**Golang知识总结**

****

**测试技术研发办**

**2016年11月17 日**

目录

[欢迎加入golang大家庭 4](#_Toc467250780)

[目标读者 4](#_Toc467250781)

[运行环境 4](#_Toc467250782)

[文档修订记录 4](#_Toc467250783)

[第一章 概述 5](#_Toc467250784)

[1.1 背景知识 5](#_Toc467250785)

[1.2 golang优点缺点 5](#_Toc467250786)

[1.2.1 优点 5](#_Toc467250787)

[1.2.2 缺点 6](#_Toc467250788)

[1.3 golang用途 6](#_Toc467250789)

[1.4 总结 7](#_Toc467250790)

[第二章 知识点总括 8](#_Toc467250791)

[2.1 golang基本信息 8](#_Toc467250792)

[2.2 安装 9](#_Toc467250793)

[1.2.2 搭建开发环境 9](#_Toc467250794)

[1.2.2 安装LiteIDE编辑器 10](#_Toc467250795)

[1.2.2 新建小项目 13](#_Toc467250796)

[2.3 基本语法介绍 14](#_Toc467250797)

[2.3.1 顺序编程 14](#_Toc467250798)

[2.3.2 面向对象 15](#_Toc467250799)

[2.3.2 并发编程 17](#_Toc467250800)

[2.3.3 网络编程 22](#_Toc467250801)

[2.3.4 安全编程 24](#_Toc467250802)

[2.3.4 工程管理 24](#_Toc467250803)

[2.3.4 go调试 25](#_Toc467250804)

[2.3.5 知识点汇总 26](#_Toc467250805)

[第三章 实例说明 34](#_Toc467250806)

[3.1 xml 35](#_Toc467250807)

[3.2 orm 35](#_Toc467250808)

[3.3 日志处理 36](#_Toc467250809)

[3.4 Cookie操作 37](#_Toc467250810)

[3.5 接口方法 38](#_Toc467250811)

[3.6 匿名组合使用 40](#_Toc467250812)

[3.7 Go并发协程 42](#_Toc467250813)

[3.8 Channel 42](#_Toc467250814)

[3.9 读取配置 43](#_Toc467250815)

# 欢迎加入golang大家庭

—— 见证你的每一次成长

## 目标读者

测试、开发工程师。

## 运行环境

Windows XP 及以上系统。

## 文档修订记录

|  |  |  |  |
| --- | --- | --- | --- |
| 日期 | 版本 | 修订人 | 修订说明 |
| 2016-11 | V1.0 | 权杨杨 | 初稿 |

# 概述

## 1.1 背景知识

Go语言于2009年11月正式宣布推出，成为开放源代码项目，并在Linux及Mac Os X平台上进行了实现，后追加Windows系统下的实现，它特有的协程并发效率高并且其本身相比线程消耗小，切换快从而能够快速处理大量委托。

## 1.2 golang优点缺点

### 1.2.1 优点

go特有的优点有：

1. 部署简单。Go 编译生成的是一个静态可执行文件，除了 glibc 外没有其他外部依赖。这让部署变得异常方便：目标机器上只需要一个基础的系统和必要的管理、监控工具，完全不需要操心应用所需的各种包、库的依赖关系，大大减轻了维护的负担。
2. 丰富的标准库，Go目前已经内置了大量的库，特别是网络库非常强大。
3. 跨平台编译，如果你写的Go代码不包含cgo，那么就可以做到window系统编译linux的应用，如何做到的呢？Go引用了plan9的代码，这就是不依赖系统的信息。
4. 内嵌C支持，因为作者是C的作者，所以Go里面也可以直接包含c代码，利用现有的丰富的C库。
5. 静态类型语言，但是有动态语言的感觉，静态类型的语言就是可以在编译的时候检查出来隐藏的大多数问题，动态语言的感觉就是有很多的包可以使用，写起来的效率很高。
6. 内置runtime，支持垃圾回收。
7. 并发性好。Goroutine（协程） 和 channel 使得编写高并发的服务端软件变得相当容易，很多情况下完全不需要考虑锁机制以及由此带来的各种问题。单个 Go 应用也能有效的利用多个 CPU 核，并行执行的性能好。
8. 良好的语言设计。但从工程的角度讲，Go 的设计是非常优秀的：规范足够简单灵活，有其他语言基础的程序员都能迅速上手。更重要的是 Go 自带完善的工具链，大大提高了团队协作的一致性。比如 gofmt 自动排版 Go 代码，很大程度上杜绝了不同人写的代码排版风格不一致的问题。把编辑器配置成在编辑存档的时候自动运行 gofmt，这样在编写代码的时候可以随意摆放位置，存档的时候自动变成正确排版的代码。
9. 执行性能好。内存占用也非常小。
10. 适用性好，可以做从底层到前端的任何工作。

### 1.2.2 缺点

1、 Go的import包不支持版本，有时候升级容易导致项目不可运行，所以需要自己控制相应的版本信息。

2、 Go的goroutine一旦启动之后，不同的goroutine之间切换不是受程序控制，runtine调整的时候，需要严谨的逻辑，否则因goroutine休眠等导致逻辑出错等情况。

3、 GC的延迟有点大，处理很大的日志，GC（垃圾回收器）没有那么快，内存回收不给力。GC比起Java还是不够成熟。

4、 图片处理库还有一些bug。

## 1.3 golang用途

Golang适合用来做什么：

1、 服务器编程，例如处理日志，数据打包，虚拟机处理，文件系统。

2、 分布式系统，数据库代理器等。

3、 网络编程，这一块目标应用最广，包括Web应用，API应用，下载应用。

4、 内存数据库，google开发的groupcache、couchbase的部分组建。

5、 云平台，目前国外很多云平台在采用Go开发，CloudFoundy的部分组建。

## 1.4 总结

golang轻松实现高并发；代码简洁、格式统一、阅读方便；开发效率也不差于python等动态语言。适用于服务器编程、网络编程、云计算，国外很多公司，包括国内阿里等一些知名公司云计算都开始广泛使用golang。Golang已成为流行趋势，golang现在还不是像java这样的成熟语言，相信在不久的以后golang一定会有更广泛的应用。

# 知识点总括

## 2.1 golang基本信息

go语言的运行环境：

|  |  |  |  |
| --- | --- | --- | --- |
| 操作系统 | Go运行环境 | 优点 | 缺点 |
| 全平台 | LiteIDE | 界面简单，易操作 | 调试功能不好用 |
| Vscode（Visual Studio Code） | 调试方便 |  |

注：目前搭建操作系统为Windows

go的一些网站：

go官网：<http://studygolang.com/Go>

beego：<https://github.com/astaxie/beego>

<http://beego.me/>

go社区：<http://www.golangtc.com/go/liteide>

beego入门文档：<http://my.oschina.net/astaxie/blog/124040>

go视频资料：www.vker.org

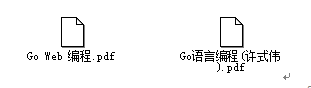
go语言学习库：<http://beego.me/docs/mvc>

beego中github下载：<https://github.com/Unknwon/com>

go中list实现：<http://studygolang.com/articles/4371>

mysql安装地址：<http://jingyan.baidu.com/album/597035521d5de28fc00740e6.html?picindex=3>

