GO? GO!

- 使用书籍 Go程序设计语言

第一章

经典helloworld

-

```
package main // 该声明为声明自己所属在哪个包， 而不是引用， 声明为main的为一个独立的可执行程序
import "fmt" //导入声明

func main(){ //程序入口  go语言换行敏感
    // first program
    fmt.Println("Hello,World")

}
```

- gofmt, 自带的格式整理，vscode配置为保存后自动运行， 但是建议直接按照要求来写

命令行参数

- os.Args, 返回是一个字符串slice s, s[0] 为命令本身

```
// 实现将参数输出出来
package main

import (
    "fmt"
    "os"
)

func main() {
    var s, sep string // 变量生成时会被初始化为空值
    for i := 1; i < len(os.Args); i++ { // for initialization; condition; post {
        s += sep + os.Args[i]
        sep = " "
    }
    fmt.Println(s)
}
```

声明方式

1. 短变量声明， `s := ""`, 用来表示初始化比较重要， 但是通常在一个函数内部使用，不适合包级别的变量。
2. 隐式初始化， `var s string`， 用来表示初始化不甚重要
3. 空标识符， `_, arg := range os.Args[1:]`, 当返回参数中有我们不需要的值时， 应该用空标识符下划线代替， go中不允许存在未使用的变量

```
package main

import (
    "fmt"
    "os"
)

func main() {
    s, sep := " ", " "
    for _, arg := range os.Args[1:] {  // 空标识符
        s += sep + arg
        sep = " "
    }
    fmt.Println(s)
}
```

- strings.Join(slice, string) 以string链接各元素

```
// 练习1.1

package main

import (
    "fmt"
    "os"
)

func main() {
    fmt.Println(os.Args[0])
}
```

```
// 练习1.2

package main

import (
    "fmt"
```

```
    "os"
)

func main() {
    for idx, s := range os.Args[1:] {
        fmt.Println(idx, s)
    }
}
```

```
// 练习1.3 暂时不会
```

基本逻辑工作

```
// 输出标准输入中出现次数大于1的行，并统计次数

package main

import (
    "bufio" // 处理输入输出， Scanner 可以读取输入，以行或者单词为单位断开， 处理以行为单位的输入内容的最简单方式，
    "fmt"
    "os"
)

func main() {
    counts := make(map[string]int) // map 将string 映射到int ， 值得注意的是， 打印map的打印结果分布是随机的，设计目的是防止程序依赖某种特殊的序列，make 是map内置的函数，多种用途
    input := bufio.NewScanner(os.Stdin)
    for input.Scan() {
        counts[input.Text()]++
    }
    for line, n := range counts {
        if n > 1 {
            fmt.Printf("%d\t%s\n", n, line)
        }
    }
}
```

- 转义字符，Go语言一般称之为verb 比较不同的是%t bool型， %q 带引号字符串， %v 内置格式的任何值， %T 任何值的类型

从文件读入

```
// 打印输入中出现多次的行的个数和文本
// 可以支持文件列表读入或者stdin读入
package main

import (
    "bufio"
    "fmt"
    "os"
)

func main() {
    counts := make(map[string]int)
    files := os.Args[1:]
    if len(files) == 0{
        countLines(os.Stdin, counts)
    } else {
        for _, arg := range files {
            f, err := os.Open(arg)  //返回两个值，一个是 *os.File 另一个是err，nil
是内置的表示没有err
            if err != nil {
                fmt.Fprintf(os.Stderr, "dup2: %v\n", err)
                continue
            }
            countLines(f, counts)
            f.Close()
        }
    }
    for line, n := range counts {
        if n > 1 {
            fmt.Printf("%d\t%s\n", n, line)
        }
    }

}

func countLines(f *os.File, counts map[string]int){ //
    input := bufio.NewScanner(f)
    for input.Scan() {
        counts[input.Text()] ++  // 这里传map 是传的副本，在函数内做修改原来的map也会
变动
    }
}
```

- 以上方法一直使用流式输入,但是也有另一种方法,直接读入一大块进内存之后再分割

```
package main

import (
    "fmt"
```

```
    "os"
    "strings"
)

func main() {
    counts := make(map[string]int)
    for _, filename := range os.Args[1:] {
        data, err := os.ReadFile(filename)
        if err != nil {
            fmt.Fprintf(os.Stderr, "dup3: %v\n", err)
            continue
        }
        for _, line := range strings.Split(string(data), "\n"){ // 读入后划分
            counts[line]++
        }
    }
    for line, n := range counts{
        if n > 1{
            fmt.Printf("%d\t%s\n", n, line)
        }
    }
}
```

```
\\ 练习1.4
// 基本逻辑，先存起来每行的信息，然后对于每个文件，重复以下行动，首先对于每个句子统计,之后判断是否重复，之后消去影响
package main

import (
    "bufio"
    "fmt"
    "os"
)

func main() {
    counts := make(map[string]int)
    files := os.Args[1:]
    if len(files) > 0{
        for _, arg := range files {
            f, err := os.Open(arg)
            if err != nil {
                fmt.Fprintf(os.Stderr, "dup2: %v\n", err)
                continue
            }
            countLines(f, counts)
            for _, n := range counts{
                if n > 1{
                    fmt.Printf("%s\n", arg)
                    break
                }
```

```
            }
            f.Close()
            f, err = os.Open(arg)
            if err != nil {
                fmt.Fprintf(os.Stderr, "dup2: %v\n", err)
                continue
            }
            decountLines(f, counts)
            f.Close()
        }
    }

}

func countLines(f *os.File, counts map[string]int){
    input := bufio.NewScanner(f)
    for input.Scan() {
        counts[input.Text()] ++
    }
}

func decountLines(f *os.File, counts map[string]int){
    input := bufio.NewScanner(f)
    for input.Scan(){
        //print("!")
        counts[input.Text()] --
    }
}
```

GIF 动画

```
// 用于产生随机利萨茹图形的GIF动画
// 本地打不开，但是联网能打开
// 联网方式，命令后添加web选项,然后访问对应端口
package main

import (
    "image"
    "image/color"
    "image/gif" // 在导入多段路径组成的包后，使用路径的最后一段来引用这个包
    "io"
    "log"
    "math"
    "math/rand"
    "net/http"
    "os"
)

var palette = []color.Color{color.White, color.Black} //复合字面量 数组
const ( // 常量命名，数字,字符串,bool
```

```
    whiteIndex = 0
    blackIndex = 1
)

func main(){
    //rand.Seed(time.Now().UTC().UnixNano())
    if len(os.Args) > 1 && os.Args[1] == "web" {
        handler := func(w http.ResponseWriter, r *http.Request) {
            lissajous(w)
        }
        http.HandleFunc("/", handler)
        log.Fatal(http.ListenAndServe("localhost:8000", nil))
        return
    }
    lissajous(os.Stdout)
}

func lissajous(out io.Writer) {
    const (
        cycles = 5
        res = 0.001
        size = 100
        nframes = 64
        delay = 8
    )
    freq := rand.Float64() * 3.0
    anim := gif.GIF{LoopCount: nframes} // 复合字面量，结构体
    phase := 0.0
    for i := 0; i < nframes; i++{
        rect := image.Rect(0, 0, 2 * size + 1, 2 * size + 1)
        img := image.NewPaletted(rect, palette)
        for t := 0.0; t < cycles * 2 * math.Pi; t += res{
            x := math.Sin(t)
            y := math.Sin(t * freq + phase)
            img.SetColorIndex(size+int(x*size+0.5), size+int(y * size + 0.5),
blackIndex)
        }
        phase += 0.1
        anim.Delay = append(anim.Delay, delay)
        anim.Image = append(anim.Image, img)
    }
    gif.EncodeAll(out, &anim)
}
```

- 练习1.5

```
//

package main
```

```
import (
    "image"
    "image/color"
    "image/gif"
    "io"
    "log"
    "math"
    "math/rand"
    "net/http"
    "os"
)

var palette = []color.Color{color.RGBA{0x3D, 0x91, 0x40, 0xff}, color.Black} // 查找RGB颜色来设置对应颜色
const (
    whiteIndex = 0
    blackIndex = 1
)

func main() {
    if len(os.Args) > 1 && os.Args[1] == "web"{
        handler := func(w http.ResponseWriter, r *http.Request){
            lissajous(w)
        }
        http.HandleFunc("/", handler)
        log.Fatal(http.ListenAndServe("localhost:8000", nil))
        return
    }
    lissajous(os.Stdout)
}

func lissajous(out io.Writer){
    const(
        cycles = 5
        res = 0.001
        size = 100
        nframes = 64
        delay = 8
    )
    freq := rand.Float64() * 3.0
    anim := gif.GIF{LoopCount: nframes} // 复合字面量，结构体
    phase := 0.0
    for i := 0; i < nframes; i++{
        rect := image.Rect(0, 0, 2 * size + 1, 2 * size + 1)
        img := image.NewPaletted(rect, palette)
        for t := 0.0; t < cycles * 2 * math.Pi; t += res{
            x := math.Sin(t)
            y := math.Sin(t * freq + phase)
            img.SetColorIndex(size+int(x*size+0.5), size+int(y * size + 0.5),
blackIndex)
        }
        phase += 0.1
        anim.Delay = append(anim.Delay, delay)
        anim.Image = append(anim.Image, img)
```

```
    }
    gif.EncodeAll(out, &anim)
}
```

- 练习1.6

```
//

package main

import (
    "image"
    "image/color"
    "image/gif"
    "io"
    "log"
    "math"
    "math/rand"
    "net/http"
    "os"
)

var palette = []color.Color{color.RGBA{0x3D, 0x91, 0x40, 0xff}, color.RGBA{0x29,
0x24, 0x21, 0xff}}
const (
    whiteIndex = 0
    blackIndex = 1
)

func main() {
    if len(os.Args) > 1 && os.Args[1] == "web"{
        handler := func(w http.ResponseWriter, r *http.Request){
            lissajous(w)
        }
        http.HandleFunc("/", handler)
        log.Fatal(http.ListenAndServe("localhost:8000", nil))
        return
    }
    lissajous(os.Stdout)
}

func lissajous(out io.Writer){
    const(
        cycles = 5
        res = 0.001
        size = 100
        nframes = 64
        delay = 8
    )
    freq := rand.Float64() * 3.0
    anim := gif.GIF{LoopCount: nframes} // 复合字面量，结构体
```

```
    phase := 0.0
    for i := 0; i < nframes; i++{
        rect := image.Rect(0, 0, 2 * size + 1, 2 * size + 1)
        img := image.NewPaletted(rect, palette)
        tmp := uint8(1)
        for t := 0.0; t < cycles * 2 * math.Pi; t += res{
            x := math.Sin(t)
            y := math.Sin(t * freq + phase)
            img.SetColorIndex(size+int(x*size+0.5), size+int(y * size + 0.5), tmp)
        }
        phase += 0.1
        tmp += 1  // 轮流出现
        tmp %= 2
        anim.Delay = append(anim.Delay, delay)
        anim.Image = append(anim.Image, img)
    }
    gif.EncodeAll(out, &anim)
}
```

