go语法

1、数据格式

1、数组和切片

1.1 有如下代码

msg：=[]int{1,2,3,4,5,6,7,8,9};

sli1:=msg[2:3:4]

sli2:=msg[2:3] 请说明两个切片的不同

答：

> 考点：切片容量
>
> 里面的数据一样，但是两个切片的容量不一样
>
> 容量=max-low

1.2 写出一下程序运行的结果

```
func main(){
    abc:=make([]int,2)
    abc=append(abc,1,2,3)
    fmt.Println(abc)
}
```

答：

> 考点：切片追加
>
> 结果：[0 0 0 0 0 0 0 0 0 0 1 2 3]

1.3 下面代码输出什么?

```
package main

import "fmt"

func main(){
  arr:=[]int{1,2,3,4,5,6,7,8}
  s1:=arr[2:6]
  s2:=s1[3:6]
  fmt.Println("s1==",s1)
  fmt.Println("s2==",s2)
}
```

答:

> 考点：切片截取
>
> 结果：
>
> s1== [3 4 5 6]
>
> s2== [6 7 8]

1.4 是否可以编译通过？如果通过，输出什么？

```
package main

import "fmt"

func main() {
  s1 := []int{1, 2, 3}
  s2 := []int{4, 5}
  s1 = append(s1, s2)
  fmt.Println(s1)
}
```

答:

> 考点：append追加不定参
>
> append切片时候别漏了'...'
>
> 结果：编译失败。

1.5 解释Slice和Array的区别

答:

> 考点：数组和切片
>
> 结果：

> array是固定长度的数组，使用前必须确定数组长度
>
> slice是一个引用类型，是一个动态的指向数组切片的指针。
> slice是一个不定长的，总是指向底层的数组array的数据结构。
>
> 数组是值类型，把一个数组赋予给另一个数组时是发生值拷贝，而切片是指针类型，拷贝的是指针。所以在golang的方法中即使是值传递切片，其实也是传递的指针。
>
> 数组大小是固定的，切片大小不是。在运行时可以动态地增加或减少切片的大小，但数组不可以。切片类似于链表，可以向切片push，pop数据，实现FIFO，LIFO。使用了内置的添加、复制功能对切片操作

1.6 slice的底层实现

答:

> 考点：slice的底层实现
>
> 结果：
>
> slice底层存储了三个内容：地址、长度、容量。
> 地址是指数据存储的位置。
> 长度表示有效数据的个数。
> 容量表示本次（不更换地址）可以扩容的最大值。
> 长度和容量随着append的添加而改变，地址可能改变。

2、map

2.1 写出一下程序运行的结果

```
package main
import (
"fmt"
)
func main(){
    dict:=map[string]int{"王五":60,"张三":43}
    modify(dict)
    fmt.Println(dict["张三"])
}
func modify(dict map[string]int){
    dict["张三"]=10
}
```

答:

> 考点：map作为函数参数，引用传递。
>
> map作为函数参数是引用传递，形参可以修改实参的值。
>
> 结果：10

2.2 编译并运行如下代码会发生什么?

```
package main

import "fmt"

type Test struct {
  Name string
}

var list map[string]Test

func main() {

  list = make(map[string]Test)
  name := Test{"xiaoming"}
  list["name"] = name
  list["name"].Name = "Hello"
  fmt.Println(list["name"])
}
```

答:

> 考点:map
>
> 结果:编译失败。
>
> 编程报错`cannot assign to struct field list["name"].Name in map`。因为list[“name”]不是一个普通的指针值,map的value本身是不可寻址的,因为map中的值会在内存中移动,并且旧的指针地址在map改变时会变得无效。 定义的是var list map[string]Test,注意哦Test不是指针,而且map我们都知道是可以自动扩容的,那么原来的存储name的Test可能在地址A,但是如果map扩容了地址A就不是原来的Test了,所以go就不允许写数据。改为var list map[string]*Test。

2.3 Map的键类型不能是哪些类型?

答:

> 考点:map的键
>
> 结果:
>
> > 字典的键类型不能是以下类型:函数类型,字典类型,切片类型
> >
> > 键类型的值之间必须支持判等操作,函数类型,字典类型,切片类型不支持判等操作
> >
> > 字典是引用类型,只声明而不初始化,值是`nil`。除了添加键-元素对,在值为`nil`的字典上操作都不会引起错误。