推荐书籍：





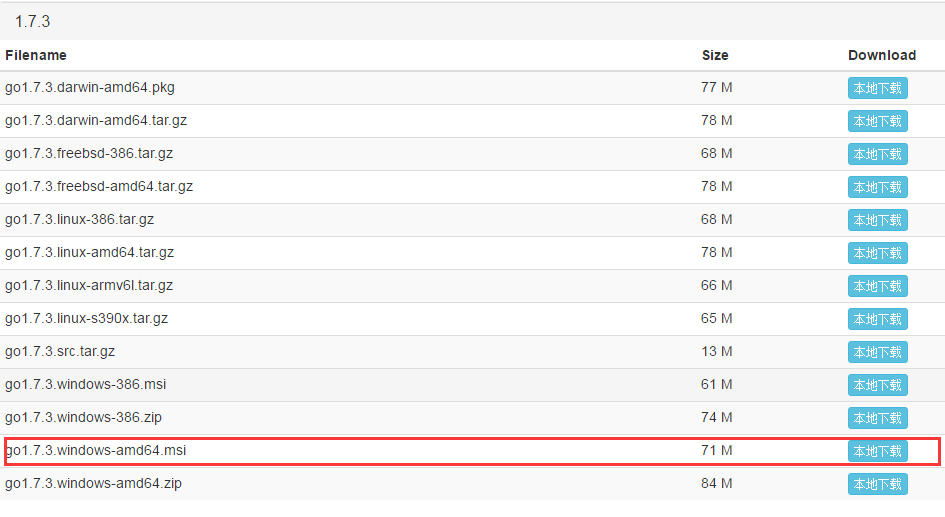
## 2.2 安装

**golang编辑器安装及安装说明：**[**http://studygolang.com/articles/7465**](http://studygolang.com/articles/7465)

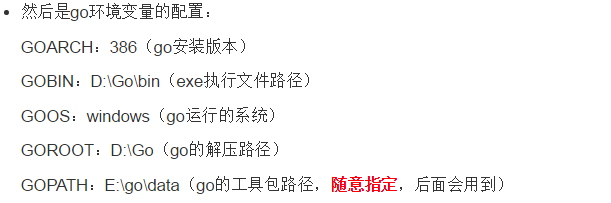
### 1.2.2 搭建开发环境

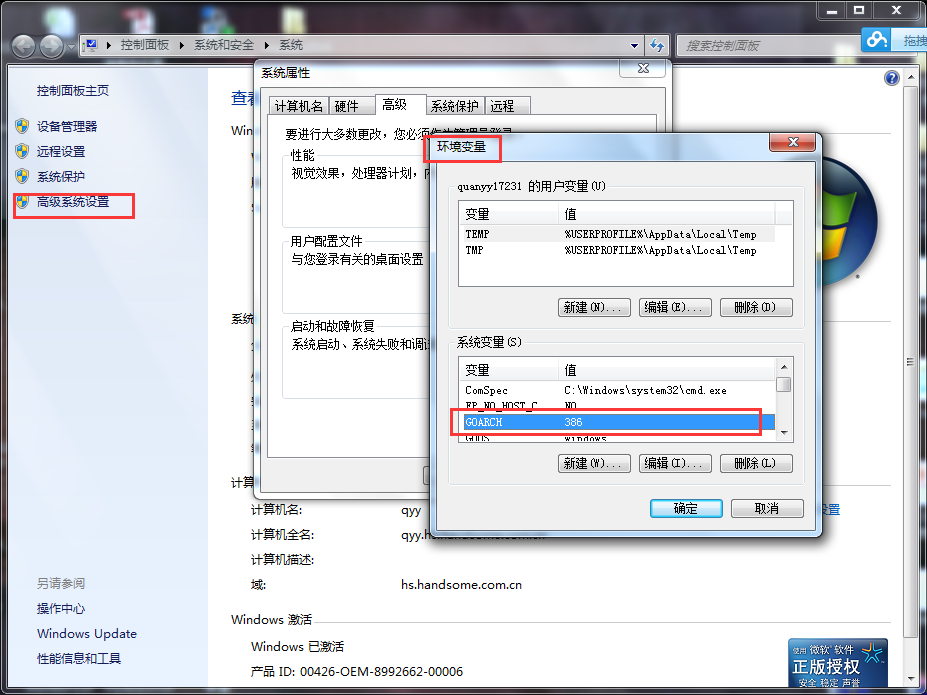
1、下载地址：

安装程序下载地址： <http://www.golangtc.com/download>



2、检查当前环境配置：





3、之后在cmd中输入go version

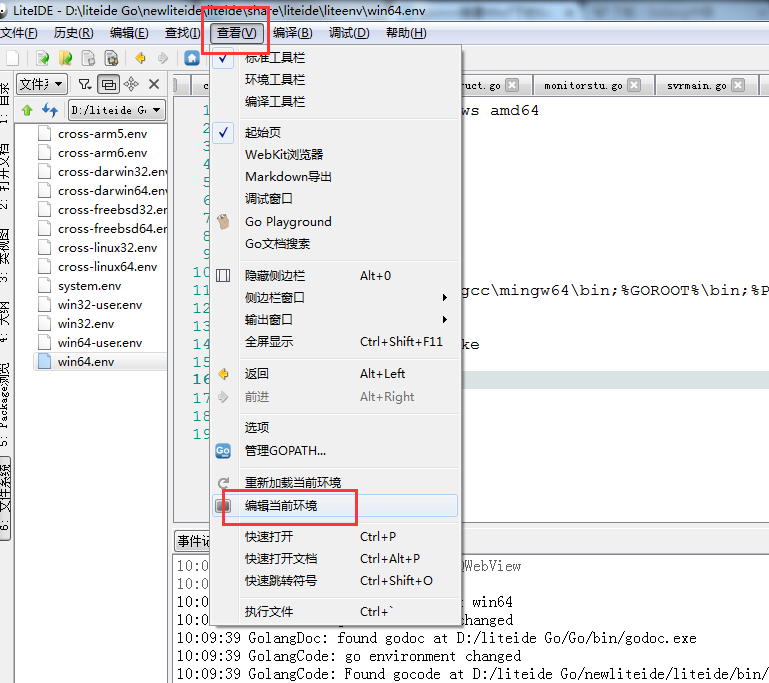


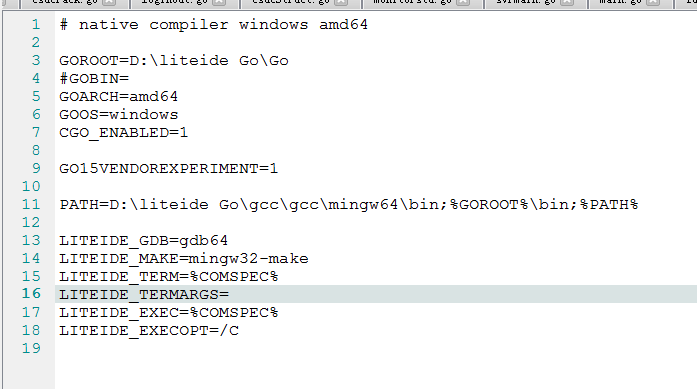
### 1.2.2 安装LiteIDE编辑器

1、LiteIDE下载地址：http://www.golangtc.com/download/liteide

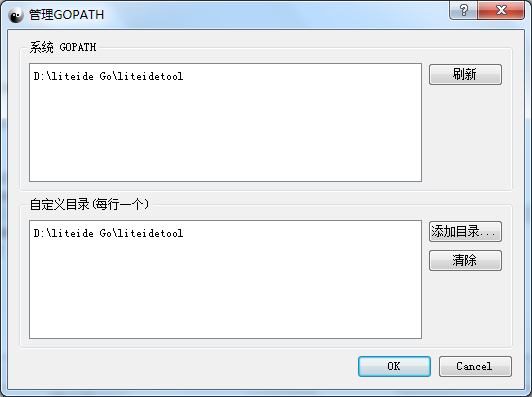


2、下载并安装完成之后，打开应用程序，“查看”-->"编辑当前环境",确认GOROOT变量与GO安装路径一致：

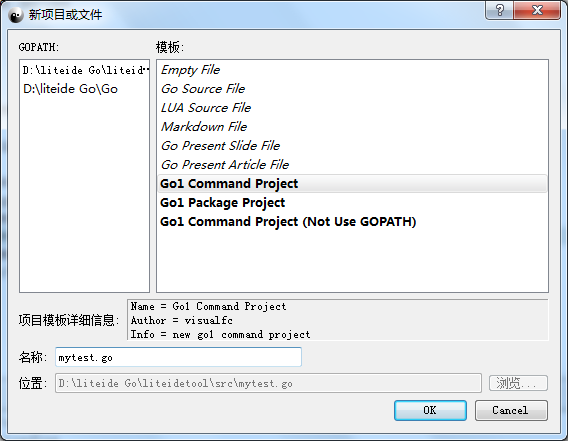




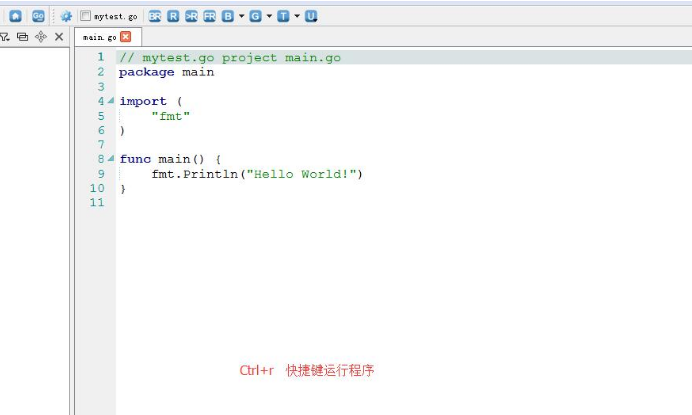
3、“查看”-->"管理GOPATH"(默认的就是安装go的时候，随意指定的那个路径)



### 1.2.2 新建小项目



运行结果



说明：具体安装说明：[**http://studygolang.com/articles/7465**](http://studygolang.com/articles/7465)

## 2.3 基本语法介绍

### 2.3.1 顺序编程

* Go支持的复合类型

指针（pointer）

数组（array）

切片（slice）

字典（map）

通道（chan）

结构体（struct）

接口（interface）

1、变量声明

|  |
| --- |
| 变量类型 |
| var v1 int |
| var v2 string |
| var v3 **[**10**]**int |
| var v4 **[]**int |
| var v5 struct**{**  f int  **}** |
| var v6 **\*** int |
| var v7 map **[**string**]**int |
| var v8 func **(**a int**)** int |
|  |

2、变量初始化

|  |
| --- |
| 变量类型 |
| var v1 int **=** 10 |
| var v2 **=** 10 |
| v3 **:=** 10 |

3、变量赋值

|  |
| --- |
| 多重赋值 |
| i **,** j **=** j **,** i |

4、匿名变量

|  |
| --- |
| 获得函数调用返回值 |
| \_**,**\_**,**nickName **:=** GetName**()** |

5、预定义常量、枚举

|  |
| --- |
| 预定义常量 |
| const**(** //itoa被重设为0  c0 **=** iota //c0 == 0  c1 //c1 == 1  c2 //c2 == 2  **)** |

6、for循环

|  |
| --- |
| For循环 |