获取一个URL

```
package main

import (
    "fmt"
    "io"
    "net/http"
    "os"
)

func main() {
    for _, url := range os.Args[1:] {
        resp, err := http.Get(url)
        if err != nil {
            fmt.Fprintf(os.Stderr, "fetch: %v\n", err)
            os.Exit(1)
        }
        b, err := io.ReadAll(resp.Body) // ioutil包现在是io包
        resp.Body.Close() //关闭数据流来避免资源泄漏,
        if err != nil {
            fmt.Fprintf(os.Stderr, "fetch reading: %v\n", err)
            os.Exit(1)
        }
        fmt.Printf("%s", b)
    }
}
```

```
// 练习1.7
package main
```

```
import (
    "fmt"
    "io"
    "net/http"
    "os"
)

func main() {
    for _, url := range os.Args[1:] {
        resp, err := http.Get(url)
        if err != nil {
            fmt.Fprintf(os.Stderr, "fetch:%v\n", err)
            os.Exit(1)
        }
        _, err = io.Copy(os.Stdout, resp.Body)
        if err != nil {
            fmt.Fprintf(os.Stderr, "copy%v\n", err)
        }
        resp.Body.Close()
    }

}
```

```
\\ 练习1.8
package main

import (
    "fmt"
    "io"
    "net/http"
    "os"
    "strings"
)

func main() {
    for _, url := range os.Args[1:] {
        if ! strings.HasPrefix(url, "http://") && ! strings.HasPrefix(url,
"https://") {
            url = "http://" + url
        }
        resp, err := http.Get(url)
        if err != nil {
            fmt.Fprintf(os.Stderr, "fetch: %v\n", err)
            os.Exit(0)
        }
        io.Copy(os.Stdout, resp.Body)
        resp.Body.Close()
    }
}
```

```
\\ 练习1.9
package main

import (
    "fmt"
    "net/http"
    "os"
)

func main() {
    for _, url := range os.Args[1:] {
        resq, err := http.Get(url)
        if err != nil {
            fmt.Fprintf(os.Stderr, "fetch %v\n", err)
            os.Exit(0)
        }
        fmt.Printf("%v\n", resq.StatusCode) // 状态码， StatusCode
        resq.Body.Close()
    }
}
```

服务器

```
package main

import (
    "fmt"
    "log"
    "net/http"
)

func main() {
    http.HandleFunc("/", handler) // 使用handler函数 处理所有请求
    log.Fatal(http.ListenAndServe("localhost:8000", nil))// 先打印日志到标准输出，
调用os.exit(1)，到那时defer函数不会被调用
}

func handler(w http.ResponseWriter, r *http.Request){ // handler格式，
    fmt.Fprintf(w, "URL.Path = %q\n", r.URL.Path)

}
```

```
package main
```

```go
import (
    "fmt"
    "log"
    "net/http"
    "sync"
)

var mu sync.Mutex
var count int


func main() {
    http.HandleFunc("/", handler)
    http.HandleFunc("/count", counter)
    log.Fatal(http.ListenAndServe("localhost:8000", nil)) // Listen
}

/*
出现问题， 使用浏览器访问的时候会调用两次接口
原因是图标也算一次
不用浏览器访问就好啦


*/


func handler(w http.ResponseWriter, r *http.Request){
    mu.Lock()
    count++
    fmt.Fprintf(w, "Count %d ffff \n", count)
    mu.Unlock()
}

func counter(w http.ResponseWriter, r *http.Request){
    mu.Lock()
    fmt.Fprintf(w, "Count %d\n", count)
    mu.Unlock()
}
```

```go
// 更完整的例子， 报告接收到的消息头和表单数据
package main

import (
    "fmt"
    "log"
    "net/http"
)


func main(){
```

```
    http.HandleFunc("/", handle)
    log.Fatal(http.ListenAndServe("localhost:8000", nil))
}

func handle(w http.ResponseWriter, r *http.Request){ // 前者输出， 后者输入

    fmt.Fprintf(w, "%s %s %s\n", r.Method, r.URL, r.Proto)
    for k, v := range r.Header{
        fmt.Fprintf(w, "Header[%q] = %q\n", k, v)
    }
    fmt.Fprintf(w, "Host = %q\n", r.Host)
    fmt.Fprintf(w, "RemoteAddr = %q\n", r.RemoteAddr)
    if err := r.ParseForm(); err != nil{ // 将err := r.ParseForm() 嵌入到if 判断条
件前， 作用域缩小
        log.Print(err)
    }
    for k, v := range r.Form {
        fmt.Fprintf(w, "Form[%q] = %q\n", k, v)
    }

}
```

http.Request 包含内容

- Header type Header map[string][]string
- Body 请求体
- GetBody 返回Body的新副本
- ContentLength int64 关联内容长度
- TransferEncoding []string 列出了从最外层到最内层的传输编码， 一般会被忽略，当发送或者接受请求时，会自动添加或者移除“chunked“传输编码
- Close bool 连接结束后是否关闭
- Host string 服务器主机地址
- Form url.Values 表单数据
- PostForm url.Values 也是表单
- MultipartForm *multipaort.Form 解析多部分表单
- Trailer Header 表示在请求体后添加附加头
- RemoteAddr string
- RequestURL string
- TLS *tls.ConnectionState
- Cancel <-chan struct{} 一个可选通道
- Response * Response 此请求的重定向响应

控制流

- if for switch(switch 不需要加break) 可以通过加fallthrough来连接到下一级
- switch 可以允许不带对象
- switch 可以紧跟简短变量声明

go doc

- 可以在本地直接命令行阅读标准库的文档。
- 建议：在源文件的开头写注释， 每一个函数之前写一个说明函数行为的注释， 容易使得被godoc这样的工具检测到

go-kit 基础服务

Others

概念完整性

- 概念的完整性，是指针对于一个领域，不仅了解该领域的所有对象，并且了解所有对象之间的关系。
- 了解所有对象之间的关系，并不是感性了解，而是理性了解，并不是将所有的信息都知道就可以了，需要达到一定的理性认识，达到一定的抽象才行。

第二章

- 实体的第一个字母的大小写决定其可见性是否跨包， 如果是大写开头， 说明是导出的， 可以被自己包之外的其他程序所调用
- 包名称永远是小写纯字母
- 名称的作用域越大，就使用越长且更有意义的名称
- 驼峰式命名法，首字母缩写词往往使用相同的大小写
- go中不允许出现未被定义的变量， 所有类型的变量都应当有直接可用的零值

```
package main

import "fmt"

func main() {
    const freezingF, boilingF = 32.0, 212.0
    fmt.Printf("freezing %g C\n",fToC(freezingF))
    fmt.Printf("boiling %g C\n", fToC(boilingF))
}


func fToC(f float64) float64{
    return (f -32) * 5 / 9
}
```

```
package main

import "fmt"

const boilingF = 212.0

func main() {
```

```
    var f = boilingF
    var c = (f - 32) * 5 / 9
    fmt.Printf("boiling point = %g F or %g C\n", f, c)
}
```

```
// 第四版
package main

import (
    "flag"
    "fmt"
    "strings"
)

var n = flag.Bool("n", false, "omit trailing newline")
var sep = flag.String("s", " ", "separator")


func main(){
    flag.Parse()
    fmt.Print(strings.Join(flag.Args(), *sep))
    if !*n {
        fmt.Println()
    }
}
```

- flag包简介：https://www.cnblogs.com/sparkdev/p/10812422.html

类型声明

- type name underlying-type
- 一般会放在函数外面全包使用， 若首字母大写则可导出包外

```
// 进行摄氏温度和华氏温度的转换
package main


type Celsius float64
type Fahrenheit float64

const (
    AbsoluteZeroC Celsius = -273.15
    FreezingC Celsius = 0
    BoilingC Celsius = 100
)
```

```
func CToF(c Celsius) Fahrenheit { return Fahrenheit(c * 9 / 5 + 32)}  // 构造时若两个底层是相同类型可以直接构造
func FtoC(f Fahrenheit) Celsius { return Celsius((f - 32) * 5 / 9)}
```

- 命名类型之后类似于继承，可以重新定义类型的行为， 类似于下面

```
func (c Celsius) String() string() {return fmt.Sprintf("%g°C", c)} // fmt 在将元素输出时，会优先调用函数的toString () 方法
```

包

- 每个包对应一个独立的命名空间， 需要明确指出包来调用， 只有名字以大写字母开头的信息才是导出的， （汉字不导出）
- 可以将之前的代码分成两个文件， 并且导出包

```
// 用于进行摄氏度与华氏度之间的转换    tempconv.go
package tempconv


type Celsius float64
type Fahrenheit float64

const (
    AbsoluteZeroC Celsius = -273.15
    FreezingC Celsius = 0
    BoilingC Celsius = 100
)
```

```
package tempconv // conv.go

// 摄氏度转华氏度
func CToF(c Celsius) Fahrenheit { return Fahrenheit(c * 9 / 5 + 32)}  // 构造时若两个底层是相同类型可以直接构造

// 华氏度转摄氏度
func FtoC(f Fahrenheit) Celsius { return Celsius((f - 32) * 5 / 9)}
```

- flag

```
// 练习2.1 注意函数复用
// 进行摄氏温度和华氏温度以及绝对温度的转换
package main
```

```
type Celsius float64
type Fahrenheit float64
type Kelvin float64

const (
    AbsoluteZeroC Celsius = -273.15
    FreezingC Celsius = 0
    BoilingC Celsius = 100
)

func CToF(c Celsius) Fahrenheit { return Fahrenheit(c * 9 / 5 + 32)}  // 构造时若两个底层是相同类型可以直接构造
func FtoC(f Fahrenheit) Celsius { return Celsius((f - 32) * 5 / 9)}
func KtoC(k Kelvin) Celsius {return Celsius(k + Kelvin(AbsoluteZeroC))}
func KtoF(k Kelvin) Fahrenheit {return Fahrenheit(CToF(KtoC(k)))}
func CtoK(c Celsius) Kelvin {return Kelvin(c - AbsoluteZeroC)}
func FtoK(f Fahrenheit) Kelvin {return CtoK(FtoC(f))}
```

