

前言

作者王慧敏，资深码农一枚，没什么名气，也没什么学历，教程中肯定会有一些错误或疏忽，大家看到请联系我，以便极时更正。Email:110male@sohu.com。

我说了，我不是什么牛人，所以写的东西追求简单，易懂，实用。写本教程的目的主要是为了跟大一起学习交流。本教程从 Go 的安装，到基本语法，再到项目中实用的一些知识点，如 XML、Json 解析，实现自己的 ORM 等。循序渐进，注重实战。如讲了“数据库操作”，“反射”之后紧接着就是如何实现一个 ORM，实用性强，注重实际应用。再比如网络编程一节，会讲设计一个 TCP 和 UDP 程序的不同，TCP 程序需自定义通信格式，UDP 需要双方约定 UDP 包的大小。本教程的第最后一章向大家介绍一个 GoMvc 框架。

官方网址：<http://www.668.cm>

目录

前言.....	1
第一章 GO 语言的安装.....	4
1.1 go 语言简介.....	4
1.2 安装 go.....	4
1.3 安装 go 语言开发工具.....	7
1.4 安装 gocode.....	15
第二章 GO 语言基础.....	18
2.1 第一个 Go 程序.....	18
2.2 基本类型.....	18
2.2 定义变量.....	19
2.3 array,slice,map.....	20
2.3.1 数组.....	20
2.3.2 切片 Slice.....	21
2.3.3 Map.....	22
2.3.4 range 遍历数组.....	23
2.4 常量.....	23
2.5 控制流.....	25
2.5.1 if else.....	25
2.5.2 Switch.....	26
2.5.3 for 循环.....	27
第三章函数.....	29
3.1 函数定义.....	29
3.2 多值返回.....	29
3.3 变参函数.....	30
3.4 defer.....	31
3.5 函数类型.....	32
3.6 错误处理.....	32
第四章 面向对象编程.....	34
4.1 struct.....	34
4.2 继承.....	35
4.3 Interface.....	39
第五章多线程.....	41
5.1 多线程.....	41
5.2 chan.....	43
5.3 进程同步.....	46
第六章 日期与定时器.....	49
6.1 日期的获取与计算.....	49
6.2 定时器.....	50
第 7 章文件操作.....	52
7.1 路径.....	52
7.2 文件读写.....	55
7.3 遍历目录下的文件.....	58

7.4 gob 序列化.....	59
第 8 章 JSON 与 XML 解析.....	61
8.1XML 序列化与解析.....	61
8.2 JSON 序列化与反序列化.....	65
第 9 章 MySQL 数据库操作.....	68
9.1 安装 MySQL 驱动.....	68
9.2 MySQL 数据库操作.....	68
9.3 事务.....	71
9.4 标准驱动的不足与改进.....	73
第 10 章 反射.....	77
10.1 反射基础.....	77
10.2 反射调用函数.....	81
10.3 反射取 Struct 的 Tag 信息.....	82
第 11 章 实现一个自己的 ORM.....	84
11.1 实现自己的 ORM.....	84
11.2 Insert 函数的实现及所有源码.....	88
第 12 章 TCP 与 UDP 网络编程.....	90
12.1 TCP 编程.....	90
12.2 TCP 编程实战.....	92
12.3 UDP 网络编程.....	97
12.4 UDP 编程实战.....	98
第十三章 WEB 编程.....	101
13.1 第一个 WEB 程序.....	101
13.2 URL 参数与 Form 表单处理.....	102
12.3 文件上传.....	104
12.4 HTML 模板处理.....	106
模板展示.....	106
第十四章 GoMvc Web 框架.....	113
14.1 MVC 简介.....	113
4.2 GoMvc 简介.....	119
14.3 GoMvc 目录结构.....	120
14.4 配置文件.....	120
14.5 路由.....	122
14.6 Controller.....	122
14.7 Action.....	123

第一章 GO 语言的安装

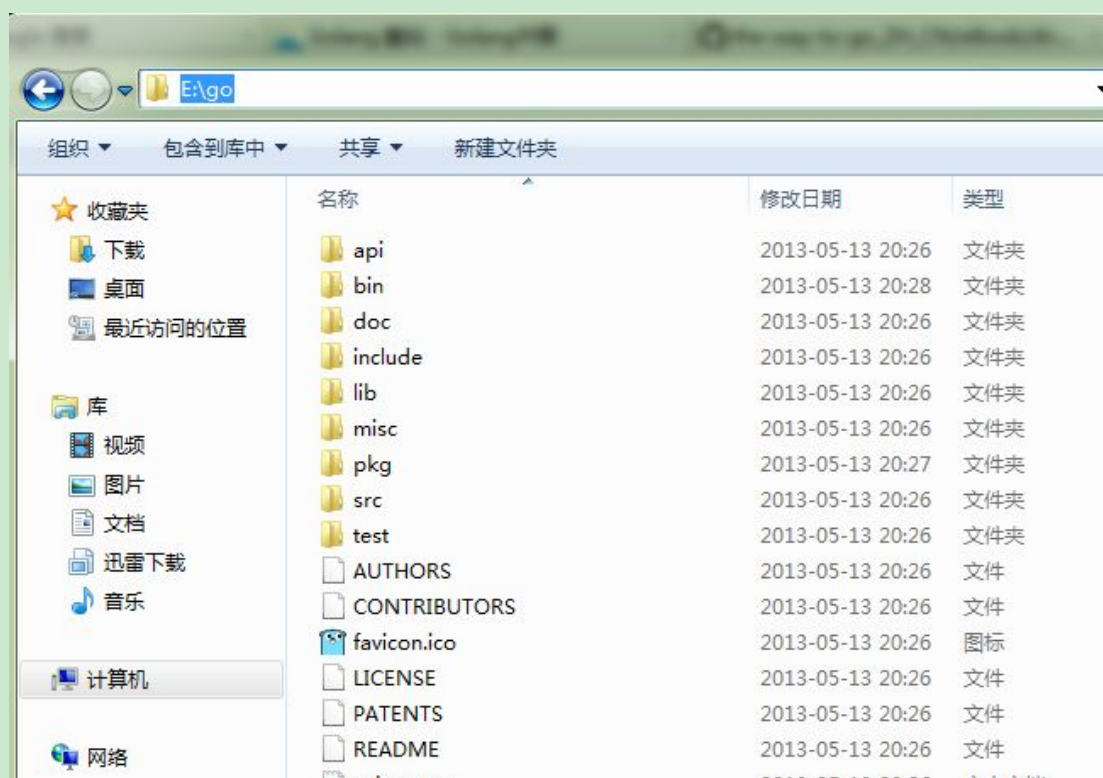
1.1 go 语言简介

Go 语言是由 Google 开发的一个开源项目,目的之一为了提高开发人员的编程效率。Go 语言语法灵活、简洁、清晰、高效。它对并发特性可以方便地用于多核处理器 和网络开发,同时灵活新颖的类型系统可以方便地编写模块化的系统。go 可以快速编译, 同时具有垃圾内存自动回收功能,并且还支持运行时反射。Go 是一个高效、静态类型, 但是又具有解释语言的动态类型特征的系统级语法。