| **for** key**,** value **:=** range oldMap **{**  newMap**[**key**]** **=** value  **}** |

### 2.3.2 面向对象

1、结构体

|  |
| --- |
| 结构体 |
| type Rect struct**{**  x**,**y float64  width**,**height float64  **}** |

2、初始化

|  |
| --- |
| 根据上面结构体进行Rect类型初始化 |
| rect1 **:=** **&**Rect**{}**  rect2 **:=** **&**Rect**{**0**,**0**,**100**,**100**}**  rect3 **:=** **&**Rect**{**width**:**100**,**height**:**200**}** |

3、匿名组合

|  |
| --- |
| 匿名组合继承 |
| type Base struct**{**  Name string  **}**  func **(**base **\***Base**)** Foo**(){**…**}**  func **(**base **\***Base**)** Bar**(){**…**}**  type Foo struct**{**  Base  **}**  func **(**foo **\*** Foo**)** Bar**(){**  foo**.**Base**.**Bar**()**  **}** |

说明：定义了一个Base类，实现了Foo()和bar()两个成员方法，然后定义了一个Foo类，该类从Base类继承并改写了Bar()方法，该方法实现时先调用了基类的Bar()方法。在派生类Foo没有改写基类Base的成员方法时，相应的方法就被继承。

4、Any类型

|  |
| --- |
| Any类型 |
| var v1 interface**{}** **=** 1  var v2 interface**{}** **=** "abc"  var v3 interface**{}** **=** **&**v2  var v4 interface**{}** **=** struct**{**x int**}{**1**}**  var v5 interface**{}** **=** **&**struct**{**x int**}{**1**}**  func Print**(**fmt string**,**arge …interface**{})**  func Println**(**args …interface**{})** |

注：可以指定任何对象，函数可以接受任意的对象实例。

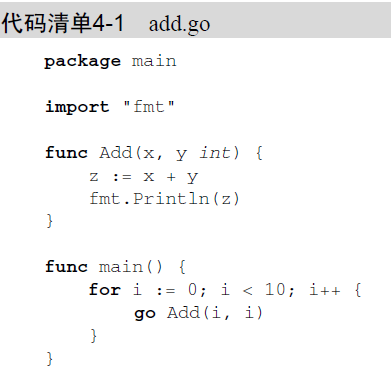
### 2.3.2 并发编程

1、协程

执行体是个抽象的概念，在操作系统层面有多个概念与之对应，比如操作系统自己掌管的进程（process）、进程内的线程（thread）以及进程内的协程（coroutine，也叫轻量级线程）。与传统的系统级线程和进程相比，协程的最大优势在于其“轻量级”，可以轻松创建上百万个而不会导致系统资源衰竭，而线程和进程通常最多也不能超过1万个。这也是协程也叫轻量级线程的原因。

Go 语言在语言级别支持轻量级线程，叫goroutine。Go 语言标准库提供的所有系统调用操作（当然也包括所有同步 IO 操作），都会出让 CPU 给其他goroutine。这让事情变得非常简单，让轻量级线程的切换管理不依赖于系统的线程和进程，也不依赖于CPU的核心数量。

如何让函数并发执行？

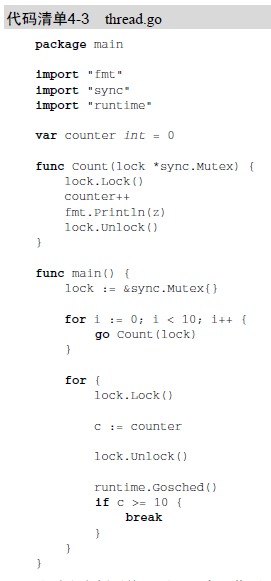


go Add(i , i)