导入包

```
// 导入tempconv包
package main

import (
    "fmt"
    "os"
    "strconv"

    "./learning/tempconv" // go 调用不同位置的包 , https://blog.csdn.net/Working_hard_111/article/details/139982343
)

func main(){
    for _, arg := range os.Args[1:]{
        t, err := strconv.ParseFloat(arg, 64)
        if err != nil {
            fmt.Fprintf(os.Stderr, "cf: %v\n", err)
            os.Exit(1)
        }
        f := tempconv.Fahrenheit(t)
        c := tempconv.Celsius(t)
        fmt.Printf("%s = %s, %s = %s\n", f, tempconv.FtoC(f), c, tempconv.CToF(c))
    }
}
```

- 包的初始化， 使用init（）函数， 该函数不能被调用或者引用， 每个文件中init初始化函数在程序执行的时候直接调用

```
// 用来统计输入数的二进制1数目
package popcount

var pc [256]byte

func init(){
    for i := range pc { // 直接可以将slice当参数
        pc[i] = pc[i / 2] + byte(i & 1) // byte 可以返回1的个数，pc[i] 表示数字i 二
进制时1的位置个数
    }
}

func PopCount(x uint64) int{
    return int(pc[byte(x >> (0 * 8))] +
        pc[byte(x >> (1 * 8))] +
        pc[byte(x >> (2 * 8))] +
        pc[byte(x >> (3 * 8))] +
        pc[byte(x >> (4 * 8))] +
        pc[byte(x >> (5 * 8))] +
        pc[byte(x >> (6 * 8))] +
        pc[byte(x >> (7 * 8))])
}
```

练习2.3

- 重写PopCount函数，用一个循环代替单一的表达式。比较两个版本的性能。（11.4节将展示如何系统地比较两个不同实现的性能。）

```
// 用来统计输入数的二进制1数目
package popcount

var pc [256]byte

func init(){
    for i := range pc { // 直接可以将slice当参数
        pc[i] = pc[i / 2] + byte(i & 1) // byte 可以返回1的个数，pc[i] 表示数字i 二
进制时1的位置个数
    }
}

func PopCount(x uint64) int{

    ans := 0
    for i := 0 ; i < 8; i++ {  // 写成循环形式
        ans += int(byte(x >> (i * 8)))
    }
    return ans
}
```

练习2.4

- 用移位算法重写PopCount函数，每次测试最右边的1bit，然后统计总数。比较和查表算法的性能差异。

```
// 用来统计输入数的二进制1数目
package popcount

var pc [256]byte

func init(){
    for i := range pc { // 直接可以将slice当参数
        pc[i] = pc[i / 2] + byte(i & 1) // byte 可以返回1的个数，pc[i] 表示数字i 二进制时1的位置个数
    }
}

func PopCount(x uint64) int{

    ans := 0
    for ; x != 0 ; x >>= 1{    // 每次右移一位
        if x & 1 == 1{
            ans ++
        }
    }
    return ans
}
```

练习2.5

- 表达式x&(x-1)用于将x的最低的一个非零的bit位清零。使用这个算法重写PopCount函数，然后比较性能。

```
// 用来统计输入数的二进制1数目
package popcount

var pc [256]byte

func init(){
    for i := range pc { // 直接可以将slice当参数
        pc[i] = pc[i / 2] + byte(i & 1) // byte 可以返回1的个数，pc[i] 表示数字i 二进制时1的位置个数
    }
}

func PopCount(x uint64) int{

    ans := 0
    for ; x != 0 ; x = x & (x - 1){    // x - lowbit(x)
            ans ++
```

```
    }
    return ans
}
```

作用域

- 作用域不等于生命周期， 作用域是编码阶段的概念，生命周期是运行时的概念
- go 中编译器会一层层地向外搜寻合适的范围，
- for, if, switch 会产生新的词法域
- 这个部分要注意好的编码习惯， 尽量不用相同的变量名， 但是go是允许使用相同的变量名的

第三章

- Go语言数据类型分为四类： 基础类型、复合类型、引用类型和接口类型
- 基础类型： 数字、 字符串、 bool型、
- 复合类型：数组、结构体、
- 引用类型： 指针、 切片、 字典、 函数、 通道
- 接口类型： 第七章

数据类型

- Go 在运算时要求比较严格，只允许相同类型的进行运算
- 整型分为 有符号和无符号， 每种都分为 8，16，32，64位
- 还有一种对应CPU平台的类型， int和 uint
- 还用一种无符号的整数类型uintptr, 没有具体的bit大小但是足以容纳指针
- 浮点数转整数的方式是丢弃小数部分， 然后向数轴方向折断
- 浮点数只有两种， float32 和float64
- Nan 的比较总是不成立， 但是！= 会成立
- 浮点数输出可以有%e 科学计数法， %f 小数点， 两种方法， 使用%g可以自动生成

运算符

- 基本上和c++ 相同
- &^ 为位清空操作

实例

```
package main

import (
    "fmt"
    "math"
)

const (
    width, height = 600, 320
    cells         = 100
```

```
    xyrange       = 30.0
    xyscale       = width / 2 / xyrange
    zscale        = height * 0.4
    angle         = math.Pi / 6
)
var sin30 = math.Sin(angle) // Go的常量是在编译之前就能确定的常量
var cos30 = math.Cos(angle)



func main() {
    fmt.Printf("<svg xmlns='http://www.w3.org/2000/svg' "+
        "style='stroke: grey; fill: white; stroke-width: 0.7' "+
        "width='%d' height='%d'>", width, height)
    for i:= 0; i < cells; i++ {
        for j := 0; j < cells; j++ {
            ax, ay := corner(i + 1, j)
            bx, by := corner(i, j)
            cx, cy := corner(i, j + 1)
            dx, dy := corner(i + 1, j + 1)
            fmt.Printf("<polygon points='%g,%g %g,%g %g,%g %g,%g'/>\n",
                ax, ay, bx, by, cx, cy, dx, dy)
        }
    }
    fmt.Println("</svg>")
}

func corner(i, j int) (float64, float64) { // 返回网格顶点的坐标参数
    x := xyrange * (float64(i) / cells - 0.5)
    y := xyrange * (float64(j) / cells - 0.5)
    z := f(x, y)
    sx := width / 2 + (x - y) * cos30 * xyscale
    sy := height / 2 + (x + y) * sin30 * xyscale - z * zscale
    return sx, sy
}

func f(x, y float64) float64 {
    r := math.Hypot(x, y)
    return math.Sin(r) / r
}
```

练习3.1

- 如果f函数返回的是无限制的float64值，那么SVG文件可能输出无效的多边形元素（虽然许多SVG渲染器会妥善处理这类问题）。修改程序跳过无效的多边形。

```
\\练习3.1 更改的代码

func main() {
    fmt.Printf("<svg xmlns='http://www.w3.org/2000/svg' "+
        "style='stroke: grey; fill: white; stroke-width: 0.7' "+
```

```
        "width='%d' height='%d'>", width, height)
    for i:= 0; i < cells; i++ {
        for j := 0; j < cells; j++ {
            ax, ay := corner(i + 1, j)
            bx, by := corner(i, j)
            cx, cy := corner(i, j + 1)
            dx, dy := corner(i + 1, j + 1)
            if math.IsNaN(ax) || math.IsNaN(ay) || math.IsNaN(bx) ||
math.IsNaN(by) || math.IsNaN(cx) || math.IsNaN(cy) || math.IsNaN(dx) ||
math.IsNaN(dy) {
                fmt.Fprintf(os.Stderr, "NAN")
            } else {
                fmt.Printf("<polygon points='%g,%g %g,%g %g,%g %g,%g'/>\n",
                ax, ay, bx, by, cx, cy, dx, dy)
            }
        }
    }
    fmt.Println("</svg>")
}
```

练习3.2

- 试验math包中其他函数的渲染图形。你是否能输出一个egg box、moguls或a saddle图案?

```
// 练习3.2 更改一下z轴函数即可


func corner(i, j int) (float64, float64) {
    x := xyrange * (float64(i) / cells - 0.5)
    y := xyrange * (float64(j) / cells - 0.5)
    //z := f(x, y)
    z := eggBox(x, y)
    sx := width / 2 + (x - y) * cos30 * xyscale
    sy := height / 2 + (x + y) * sin30 * xyscale - z * zscale
    return sx, sy
}


func eggBox(x, y float64) float64 {
    return math.Sin(x) + math.Sin(y) / 10
}
```

练习3.3

- 根据高度给每个多边形上色，那样峰值部将是红色（#ff0000），谷部将是蓝色（#0000ff）。

```
\\ 练习3.3
package main
```

```
import (
    "fmt"
    "math"
    "os"
)

const (
    width, height = 600, 320
    cells         = 100
    xyrange       = 30.0
    xyscale       = width / 2 / xyrange
    zscale        = height * 0.4
    angle         = math.Pi / 6
)
var sin30 = math.Sin(angle) // Go的常量是在编译之前就能确定的常量
var cos30 = math.Cos(angle)



func main() {
    fmt.Printf("<svg xmlns='http://www.w3.org/2000/svg' "+
        "style='stroke: grey; fill: white; stroke-width: 0.7' "+
        "width='%d' height='%d'>", width, height)
    for i:= 0; i < cells; i++ {
        for j := 0; j < cells; j++ {
            ax, ay, az := corner(i + 1, j)
            bx, by, bz := corner(i, j)
            cx, cy, cz := corner(i, j + 1)
            dx, dy, dz := corner(i + 1, j + 1)
            if math.IsNaN(ax) || math.IsNaN(ay) || math.IsNaN(bx) ||
math.IsNaN(by) || math.IsNaN(cx) || math.IsNaN(cy) || math.IsNaN(dx) ||
math.IsNaN(dy) {
                fmt.Fprintf(os.Stderr, "NAN")
            } else {
                //将z映射到一个较大范围

                fmt.Printf("<polygon style='fill: ")

                avgz := int((az + bz + cz + dz) * 10.0 + 8.0) * 18

                redv, bluev := 0, 0
                if avgz <= 255 {
                    redv = 0
                    bluev = 255 - avgz
                } else {
                    redv = avgz - 255
                    bluev = 0
                }
                if redv > 255 {
                    redv = 255
                }
                if bluev > 255{
                    bluev = 255
```

```
                }

                fmt.Printf("#%02X00", redv)
                fmt.Printf("%02X", bluev)
                fmt.Printf("' points='%g,%g %g,%g %g,%g %g,%g'/>\n",ax, ay, bx,
by, cx, cy, dx, dy)

            }
        }
    }
    fmt.Println("</svg>")
}
$$
func corner(i, j int) (float64, float64, float64) {
    x := xyrange * (float64(i)/cells - 0.5)
    y := xyrange * (float64(j)/cells - 0.5)

    z := f(x, y)
    sx := width/2 + (x-y)*cos30*xyscale
    sy := height/2 + (x+y)*sin30*xyscale - z*zscale
    return sx, sy, z
}

func f(x, y float64) float64 {
    r := math.Hypot(x, y)
    return math.Sin(r) / r
}
```