3、数据定义

3.1 是否可以编译通过？如果通过，输出什么？

```
const (
  x = iota
  y
  z = "zz"
  k
  p = iota
)

func main()  {
  fmt.Println(x,y,z,k,p)
}
```

答:

> 考点：常量定义和iota枚举赋值
>
> 结果：0 1 zz zz 4
>
> iota 换行值+1

3.2 编译执行下面代码会出现什么?

```
package main
var(
    size :=1024
    max_size = size*2
)
func main()  {
    println(size,max_size)
}
```

答:

> 考点：全局变量定义
>
> 结果：编译失败。
>
> 常量不允许使用自动推导类型。

3.3 下面函数有什么问题?

```
package main
const cl = 100

var bl = 123

func main()  {
    println(&bl,bl)
    println(&cl,cl)
}
```

答:

> 考点:常量
>
> 结果：编译失败。
>
> 常量不同于变量的在运行期分配内存，常量通常会被编译器在预处理阶段直接展开，作为指令数据使用，报错：cannot take the address。

3.4 编译执行下面代码会出现什么?

```
package main
import "fmt"

func main()  {
    type MyInt1 int
    type MyInt2 = int
    var i int = 9
    var i1 MyInt1 = i
    var i2 MyInt2 = i
    fmt.Println(i1,i2)
}
```

答:

> 考点：type起别名
>
> 基于一个类型创建一个新类型，称之为defintion；基于一个类型创建一个别名，称之为alias。MyInt1为称之为defintion，虽然底层类型为int类型，但是不能直接赋值，需要强转； MyInt2称之为alias，可以直接赋值。
>
> 结果:编译失败。报错：annot use i (type int) as type MyInt1 in assignment。

3.5 string和[]byte的区别

答:

> 考点：string和[]byte的区别
>
> 结果:

> 共同点：可以互相转化、都可以通过下标索引
>
> 不同点：
> 1、[]byte可以通过下标修改值,string不可以
>
> 2、string可以比较，[]byte不可以比较,所以[]byte不能作为map的key值
>
> 3、[]byte在传输性能方面要比string好
>
> 4、string的值不可以为nil，所以如果需要nil的特性，就得用[]byte

4、指针

4.1 是否可以编译通过？如果通过，输出什么？

```go
func main() {
  list := new([]int)
  list = append(list, 1)
  fmt.Println(list)
}
```

答：

> 考点：new和make
>
> 结果：编译失败。
>
> 切片指针的解引用。
>
> 可以使用list:=make([]int,0) list类型为切片
>
> 或使用*list = append(*list, 1) list类型为指针

4.2 make和new的区别

答：

> 考点：new和make
>
> 结果：
>
> > 二者都是内存的分配（堆上），但是make只用于slice、map以及channel的初始化（非零值）；而new用于类型的内存分配，并且内存置为零。所以在我们编写程序的时候，就可以根据自己的需要很好的选择了。
> >
> > make返回的还是这三个引用类型本身；而new返回的是指向类型的指针。

5、interface

5.1 以下代码输出什么?

```
func main() {
  i := GetValue()

  switch i.(type) {
  case int:
    println("int")
  case string:
    println("string")
  case interface{}:
    println("interface")
  default:
    println("unknown")
  }
}

func GetValue() int {
  return 1
}
```

答:

> 考点:interface{}类型和函数类型
>
> 结果:编译失败。
>
> 因为type只能使用在interface

5.2 是否可以编译通过?如果通过,输出什么?

```
func Foo(x interface{}) {
  if x == nil {
    fmt.Println("empty interface")
    return
  }
  fmt.Println("non-empty interface")
}
func main() {
  var x *int = nil
  Foo(x)
}
```

答:

> 考点:interface内部结构

> 结果：non-empty interface

5.3 ABCD中哪一行存在错误?

```
type S struct {
}

func f(x interface{}) {
}

func g(x *interface{}) {
}

func main() {
  s := S{}
  p := &s

  f(s) //A
  g(s) //B
  f(p) //C
  g(p) //D

}
```

答:

> 考点：interface
>
> 结果：B和D错误。
>
> 看到这道题需要第一时间想到的是Golang是强类型语言，interface是所有golang类型的父类， 函数中`func f(x interface{})`的`interface{}`可以支持传入golang的任何类型，包括指针，但是函数`func g(x *interface{})`只能接受`*interface{}`.