Golang 官网 <http://golang.org/>, 在国内可能打不开, 可以使用 goagent 代理。对于英文比较好的, 可以看官网的文档。国内也有对应的翻译文档, 但不全。

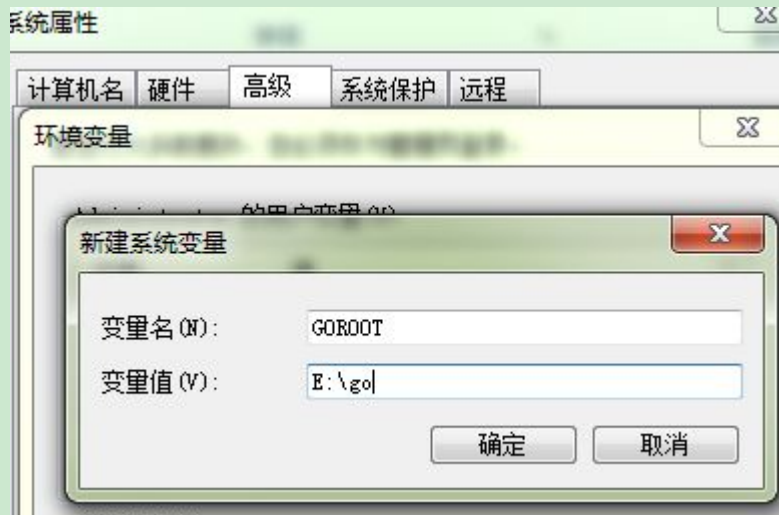
1.2 安装 go

以 windows 为例, 首先从 go 语言官网 <http://code.google.com/p/go/downloads/list> 下载最新的 go 安装包。将其解压到本地硬盘。

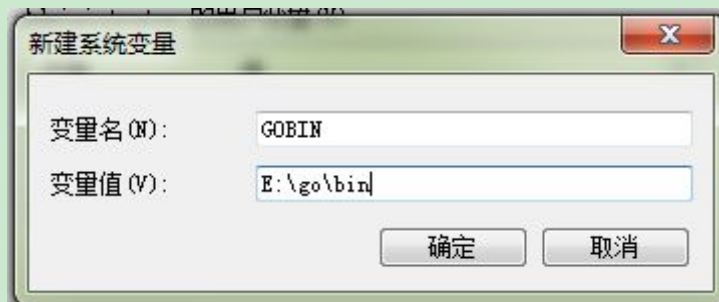


Go 的安装只要设置相应环境变量就可以了, 相关环境变量如下:

GOROOT: 表示 Go 在你的电脑上的安装位置, 如上图, 我是在 E:\go 目录下的。



GOBIN: 表示编译器和链接器的安装位置，默认是 \$GOROOT/bin，如果你使用的是 Go 1.0.3 及以后的版本，一般情况下你可以将它的值设置为空，Go 将会使用前面提到的默认值。



GOARCH: 表示目标机器的处理器架构，它的值可以是 386, amd64 或 arm。我的机器是 windows 64 位，所以设置为 amd64。如果你的机器是 32 位的设置为 386。



GOOS: 表示目标机器的操作系统，它的值可以是 darwin, freebsd, linux 或 windows。我用的是 windows，所以设置为 windows。



GOMAXPROCS: 用于设置应用程序可使用的处理器个数与核数。

尽管在程序中使用了多线程，默认情况下，如果没有设定 GOMAXPROCS 环境变量，程序只使用一个线程。为了利用全部 CPU 内核，则必须制定它的值。
我的 CPU 是 i7,四核，所以该变量本人设置为 4。

GOPATH: 用来设置 Go 项目源代码和二进制文件的目录。GOPATH 允许多个目录，当有多个目录时，请注意分隔符，多个 GOPATH 的时候 Windows 是分号，Linux 系统是冒号，当有多个 GOPATH 时，默认会将 go get 的内容放在第一个目录下。

以上 \$GOPATH 目录约定有三个子目录：

- src 存放源代码（比如：.go .c .h .s 等）
- pkg 编译后生成的文件（比如：.a）
- bin 编译后生成的可执行文件（为了方便，可以把此目录加入到 \$PATH 变量中）

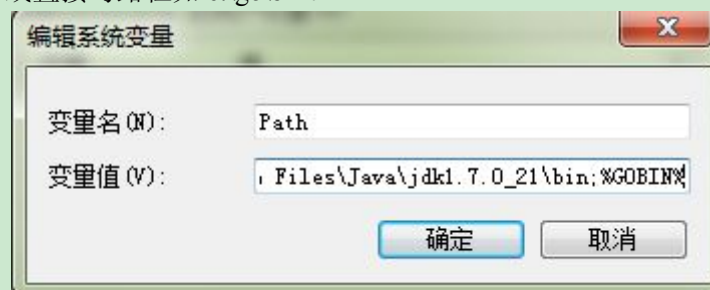
在程序开发时经常会出某包找不到的情况，将指定的目录，添加到该变量中即可解决。
这里我把 k:\go 做我工作目录。如下图所示：



GOPATH 在 1.1 前可以跟 GOROOT 一样，1.1 以后不能设置跟 GOROOT 一样，否则会有如下的提示。

```
warning: GOPATH set to GOROOT (E:\go) has no effect
OT. For more details see: go help gopath
```

最后设置 PATH 变量，这样就可以在任何位置运行 Go 命令。如果设置了 GOBIN，可以填 %GOBIN%，或直接写路径如 e:\go\bin。



开始->运行->输入 cmd, 打开命令行窗口，输入 go 回车，出现下图信息，说明安装成功。



```
C:\Windows\system32\cmd.exe
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>go
Go is a tool for managing Go source code.

Usage:

    go command [arguments]

The commands are:

    build      compile packages and dependencies
    clean      remove object files
    doc        run godoc on package sources
    env        print Go environment information
    fix        run go tool fix on packages
    fmt        run gofmt on package sources
    get        download and install packages and dependencies
    install    compile and install packages and dependencies
    list       list packages
    run        compile and run Go program
    test       test packages
    tool       run specified go tool
    version    print Go version
    vet        run go tool vet on packages
```

