函数

函数声明

- 基本声明
  - 每一个函数都会有函数名称，一组形参、可选的返回参数列表和函数体
  - Go函数入参类型声明与C、Java均不相同，参数名称在前，而参数类型在后，并且函数返回值可以添加可选的名称

```
func Name(parameters) (results) {
    body
}
```

```
func Square(x, y int) (result int){
    result = x * y
    return result
}
```

- 函数签名
  - 函数签名即函数类型，只要函数的形参和返回值的个数、顺序以及类型均相同的话，就认为是同一个函数类型，或者说，它们的函数签名相同
  - 函数名称、参数名称返回值名称并不会影响函数签名
- 函数参数
  - Go函数并没有默认参数支持，同时也不能使用具名参数传递
  - 支持变长参数的传递，所有参数将作为slice传入至函数内部
    - 并且，支持将一个数组或者是slice传递进支持变长参数的函数

```
func Sum(value ...int) int {
    sum := 0
    for _, v := range value {
        sum += v
    }
    return sum
}
```

```
arr := []int{1,2,3,4,5,6}
Sum(arr...)
```

  - 实参按值传递，函数接收到的其实是每个实参的拷贝
    - 基本数据类型、结构体以及数组传入函数数，函数实际的接收值为其拷贝
      - 数组的函数传递与C语言不同，C中数组名为指向数组第一个元素的指针，而Go数组则不同
      - 通常来讲，如果一个结构体本身占用较大的内存空间，采用指针则能有效地节省内存
    - 而对于slice、map、函数或者是通道，由于这些数据类型中包含了指针，则在函数内部的修改可能会间接地修改变量
- 函数返回值
  - Go允许函数返回多个值，这一点与C、Java完全不同
    - 由于C语言没有异常机制且无多返回值，所以只能采取函数返回错误码的方式来判断函数调用是否出错
  - 在接收函数返回时，有多少个返回值就用多少个变量接收。如果对其中一个返回值不感兴趣，可以使用`_`进行屏蔽

```
response, _ := http.Get("")
```

函数错误处理

- 错误
  - Go同C语言一样，没有诸如Python、Java一样的异常抛出/捕获机制，所以通常使用显式错误返回的方式来表明函数调用出错
    - 即Go函数通常会返回2个值: 正常调用返回的数据+错误信息
  - Go不同于C的地方在于没有全局的errno变量用于指示具体是何种错误信息，相反，Go使用error类型作为错误的返回值
    - 当返回的error不为nil时，表示函数调用出错，所以需要时刻地检查函数所返回的error参数

```
response, err := http.Get("")

if err != nil {
    return nil, err
}
```

  - error类型将在接口一节中详细描述，目前只需要知道可以使用fmt.Println(err)或fmt.Printf("%v", error)来输出错误信息
- 文件结束标识
  - 在I/O中有一种特殊的读取情况，就是读取到了文件末尾
  - Go则直接返回errors.New("EOF")来表示已读取至文件末尾

```
in := bufio.NewReader(os.Stdin)
for {
    r, _, err := in.ReadRune()
    if err == io.EOF {
        break
    }
    if err != nil {
        return nil, fmt.Errorf("Read failed: %v", err)
    }
}
```

将函数作为变量

- Go同Python一样，函数可以赋值给其它变量和作为参数传递进另一个函数中
  - 函数在Go语言中也是一等公民

```
func Multi(x, y int) int { return x * y }

func Add(x, y int) int { return x + y }

func main() {
    f := Multi
    fmt.Println(f(10, 20))

    f = Add
    fmt.Println(f(10, 20))
}
```

    - 只要函数签名相同，就可以随意的赋给同一变量
    - 从某种意义上来说，这就是基于函数的策略模式，或者说，部分的函数式编程
- 由于Go未有泛型的实现，所以若想要实现通用的map、reduce以及filter(均为Python内置函数)等小工具仍然比较麻烦

匿名函数

- 闭包
  - Go不允许函数的嵌套定义，但是可以通过匿名函数来实现闭包

```
func Squares() func() int {
    var x int
    return func() int {
        x++
        return x * x
    }
}
```