6、函数和闭包

6.1 下面函数有什么问题?

```
func funcMui(x,y int)(sum int,error){
    return x+y,nil
}
```

答:

> 考点：函数返回值命名
>
> 结果：编译出错。

> 在函数有多个返回值时，只要有一个返回值有指定命名，其他的也必须有命名。 如果返回值有有多个返回值必须加上括号； 如果只有一个返回值并且有命名也需要加上括号； 此处函数第一个返回值有sum名称，第二个未命名，所以错误。

6.2 是否可以编译通过？如果通过，输出什么？

```
func GetValue(m map[int]string, id int) (string, bool) {
  if _, exist := m[id]; exist {
    return "存在数据", true
  }
  return nil, false
}
func main()  {
  intmap:=map[int]string{
    1:"a",
    2:"bb",
    3:"ccc",
  }

  v,err:=GetValue(intmap,3)
  fmt.Println(v,err)
}
```

答:

> 考点：函数返回值类型
>
> 结果：编译失败。
>
> nil 可以用作 interface、function、pointer、map、slice 和 channel 的“空值”。但是如果不特别指定的话，Go 语言不能识别类型，所以会报错。

6.3 编译执行下面代码会出现什么？

```
package main

func main()  {

    for i:=0;i<10 ;i++  {
    loop:
        println(i)
    }
    goto loop
}
```

答:

> 考点：goto跳转语句

> 结果：编译失败。
>
> goto不能跳转到其他函数或者内层代码，报错：goto loop jumps into block starting at。

6.4 编译执行下面代码会出现什么?

```
package main

func test() []func()  {
    var funs []func()
    for i:=0;i<2 ;i++  {
        funs = append(funs, func() {
            println(&i,i)
        })
    }
    return funs
}

func main(){
    funs:=test()
    for _,f:=range funs{
        f()
    }
}
```

答:

> 考点：闭包延迟求值 for循环复用局部变量i，每一次放入匿名函数的应用都是同一个变量。
>
> 结果:
>
> ```
> 0xc042046000 2
> 0xc042046000 2
> ```
>
> 如果想不一样:
>
> ```
> func test() []func()  {
> var funs []func()
> for i:=0;i<2 ;i++  {
>   x:=i
>   funs = append(funs, func() {
>       println(&x,x)
>   })
> }
> return funs
> }
> ```
>
> 结果:
> 0xc000096000 0

```
0xc000096008 1
```

6.5 编译执行下面代码会出现什么?

```
package main

func test(x int) (func(),func())  {
    return func() {
        println(x)
        x+=10
    }, func() {
        println(x)
    }
}

func main()  {
    a,b:=test(100)
    a()
    b()
}
```

答案:

> 考点：闭包引用相同变量 结果：
>
> ```
> 100
> 110
> ```

6.6 输出什么?

```
package main

import (
  "fmt"
)

func main() {
  fmt.Println(len("你好bj!"))
}
```

答:

> 考点：len函数的返回值，编码长度。
>
> 一个中文占三个字节。
>
> 结果：9

6.7 实现函数

```
func main(){
    var num interface{} = 1.2345
    //使用reflect输出类型与值

}
```

答：

> 考点：反射
>
> 结果：
>
> ```
> func main(){
>   var num float64 = 1.2345
>   //使用reflect输出类型与值
> //类型
>   fmt.Println(reflect.TypeOf(num))
> //值
>   fmt.Println(reflect.ValueOf(num))
> }
> ```

6.8 编写一个函数，实现字符串反转，函数输入类型为string

示例：锄禾日当午 转换后 午当日禾锄

答：

> 考点：函数实现
>
> 结果：
>
> ```
> func reverse(str string) string {
> rs := []rune(str)
> len := len(rs)
> var tt []rune
>
> tt = make([]rune, 0)
> for i := 0; i < len; i++ {
>   tt = append(tt, rs[len-i-1])
> }
> return string(tt[0:])
> }
> func main(){
>  str:="锄禾日当午"
>  reverse(str)
> }
> ```

2、面向对象

1、封装(方法)