至于其它系统的安装，大同小异，如在安装过程中遇到问题可以 google 一下。

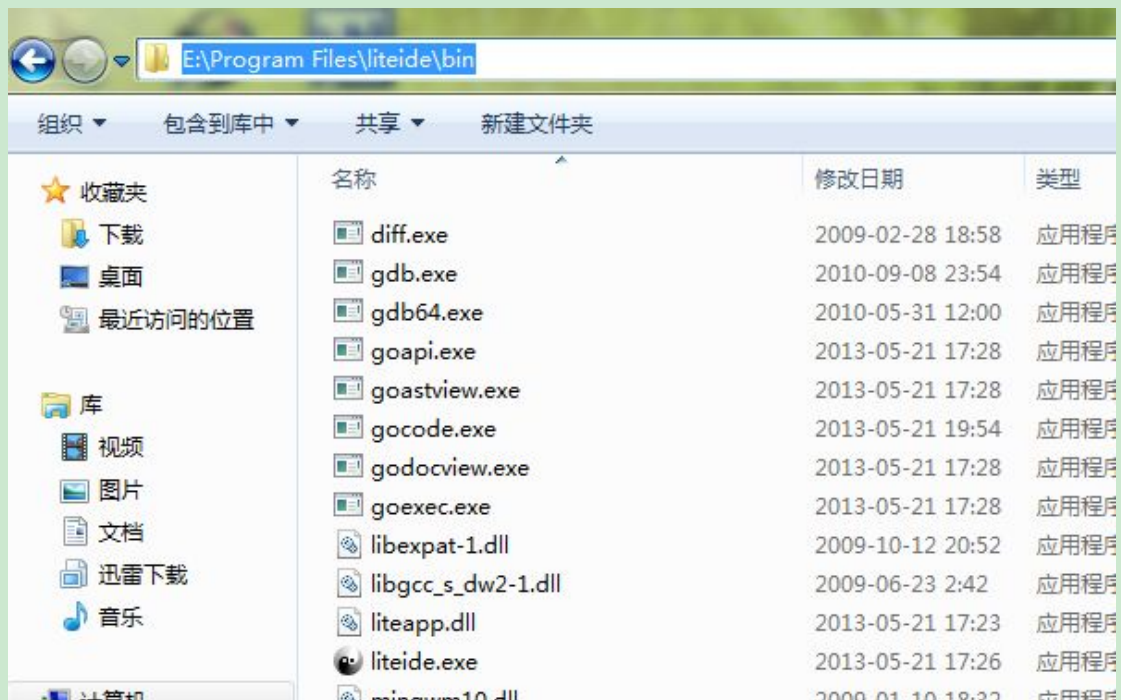
1.3 安装 go 语言开发工具

Go 语言的开发工具有很多，这里只介绍 LiteIDE, IntelliJ IDEA 。

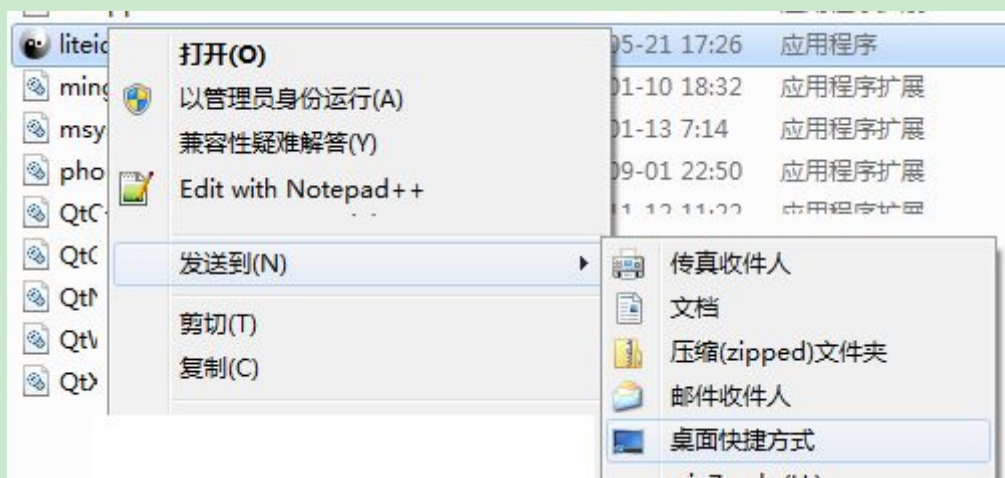
1) LiteIDE

LiteIDE 是一款专为 Go 语言开发而设计的跨平台轻量级集成开发环境 (IDE)，基于 Qt 开发，支持 Windows、Linux 和 Mac OS X 平台。LiteIDE 的第一个版本发布于 2011 年 1 月初，是最早的面向 Go 语言的 IDE 之一。

<https://code.google.com/p/golangide/> 下载最新版本。解压到本地硬盘。

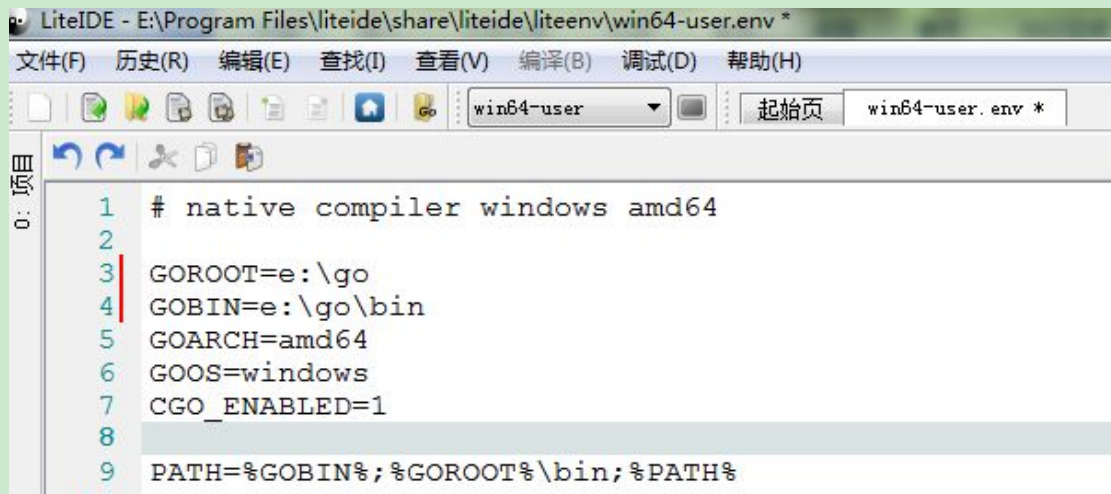


我是放在了 E:\Program Files\liteide 目录下。右击 liteide.exe，在弹出的右键菜单中选择，发送到->桌面快捷方式。在桌面上他建快捷方式。



双击 liteide 的桌面快捷方式，打开 liteide,在如下图所示的位置选择 windows64-user,点旁边的黑色按钮，编 lite 的环境配置。

```
# native compiler windows amd64
GOROOT=e:\go
GOBIN=e:\go\bin
GOARCH=amd64
GOOS=windows
CGO_ENABLED=1
PATH=%GOBIN%;%GOROOT%\bin;%PATH%
```