练习3.4

- 参考1.7节Lissajous例子的函数，构造一个web服务器，用于计算函数曲面然后返回SVG数据给客户端。
- 服务器必须设置Content-Type头部： `w.Header().Set("Content-Type", "image/svg+xml")`

```
// 直接返回给浏览器
package main

import (
    "fmt"
    "io"
    "log"
    "math"
    "net/http"
    "os"
)

const (
    width, height = 600, 320
    cells         = 100
    xyrange       = 30.0
    xyscale       = width / 2 / xyrange
    zscale        = height * 0.4
```

```
    angle         = math.Pi / 6
)
var sin30 = math.Sin(angle) // Go的常量是在编译之前就能确定的常量
var cos30 = math.Cos(angle)



func main() {
    handler := func(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("Content-Type", "image/svg+xml")
        getXML(w)
    }
    http.HandleFunc("/", handler)
        log.Fatal(http.ListenAndServe("localhost:8000", nil))
}

func getXML(out io.Writer){
    fmt.Fprintf(out, "<svg xmlns='http://www.w3.org/2000/svg' "+
    "style='stroke: grey; fill: white; stroke-width: 0.7' "+
    "width='%d' height='%d'>", width, height)
    for i:= 0; i < cells; i++ {
        for j := 0; j < cells; j++ {
            ax, ay := corner(i + 1, j)
            bx, by := corner(i, j)
            cx, cy := corner(i, j + 1)
            dx, dy := corner(i + 1, j + 1)
            if math.IsNaN(ax) || math.IsNaN(ay) || math.IsNaN(bx) ||
math.IsNaN(by) || math.IsNaN(cx) || math.IsNaN(cy) || math.IsNaN(dx) ||
math.IsNaN(dy) {
                fmt.Fprintf(os.Stderr, "NAN")
            } else {
                fmt.Fprintf(out, "<polygon points='%g,%g %g,%g %g,%g
%g,%g'/>\n",ax, ay, bx, by, cx, cy, dx, dy)
            }
        }
    }
    fmt.Fprintln(out, "</svg>")
}

func corner(i, j int) (float64, float64) {
    x := xyrange * (float64(i) / cells - 0.5)
    y := xyrange * (float64(j) / cells - 0.5)
    z := f(x, y)
    sx := width / 2 + (x - y) * cos30 * xyscale
    sy := height / 2 + (x + y) * sin30 * xyscale - z * zscale
    return sx, sy
}

func f(x, y float64) float64 {
    r := math.Hypot(x, y)
    return math.Sin(r) / r
}
```

复数

- 附属包含complex64 和 complex128, 注意分别对应的是float32 和 float64 是两倍的关系
- Mandelbrot图像， 对每个点进行$z_{k+1} = z^2_k + c$迭代测试， 迭代次数越多出范围的颜色越深形成的图形

```
\\ web示例
// png格式的mandelbrot 图像
package main

import (
    "image"
    "image/color"
    "image/png"
    "log"
    "math/cmplx"
    "net/http"
)

func main(){
    const (
        xmin, ymin, xmax, ymax = -2, -2,  2, 2
        width, height = 1024, 1024
    )
    img := image.NewRGBA(image.Rect(0,0,width, height))
    for py := 0; py < height; py++ {
        y := float64(py) / height * (ymax - ymin) + ymin
        for px := 0; px < width; px++ {
            x := float64(px) / width * (xmax - xmin) + xmin
            z := complex(x, y)
            img.Set(px, py, mandelbrot(z))
        }
    }
    handler := func(w http.ResponseWriter, r *http.Request) {
        png.Encode(w, img)
    }
    http.HandleFunc("/", handler)
        log.Fatal(http.ListenAndServe("localhost:8000", nil))

}

func mandelbrot(z complex128) color.Color{
    const iterations = 200
    const contrast = 15
    var v complex128
    for n := uint8(0); n < iterations; n++ {
        v = v * v + z
        if cmplx.Abs(v) > 2 {
            return color.Gray{255 - contrast * n}
        }
    }
```

```
    return color.Black
}
```

练习3.5

- 实现一个彩色的Mandelbrot图像，使用image.NewRGBA创建图像，使用color.RGBA或color.YCbCr生成颜色。

```
// 随便调调参， 颜色还挺好看
// 练习3.5 实现彩色效果
package main

import (
    "image"
    "image/color"
    "image/png"
    "log"
    "math/cmplx"
    "net/http"
)

func main(){
    const (
        xmin, ymin, xmax, ymax = -2, -2,  2, 2
        width, height = 1024, 1024
    )
    img := image.NewRGBA(image.Rect(0,0,width, height))
    for py := 0; py < height; py++ {
        y := float64(py) / height * (ymax - ymin) + ymin
        for px := 0; px < width; px++ {
            x := float64(px) / width * (xmax - xmin) + xmin
            z := complex(x, y)
            img.Set(px, py, mandelbrot(z))
        }
    }
    handler := func(w http.ResponseWriter, r *http.Request) {
        png.Encode(w, img)
    }
    http.HandleFunc("/", handler)
        log.Fatal(http.ListenAndServe("localhost:8000", nil))

}

func mandelbrot(z complex128) color.Color{
    const iterations = 200
    const contrast = 15
    var v complex128
    for n := uint8(0); n < iterations; n++ {
        v = v * v + z
        if cmplx.Abs(v) > 2 {
```

```
            return color.RGBA{50, 100, 100 + n, 255 - contrast * n} // 生成RGB效果
        }
    }
    return color.RGBA{200, 200, 100, 0}
}
```

练习3.6

- 升采样技术可以降低每个像素对计算颜色值和平均值的影响。简单的方法是将每个像素分成四个子像素，实现它。
- 升采样技术， 这里要求分成四个像素， 已知本来像素的中心点和宽度， 计算其他四个中心点其实不难， 结果取个平均值

```
// 练习3.6
package main

import (
    "image"
    "image/color"
    "image/png"
    "log"
    "math/cmplx"
    "net/http"
)

func main(){
    const (
        xmin, ymin, xmax, ymax = -2, -2,  2, 2
        width, height = 1024, 1024
    )
    img := image.NewRGBA(image.Rect(0,0,width, height))
    for py := 0; py < height; py++ {
        y := float64(py) / height * (ymax - ymin) + ymin
        for px := 0; px < width; px++ {
            x := float64(px) / width * (xmax - xmin) + xmin

            xn := []float64{x - (xmax - xmin) / width / 4, x + (xmax - xmin) /
width / 4}
            yn := []float64{y - (ymax - ymin) / height / 4, y + (ymax - ymin) /
height / 4}
            var rnow, gnow, bnow, anow uint32 // 因为有相加操作， 所以要大一点
            //fmt.Fprintf(os.Stderr, "%g\n", xn[0])
            for _, xnow := range xn {
                for _, ynow := range yn {
                    rtmp, gtmp, btmp, atmp := mandelbrot(complex(xnow,
ynow)).RGBA()
                    //fmt.Fprintf(os.Stderr, "%d\n", atmp)
                    rnow += rtmp >> 8
                    gnow += gtmp >> 8
```

```
                    bnow += btmp >> 8
                    anow += atmp >> 8
                }
            }
            rnow /= 4
            gnow /= 4
            bnow /= 4
            anow /= 4
            //fmt.Fprintf(os.Stderr, "%d\n", anow)
            img.SetRGBA(px, py, color.RGBA{uint8(rnow), uint8(gnow), uint8(bnow),
uint8(anow)})
        }
    }
    handler := func(w http.ResponseWriter, r *http.Request) {
        png.Encode(w, img)
    }
    http.HandleFunc("/", handler)
        log.Fatal(http.ListenAndServe("localhost:8000", nil))

}

func mandelbrot(z complex128) color.Color{
    const iterations = 200
    const contrast = 15
    var v complex128
    for n := uint8(0); n < iterations; n++ {
        v = v * v + z
        if cmplx.Abs(v) > 2 {
            return color.Gray{255 - contrast * n}
        }
    }
    return color.Black
}
```

练习3.7

- 另一个生成分形图像的方式是使用牛顿法来求解一个复数方程，例如$z^4-1=0$。每个起点到四个根的迭代次数对应阴影的灰度。方程根对应的点用颜色表示。
- $f(z) = z^4-1$, 已知幂函数为解析函数， 故 $f'(z) = 4z^3$

```
// 练习3.7
package main

import (
    "image"
    "image/color"
    "image/png"
    "log"
    "math/cmplx"
```

```
    "net/http"
)

func main(){
    const (
        xmin, ymin, xmax, ymax = -2, -2,  2, 2
        width, height = 1024, 1024
    )
    img := image.NewRGBA(image.Rect(0,0,width, height))
    for py := 0; py < height; py++ {
        y := float64(py) / height * (ymax - ymin) + ymin
        for px := 0; px < width; px++ {
            x := float64(px) / width * (xmax - xmin) + xmin
            z := complex(x, y)
            img.Set(px, py, newton(z))
        }
    }
    handler := func(w http.ResponseWriter, r *http.Request) {
        png.Encode(w, img)
    }
    http.HandleFunc("/", handler)
        log.Fatal(http.ListenAndServe("localhost:8000", nil))

}

func newton(z complex128) color.Color{ // x_{i + 1} = x_i - f(x) / f'(x)
    const iterations = 200
    const contrast = 15
    var v complex128
    v = z
    eps := 1e-8
    ans1, ans2, ans3, ans4 := complex(1, 0), complex(-1, 0), complex(0, 1),
complex(0, -1)
    for n := uint8(0); n < iterations; n++ {
        v = v - f(v) / diff(v)
        if cmplx.Abs(v - ans1) < eps || cmplx.Abs(v - ans2) < eps || cmplx.Abs(v -
ans3) < eps || cmplx.Abs(v - ans4) < eps {
            return color.Gray{255 - contrast * n}
        }
    }
    return color.Black
}

func f(z complex128) complex128 {
    return z * z * z * z - complex(1,0)
}

func diff(z complex128) complex128 {
    return 4 * z * z * z
}
```

练习3.8

- 通过提高精度来生成更多级别的分形。使用四种不同精度类型的数字实现相同的分形：complex64、complex128、big.Float和big.Rat。（后面两种类型在math/big包声明。Float是有指定限精度的浮点数；Rat是无限精度的有理数。）它们间的性能和内存使用对比如何？当渲染图可见时缩放的级别是多少？