“go”这个单词是关键，在一个函数调用前加上go关键字，这次调用就会在一个新的goroutine中并发执行。当被调用的函数返回时，这个goroutine也自动结束了。

2、并发通信

* 加锁



* channel通信

channel是go语言中一个独特的存在，是go语言在语言级别提供的goroutine间的通信方式，我们可以使用channel在两个或多个goroutine之间传递消息。Channel为先进先出。

Channel的声明形式：

* var chanName chan ElementType
* var ch chan int

与一般的变量声明不同的地方仅仅是在类型之前加了chan关键字

* 声明一个map，元素是bool型的channel：**var m map [string] chan bool**
* 定义一个channel，使用内置的函数make()：**ch := make(chan int)**
* 将一个数据写入（发送）至channel：

ch <- value

value := <-ch

* 单项channel定义：

Ch4 **:=** make**(**chan int**)**

Ch5 **:=** **<-** chan int**(**ch4**)** //ch5为单向的读取channel

Ch6 **:=** chan **<-** int**(**ch4**)** //ch6是单向的写入channel

* Channel的超时机制：timeout := make(chan bool, 1)
* Channel关闭：

x**,** ok **:=** **<-**ch //第二个返回false

close**();**

* 同步

倡导用通信来共享数据，而不是通过共享数据来进行通信。Mutex是最简单的一种锁类型，同时也比较暴力，当一个goroutine获得了Mutex后，其他goroutine就只能乖乖等到这个goroutine释放该Mutex。RWMutex相对友好些，是经典的单写多读模型。

* Once操作：

保证只调用一次

var a string

var once sync**.**Once

func setup**()** **{**

a **=** "hello, world"

**}**

func doprint**()** **{**

once**.**Do**(**setup**)**

print**(**a**)**

**}**

func twoprint**()** **{**

go doprint**()**

go doprint**()**

**}**

* Select

select **{**

**case** **<-**chan1**:**

// 如果chan1成功读到数据，则进行该case处理语句

**case** chan2 **<-** 1**:**

// 如果成功向chan2写入数据，则进行该case处理语句

**default:**

// 如果上面都没有成功，则进入default处理流程

**}**

* 缓冲机制

c := make(chan int, 1024)

创建了一个大小为1024的int类型channel，写入方一直往channel里写入，在缓冲区被填完之前都不会阻塞。

* 非侵入式接口

type File struct **{**

// ...

**}**

func **(**f **\***File**)** Read**(**buf **[]**byte**)** **(**n int**,** err error**)**

func **(**f **\***File**)** Write**(**buf **[]**byte**)** **(**n int**,** err error**)**

func **(**f **\***File**)** Seek**(**off int64**,** whence int**)** **(**pos int64**,** err error**)**

func **(**f **\***File**)** Close**()** error

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

type IFile interface **{**

Read**(**buf **[]**byte**)** **(**n int**,** err error**)**

Write**(**buf **[]**byte**)** **(**n int**,** err error**)**

Seek**(**off int64**,** whence int**)** **(**pos int64**,** err error**)**

Close**()** error

**}**

type IReader interface **{**

Read**(**buf **[]**byte**)** **(**n int**,** err error**)**

**}**

type IWriter interface **{**

Write**(**buf **[]**byte**)** **(**n int**,** err error**)**

**}**

type ICloser interface **{**

Close**()** error

**}**

var file1 IFile **=** **new(**File**)**

var file2 IReader **=** **new(**File**)**

var file3 IWriter **=** **new(**File**)**

var file4 ICloser **=** **new(**File**)**

* 接口赋值

package one

type ReadWriter interface **{**

Read**(**buf **[]**byte**)** **(**n int**,** err error**)**

Write**(**buf **[]**byte**)** **(**n int**,** err error**)**

**}**

package two

type IStream interface **{**

Write**(**buf **[]**byte**)** **(**n int**,** err error**)**

Read**(**buf **[]**byte**)** **(**n int**,** err error**)**

**}**

var file1 two**.**IStream **=** **new(**File**)**

var file4 Writer **=** file1

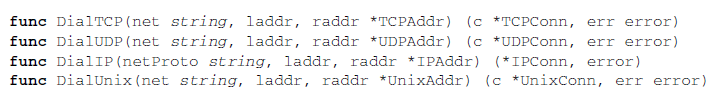
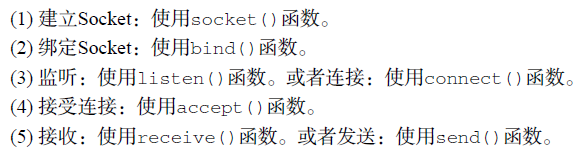
var file1 Writer **=** **new(**File**)**

var file5 two**.**IStream **=** file1 // 编译不能通过

//字段多的可以给字段少的进行赋值操作，反过来则不行

### 2.3.3 网络编程

* Socket编程



* Json处理

可以对一组数据进行Json格式的编码

Json.Marshal();

**举例：**将结构体编码成json数据







### 2.3.4 安全编程

数据加密

单密钥的加密算法：对称加密

双密钥的加密算法：非对称加密

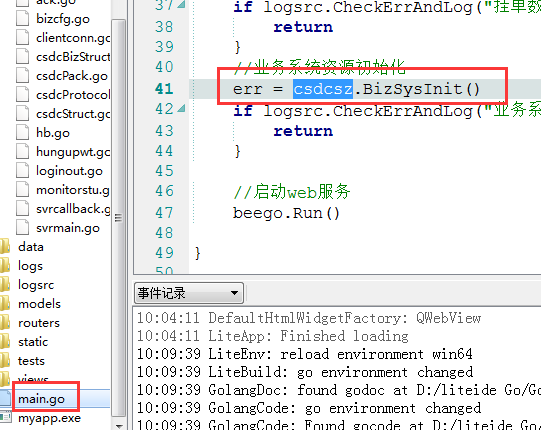
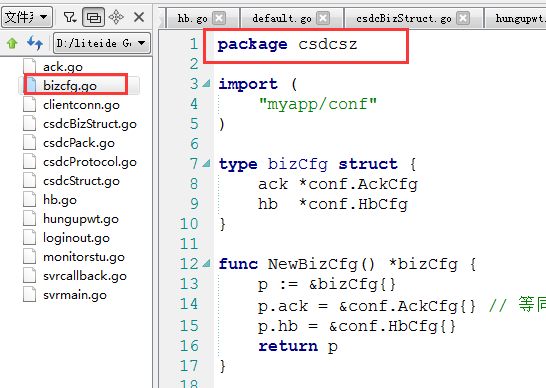
MD5、SHA-1加密

### 2.3.4 工程管理

* Package包

golang package是基本的管理单元，

同一个package下面，可以有非常多的不同文件，只要每个文件的头部都有如 "package xxx" 的相同name，

就可以在主方法中使用 xxx.Method()调用不同文件中的方法了。

* 反射

**import** **(**

"reflect"