1.1 以下代码能编译过去吗？为什么？

```
package main
import ("fmt")
type People interface {
    Speak(string) string
}
type Student struct{}
func(stu*Student)Speak(think string)(talk string) {
if think == "法师" {
        talk = "法师，我爱你哟～"
    } else {
        talk = "hi"
    }
    return
}
func main() {
    var peo People = Student{}
    think := "法师"
   fmt.Println(peo.Speak(think))
}
```

答：

> 考点：golang的方法集
>
> 结果： 编译不通过！golang的方法集仅仅影响接口实现和方法表达式转化，与通过实例或者指针调用方法无关。

1.2 写一个方法,输入n,输出第n个斐波那契数

答：

> 考点：方法定义和使用
>
> 结果：
>
> ```
> type Int int
> func (n Int)fibonacci() (res Int) {
>   if n <= 1 {
>     res = 1
>   } else {
>     res = (n-1).fibonacci() + (n-2).fibonacci()
>   }
>   return
> }
> func main(){
> ```

```
    var n Int=11
    v:=n.fibonacci()
    fmt.Println(v)
}
```

1.3 方法中，方法的接受者类型，带与不带*的区别

答:

> 考点：方法定义和使用
>
> 结果：
>
> ```
> （1）结构指针接收者，顾名思义，会在方法内部改变该结构内部变量的值。
> （2）结构值接收者，在方法内部对变量的改变不会影响该结构。
> （3）对于指针接收者，如果你调用的是值方法，即使你是指针调用者，也不会改变你的结构内的变量值。
> （4）对于值接收者，如果你调用的是指针方法，即使你是值调用者，也会改变你的结构内的变量值。
> ```

2、继承（匿名字段）

2.1 go语言中是如何实现面向对象的继承的？请写出一个例子说明

答:

> 考点：面向对象的继承（匿名字段实现继承关系）
>
> 实例: 面向对象的计算器实现
>
> ```
> 1.定义父类
> 2.定义子类,以及子类的方法 运算实现
> 3.定义接口，归纳 子类方法
> 4.定义空类，定义空类的方法,即 工厂模式，将 运算符 与 数值 分开处理，以运算符来分发方法，方便调用
> 5.定义一个多态，将接口归纳，方便调用
> 6.主函数，初始化，调用工厂模式，进行验证
> ```

```
package main

import "fmt"

 //父类
 type BaseNum struct {
     num1 int
     num2 int
 }
```

```go
//加法子类
type Add struct {
    BaseNum
}

//减法子类
type Sub struct {
    BaseNum
}

//子类方法
func (a *Add)Opt() int {
    return a.num1 + a.num2
}

func (s *Sub)Opt() int {
    return s.num1 - s.num2
}

//定义接口，即封装

type Opter interface {
    Opt() int
}

//定义多态
func MultiState(o Opter) int{
    value:=o.Opt()
    return value
}

//定义空类 以产生 工厂模式 的方法
type Factory struct {

}

func (f *Factory)FacMethod(a,b int,operator string) (value int){
    var i Opter
    switch operator {
    case "+":
            var AddNum Add = Add{BaseNum{a,b}}
            i = &AddNum
    case "-":
            var SubNum Sub = Sub{BaseNum{a,b}}
            i = &SubNum
    }
    //接口实现 : value = i.Opt()
    value = MultiState(i) //多态实现
    return
```

```
 }

func main() {
    var a Factory
    value := a.FacMethod(20,3,"-")
    fmt.Println(value)
}
```

2.2 下面代码会输出什么?

```
type People struct{}
func (p *People)ShowA() {
    fmt.Println("showA")
    p.ShowB()
}
func(p*People)ShowB() {
    fmt.Println("showB")
}
typeTeacher struct {
    People
}
func(t*Teacher)ShowB() {
    fmt.Println("teachershowB")
}
func main() {
    t := Teacher{}
    t.ShowA()
}
```

答:

> 考点：go的组合继承 这是Golang的组合模式，可以实现OOP的继承。 被组合的类型People所包含的方法虽然升级成了外部类型Teacher这个组合类型的方法（一定要是匿名字段），但它们的方法(ShowA())调用时接受者并没有发生变化。 此时People类型并不知道自己会被什么类型组合，当然也就无法调用方法时去使用未知的组合者Teacher类型的功能。 结果： showA showB

2.3 编译执行下面代码会出现什么?

```
package main

import "fmt"

type T1 struct {
}
func (t T1) m1(){
    fmt.Println("T1.m1")
}
type T2 = T1
type MyStruct struct {
```

```
    T1
    T2
}
func main() {
    my:=MyStruct{}
    my.m1()
}
```