练习3.9

- 编写一个web服务器，用于给客户端生成分形的图像。运行客户端通过HTTP参数指定x、y和zoom参数。

```
// 实现http传参， 先处理参数再绘制

package main

import (
    "fmt"
    "image"
    "image/color"
    "image/png"
    "log"
    "math/cmplx"
    "net/http"
    "strconv"
)

func main(){
    const (
    //  xmin, ymin, xmax, ymax = -2, -2,  2, 2
        width, height = 1024, 1024
    )
    params := map[string] float64 { // 使用map直接存， 要是在之后使用多个if判断， 耗时反而会更久
        "xmin": -2,
        "xmax": 2,
        "ymin": -2,
        "ymax": 2,
        "zoom": 1,
    }

    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request){
            for name := range params {
                s := r.FormValue(name)
                if s == "" {
                    continue
                }
                f, err := strconv.ParseFloat(s, 64)
                if err != nil {
                    http.Error(w, fmt.Sprintf("query param %s: %s", name, err), http.StatusBadRequest)
                    return
                }
                params[name] = f  // 读取信息的方式， 来自Gopl-homework
            }
            if params["xmax"] <= params["xmin"] || params["ymax"] <=
```

```
params["ymin"] {
                http.Error(w, fmt.Sprintf("min coordinate greater than max"),
http.StatusBadRequest)
                    return
            }
            xmin, xmax, ymin, ymax, zoom :=
params["xmin"],params["xmax"],params["ymin"],params["ymax"],params["zoom"]
            lenX, lenY := xmax - xmin, ymax - ymin
            midX, midY := xmin + lenX / 2, ymin + lenY / 2
            xmin, xmax, ymin, ymax = midX - lenX / 2 / zoom, midX + lenX / zoom /
2, midY - lenY / 2 / zoom, midY + lenY / 2 / zoom
            //fmt.Fprintf(os.Stderr, "%g %g %g %g\n", xmin, xmax, ymin, ymax)
            img := image.NewRGBA(image.Rect(0,0,width, height))
            for py := 0; py < height; py++ {
                y := float64(py) / height * (ymax - ymin) + ymin
                for px := 0; px < width; px++ {
                    x := float64(px) / width * (xmax - xmin) + xmin
                    z := complex(x, y)
                    img.Set(px, py, mandelbrot(z))
                }
            }
            png.Encode(w, img)
    })
    log.Fatal(http.ListenAndServe("localhost:8000", nil))

}

func mandelbrot(z complex128) color.Color{
    const iterations = 200
    const contrast = 15
    var v complex128
    for n := uint8(0); n < iterations; n++ {
        v = v * v + z
        if cmplx.Abs(v) > 2 {
            return color.Gray{255 - contrast * n}
        }
    }
    return color.Black
}
```

- Go 同C含有表达式短路机制

字符串

- 首先 Go中的字符串是不允许被修改的
- 字符串S[i]表示读第i个字节， 并不一定是第i个字符
- Unicode 编码标准优良， 可以不用解码检查前后缀
- 在程序内部使用rune序列可以更加方便， 大小一致， 便于切割
- 书上的示例

```go
// 三个示例写一个文件里了

// 将看起来像是 系统目录的前缀删除， 并将看起来像后缀名的部分删除
func basename (s string) string {
    for i := len(s); i >= 0; i-- {
        if s[i] == '/' {
            s = s[i + 1: ]
            break
        }
    }
    for i := len(s) - 1; i >= 0; i-- {
        if s[i] == '.' {
            s = s[:i]
            break
        }
    }
    return s
}

// 使用strings.LastIndex
func basename1 (s string) string {
    slash := strings.LastIndex(s, "/")
    s = s[slash + 1:]
    if dot := strings.LastIndex(s, "."); dot >= 0 {
        s = s[:dot]
    }
    return s
}

// comma

func comma(s string) string {
    n := len(s)
    if n <= 3 {
        return s
    }
    return comma(s[:n - 3] + "," + s[n - 3:])
}
```

练习3.10

- 编写一个非递归版本的comma函数，使用bytes.Buffer代替字符串链接操作。

```go
// 实现非递归版本的comma函数

package main

import "bytes"
```

```
func main(){
    for _, s := range []string{"1", "12", "123", "1234", "12345678901234",
"123456789012345", "1234567890123456", "12345678901234567", "123456789012345678",
"1234567890123456789"} {
        println(comma(s))
    }
}

func comma(s string) string {
    var buf bytes.Buffer
    len := len(s)
    mod1 := len % 3
    if mod1 == 0 && len >= 3 {
        mod1 = 3
    }
    for be := 0; be + mod1 <= len ;{
        buf.WriteString(s[be:be+mod1])
        if be + mod1 != len {
            buf.WriteString(",")
        }
        be += mod1
        mod1 = 3
    }

    return buf.String()
}
```

练习3.11

- 完善comma函数，以支持浮点数处理和一个可选的正负号的处理。

```
// 实现非递归版本的comma函数

package main

import (
    "bytes"
    "strings"
)

func main(){
    for _, s := range []string{"1", "12", "123", "1234", "12345678.901234",
"-123456789012.345", "123456789012.3456", "123456789012345.67",
"12345678901234567.8", "12.34567890123456789"} {
        println(comma(s))
    }
}

// 更改后的comma 函数支持将浮点数，小数部分是从左往右
func comma(s string) string {
```

```
    var buf bytes.Buffer
    len := len(s)
    if len > 0 && s[0] == '-' {
        buf.WriteByte('-')
        s = s[1:]
    }
    dotPlace := strings.IndexByte(s, '.')
    if dotPlace > 0 {
        buf.WriteString(comma1(s[:dotPlace]))
        buf.WriteByte('.')
        buf.WriteString(comma2(s[dotPlace + 1:]))
    } else {
        buf.WriteString(comma1(s))
    }
    return buf.String()
}
// 整数部分
func comma1(s string) string {
    var buf bytes.Buffer
    len := len(s)
    mod1 := len % 3
    if mod1 == 0 && len >= 3 {
        mod1 = 3
    }
    for be := 0; be + mod1 <= len ;{
        buf.WriteString(s[be:be+mod1])
        if be + mod1 != len {
            buf.WriteString(",")
        }
        be += mod1
        mod1 = 3
    }
    return buf.String()
}
// 小数部分
func comma2(s string) string {
    var buf bytes.Buffer
    len := len(s)

    for be := 0; be < len ; be += 3{
        if be + 3 < len {
            buf.WriteString(s[be:be+3])
            buf.WriteString(",")
        } else {
            buf.WriteString(s[be:])
        }
    }
    return buf.String()
}
```

练习3.12

- 编写一个函数，判断两个字符串是否是相互打乱的，也就是说它们有着相同的字符，但是对应不同的顺序。

```
// 使用两个map记录两个字符串字符出现次数
//

package main

import "fmt"

func main(){
    s1, s2 := "abc", "cba"
    fmt.Println(cmp(s1, s2))
}

func cmp(s1, s2 string) bool {
    if s1 == s2{
        return false
    }
    m1, m2 := make(map[string]int), make(map[string]int)
    for i := 0; i < len(s1); i++ {
        m1[string(s1[i])]++
    }
    for i := 0; i < len(s2); i++ {
        m2[string(s2[i])]++
    }
    for k, v := range m1 {
        if m2[k] != v {
            return false
        }
    }
    return true
}
```

常量

iota 常量生成器

- iota 常量生成器， 可以在常量声明中用iota作为常量值， 它会自增， 从0开始

```
type Weekday int

const (
    Sunday Weekday = iota
    Monday
    Tuesday
    Wednesday
    Thursday
    Friday
```

```
    Saturday
)
```

- 常量可以表示为无类型的

练习3.13

- 编写KB、MB的常量声明，然后扩展到YB。

```
// 练习3.13 编写KB、MB的常量声明，然后扩展到YB。
// 拆分二进制
package main

func main() {
    const (
        KB = 1000
        MB = 1000 * KB
        GB = 1000 * MB
        TB = 1000 * GB
        PB = 1000 * TB
        EB = 1000 * PB
        ZB = 1000 * EB
        YB = 1000 * ZB
    )
    //fmt.Println(KB, MB, GB, TB, PB, EB, ZB,YB)
}
```

第四章

数组

- 默认情况下，数组的每个元素都被初始化为元素类型对应的0值。
- 示例

```
q := [3]int{1, 2, 3}

type Currency int
const (
    USD Currency = iota
    EUR
    GBP
    RMB
)
symbol := [4]string{USD: "$", EUR: "€", GBP: "£", RMB: "¥"}
fmt.Println(RMB, symbol[RMB])
```

- 数组比较时， 只有各元素都可比较且一致时， 才认为数组相等
- 这里有个数组比较的例子

```
package main

import (
    "crypto/sha256"
    "fmt"
)

func main(){
    c1 := sha256.Sum256([]byte("x"))
    c2 := sha256.Sum256([]byte("X"))
    fmt.Printf("%x\n%x\n%t\n%T\n", c1, c2, c1 == c2, c1)
}
```

练习4.1

- 编写一个函数，计算两个SHA256哈希码中不同bit的数目。（参考2.6.2节的PopCount函数。）

```
package main

import (
    "crypto/sha256"
    "fmt"
)

var pc [256]byte

func init() {
    for i := range pc {
        pc[i] = pc[i/2] + byte(i&1)
    }
}

func main() {
    c1 := sha256.Sum256([]byte("x"))
    c2 := sha256.Sum256([]byte("X"))
    ans := 0
    for i := 0; i < len(c1); i++ {
        ans += int(pc[c1[i]^c2[i]])
    }
    fmt.Println(ans)
}
```

练习4.2

- 编写一个程序，默认情况下打印标准输入的SHA256编码，并支持通过命令行flag定制，输出SHA384或SHA512哈希算法。

```
//编写一个程序，默认情况下打印标准输入的SHA256编码，并支持通过命令行flag定制，输出SHA384或SHA512哈希算法。

package main

import (
    "crypto/sha256"
    "crypto/sha512"
    "flag"
    "fmt"
)

var hashMethod = flag.Int("s", 256, "hash method default:256 other:384, 512")
func main() {
    flag.Parse()
    var s string
    fmt.Printf("输入字符串")
    fmt.Scanln(&s)
    switch *hashMethod{
        case 256:
            fmt.Printf("%x 1\n", sha256.Sum256([]byte(s)))
        case 384:
            fmt.Printf("%x 2\n", sha512.Sum384([]byte(s)))
        case 512:
            fmt.Printf("%x 3\n", sha512.Sum512([]byte(s)))
        default:
            fmt.Printf("输入错误"u
}
```

Slice

共通点

- 语法相近,slice只是没有固定长度。

区别

- slice的第一个元素不一定是数组的第一个元素。
- slice的容量是指从slice开始地址到数组结束地址的距离。 使用cap函数可以获取slice的容量。
- slice的容量和长度可以不一样。多个slice可以指向同一个数组。
- slice不能直接判断是否相等，但是可以通过比较其长度和元素是否相等来判断。
-