**)**

reflect**.**TypeOf**()**

reflect**.**ValueOf**(**x**)** //参数传递，这是一个值传递，传递的是x的值的一个副本，不是x本身。

### 2.3.4 go调试

* Go中主要有两种调试模式，debug和delve
* 调试快捷键

F9打断点

F5进入调试状态

F10下一步

F2是快捷键

### 2.3.5 知识点汇总

* 每个中文字符在UTF-8中占3个字节
* go语言支持goto关键字
* 判断传入参数的类型

package main

**import** "fmt"

func MyPrintf**(**args **...** interface**{})** **{**

**for** \_**,** arg **:=** range args **{**

**switch** arg**.** **(**type**)** **{**

**case** int**:**

fmt**.**Println**(**arg**,** "is an int value."**)**

**case** string**:**

fmt**.**Println**(**arg**,** "is a string value."**)**

**case** int64**:**

fmt**.**Println**(**arg**,** "is an int64 value."**)**

**default:**

fmt**.**Println**(**arg**,** "is an unknown type."**)**

**}**

**}**

**}**

func main**()** **{**

var v1 int **=** 1

var v2 int64 **=** 234

var v3 string **=** "hello"

var v4 float32 **=** 1.234

MyPrintf**(**v1**,** v2**,** v3**,** v4**)**

**}**

* 错误处理

type error interface **{**

Error**()** string

**}**

* defer延迟函数
* panic函数

当在一个函数执行过程中调用panic() 函数时，正常的函数执行流程将立即终止，但函数中之前使用defer关键字延迟执行的语句将正常展开执行

* 关键包

values **=** make**([]**int**,** 0**)** //创建数组切片

**import** "bufio"

**import** "flag"

**import** "fmt"

**import** "io"

**import** "os"

**import** "strconv"

data\_path **:=** flag**.**String**(**"D"**,**"/home/manu/sample/"**,**"DB data path"**)**

其中flag可以解析路径

os获取系统文件

bufio 处理缓存I/O

strconv字符串转换的一个包

* 引用

var a **=** **[**3**]**int**{**1**,** 2**,** 3**}**

var b **=** **&**a

b**[**1**]++**

fmt**.**Println**(**a**,** **\***b**)**

**[**1 3 3**]** **[**1 3 3**]**

* 分割字符串

tokens := strings.Split(line, " ") //分割字符串

* 文件操作

在Go语言中，文件是使用一个os.File类的对象指针表示的，也可以称这指针为文件句柄（filehandle），os.Stdin和os.Stdout也是属于这个\*os.File类型的。先得到文章的句柄，然后再定义读的对象

* 时间包

Import "time"

time**.**Sleep**(**5 **\*** time**.**Second**)**

* Strconv

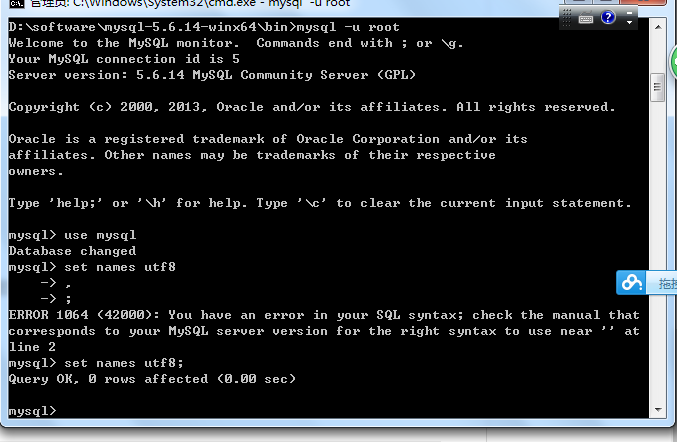
strconv.ParseInt(id,10,64);

//id是string类型，10代表十进制，64代表64位。

* Mysql

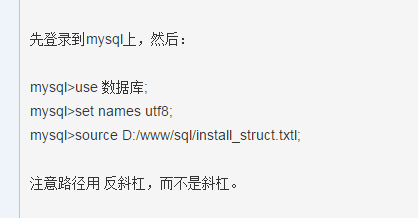
mysql安装地址：

<http://jingyan.baidu.com/album/597035521d5de28fc00740e6.html?picindex=3>



其中set names utf8 设置客户端编码

导入sql文件：

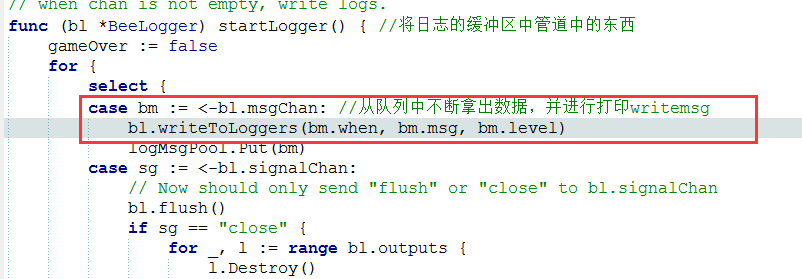
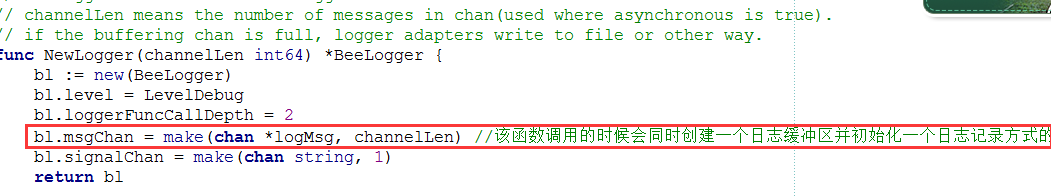