答:

> 考点：继承关系和type。
>
> 结果：编译失败。
>
> 是不能正常编译的,异常：ambiguous selector my.m1
>
> 结果不限于方法，字段也也一样；也不限于type alias，type defintion也是一样的，只要有重复的方法、字段，就会有这种提示，因为不知道该选择哪个。 改为:
>
> my.T1.m1()
>
> my.T2.m1()

3、多态（接口）

3.1 写一个简单的示例代码，实现面向对象的多态

答:

> 考点：面向对象多态实现

```
type API interface{
    show()
}
type Person struct{}
type Student struct{}
func (*Person)show(){
    fmt.Println("这是Person类实现了接口的show方法")
}
func (*Student)show(){
    fmt.Println("这是Student类实现了接口的show方法")
}
func display(i API){
    i.show()
}
func main(){
    display(&Person{}) // 这是Person类实现了接口的show方法
    display(&Student{}) // 这是Student类实现了接口的show方法
}
```

3、异常处理

1、panic

1.1 写出下列程序输出内容

```
package main
import (
"fmt"
)
func main() {
    defer_call()
}
func defer_call() {
    defer func() {fmt.Println("打印前")}()
    defer func() {fmt.Println("打印中")}()
    defer func() {fmt.Println("打印后")}()
    panic("触发异常")
}
```

答:

> 考点：panic异常处理
>
> panic 需要等defer 结束后才会向上传递。 出现panic恐慌时候，会先按照defer的后入先出的顺序执行，最后才会执行panic。
>
> 结果:
>
> 打印后 打印中 打印前 panic: 触发异常

1.2 哪些情况会导致panic，如何预防panic导致程序崩溃

答:

> 考点：panic异常处理
>
> 结果:
>
> ```
> panic的产生：数组访问越界，空指针引用，除数为0。
> panic一般会导致 程序挂掉，除非使用recover捕获异常
> 异常捕获：在函数的开始使用defer函数里调用内置函数recover，recover会使程序从panic中恢复，并返回panic value. 导致panic的函数不会继续运行，但能正常返回. 在为发生panic时调用recover会返回nil。
> ```

2、defer

2.1 写出一下程序运行的结果

```
package main
import (
"fmt"
)
func main() {
    defer_call()
}
func defer_call() {
    defer func() {fmt.Println("大")}()
    defer func() {fmt.Println("法")}()
    defer func() {fmt.Println("师")}()
}
```

答:

> 考点：defer和函数组合调用方式
>
> 结果：
>
> 师
>
> 法
>
> 大

2.2 下面代码输出什么?

```
package main

import "fmt"

func main(){
  var name="zhangsan"
  fmt.Println(name)
  defer fmt.Println(name)
  name="lisa"
  fmt.Println(name)
  defer fmt.Println(name)
}
```

答:

> 考点：defer和函数组合调用方式
>
> 结果：
>
> zhangsan lisa lisa zhangsan

4、设计模式

1、单例

1.1 实现一个单例

答：

> 考点：单例设计模式
>
> 结果：

```
package main

import "sync"

// 实现一个单例

type singleton struct{}

var ins *singleton
var mu sync.Mutex

//懒汉加锁:虽然解决并发的问题，但每次加锁是要付出代价的
func GetIns() *singleton {
  mu.Lock()
  defer mu.Unlock()

  if ins == nil {
    ins = &singleton{}
  }
  return ins
}

//懒汉双重锁:避免了每次加锁，提高代码效率
func GetIns1() *singleton {
  if ins == nil {
    mu.Lock()
    defer mu.Unlock()
    if ins == nil {
      ins = &singleton{}
    }
  }
  return ins
}

//sync.Once实现
var once sync.Once
```

```
func GetIns2() *singleton {
  once.Do(func() {
    ins = &singleton{}
  })
  return ins
}
```