第五章

- 函数的四种写法——你懂么?

```
func add(x int, y int) int {return x + y}
func sub(x, y int) int {return x - y})
func first(x int, _ int) int {return x}
func zero(int, int ) int {return 0}
```

- 函数的类型被称为函数的签名， 由两个部分，参数列表和返回值列表决定。

```
func Sin(x float64) float // 该函数没有函数体， 为函数声明，表示功能不是由GO实现的，定义了函数签名（可能是汇编语言）
```

- 遍历dom树的递归函数

```
// 遍历dom树查找herf
// 遍历dom树查找herf
package main

import (
    "fmt"
    "os"

    "golang.org/x/net/html"
)

func main(){
    //fmt.Println(fetch())
    doc, err := html.Parse(os.Stdin)  // html.Parse 输入是io.Reader 常见来源有os.Open, strings.NewReader, http.Request.body, bytes.Buffer
    if err != nil {
        fmt.Fprintf(os.Stderr, "findlink: %v\n", err)
        os.Exit(1)
    }
    for _, link := range visit(nil, doc) { //
        fmt.Println(link)
    }
}

func visit(links []string, n *html.Node) []string {
    if n.Type == html.ElementNode && n.Data == "a" {
        for _, a := range n.Attr {
```

```
            if a.Key == "href" {
                links = append(links, a.Val)
            }
        }
    }
    for c := n.FirstChild; c != nil; c = c.NextSibling {
        links = visit(links, c) // 递归调用,
    }
    return links
}
```

- 输出整个dom树结构,

练习5.1

- 修改findlinks代码中遍历n.FirstChild链表的部分，将循环调用visit，改成递归调用。

```
// 递归子节点和兄弟节点

package main

import (
    "fmt"
    "os"

    "golang.org/x/net/html"
)

func main(){
    doc, err := html.Parse(os.Stdin)
    if err != nil {
        fmt.Fprintf(os.Stderr, "findlink: %v\n", err)
        os.Exit(1)
    }
    for _, link := range visit(nil, doc) { //
        fmt.Println(link)
    }
}

func visit(links []string, n *html.Node) []string {
    if n.Type == html.ElementNode && n.Data == "a" {
        for _, a := range n.Attr {
            if a.Key == "href" {
                links = append(links, a.Val)
            }
        }
    }
    if n.FirstChild != nil {
        links = visit(links, n.FirstChild)
```

```
    }
    if n.NextSibling != nil {
        links = visit(links, n.NextSibling)
    }
    return links
}
```

练习5.2

- 编写函数，记录在HTML树中出现的同名元素的次数。

```
// 编写函数，记录在HTML树中出现的同名元素的次数。
// 同名元素是指 Node.Data 相同的， 使用map统计即可
package main

import (
    "fmt"
    "os"

    "golang.org/x/net/html"
)


func main(){
    //fmt.Println(fetch())
    doc, err := html.Parse(os.Stdin)  // html.Parse 输入是io.Reader 常见来源有
os.Open, strings.NewReader, http.Request.body, bytes.Buffer
    if err != nil {
        fmt.Fprintf(os.Stderr, "findlink: %v\n", err)
        os.Exit(1)
    }
    for name, total := range count(make(map[string]int), doc) { //
        fmt.Printf("%s, %d\n", name, total)
    }
}

func count(m map[string]int, n *html.Node) map[string]int{
    if n.Type == html.ElementNode{
        m[n.Data]++
    }
    for c := n.FirstChild; c != nil; c = c.NextSibling {
        count(m, c)
    }
    return m
}
```

练习5.3

- 编写函数输出所有text结点的内容。注意不要访问<script>和<style>元素，因为这些元素对浏览者是不可见的。

```
// text节点判断方法— 为html.text Node, 输出非英文有的是乱码
package main

import (
    "fmt"
    "os"

    "golang.org/x/net/html"
)

func main(){
    //fmt.Println(fetch())
    doc, err := html.Parse(os.Stdin)  // html.Parse 输入是io.Reader 常见来源有
os.Open, strings.NewReader, http.Request.body, bytes.Buffer
    if err != nil {
        fmt.Fprintf(os.Stderr, "findlink: %v\n", err)
        os.Exit(1)
    }
    for _, link := range visit(nil, doc) { //
        fmt.Println(link)
    }
}

func visit(links []string, n *html.Node) []string {
    if n.Type == html.TextNode  {
        fmt.Println(n.Data)
    }
    for c := n.FirstChild; c != nil; c = c.NextSibling {
        if c.Data == "style" || c.Data == "script" {
            continue
        }
        links = visit(links, c)
    }
    return links
}
```

练习5.4

- 扩展visit函数，使其能够处理其他类型的结点，如images、scripts和style sheets。
-

```
// 扩展visit函数，使其能够处理其他类型的结点，如images、scripts和style sheets。
package main
```

```
import (
    "fmt"
    "os"

    "golang.org/x/net/html"
)

func main(){
    doc, err := html.Parse(os.Stdin)
    if err != nil {
        fmt.Fprintf(os.Stderr, "findlink: %v\n", err)
        os.Exit(1)
    }
    for _, link := range visit(nil, doc) { //
        fmt.Println(link)
    }
}

func visit(links []string, n *html.Node) []string {
    if n.Type == html.ElementNode && n.Data == "a" {
        for _, a := range n.Attr {
            if a.Key == "href" {
                links = append(links, a.Val)
            }
        }
    }
    if n.Type == html.ElementNode && (n.Data == "img" || n.Data == "script") {
        for _, a := range n.Attr {
            if a.Key == "src" {
                links = append(links, a.Val)
            }
        }
    }
    for c := n.FirstChild; c != nil; c = c.NextSibling {
        links = visit(links, c) // 递归调用,
    }
    return links
}
```

多返回值

- 多返回值可以做返回值
- 多返回值函数可以做参数， 下面两种写法等同

```
log.Println(findLinks(url))
```

```
links, err := findLinks(url)
log.Println(links, err)
```

- 新版本findlinks

```
// 遍历dom树查找herf
package main

import (
    "fmt"
    "net/http"
    "os"

    "golang.org/x/net/html"
)

func main(){
    for _, url := range os.Args[1:] {
        links, err := findLinks(url)
        if err != nil {
            fmt.Fprintf(os.Stderr, "findlinks2: %v\n", err)
            continue
        }
        for _, link := range links {
            fmt.Println(link)
        }
    }
}

func findLinks(url string) ([]string, error) {
    resp, err := http.Get(url)
    if err != nil {
        return nil, err
    }
    if resp.StatusCode != http.StatusOK {
        resp.Body.Close()
        return nil, fmt.Errorf("getting %s: %s", url, resp.Status)
    }
    doc, err := html.Parse(resp.Body)
    resp.Body.Close() // Go的垃圾回收不包括
    if err != nil {
        return nil, err
    }
    return visit(nil, doc), nil // 返回值有好几种,
}

func visit(links []string, n *html.Node) []string {
    if n.Type == html.ElementNode && n.Data == "a" {
        for _, a := range n.Attr {
            if a.Key == "href" {
                links = append(links, a.Val)
```

```
            }
        }
    }
    for c := n.FirstChild; c != nil; c = c.NextSibling {
        links = visit(links, c) // 递归调用,
    }
    return links
}
```

- 如果一个函数的所有返回值都有显示的变量名， 那么该函数的return语句可以省略变量名。 bare return

练习5.6

- 修改gopl.io/ch3/surface（§3.2）中的corner函数，将返回值命名，并使用bare return。
-

```
func corner(i, j int) (sx float64,sy float64) {
    x := xyrange * (float64(i) / cells - 0.5)
    y := xyrange * (float64(j) / cells - 0.5)
    z := f(x, y)
    sx = width / 2 + (x - y) * cos30 * xyscale
    sy = height / 2 + (x + y) * sin30 * xyscale - z * zscale
    return
}
```

错误处理

- 所有错误在本层分层时， 都需要添加本层的前缀， 错误信息
- 错误一般分为五种
- 传播错误， 错误会使得整个功能失败。 整个错误返回给调用者
- 错误时偶然性，不可预知的问题产生。 重试时， 我们需要限制重试的时间间隔或者重试的次数， 避免无限制的重试 （例子：下面的wait函数）
- 错误时整个程序无法运行、需要输出错误并且结束程序， 只应该在main中执行。
- 错误时， 只需要输出错误， 不需要结束程序。 log.printf("message", error)
- 直接忽略掉错误。
- 文件结尾错误一般不需要报错

```
package main

import (
    "fmt"
    "log"
    "net/http"
    "time"
)
```

```
func main() {

}

func WaitForServer(url string) error {
    const timeout = 1 * time.Minute
    deadline := time.Now().Add(timeout)
    for tries := 0; time.Now().Before(deadline); tries++ {
        _, err := http.Head(url)
        if err == nil {
            return nil
        }
        log.Printf("server not responding (%s); retrying...", err)
        time.Sleep(time.Second << uint(tries))
    }
    return fmt.Errorf("server %s failed to respond after %s", url, timeout)
}
```

函数变量

- Go语言中，函数也是值， 可以赋值给变量， 函数变量可以作为参数传递给其他函数
- 例子

```
func square(n int) int { return n * n }
func negative(n int) int { return -n }
func product(m, n int) int { return m * n }
f := square
fmt.Println(f(3))
f = negative
fmt.Println(f(3))
fmt.Printf("%T\n", f))
f = product // 这里会报错， 因为两种函数不是同一类型（类型由参数列表和返回值列表决定）
```

- %*s 会在字符串之前填充一些空格，后面写数目
- 利用函数变量重写outline如下

```
package main

import (
    "fmt"
    "os"

    "golang.org/x/net/html"
)
var depth int
func main(){
    //fmt.Println(fetch())
```

```
    doc, err := html.Parse(os.Stdin)
    if err != nil {
        fmt.Fprintf(os.Stderr, "findlink: %v\n", err)
        os.Exit(1)
    }
    forEachNode(doc, startElement, ElementNode)
}


func forEachNode(n *html.Node, pre, post func(n *html.Node)) {
    if pre != nil {
        pre(n)
    }
    for c := n.FirstChild; c != nil; c = c.NextSibling {
        forEachNode(c, pre, post)
    }
    if post != nil {
        post(n)
    }
}

func startElement(n *html.Node) {
    if n.Type == html.ElementNode {
        fmt.Printf("%*s<%s>\n", depth * 2, "", n.Data)
        depth++
    }
}

func ElementNode(n *html.Node) {
    if n.Type == html.ElementNode {
        depth--
        fmt.Printf("%*s<%s>\n", depth * 2, "", n.Data)
    }
}
```