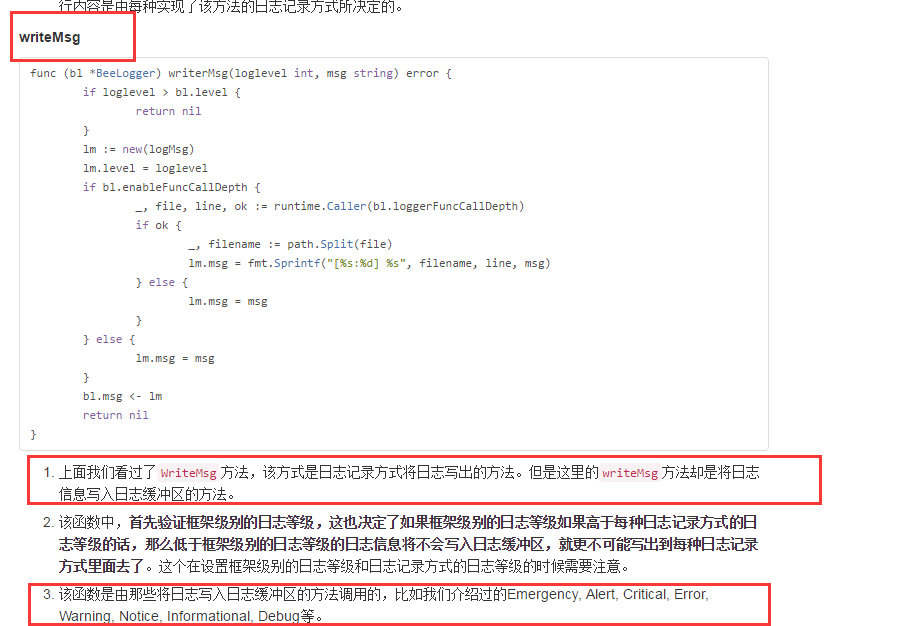
* Log.go文件的理解

具体路径为：github.com\astaxie\beego\logs\log.go

log.go文件理解：<http://golanghome.com/post/284>

分析log.go的流程：



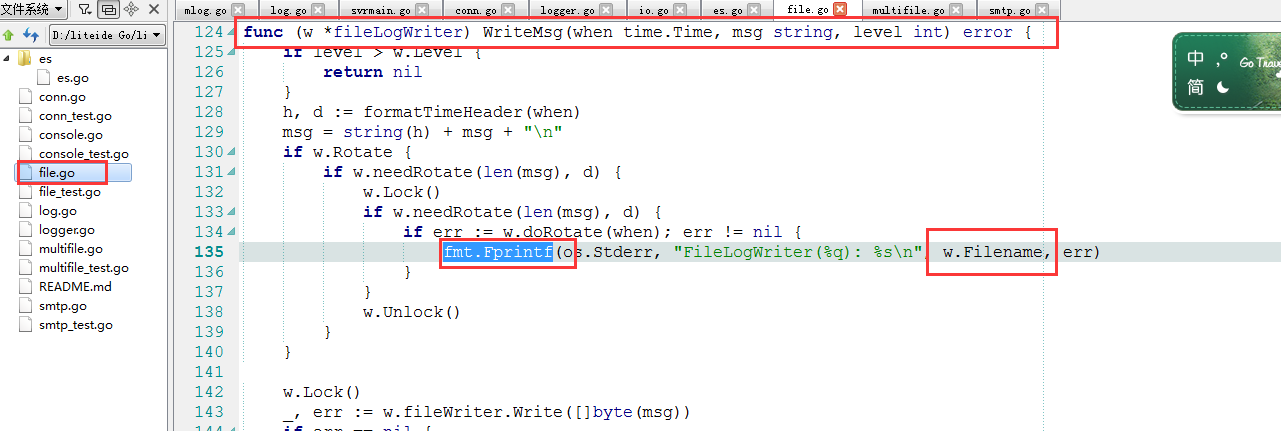
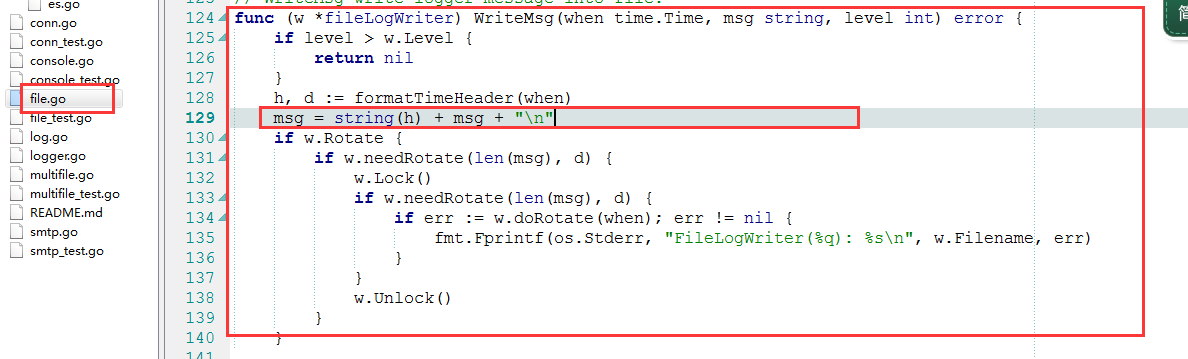




Log.go当前目录下各个文件的用途：

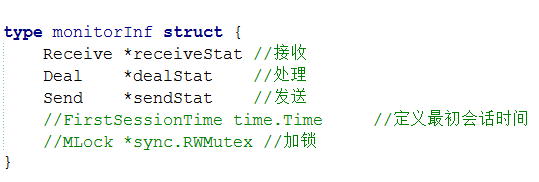


File.go日志记录方式之文件输出的实现文件



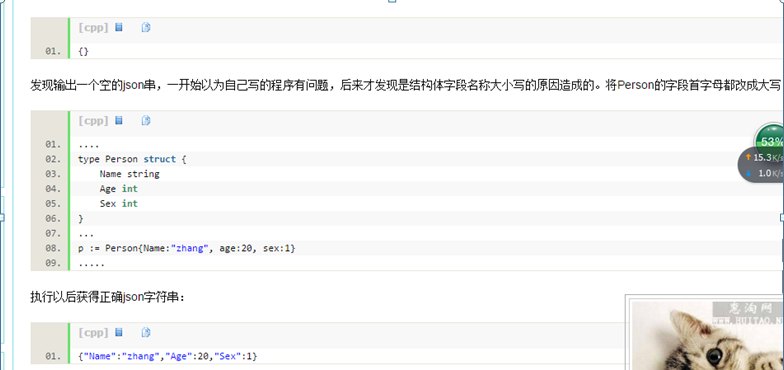
* 结构体字段的私有性

同样，首字母大写的字段可以被导出，也就是说，在其他包中可以进行读写。字段名以小写字母开头是当前包的私有的，函数定义也是类似的。



**举例**：Json生成：



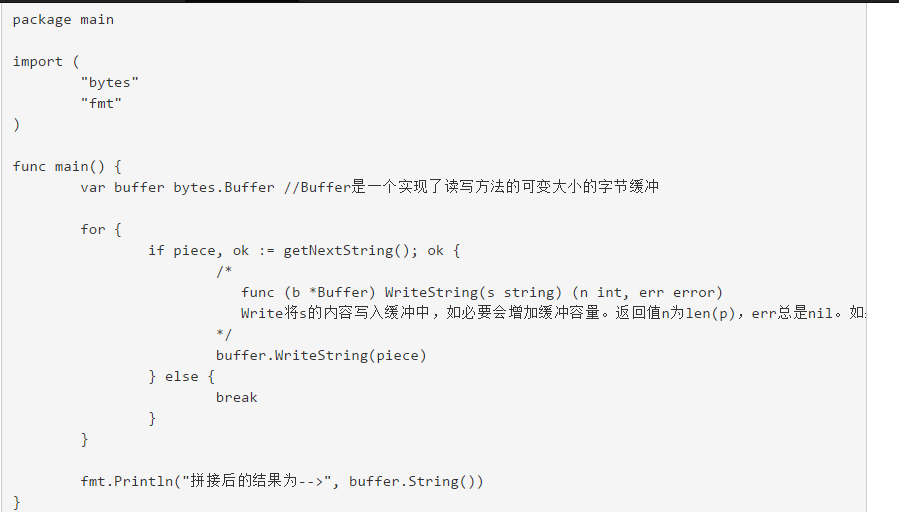


* 实现路径参数获取

//http://localhost:8080/?IsFirst=false

**if** **this.**GetString**(**"IsFirst"**)** **==** "false" **{}**

* 高效拼接字符



附录原网址<http://studygolang.com/articles/3427>

# 实例说明

本章主要结合中登D-COM网关模拟项目对go语言的具体使用来进行说明。整体架构类似于java的mvc：model、view、controller。

整体项目构架：

直接调用beego.run。切入函数run中，run中设置了监听http。主要功能是开了8080端口，进行tcp监听。

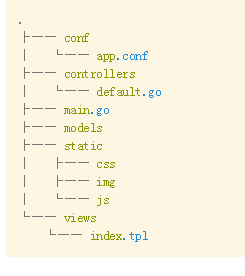
Onconnect函数中，主要是用来连接TCP进行调用。

Onmessage函数中，主要是TCP接收到消息进行调用

系统会去router.go中进行路由器注册，给每个控制器controller注册路径，然后去controller中进行执行，controller中有对应的view页面，view页面会通过model中的数据、controller中的执行条件进行页面渲染。