2、工厂

5、并发编程

1、select

1.1 Select的语句分为几种？分别是什么？Select的执行顺序是怎样的？

答：

> 考点：select使用方式
>
> 结果：

```
1.超时处理

select {
case str := <- resultChan:
   fmt.Println("receive str", str)
case <- time.After(time.Second * 5):
   fmt.Println("timeout!!")
}

2. 退出
select {
   case <- quitChan:
       cleanUp()
       return
   default:
}

3.判断channel是否阻塞
var ch chan int = make(chan int, 5)
select {
   case ch <- data:
        fmt.Println("add success")
```

```
    default: //channel满了
}
```

select{}是一个没有任何case的select，它会一直阻塞
select 是 Go 提供的一个关键字。可以监听channel上的数据流动。
select的语法与switch类似。每个选择条件由case来描述。
每次执行一个case分支
通常将select放置到循环中去
每个case语句里必须是一个IO操作。
select 通常对应一个异步事件处理。
可以利用select来设置超时。
如果监听中的case不满足--当前case阻塞
如果监听中的case同时有多个满足，select选择任意一个执行
select语法中的default是在所有case不满足情况的条件下，设置的默认处理动作。通常不设置，防止忙轮询消耗系统资源。
break只能跳出一个case分支不能跳出select外面的for

1.2 下面代码输出什么?

```
package main

import (
  "runtime"
  "fmt"
)

func main()  {
  runtime.GOMAXPROCS(1)
  int_chan:=make(chan int,1)
  string_chan:=make(chan string,1)
  int_chan<- 1
  string_chan<- "hello"
  select {
  case value:=<-int_chan:
  fmt.Println(value)
  case value:=<-string_chan:
    panic(value)
  }
}
```

答:

考点：select的随机性

结果：随机输出panic或者1

select会随机选择一个可用通用做收发操作。 所以代码是有肯触发异常，也有可能不会。 单个chan如果无缓冲时，将会阻塞。但结合 select可以在多个chan间等待执行。有三点原则：

> select 中只要有一个case能return，则立刻执行。 当如果同一时间有多个case均能return则伪随机方式抽取任意一个执行。 如果没有一个case能return则可以执行”default”块

1.3 下面这个程序的现象

```
func main(){
    select{}
}
```

答:

> 考点：select 阻塞。
>
> 结果：select{}是一个没有任何case的select，它会一直阻塞。

1.4 如果多个case分支同时处于ready状态，那么select的执行情况是怎样的?

答:

> 考点：select 随机性。
>
> 结果：
>
> ```
> 如果监听中的case不满足,当前case阻塞。
> 如果监听中的case同时有多个满足，select选择任意一个执行。
> ```

2、channel

2.1 有缓冲channel与无缓冲channel的区别

答:

> 考点：channel
>
> 结果：
>
> ```
> 无缓冲channel：
>
> 接收前没有能力保存任何数据的通道。要求发送goroutine和接收goroutine同时准备好，才能完成发送和接收操作。
>
> 否则，通道会导致先执行发送或接收操作的 goroutine 阻塞等待。
>
> 通常对应 同步操作。会发生阻塞。
>
> 有缓冲channel：
>
> 在接收前能存储一个或者多个数据值的通道。不强制要求 goroutine 之间必须同时完成发送和接收。
> ```

只有通道中没有要接收的值时，接收端才会阻塞。通道没有可用缓冲区容纳被发送的数据时，发送端才会阻塞。

通常对应 异步操作。会发生阻塞。

2.2 列举golang协程间通信常见的几种方法

答：

考点：channel和context中cancel函数

结果：

使用channel机制，每个goroutine传一个channel进去然后往里写数据，在再主线程中读取这些channel，直到全部读到数据了子goroutine也就全部运行完了，那么主goroutine也就可以结束了。这种模式是子线程去通知主线程结束。

使用context中cancel函数，这种模式是主线程去通知子线程结束。
sync.WaitGroup模式，Add方法设置等待子goroutine的数量，使用Done方法设置等待子goroutine的数量减1，当等待的数量等于0时，Wait函数返回。

3、goroutine

3.1 goroutine与协程的区别

答：

考点：goroutine

结果：

goroutine是协程的go语言实现，相当于把别的语言的类库的功能内置到语言里。

不同的是：
Golang在runtime，系统调用等多方面对goroutine调度进行了封装和处理，即goroutine不完全是用户控制，一定程度上由go运行时（runtime）管理。

好处：当某goroutine阻塞时，会让出CPU给其他goroutine。

3.2 goroutine连接池

答：

考点：goroutine 连接池

结果：

```
package main
```

```
const poolSize = 10
func main() {
  // 初始化创建poolSize连接池的
  channel pool := make(chan redis.Conn, poolSize)
  // 通过goroutine并发执行1000个协程
  for i:=0; i<1000; i++ {
  go func() {
    // 先从连接池中取出一个连接，如果取不到则挂起等待
    rcon := <- pool
    // 处理逻辑
    rcon.dosomething
    // 完成后将链接放回连接池
    pool <- rcon
    }()
  }
}
```

3.3 go语言怎么实现并发，并发的管理方式有哪些?