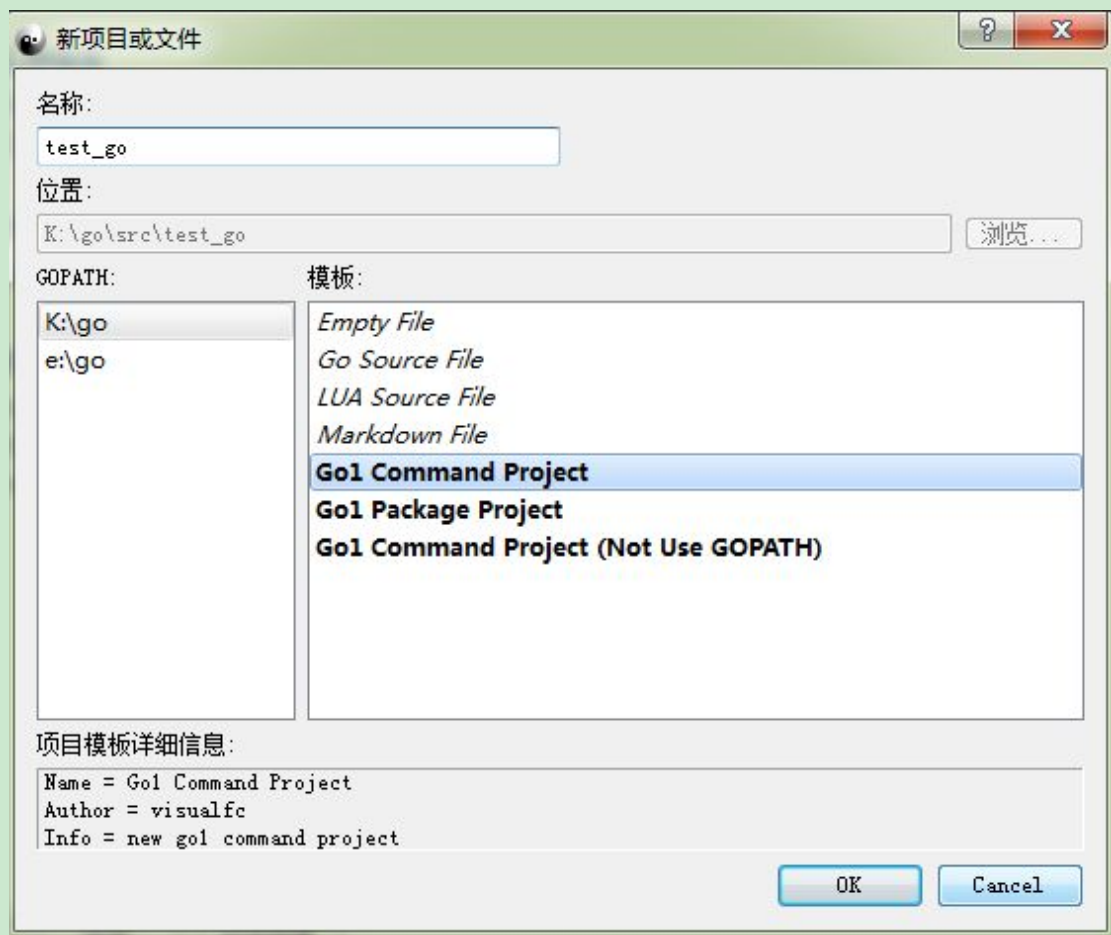



The screenshot shows the LiteIDE application window with the file 'win64-user.env' open. The menu bar includes '文件(F)', '历史(R)', '编辑(E)', '查找(I)', '查看(V)', '编译(B)', '调试(D)', and '帮助(H)'. The toolbar contains icons for file operations and a 'Go' button. The editor displays the following content:

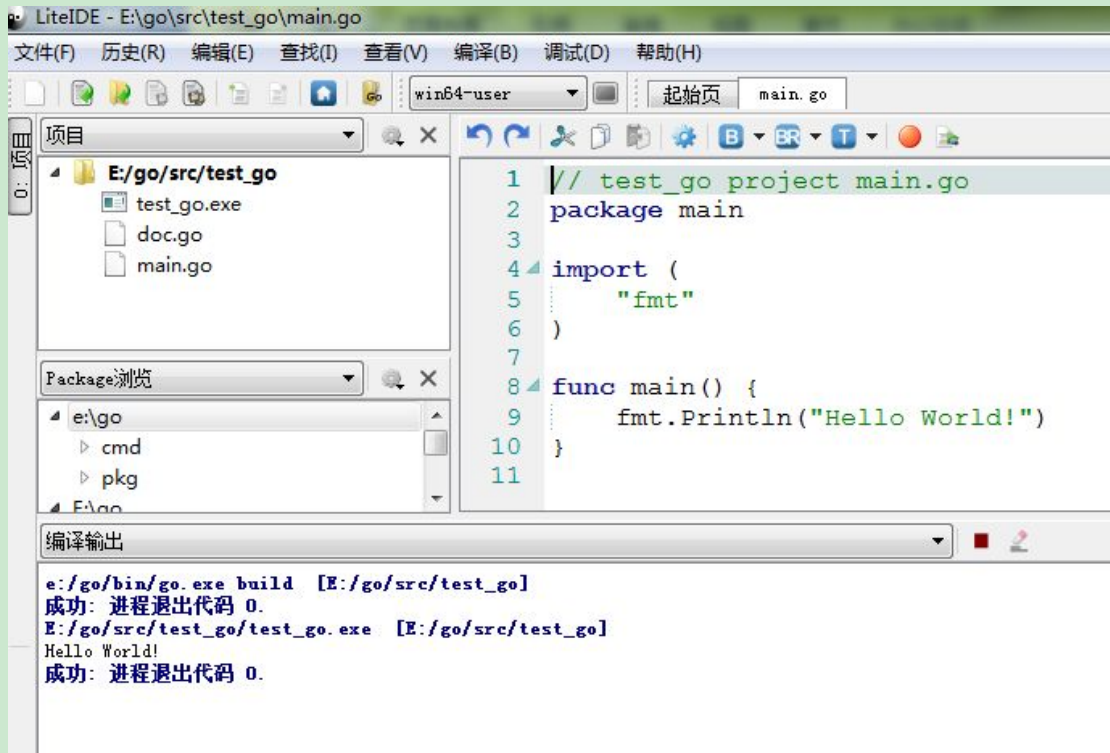
```
1 # native compiler windows amd64
2
3 GOROOT=e:\go
4 GOBIN=e:\go\bin
5 GOARCH=amd64
6 GOOS=windows
7 CGO_ENABLED=1
8
9 PATH=%GOBIN%;%GOROOT%\bin;%PATH%
```