Liteide基本结构：



## 3.1 xml

type commonHbMsg struct **{**

XMLName xml**.**Name `xml**:**"Msg"` // 不加该行，根节点为 djdjhbMsg

AppHdr appHdr `xml**:**"AppHdr"`

BizTp string `xml**:**"Document>Data>BizTp"`

InstrTp string `xml**:**"Document>Data>InstrTp"`

SrlNo string `xml**:**"Document>Data>ConfInf>ConfDtls>SrlNo"`

OrgnlOrdrDtls IOrgOrd `xml**:**"Document>Data>ConfInf>ConfDtls>OrgnlOrdrDtls"`

RcvTm string `xml**:**"Document>Data>ConfInf>ConfDtls>RcvTm"`

Cd string `xml**:**"Document>Data>ConfInf>ConfDtls>Rslt>Cd"`

Desc string `xml**:**"Document>Data>ConfInf>ConfDtls>Rslt>Desc"`

**}**

## 3.2 orm

* **Orm数据库创建**

model**.**go

package models

**import** **(**

"github.com/astaxie/beego/orm"

\_ "github.com/mattn/go-sqlite3"

）

const **(**

\_DB\_NAME **=** "data/zhongdeng.db" //数据库

\_SQLITE3\_DRIVER **=** "sqlite3" //驱动

**)**

func RegisterDB**()** **{** //创建数据库

**if** **!**com**.**IsExist**(**\_DB\_NAME**)** **{** //如果数据库不存在

os**.**MkdirAll**(**path**.**Dir**(**\_DB\_NAME**),** os**.**ModePerm**)**

os**.**Create**(**\_DB\_NAME**)**

**}**

//注册模型model

orm**.**RegisterModel**(new(**Order**),** **new(**Monitor**))**

orm**.**RegisterDriver**(**\_SQLITE3\_DRIVER**,** orm**.**DRSqlite**)**

//第二个参数是类型

orm**.**RegisterDataBase**(**"default"**,** \_SQLITE3\_DRIVER**,** \_DB\_NAME**,** 30**)** //( 表名(必须有)，驱动，数据库，最大连接数)

logsrc**.**WLog**(**logsrc**.**LevelInformational**,** "数据库建立成功！"**)**

**}**

main**.**go

**import** "github.com/astaxie/beego/orm" //数据库需要使用的

func init**()** **{**

models**.**RegisterDB**()** //注册数据库及表结构

**}**

func main**()** **{**

err **:=** orm**.**RunSyncdb**(**"default"**,** **false,** **true)**

//自动建表,第二个参数为true时每次运行都要删除表后重新建表

beego**.**Run**()** //启动web服务

**}**

* **orm数据库结构体定义**

type Online struct **{**

Id int

Orderid int64 `orm**:**"index"` //挂单编号，索引

BizTp string //业务类型

OrdrInf string //委托数据

OrdrDtls string //委托明细

SrlNo string //业务流水号

ReplyTime time**.**Time

**}**

* **Orm数据查询**

对表信息进行过滤

//根据id查询出记录

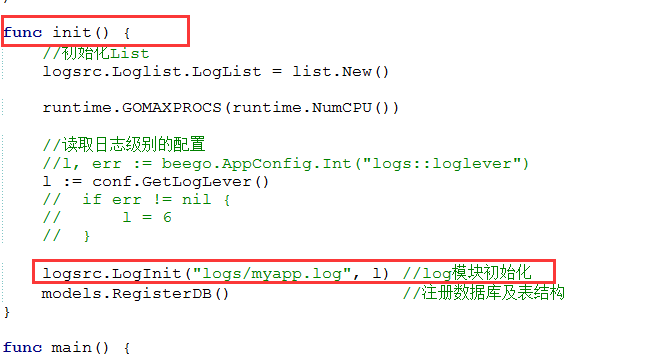
o **:=** orm**.**NewOrm**()**

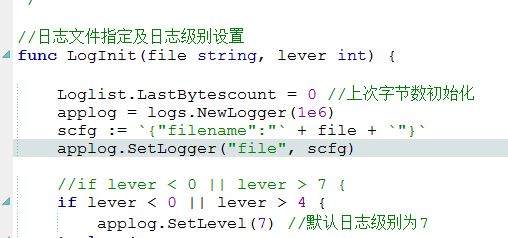
rec **:=** **&**models**.**Order**{}**

err **=** o**.**QueryTable**(**"Order"**).**Filter**(**"Id"**,** cid**).**Limit**(**1**).**One**(**rec**)**

## 3.3 日志处理

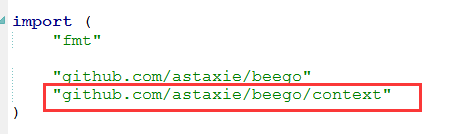
Main.go

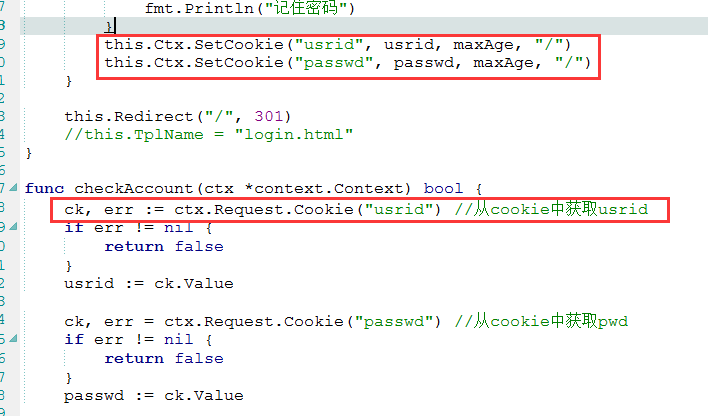




## 3.4 Cookie操作

导入包





## 3.5 接口方法

使用interface中的any接口，可以给它赋值任意类型，并可以调用对应类型的方法。

package csdcsz

**import** **(**

"encoding/xml"

"fmt"

"time"

**)**

func autoHbLine**()** **{**

**for** **{**

**if** gSvrStop **{**

**break**

**}**

**if** gConn**.**IsLogged**()** **==** **false** **{**

time**.**Sleep**(**HeartBeat **/** 10**)**

**continue**

**}**

mypack **:=** **<-**ghbQueue //chan关闭？pitodo

fmt**.**Println**(**"autoHbLine--出队列"**)**

err **:=** dohb**(**mypack**)** //按业务进行回报处理

**if** err **!=** nil **{**

fmt**.**Println**(**"hbDeal失败" **+** err**.**Error**())**

**}**

**}**

fmt**.**Println**(**"autoHbLine stop"**)**

**}**

func dohb**(**wt **\***tMyPack**)** error **{**

m **:=** wt**.**getXmlType**()** //xml类型

var p INewHbPack

**switch** m **{**

**case** "XHDJWT"**:** //冻结类

p **=** **&**xhdjhbMsg**{}**

**case** "XHHZWT"**:** //划转类

p **=** **&**xhhzhbMsg**{}**

**default:**

fmt**.**Println**(**"Unsupported type"**,** m**)**

**return** nil //pitodo

**}**

anspack **:=** p**.**newHbPack**(**wt**)** //组回报报文

**if** anspack **!=** nil **{**

err **:=** gConn**.**sendXmlPack**(**anspack**)** //发送

**return** err

**}** **else** **{**

fmt**.**Println**(**"组包失败"**)** //pitodo

**return** nil

**}**

**}**

//--各种业务回报消息需要实现如下接口-----------------

type INewHbPack interface **{**

newHbPack**(**wt **\***tMyPack**)** **[]**byte

**}**

//现货冻结消息类实现INewHbPack接口-----------------

func **(**p **\***xhdjhbMsg**)** newHbPack**(**wt **\***tMyPack**)** **[]**byte **{}**

