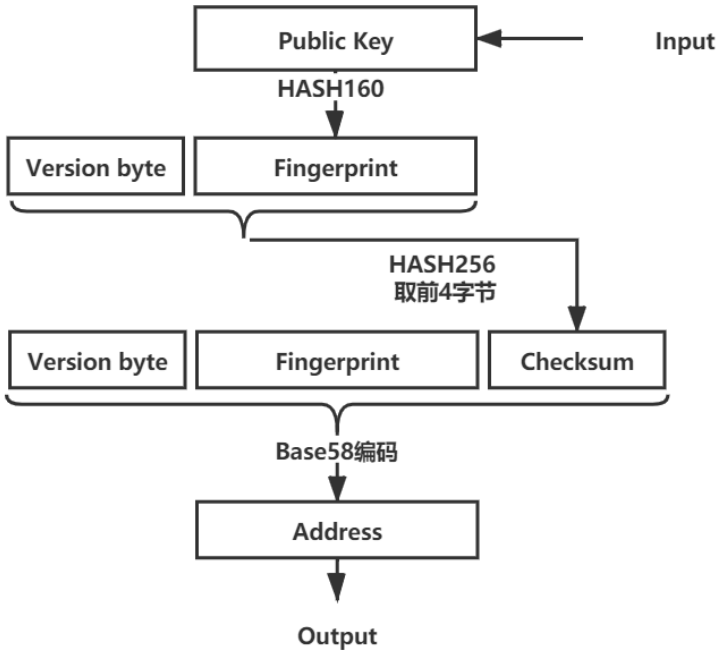
```
}
type BydCar struct {
}
func (bc BydCar) drive() {
  fmt.Println("I am BYD!")
}
func main() {
  var car Car
  car = new(TeslaCar)       //接口变量可以直接被赋值实现了该接口的类型实例。
  car.drive()
  car = new(BydCar)
  car.drive()
}

//接口具有广泛的用处，例如 go 中的错误处理就是用接口实现的
type error interface {
  Error() string
}
```

```
//并发
//go 支持高并发的原因就在于其 goroutine，是一种轻量级线程，可用 go 关键字开启。
go f(x,y,z) //创建一个 goroutine 并在内执行 f(x,y,z)函数
//由于 goroutine 之间是相互独立的，因此连续开启两个 goroutine 后，二者内部运行是没有先后关系的。
//并发部分涉及内容较多，例如 goroutine 之间的通信，暂不在 Go 语言入门考虑内，感兴趣的可自行了解。
```

五、 实验内容

实验1 比特币测试网地址的生成

参考以下比特币地址生成流程，用Go语言实现如下操作：



使用RIPEMD-160、SHA-256哈希算法以及Base58编码对给定公钥生成地址

给定公钥：

public key 1:

02b1ebcdbac723f7444fdfb8e83b13bd14fe679c59673a519df6a1038c07b719c6

public key 2:

036e69a3e7c303935403d5b96c47b7c4fa8a80ca569735284a91d930f0f49afa86

提示：

比特币中有两种复合式的哈希函数，分别为：

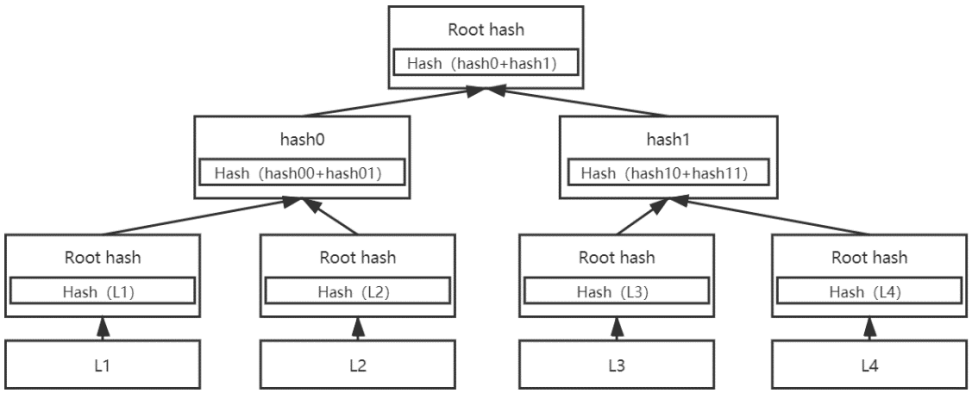
HASH160，即先对输入做一次SHA256，再做一次RIPEMD160；

HASH256，即先对输入做一次SHA256，再做一次SHA256。

本练习要求的version byte为0x6f(测试网)。

实验2 Merkle Tree

Merkle Tree是比特币中用来存储交易单的一种数据结构，它是一种二叉树，所有叶子节点均为交易数据块，而非叶子节点则存储了该节点两个子节点的Hash值，经过层层传递，最终得到根Hash值，这样，当任何叶子节点的交易数据发生改变时，都会导致根Hash值的改变，这对于验证和定位被修改的交易十分高效：



请用Go语言实现一棵叶子节点数为16的Merkle Tree，并在叶子节点存储任意字符串，并在所有非叶节点计算相应Hash值

请将上一步生成的Merkle Tree任一叶子节点数据进行更改，并重新生成其余Hash值。利用Merkle Tree的特点对该修改位置进行快速定位

即设计函数func compareMerkleTree(*MTree tree1, *MTree tree2) (int index) { }