别忘了保存编辑后的文件。

文件->新建，打开新建对话框，名称输入 test_go。点击 OK 创建一个命令行项目。



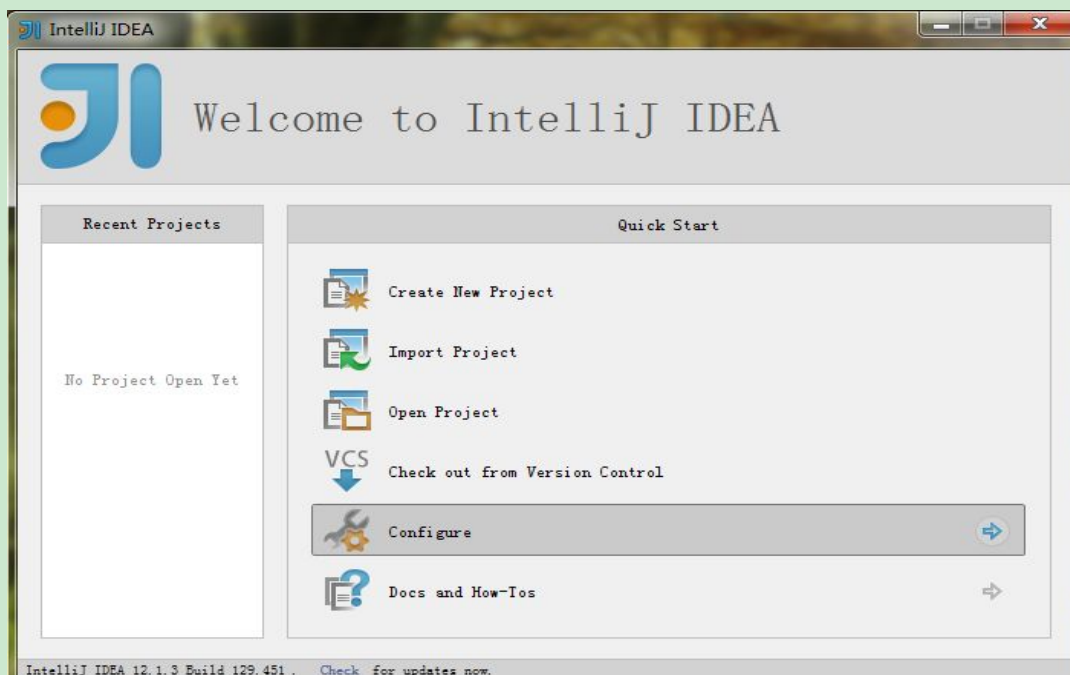
在菜单中选择 Build->BuildAndRun,编译并运行程序。



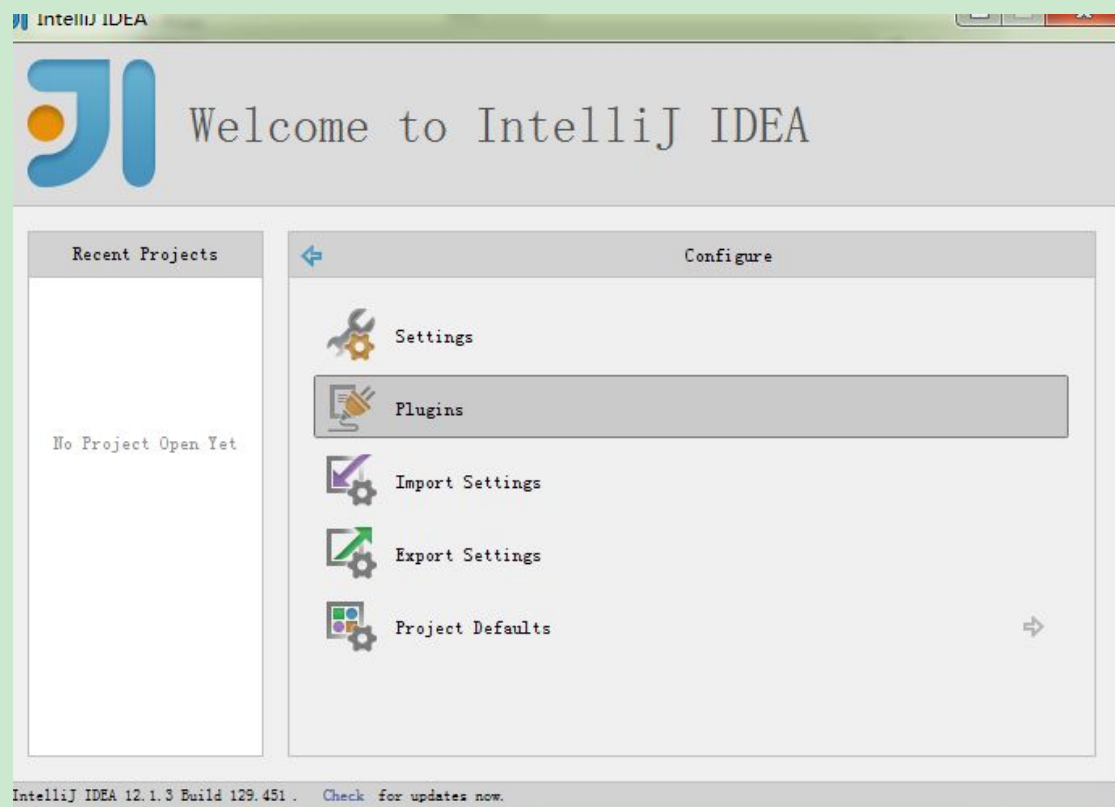
2) IntelliJ IDEA

IntelliJ IDEA 被认为是当前 Java 开发效率最快的 IDE 工具。它整合了开发过程中实用的众多功能，几乎可以不用鼠标可以方便的完成你要做的任何事情，最大程度的加快开发的速度。简单而又功能强大。与其他的一些繁冗而复杂的 IDE 工具有鲜明的对比。IntelliJ IDEA 下有 golang 插件，支持 go 的开发。