练习5.7

- 完善startElement和endElement函数，使其成为通用的HTML输出器。要求：输出注释结点，文本结点以及每个元素的属性（< a href='...'>）。使用简略格式输出没有孩子结点的元素（即用代替）。编写测试，验证程序输出的格式正确。（详见11章）

```
// 在前括号信息内增加所需信息， 并且通过有无子节点判断是否增添后括号
package main

import (
    "fmt"
    "os"

    "golang.org/x/net/html"
)
var depth int
```

```
func main(){
    //fmt.Println(fetch())
    doc, err := html.Parse(os.Stdin)
    if err != nil {
        fmt.Fprintf(os.Stderr, "findlink: %v\n", err)
        os.Exit(1)
    }
    forEachNode(doc, startElement, ElementNode, leaveNode)
}


func forEachNode(n *html.Node, pre, post, now func(n *html.Node)) {
    if n.FirstChild == nil {
        now(n)
        return
    }
    if pre != nil {
        pre(n)
    }
    for c := n.FirstChild; c != nil; c = c.NextSibling {
        forEachNode(c, pre, post, now)
    }
    if post != nil && n.FirstChild != nil{
        post(n)
    }
}

func startElement(n *html.Node) {
    if n.Type == html.ElementNode {
        fmt.Printf("%*s<%s ", depth * 2, "", n.Data)
        for _, a := range n.Attr {
            fmt.Printf("%s=%s ", a.Key, a.Val)
        }
        depth++
        fmt.Printf(">\n")
    }
}

func ElementNode(n *html.Node) {
    if n.Type == html.ElementNode {
        depth--
        fmt.Printf("%*s</%s>\n", depth * 2, "", n.Data)
    }
}

func leaveNode(n *html.Node) {
    if n.Type == html.ElementNode {
        fmt.Printf("%*s<%s/ ", (depth + 1) * 2, "", n.Data)
        for _, a := range n.Attr {
            fmt.Printf("%s=%s ", a.Key, a.Val)
        }
        fmt.Printf(">\n")
    }
}
```

练习5.8

- 修改pre和post函数，使其返回布尔类型的返回值。返回false时，中止forEachNoded的遍历。使用修改后的代码编写ElementByID函数，根据用户输入的id查找第一个拥有该id元素的HTML元素，查找成功后，停止遍历。

```
package main

import (
    "fmt"
    "os"

    "golang.org/x/net/html"
)
var depth int
func main(){
    //fmt.Println(fetch())
    doc, err := html.Parse(os.Stdin)
    if err != nil {
        fmt.Fprintf(os.Stderr, "findlink: %v\n", err)
        os.Exit(1)
    }
    _, ok := ElementByID(doc, "lg", startElement, ElementNode)
    if !ok {
        fmt.Println("not found")
    }

}

func ElementByID(n *html.Node, id string,  pre, post func(n *html.Node,id string)
bool) (*html.Node, bool) {
    if pre != nil {
        ok := pre(n, id)
        if ok {
            return n, ok
        }
    }
    for c := n.FirstChild; c != nil; c = c.NextSibling {
        ans1, ok := ElementByID(c, id, pre, post)
        if ok {
            return ans1, ok
        }
    }
    if post != nil {
        post(n, id)
    }
    return nil, false
}
```

```
func startElement(n *html.Node, id string) bool{
    if n.Type == html.ElementNode {
        fmt.Printf("%*s<%s>\n", depth * 2, "", n.Data)
        depth++
    }
    for _, a := range n.Attr {
        if a.Key == "id" && a.Val == id{
            fmt.Printf("%*s found here\n", (depth + 1) *2, "")
            return true
        }
    }
    return false
}

func ElementNode(n *html.Node, id string) bool{
    if n.Type == html.ElementNode {
        depth--
        fmt.Printf("%*s</%s>\n", depth * 2, "", n.Data)
    }
    return false
}
```

练习5.9

- 编写函数expand，将s中的"foo"替换为f("foo")的返回值。

```
// 编写函数expand，将s中的"foo"替换为f("foo")的返回值

package main

import (
    "fmt"
    "strings"
)

func main() {

    s := "fooofffofofooffooofofofofofo"
    fmt.Printf("%s\n%s\n", s, expand(s, f))
}

func expand(s string, f func(string) string) string {
    newS := f("foo")
    return strings.Replace(s, "foo", newS, -1)
}

func f(s string) string {
    return "?" + s + "?"
}
```

匿名函数

- 命名函数只能在包级别进行声明， 匿名函数函数后面没有名称， 能够获取到整个词法环境
- 示例

```
package main

func main() {
    f := squares()
    for i := 0; i < 10; i++ {
        println(f())
    }
}

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

- 拓扑排序

```
package main

import (
    "fmt"
    "sort"
)

var prereqs = map[string][]string{
    "algorithms": {"data structures"},
    "calculus":   {"linear algebra"},
    "compilers": {
        "data structures",
        "formal languages",
        "computer organization",
    },
    "data structures":       {"discrete math"},
    "databases":             {"data structures"},
    "discrete math":         {"intro to programming"},
    "formal languages":      {"discrete math"},
    "networks":              {"operating systems"},
    "operating systems":     {"data structures", "computer organization"},
    "programming languages": {"data structures", "computer organization"},
}

func main() {
    for i, course := range topoSort(prereqs) {
```

```
        fmt.Printf("%d:\t%s\n", i+1, course)
    }
}

// topoSort 实现了对有向无环图（DAG）的节点进行拓扑排序。
// m 是一个映射，其中每个键代表图中的一个节点，对应的值是该节点指向的其他节点的列表。
// 返回值是节点的拓扑排序列表。
func topoSort(m map[string][]string) []string {
    // order 用于存储拓扑排序的结果。
    var order []string
    // seen 用于记录已经访问过的节点，以避免重复访问。
    seen := make(map[string]bool)

    // visitAll 是一个递归函数，用于遍历节点并将其按拓扑顺序添加到 order 中。
    var visitAll func(items []string)
    visitAll = func(items []string) {
        for _, item := range items {
            // 如果节点尚未被访问，则递归访问其依赖项，并将该节点添加到排序顺序中。
            if !seen[item] {
                seen[item] = true
                visitAll(m[item])
                order = append(order, item)
            }
        }
    }

    // keys 用于存储 m 中所有键（节点）的列表，以便进行排序。
    var keys []string
    for key := range m {
        keys = append(keys, key)
    }

    // 对节点进行排序，以便按照一定的顺序访问它们。
    sort.Strings(keys)

    // 使用排序后的节点列表调用 visitAll，以确保节点的处理顺序符合排序结果。
    visitAll(keys)  // 这里是访问入口， 从这里开始拓扑排序

    // 返回拓扑排序结果。
    return order
}
```

练习5.10

- 重写topoSort函数，用map代替切片并移除对key的排序代码。验证结果的正确性（结果不唯一）。

```
// 重写topoSort函数，用map代替切片并移除对key的排序代码。验证结果的正确性（结果不唯一）。

package main
```

```go
import (
    "fmt"
)

var prereqs = map[string][]string{
    "algorithms": {"data structures"},
    "calculus":   {"linear algebra"},
    "compilers": {
        "data structures",
        "formal languages",
        "computer organization",
    },
    "data structures":       {"discrete math"},
    "databases":             {"data structures"},
    "discrete math":         {"intro to programming"},
    "formal languages":      {"discrete math"},
    "networks":              {"operating systems"},
    "operating systems":     {"data structures", "computer organization"},
    "programming languages": {"data structures", "computer organization"},
}

func main() {
    for i, course := range topoSort(prereqs) {
        fmt.Printf("%d:\t%s\n", i+1, course)
    }
}
func topoSort(m map[string][]string) []string {
    var order []string
    seen := make(map[string]bool)

    var visitAll func(m map[string][]string, s string)
    visitAll = func(m map[string][]string, s string) {
        if seen[s] {  // 多入口， 把判断改到循环开始
            return
        }
        seen[s] = true
        order = append(order, s)
        items := m[s]
        for _, item := range items {
            //fmt.Println(s + "!" + item + "!")
            visitAll(m , item)
        }
    }
    for key := range m {
        visitAll(m, key)
    }
    return order
}
```

练习5.11

- 现在线性代数的老师把微积分设为了前置课程。完善topSort，使其能检测有向图中的环。

```
// 使用带度数的拓扑排序，最后要是有度数不为零的，就是环上的。

package main

import (
    "fmt"
    "sort"
)

var prereqs = map[string][]string{
    "algorithms": {"data structures"},
    "calculus":   {"linear algebra"},
    "compilers": {
        "data structures",
        "formal languages",
        "computer organization",
    },
    "data structures":       {"discrete math"},
    "databases":             {"data structures"},
    "discrete math":         {"intro to programming"},
    "formal languages":      {"discrete math"},
    "networks":              {"operating systems"},
    "operating systems":     {"data structures", "computer organization"},
    "programming languages": {"data structures", "computer organization"},
}

func main() {
    for i, course := range topoSort(prereqs) {
        fmt.Printf("%d:\t%s\n", i+1, course)
    }
}
func topoSort(m map[string][]string) []string {
    du := make(map[string]int)
    for _, v := range m {
        for _, tmp := range v{
            du[tmp] ++
        }
    }

    var order []string
    var visitAll func(now string)
    visitAll = func(now string) {
        order = append(order, now)
        items := m[now]
        for _, item := range items{
            du[item] --
            if du[item] == 0 {
            //  fmt.Println("?")
                visitAll(item)
            }
        }
```

```
    }
    var keys []string
    for key := range m {
        if du[key] == 0 {
            keys = append(keys, key)
        //  fmt.Println(key + "!")
        }
    }
    sort.Strings(keys)
    for _, key := range keys {
        visitAll(key)
    }
    for _, v := range du {
        if v != 0 {
            panic("有环")
        }
    }
    return order
}
```

练习5.12

- gopl.io/ch5/outline2（5.5节）的startElement和endElement共用了全局变量depth，将它们修改为匿名函数，使其共享outline中的局部变量。

```
// 将outline2 中的startElement和endElement改成匿名函数

package main

import (
    "fmt"
    "os"

    "golang.org/x/net/html"
)
func main(){
    //fmt.Println(fetch())
    doc, err := html.Parse(os.Stdin)
    if err != nil {
        fmt.Fprintf(os.Stderr, "findlink: %v\n", err)
        os.Exit(1)
    }
    var depth int
    forEachNode(doc, func(n *html.Node){
        if n.Type == html.ElementNode {
            fmt.Printf("%*s<%s>\n", depth * 2, "", n.Data)
            depth++
        }
    }, func(n *html.Node){
        if n.Type == html.ElementNode {
```

