入门

（1）Hello,World

1.示例

运行一个程序

编译一个程序

2. package(常见package,例如fmt; 特殊的，例如main)

3. go具有非常严格的代码格式要求，gofmt

（2）命令行参数，实现一个类似echo的程序

1. os.Args[1:]，os.Args[0]

2.for循环(\*)

for \_.arg := range os.Args[1:] {

…

}

3. strings.Join()，这个函数我的理解是，将目前属组拼接成一个字符串，(\*)

本文是用 fmt.Println(strings.Join(os.Args[1:],””)) , 本节最终版的echo 核心代码。简易

（3）查找重复的行，实现一个类似uniq的程序

1. map, go map基础，创建，常用使用方法 counts := make(map[string]int)

2. 介绍了bufio.NewScanner

bufio包，它使处理输入和输出方便又高效。Scanner类型是该包最有用的特性之一，它读取输入并将其拆成行或单词；通常是处理行形式的输入最简单的方法 \*

input := bufio.NewScanner(os.Stdin) 将标准输入读入input,

input.Scan 则是输出读入的值，非常方便

3. 文件操作 os.Open(file)

4. 常规文件操作流程

1. **for** \_, arg := range files { //进行文件进行数据读取，可多个;arg表示文件名
2. f, err := os.Open(arg) //打开文件操作
3. **if** err != nil { //常规的非true判定
4. fmt.Fprintf(os.Stderr, "dup2:%v\n", err)
5. **continue**
6. }
7. countLines(f, counts) //函数引用
8. f.Close()             //文件关闭
9. }

map是一个由make函数创建的数据结构的引用。map作为为参数传递给某函数时，该函数接收这个引用的一份拷贝（copy，或译为副本），被调用函数对map底层数据结构的任何修改，调用者函数都可以通过持有的map引用看到。在我们的例子中，countLines函数向counts插入的值，也会被main函数看到。（译注：类似于C++里的引用传递，实际上指针是另一个指针了，但内部存的值指向同一块内存）

4. 常规的非true判定

5. 在dup3版本中引入ReadFile(),该函数来源于io/ioutil，读取指定文件的全部内容

所以在改造完的dup3版本中，又引入了strings.Split该函数（与strings.Join相反）把字符串分割成子串的切片

1. **for** \_, filename := range os.Args[1:] {
2. data, err := ioutil.ReadFile(filename)   //将文件内容整个读入内存
3. **if** err != nil {
4. fmt.Fprint(os.Stderr, "dup3:%v\n", err)
5. **continue**
6. }
7. **for** \_, line := range strings.Split(string(data), "\n") {   //然后将读入内容按照\n切成独立的字符串，并入counts中
8. counts[line]++
9. }
10. **for** line, n := range counts {
11. **if** n > 1 {
12. fmt.Printf("%d\t%s\n", n, line)
13. }
14. }
15. }

5.分析

从dup3 和 dup2的程序上来分析，其实逻辑是不太一样的，

1. 如果同时操作两个文件，那么dup3在处理第二文件的时候，内存中也加载这第一个文件，耗费内存（缺点）
2. 但是ioutil.ReadFile(filename)，要比os.Open(filename)省事很多，再配合strings.Split(strings(data),”\n”)对文件行处理，非常方便

（4）只做一个GIF动画 [该课程程序结构无法演示，尴尬]

数学构造没有看明白，介绍了image, 提及了复合声明

（5）获取URL, 编写一个简单的程序事项curl (这个小节小有意思)

重要package net

1. resp.err := http.Get(url)
2. b.err := ioutil.ReadAll(resp.Body)
3. io.Copy，这个比较方便，不用开拓内存
4. ioutil.ReadAll 写法如下
5. b, err := ioutil.ReadAll(resp.Body)
6. resp.Body.Close()
7. **if** err != nil {
8. fmt.Fprintf(os.Stderr, "fetch:reading %s,%v\n", url, err)
9. os.Exit(1)
10. }
11. io.Copy()写法如下
12. \_, err = io.Copy(os.Stdout, resp.Body)     //将Copy的结果，输出到标准输出 1
13. **if** err != nil {
14. fmt.Fprintf(os.Stderr, "fetch:reding%s,%v\n", url, err)
15. }
17. //两种写法
18. **if** \_,err := io.Copy(os.Stdout, resp.Body) err != nil {    // 2
19. fmt.Fprintf(os.Stderr, "fetch:reding%s,%v\n", url, err)
20. }

4. strings.HasPrefix()，也很好用

1. **if** strings.HasPrefix(url, "http") != **true** {
2. url = "http://" + url
3. }

5. resp.Status，记录获取url的响应码Code，在最终的fmt.Printf中直接输出

1. fmt.Printf("%s\nCode-Status:%s\n",b,resp.Status)

（6）并发获取多个URL, em.. 从这节开始 我慢慢的感受到了go的魅力了

1. 重点：go的并发编程（goroutine）的两个元素

a. channel是用来在goroutine之间传递参数的

b. goroutine是一种函数的并发执行方式，main()函数本身也运行在一个goroutine中

2. 本章示例如下（4个核心句和1个核心逻辑）：

1. 初始化一个channel（通道）

1. ch := make(chan string)   //初始化一个 channel（通道），记住初始化写法
2. go 并发函数的使用
3. **for** \_, url := range os.Args[1:] {
4. **if** strings.HasPrefix(url, "http") != **true** {
5. url = "https://" + url
6. }
7. go fecth(url, ch) //go既是并发函数，看出来并发在go中是对么重要了吧!
8. }
9. 并发通过的传参方式
10. func fecth(url string, ch chan<- string) {
11. start := time.Now()
12. resp, err := http.Get(url)
13. **if** err != nil {
14. ch <- fmt.Sprint(err)
15. **return**
16. }
17. ...
18. }
19. gorutine其实只是并发，但是并发顺序由ch控制的（传参的本质-控制程序运行顺序）
20. **for** range os.Args[1:] {
21. fmt.Println(<-ch)   //本示例的数据处理的入口
22. }
23. 核心逻辑

示例的核心逻辑就是并发处理从命令行输入的url，然后通过ch 将命令行的url传入fecth处理，由于goroutine的运行逻辑，那么一个线程中所有资源通过channel(通道)进行处理，并且同一个时间一份用户数据之被一个goroutine的处理（这个本例子没用的，其他地方看的），并且fecth()中处理的所有结果都统一返回給man()进行输出，示例非常精炼

并发不会超出线程的总体运行时间

1. start := time.Now()   //记录start
2. ..
3. fmt.Printf("%.2fs elapsed\n", time.Since(start).Seconds())   //输出总时间
4. 本节课后题嘛，试了试 效果不大，没有展示