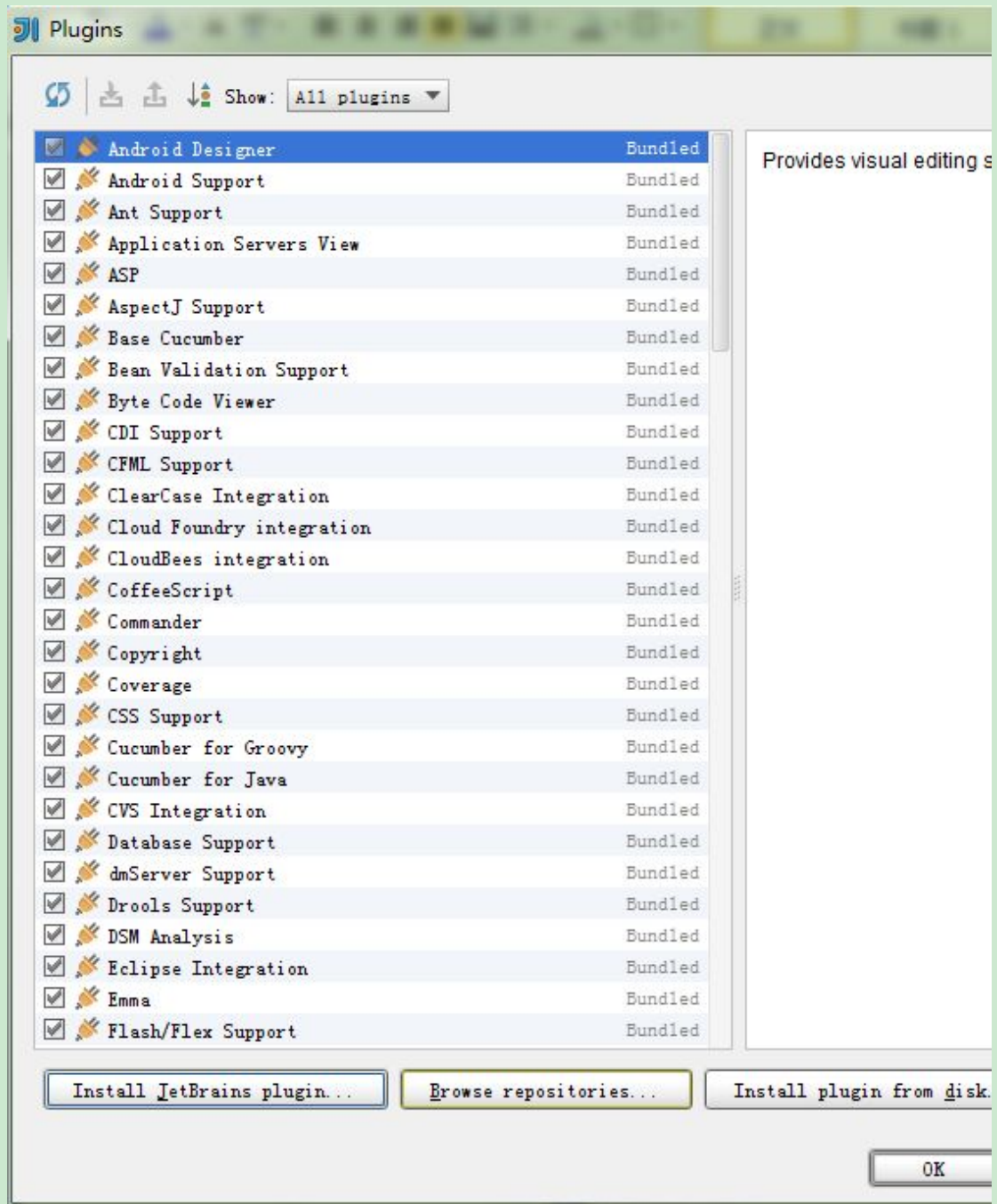
打开 IntelliJ IDEA，點選 Configure,如下图所示：



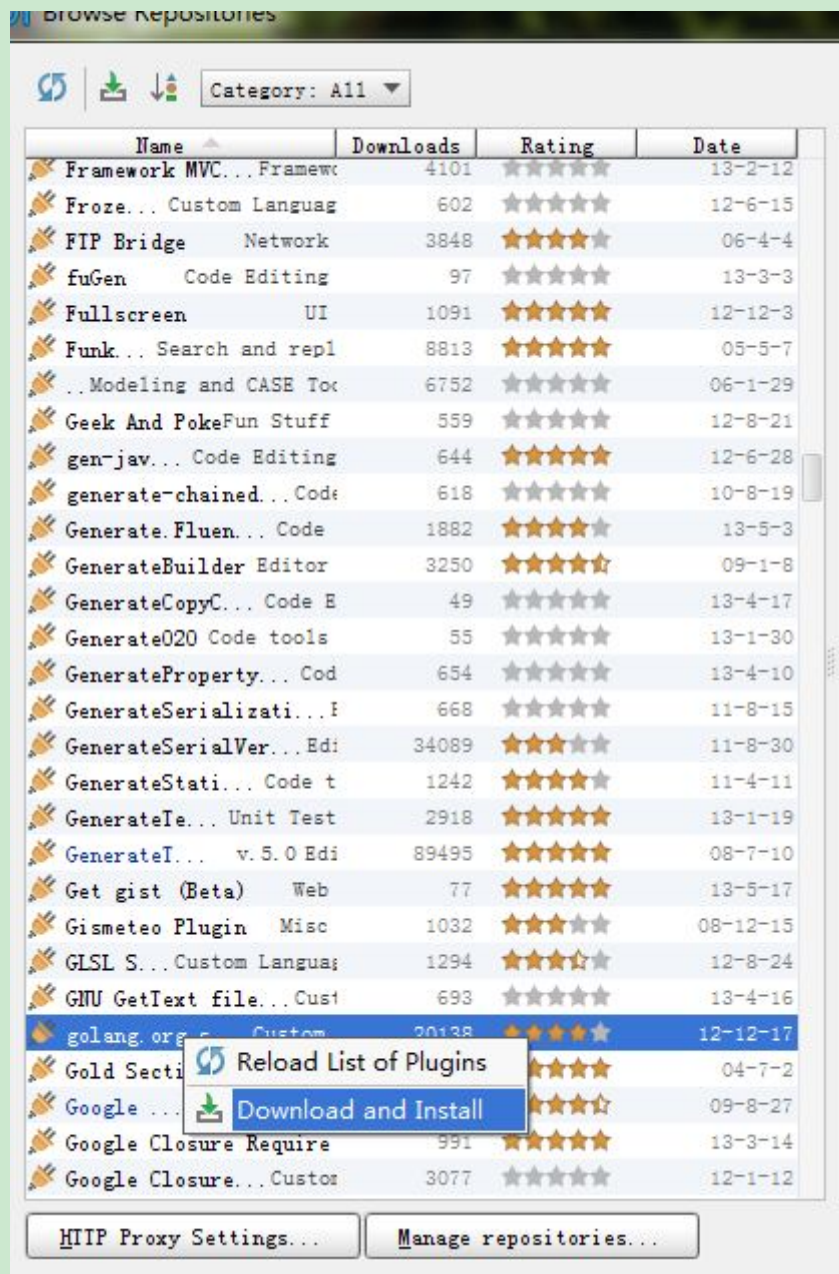
再选 Plugins，如下图所示：



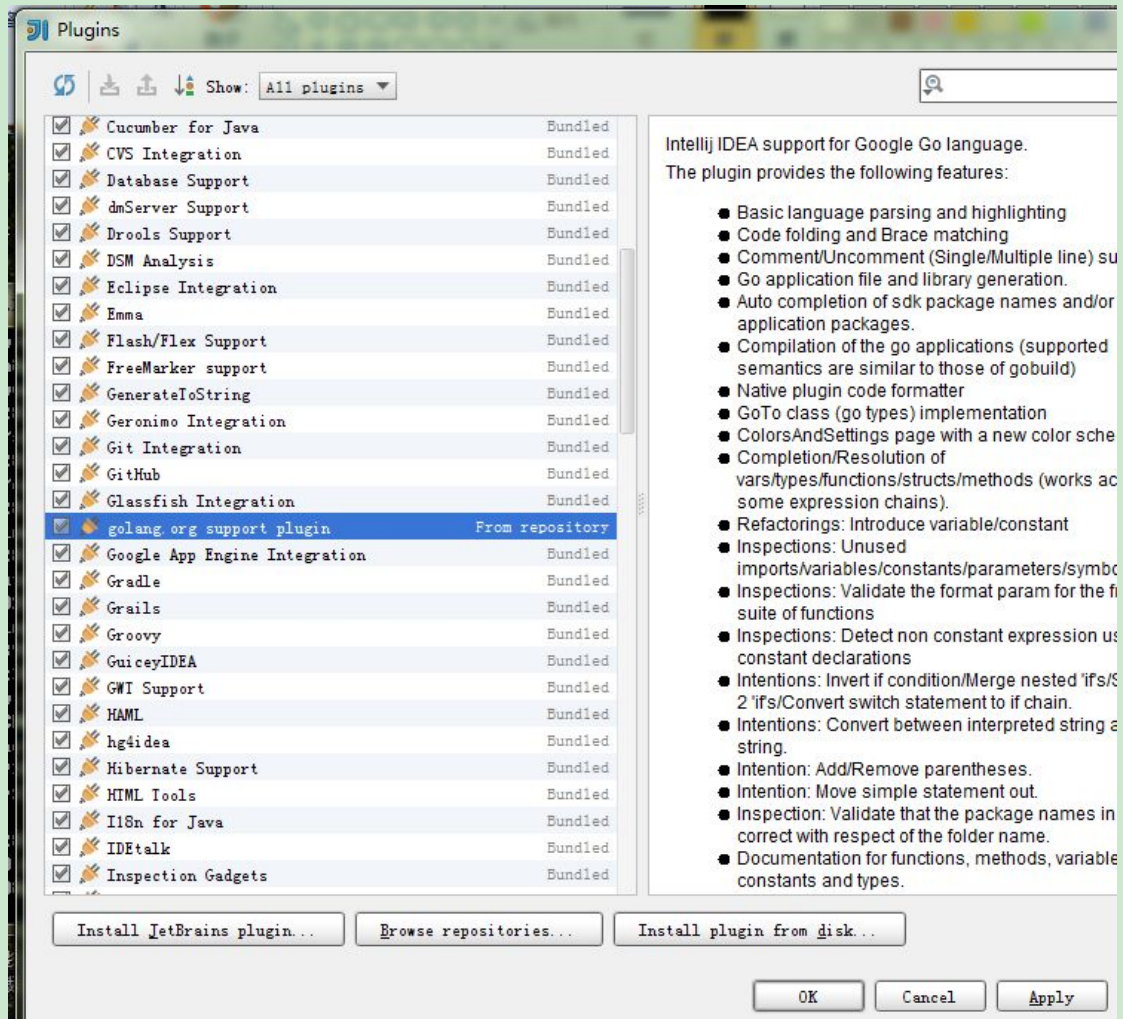