```
            depth--
            fmt.Printf("%*s</%s>\n", depth * 2, "", n.Data)
        }
    })
}


func forEachNode(n *html.Node, pre, post func(n *html.Node)) {
    if pre != nil {
        pre(n)
    }
    for c := n.FirstChild; c != nil; c = c.NextSibling {
        forEachNode(c, pre, post)
    }
    if post != nil {
        post(n)
    }
}
```

练习5.13

- 修改crawl，使其能保存发现的页面，必要时，可以创建目录来保存这些页面。只保存来自原始域名下的页面。假设初始页面在golang.org下，就不要保存vimeo.com下的页面。
- 暂时略过

练习5.14

- 使用breadthFirst遍历其他数据结构。比如，topoSort例子中的课程依赖关系（有向图）、个人计算机的文件层次结构（树）；你所在城市的公交或地铁线路（无向图）。
- 暂时略过
- 作用域问题， 由于匿名函数使用的变量常常是传址

可变参数

- 在声明可变参数函数时， 需要在参数列表的最后一个参数类型前面加上省略号...
- 例子

```
package main

import "fmt"

func main() {
    fmt.Println(sum(1, 2, 3, 4, 5))
    fmt.Println(sum())
    fmt.Println(sum(1))
}

func sum(vals ...int) int {
```

```
    total := 0
    for _, val := range vals {
        total += val
    }
    return total
}
```

练习5.15

- 编写类似sum的可变参数函数max和min。考虑不传参时，max和min该如何处理，再编写至少接收1个参数的版本。

```
// 编写类似sum的可变参数函数max和min。考虑不传参时，max和min该如何处理，再编写至少接收1个参数的版本。
package main

import "fmt"


func main(){
    fmt.Println(max(1,2,3,4,5,6,7,8,9,10))
    fmt.Println(min())
}

func max(x ...int) int {
    if len(x) == 0 {
        return 0
    }
    max := x[0]
    for _, v := range x {
        if v > max {
            max = v
        }
    }
    return max
}

func min(x ...int) int {
    if len(x) == 0 {
        return 0
    }
    min := x[0]
    for _, v := range x {
        if v < min {
            min = v
        }
    }
    return min
}
```

练习5.16

- 编写多参数版本的strings.Join。

```
// 将 多个字符串数组拼接成一个字符串
package main

import (
    "fmt"
)
func main() {
    strs := []string{"a", "b", "c"}
    strs2 := []string{"d", "e", "f"}
    fmt.Println(myJoin(","))
    fmt.Println(myJoin("?", strs))
    fmt.Println(myJoin("?", strs, strs))
    fmt.Println(myJoin("?", strs, strs2, strs))

}

func myJoin(s string, elems ...[]string) string {
    if len(elems) == 0 {
        return ""
    }
    var str string
    for _, elem := range elems {
        for _, e := range elem {
            str += e + s
        }
    }
    return str[:len(str)-len(s)]
}
```

练习5.17

- 编写多参数版本的ElementsByTagName，函数接收一个HTML结点树以及任意数量的标签名，返回与这些标签名匹配的所有元素。下面给出了2个例子：

```
package main

import (
    "fmt"
    "net/http"
    "os"

    "golang.org/x/net/html"
)

func main() {
    for _, url := range os.Args[1:] {
```

```
        resp, err := http.Get(url)
        if err != nil {
            fmt.Fprintf(os.Stderr, "fetch: %v\n", err)
            continue
        }
        if resp.StatusCode != http.StatusOK {
            resp.Body.Close()
            fmt.Fprintf(os.Stderr, "fetch: %s: %v\n", url, resp.Status)
            continue
        }
        doc, err := html.Parse(resp.Body)
        resp.Body.Close()
        if err != nil {
            fmt.Fprintf(os.Stderr, "fetch: parsing %s: %v\n", url, err)
            continue
        }
        images := ElementsByTagName(doc, "img")
        headings := ElementsByTagName(doc, "h1", "h2", "h3", "h4")
        fmt.Println(images)
        fmt.Println(headings)
    }
}

func ElementsByTagName(doc *html.Node, name ...string) []*html.Node {
    return visit(nil, doc, name)
}

func visit(links []*html.Node, n *html.Node, v []string) []*html.Node{
    if n.Type == html.ElementNode {
        for _, a := range v {
            if n.Data == a {
                links = append(links,  n)
                return links
            }
        }
    }
    for c := n.FirstChild; c != nil; c = c.NextSibling {
        links = visit(links, c, v) // 递归调用,
    }
    return links
}
```

Deffered 函数

- 一般判断出错的方法如下

```
package main

import (
    "fmt"
```

```
    "net/http"
    "strings"

    "golang.org/x/net/html"
)

func main() {

}
func title(url string) error {
    resp, err := http.Get(url)
    if err != nil {
        return err
    }
    // Check Content-Type is HTML (e.g., "text/html;charset=utf-8").
    ct := resp.Header.Get("Content-Type")
    if ct != "text/html" && !strings.HasPrefix(ct, "text/html;") {
        resp.Body.Close() // 多次调用关闭， 确保各种情况都会正常退出， 但是很麻烦
        return fmt.Errorf("%s has type %s, not text/html", url, ct)
    }
    doc, err := html.Parse(resp.Body)
    resp.Body.Close()
    if err != nil {
        return fmt.Errorf("parsing %s as HTML: %v", url, err)
    }
    visitNode := func(n *html.Node) {
        if n.Type == html.ElementNode && n.Data == "title" && n.FirstChild != nil
{
            fmt.Println(n.FirstChild.Data)
        }
    }
    forEachNode(doc, visitNode, nil)
    return nil
}


func forEachNode(n *html.Node, pre, post func(n *html.Node)) {
    if pre != nil {
        pre(n)
    }
    for c := n.FirstChild; c != nil; c = c.NextSibling {
        forEachNode(c, pre, post)
    }
    if post != nil {
        post(n)
    }
}
```

- 可以使用defer函数， 在调用普通函数或者方法的前面加上defer ， 当执行到该条语句时， 函数和参数表达式得到计算， 但函数并不执行。当函数返回时， 函数和参数表达式被执行。
- 执行顺序和声明顺序相反

- 在一些复杂的情况下， 可以使用defer函数， 确保函数在程序退出时执行。
- 示例

```
package main

import (
    "fmt"
    "net/http"
    "strings"

    "golang.org/x/net/html"
)

func main() {

}
func title(url string) error {
    resp, err := http.Get(url)
    if err != nil {
        return err
    }
    defer resp.Body.Close() //
    ct := resp.Header.Get("Content-Type")
    if ct != "text/html" && !strings.HasPrefix(ct, "text/html;") {
        return fmt.Errorf("%s has type %s, not text/html", url, ct)
    }
    doc, err := html.Parse(resp.Body)
    if err != nil {
        return fmt.Errorf("parsing %s as HTML: %v", url, err)
    }
    visitNode := func(n *html.Node) {
        if n.Type == html.ElementNode && n.Data == "title" && n.FirstChild != nil
{
            fmt.Println(n.FirstChild.Data)
        }
    }
    forEachNode(doc, visitNode, nil)
    return nil
}


func forEachNode(n *html.Node, pre, post func(n *html.Node)) {
    if pre != nil {
        pre(n)
    }
    for c := n.FirstChild; c != nil; c = c.NextSibling {
        forEachNode(c, pre, post)
    }
    if post != nil {
        post(n)
    }
```

```
}
```

- defer 也可以用来记录何时进入和退出函数

```
package main

import (
    "log"
    "time"
)

func main() {
    bigSlowOperation()
}

func bigSlowOperation() {
    defer trace("bigSlowOperation")() // 如果不加括号的的话， 则表示在退出时调用trace

    time.Sleep(10 * time.Second)
}

func trace(msg string) func() {
    start := time.Now()
    log.Printf("enter %s", msg)
    return func() {
        log.Printf("exit %s (%s)", msg, time.Since(start))
    }
}
```

- 注意defer函数是在函数结束后才执行， 而不是其他代码域
- 使用defer 改进fetch ， 将http相应信息写入本地文件而不是标准输出流

```
// fetch 改进版 将http相应信息写入本地文件而不是标准输出流

package main

import (
    "fmt"
    "io"
    "net/http"
    "os"
    "path"
)
//
func main() {
    for _, url := range os.Args[1:] {
        local, n, err := fetch(url)
```

```
        if err != nil {
            fmt.Fprintf(os.Stderr, "fetch: %v\n", err)
            return
        }
        fmt.Printf("%s %s %d\n", url, local, n)
    }
}

func fetch(url string) (filename string, n int64, err error) {
    resp, err := http.Get(url)
    if err != nil {
        return "", 0, err
    }
    defer resp.Body.Close() // 延迟关闭
    local := path.Base(resp.Request.URL.Path)
    if local == "/" {
        local = "index.html"
    }
    f, err := os.Create(local)
    if err != nil {
        return "", 0, err
    }
    n, err = io.Copy(f, resp.Body)
    if closeErr := f.Close(); err == nil {
        err = closeErr
    }
    return local, n, err
}
```

练习5.18

- 不修改fetch的行为，重写fetch函数，要求使用defer机制关闭文件。

```
//  不修改fetch的行为，重写fetch函数，要求使用defer机制关闭文件。

package main

import (
    "fmt"
    "io"
    "net/http"
    "os"
    "path"
)

func main() {
    for _, url := range os.Args[1:] {
        local, n, err := fetch(url)
        if err != nil {
            fmt.Fprintf(os.Stderr, "fetch: %v\n", err)
            return
```

```
        }
        fmt.Printf("%s %s %d\n", url, local, n)
    }
}

func fetch(url string) (filename string, n int64, err error) {
    resp, err := http.Get(url)
    if err != nil {
        return "", 0, err
    }
    defer resp.Body.Close() // 延迟关闭
    local := path.Base(resp.Request.URL.Path)
    if local == "/" {
        local = "index.html"
    }
    f, err := os.Create(local)
    if err != nil {
        return "", 0, err
    }
    n, err = io.Copy(f, resp.Body)
    defer func() {   // defer 执行顺序在return 之后， 但是在返回值赋值给调用方之前
        // 为什么defer能调用返回值，因为这里返回值是有名的， defer 函数只能访问有名返回
值
        if closeErr := f.Close(); err == nil {
            err = closeErr
        }
    }()
    return local, n, err
}
```

panic 异常 （宕机）

- 一般来说， panic异常是只能在运行时才能检查到的错误， 比如说数组访问越界， 空指针引用， 当panic发生时， 程序中断运行， 并立即执行在goroutine中被延迟的函数， 然后崩溃输出日志信息
- 一般来说， 不应用 panic 检查哪些运行时会检查的信息。而且只有比较严重的错误才应用panic
- 为了方便诊断问题，runtime包允许程序员输出堆栈信息。在下面的例子中，我们通过在main函数中延迟调用printStack输出堆栈信息。
-