//现货划转消息类实现INewHbPack接口-----------------

func **(**p **\***xhhzhbMsg**)** newHbPack**(**wt **\***tMyPack**)** **[]**byte **{}**

## 3.6 匿名组合使用

type hkHbMsgBase struct **{**

XMLName xml**.**Name `xml**:**"Msg"` // 不加该行，根节点为 hkHbMsgBase

AppHdr appHdr `xml**:**"AppHdr"`

SrlNo string `xml**:**"Document>Data>SrlNo"`

RcvTm string `xml**:**"Document>Data>RcvTm"`

RsltCd string `xml**:**"Document>Data>RsltCd"`

Desc string `xml**:**"Document>Data>Desc"`

**}**

type hkWt\_ca struct **{**

BizTp string `xml**:**"Document>Data>BizTp"` //可以知道业务类别

InstrctnTp string `xml**:**"Document>Data>InstrctnTp"` //可以知道是否撤单

RefNo string `xml**:**"Document>Data>RefNo"`

CrpActCd string `xml**:**"Document>Data>CrpActCd"`

PrvSrlNo string `xml**:**"Document>Data>PrvSrlNo"`

InvstrAcct string `xml**:**"Document>Data>InvstrAcct"`

CtdnUnt string `xml**:**"Document>Data>CtdnUnt"`

Qty string `xml**:**"Document>Data>Qty"`

**}**

//匿名组合等于继承

type hkHbMsg\_ca struct **{**

hkHbMsgBase

hkWt\_ca

**}**

func dohb**(**wt **\***tMyPack**)** error **{**

m **:=** wt**.**getXmlType**()** //xml类型

var anspack **[]**byte

**if** m**[:**2**]** **==** "HK" **{**

p **:=** **&**hkHbMsgBase**{}**

anspack **=** p**.**newHbPack**(**wt**)** //组回报报文

**}** **else** **{**

p **:=** **&**commonHbMsg**{}**

anspack **=** p**.**newHbPack**(**wt**)** //组回报报文

**}**

**}**

func **(**p **\***hkHbMsgBase**)** newHbPack**(**wt **\***tMyPack**)** **[]**byte **{**

m **:=** wt**.**getXmlType**()** //xml类型

var **(**

phb interface**{}** //指向整个回报结构体对象

pbase interface**{}** //指向回报对象的公共部分

pwt interface**{}** //指向回报对象的业务委托字段部分

**)**

**switch** m **{**

**case** "HKCAWT"**:**

phb **=** **&**hkHbMsg\_ca**{}** //创建一个ca对象

pbase **=** **&(**phb**.(\***hkHbMsg\_ca**)).**hkHbMsgBase //pbase指向ca对象的base

pwt **=** **&(**phb**.(\***hkHbMsg\_ca**)).**hkWt\_ca //pwt指向ca对象的hkWt\_ca

**case** "HKTPWT"**:**

phb **=** **&**hkHbMsg\_tp**{}** //创建一个tp对象

pbase **=** **&(**phb**.(\***hkHbMsg\_tp**)).**hkHbMsgBase //pbase指向tp对象的base

pwt **=** **&(**phb**.(\***hkHbMsg\_tp**)).**hkWt\_tp //pwt指向tp对象的hkWt\_tp

**default:**

logsrc**.**WLog**(**logsrc**.**LevelError**,** "暂不支持的消息类型："**+**m**)**

**return** nil

**}**

p**.**setHKHbMsgBase**(**wt**,** pbase**)**

err **:=** xml**.**Unmarshal**(**wt**.**pack**.**GetXml**(),** pwt**)** //直接把委托报文中的业务字段解析到pwt对象里

**if** logsrc**.**CheckErrAndLog**(**"回报业务节点赋值异常"**+**m**,** err**)** **{**

**return** nil

**}** **else** **{**

**return** p**.**packHkHbMsg**(**phb**)**

**}**

**}**

注：

hkHbMsg\_ca的结构体中有两个匿名变量：hkHbMsgBase、hkWt\_ca，则hkHbMsg\_ca分别继承了hkHbMsgBase、hkWt\_ca两个的特性，方法。

## 3.7 Go并发协程

go preCheckLine**()** //预检线程

go heartbeatLine**()** //心跳维护线程

go ackLine**()** //ACK线程

go hungupLine**()** //挂单写入数据库

go autoHbLine**()** //自动回报线程

go speedRefresh**()**

## 3.8 Channel

var **(**

gSvr **\***gotcp**.**Server

gSvrStop bool **=** **true**

gPreCheckQueue chan **\***tMyPack

gConn **\***cltConn //现只考虑：只支持一个svr且一个端口，只考虑一个客户端连接

gACKQueue chan **\***tMyPack

gBizCfg **\***bizCfg

gHungupQueue chan **\***tMyPack //挂单写入数据库之前的缓存队列

gHbQueue chan **\***tMyPack //自动回报队列

gWtIdUsed map**[**string**]**bool

gAckCurrNum int64

gHBCurrNum int64

gCtlMsgNum int64

**)**

func BizSysInit**()** error **{**

gBizCfg **=** NewBizCfg**()**

initmonitor**()**

//资源初始化

gPreCheckQueue **=** make**(**chan **\***tMyPack**,** 1e7**)** //预检队列

gACKQueue **=** make**(**chan **\***tMyPack**,** 1e7**)** //ACK队列

gHungupQueue **=** make**(**chan **\***tMyPack**,** 1000**)**

gHbQueue **=** make**(**chan **\***tMyPack**,** 1e7**)** //自动回报队列

gWtIdUsed **=** make**(**map**[**string**]**bool**)**

**return** nil

**}**

gHungupQueue **<-** mypack //入队列

gACKQueue **<-** mypack //入队列

gHbQueue **<-** mypack //入队列

mypack **:=** **<-**gHungupQueue //出队列

mypack **:=** **<-**gACKQueue //出队列

mypack **:=** **<-**gHbQueue //出队列

声明、定义channel，往channel中加入数据（入队列），从channel中取出数据（出队列）。

## 3.9 读取配置

* Config方法

**import** "github.com/astaxie/beego/config"

ca **:=** cnf**.**String**(**"bizack::cause"**)** //取配置

c**.**Svcing**,** err **=** cnf**.**Bool**(**"bizack::svcing"**)** //取配置

c**.**AckProb**,** err **=** cnf**.**Int**(**"bizack::ackprob"**)** //取配置

err **=** cnf**.**Set**(**"bizack::svcing"**,** "1"**)**

err **=** cnf**.**Set**(**"bizack::cause"**,** c**.**Cause**)**

err **=** cnf**.**SaveConfigFile**(**"./conf/app.conf"**)**

* Beego

"github.com/astaxie/beego" //包

beego**.**AppConfig**.**String**(**"bizack::svcing"**)** //取字段

beego**.**AppConfig**.**Set**(**"bizack::ackmodel"**,** "0"**)** //设置

beego**.**AppConfig**.**Set**(**"bizack::chktypesupport"**,** "0"**)** //设置

beego**.**AppConfig**.**SaveConfigFile**(**"conf/app.conf"**)** //保存

因为config和beego都是在缓冲区里面进行保存的，当两者在一起使用时，有可能会出现错误，所以尽量避免两者一起使用。