在 Plugins 对话框中，点 Browse repositories...，如下图所示



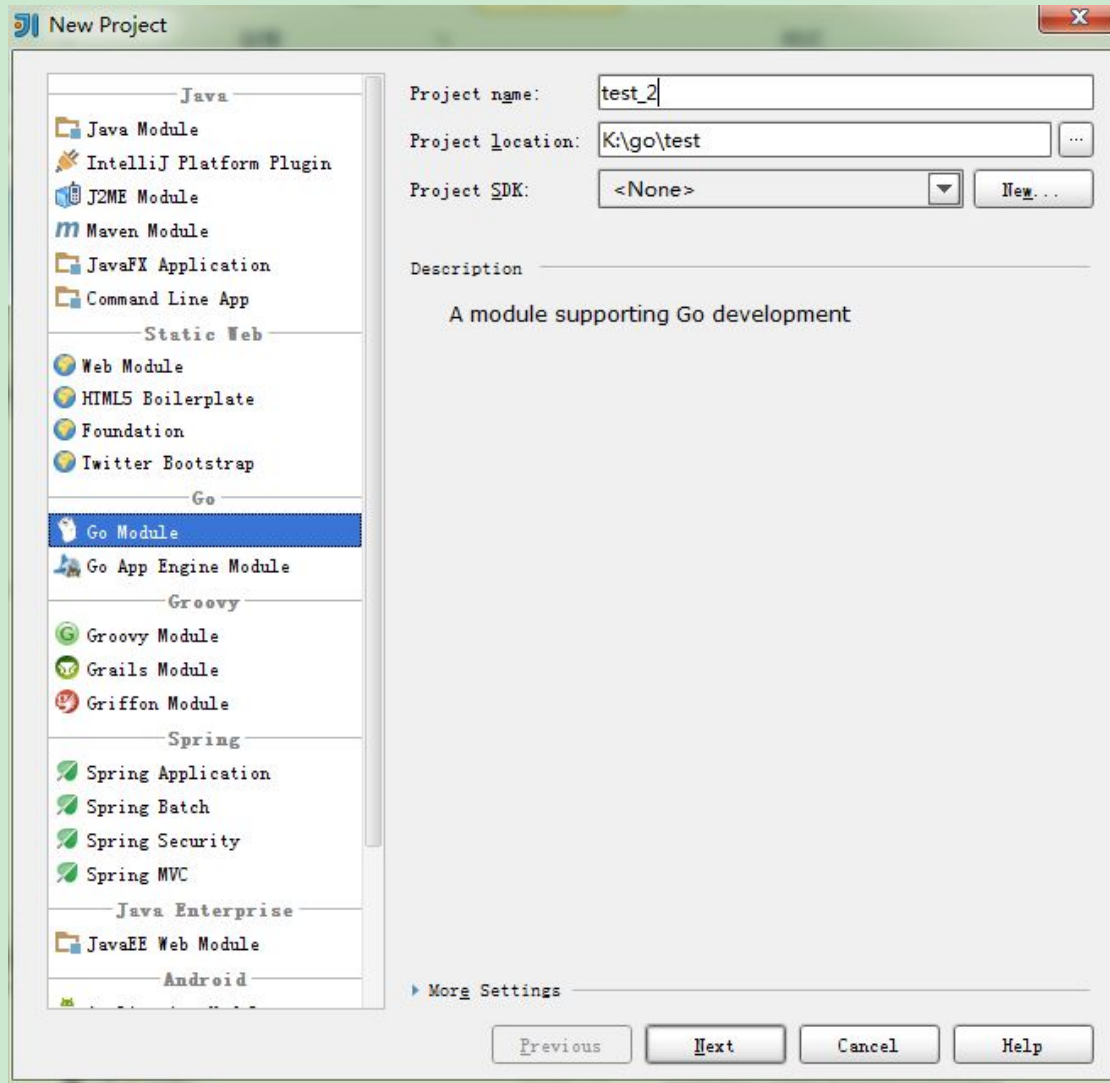
在新窗口中找到 golang.org 的插件，右击->选择”Download and Install”。