答:

> 考点：goroutine
>
> 结果：
>
> 使用 go 关键字用来创建 goroutine 。将go声明放到一个需调用的函数之前，在相同地址空间调用运行这个函数，这样该函数执行时便会作为一个独立的并发线程。这种线程在Go语言中称作goroutine。

```
//go 关键字放在方法调用前新建一个 goroutine 并执行方法体
go GetThingDone(param1, param2);

//新建一个匿名方法并执行
go func(param1, param2) {
}(val1, val2)

//直接新建一个 goroutine 并在 goroutine 中执行代码块
go {
 //do someting...
}
```

6、垃圾回收和性能调优

1、垃圾回收（gc）

1.1 go中的垃圾回收机制原理

答:

> 考点：垃圾回收

> 结果：
>
> ```
> 1、引用计数法
>
> 原理是在每个对象内部维护一个整数值，叫做这个对象的引用计数，当对象被引用时引用计数加一，当对象不被引用时引用计数减一。当引用计数为 0 时，自动销毁对象。另外的缺陷是，每次对象的赋值都要将引用计数加一，增加了消耗。
>
> 2、Mark-Sweep法（标记清除法）
>
> 这个算法分为两步，标记和清除。
> 标记：从程序的根节点开始， 递归地 遍历所有对象，将能遍历到的对象打上标记。
> 清除：讲所有未标记的的对象当作垃圾销毁。
>
> 3、三色标记法
>
> 三色标记法是传统 Mark-Sweep 的一个改进，它是一个并发的 GC 算法。
> 原理如下，
> 一、首先创建三个集合：白、灰、黑。
> 二、将所有对象放入白色集合中。
> 三、然后从根节点开始遍历所有对象（注意这里并不递归遍历），把遍历到的对象从白色集合放入灰色集合。
> 四、之后遍历灰色集合，将灰色对象引用的对象从白色集合放入灰色集合，之后将此灰色对象放入黑色集合
> 五、重复 【四】直到灰色中无任何对象
> 六、收集所有白色对象（垃圾）
> ```

2、性能调优

2.1 go的性能分析优化

答：

> 考点：性能调优
>
> 结果：
>
> ```
> 用 go test 调试
> 如果代码使用了 Go 中 testing 包的基准测试功能，我们可以用 gotest 标准的 -cpuprofile 和 -memprofile 标志向指定文件写入 CPU 或 内存使用情况报告。
> 使用方式：go test -x -v -cpuprofile=prof.out -file x_test.go
> 编译执行 x_test.go 中的测试，并向 prof.out 文件中写入 cpu 性能分析信息。
> 用 pprof 调试
>
> 你可以在单机程序 progexec 中引入 runtime/pprof 包；这个包以 pprof 可视化工具需要的格式写入运行时报告数据。对于 CPU 性能分析。
> ```

7、go语言包和包管理

1、go语言包

1.1 包里首字母大小写区别

答:

> 考点：包里首字母大小写区别
>
> 结果：
>
> ```
> 包中成员以名称首字母大小写决定访问权限：
>
> public：首字母大写，可被包外访问。
>
> private：首字母小写，仅包内成员可以访问。
> ```

2、包管理

2.1 使用的第三方包, 如何管理(使用go vender)

答:

> 考点：包管理
>
> 结果：
>
> 为了能让项目继续使用这些依赖包，有这么几个办法：
>
> ```
> 将依赖包拷贝到项目源码树中，然后修改import
> 将依赖包拷贝到项目源码树中，然后修改GOPATH
> 在某个文件中记录依赖包的版本，然后将GOPATH中的依赖包更新到对应的版本(因为依赖包实际是个git库，可以切换版本)
> ```
>
> 简单来说，vendor属性就是让go编译时，优先从项目源码树根目录下的vendor目录查找代码(可以理解为切了一次GOPATH)，如果vendor中有，则不再去GOPATH中去查找。