- Go匿名函数多用于goroutine以及延迟函数调用中，或者是在函数内部定义一匿名函数，但后者使用频率不高

延迟函数调用

- 保证资源的释放
  - 诸如文件等I/O在使用完毕后均需要进行关闭，在C和Java中由程序员手动执行关闭。在Python中，则可以使用上下文管理器由函数提供者在退出时关闭相关资源

```
with open("file", "r") as f:
    data = f.read()
# 上下文退出时即调用 __exit__ 方法关闭相关资源
```

  - **Go虽未提供类似于Python中的上下文管理器，但是提供了 defer 关键字，defer 后的函数调用将在函数结束(return 语句)之后调用**
    - 正因为如此，我们可以在打开一个文件流之后，紧跟 defer 语句来延迟关闭对应的资源，算是一种变相的提醒和便利

```
func Print() {
    defer fmt.Println("defer")
    fmt.Println("func body invoke")
}
```

    - 调用Print函数时，将首先打印"func body invoke"，而后才会打印出"defer"
  - defer常用于成对的操作，例如打开和关闭，连接和断开，加锁和解锁。defer语句使得这些成对操作在代码中能处于相同的位置，而不会像C语言一样，close()方法只能在资源使用完毕后才能调用
- 和闭包来实现装饰器

```
func RecordInvokeTime(funcName string) func() {
    start := time.Now()
    return func() {
        log.Printf("func %s spent %s", funcName, time.Since(start))
    }
}

func Simulation() {
    defer RecordInvokeTime("Simulation")()  // 不要忘记后面这对括号，defer接函数调用，而不是函数
    time.Sleep(2 * time.Second)
}
```

  - 虽然defer为延迟调用，但是也不是什么都不做，至少函数原型需要加载至栈中，那么此时就可以使用闭包来做一些额外的前置工作
- 改变函数返回值
  - defer所"装饰"的函数将会在return语句之后调用，那么就可以在defer函数中改变原有的函数返回值

```
func Add(x, y int) (result int) {
    defer func() {result += 1}()
    return x + y
}
```

  - 可以使用此特性来实现函数的代理模式，对原有代码改动较小，且可以随意拓展
- 多个defer的调用顺序

```
func Print() {
    defer fmt.Println("line 1")
    defer fmt.Println("line 2")
    defer fmt.Println("line 3")
}
```

  - **当一个函数存在多个defer语句时，其执行顺序遵循"先进后出"，也就是说，类似于栈调用**
- 一些小坑
  - **不在循环中直接使用defer**
  - 有时候程序可能会迭代的同步读取多个文件，在迭代中，不要直接使用defer调用，而是将defer调用转移至另一个函数中

```
func ReadAllFiles(filenames []string){
    for _, name := range filenames {
        f, _ := os.Open(name)  // 忽略错误处理
        defer f.Close()
    }
}
```

    - 由于defer会在return结束时调用，那么文件句柄f最终所指向的文件都是同一个文件，而不是循环中不同的文件。所以该函数仅关闭了最后一个文件描述符而已

宕机

- 尽管Go没有提供异常抛出/捕获机制，但是提供了宕机机制。当异常发生时，例如数组越界，解引用空指针等，Go将终止goroutine的执行，但是延迟函数将会执行，并异常退出打印出错误的堆栈信息

```
q := make([]string, 0)
fmt.Println(q[0])
// panic: runtime error: index out of range
```

- panic 中文释义为"恐慌"，"慌乱"，只有当程序完全无法运行时，才应主动使用panic来终止goroutine的运行

恢复

- panic将会直接导致goroutine异常退出，并打印出堆栈信息。但是在一些特殊情况中，例如Web服务器，应当返回有价值的信息
- **recover通常会在嵌入在defer中进行调用，因为不管怎么panic，延迟函数始终都会被调用**

```
func RecoverFunc() (err error) {
    defer func() {
        if p := recover(); p != nil {
            err = fmt.Errorf("internal error: %v", p)
        }
    }()
    q := make([]string, 0)
    fmt.Println(q[0]) // panic
    return nil
}
```