关闭当前窗口，返回 Plugins 对话框，找到 golang 插件，点击选择，再点右下角的 Apply 按钮重启 IntelliJ IDEA，如下图所示：



重启后，点 Create New Project，打开新建项目的对话，要左则选择 Go Module,在右边输入项目名称，选择项目位置。



点 Project SDK 右边的”New...”按钮，选择 golang 的安装位置，我的是装在 E:\go 目录下的。点下一步完成项目创建。

1.4 安装 gocode

Gocode 是的 Go 语言的一个代码自动补全的工具，对于 windows 程序员应该很熟悉 VS 的代码提示功能，很强大。在不安装 gocode 的情况下，LiteIDE 是没有代码提示的，所以最好能装一下 gocode。经测试 IntelliJ IDEA 不需要安装 gocode 就可以代码提示，而且 IntelliJ IDEA 的代码提示功能要比 LiteIDE 强大。

Gocode 是 github.com 上的一个开源项目，地址是：<https://github.com/nsf/gocode>。开始->运行->输入 cmd 打开命令行窗口。输入如下命令来安装 gocode。

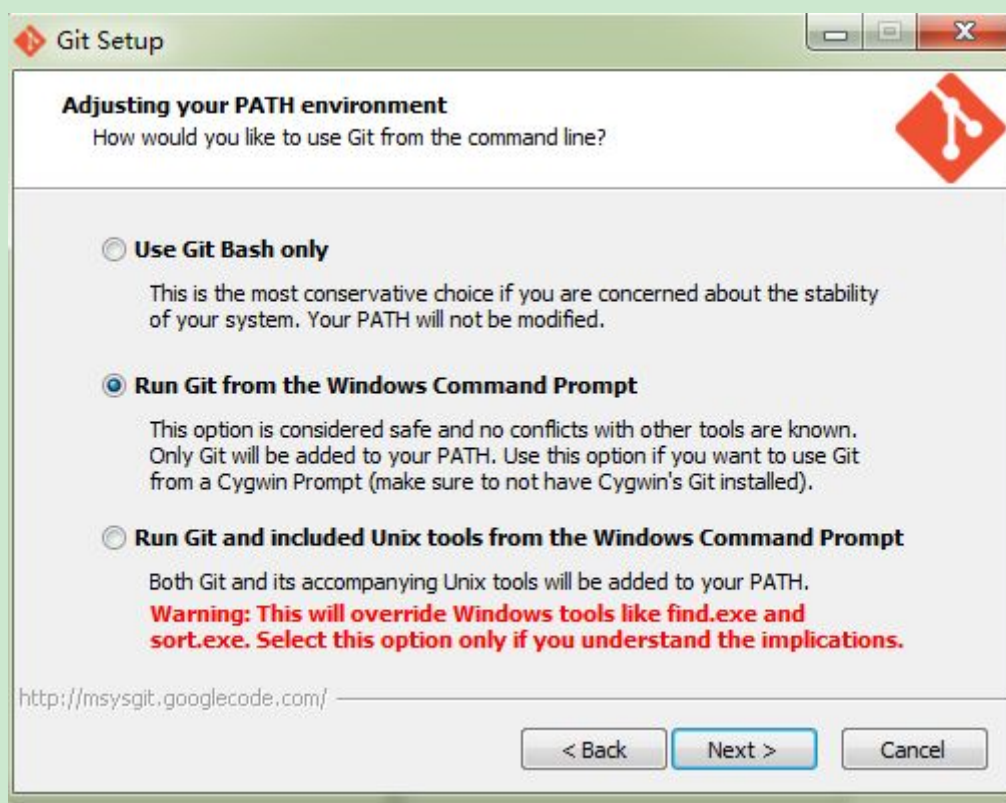
```
go get github.com/nsf/gocode
go install github.com/nsf/gocode
```

如果提示下面的信息，说明没有安装 git。在 <http://git-scm.com/download> 下载最新版本，

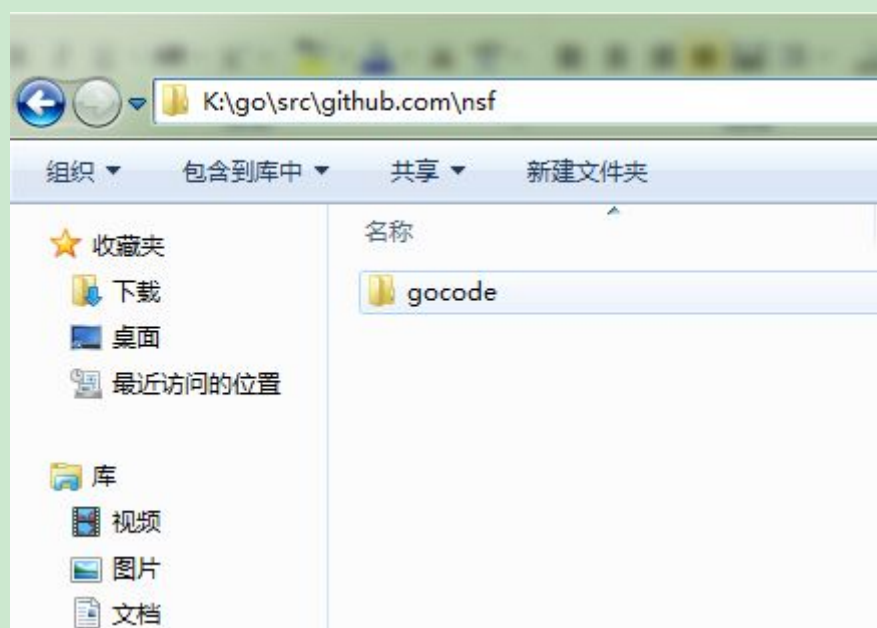
并安装。

```
go: missing Git command. See http://golang.org/s/gogetcmd
package github.com/nsf/gocode: exec: "git": executable file not found in %PATH%
```

如下图所示，在安装过程中，选择 Run Git from the Windows Command Prompt，这样就不需要手动在 Path 里面设置 git 的路径，可以在命令行运行 git 命令。



安装成功后，关闭原来打开的命令窗口，再重新打开一个命令窗口，重新运行上面说过的安装 gocode 的命令。命令执行成功后，gocode 将安装在 GOPATH 的位置，我设置的是 K:\go，gocode 将会安装在这个目录。如下图所示：



第二章 GO 语言基础

2.1 第一个 Go 程序

```
package main
import (
    "fmt"
)
func main() {
    fmt.Println("Hello World!")
}
```

第一行，package 定义了程序包 main。第二行 import 引入了 fmt 包。func main 定义了 main 函数，func 是函数定义的关键字。在 main 函数调用 fmt 包的 Println 函数，输出了”Hello World!”字符串。

在所有初始化完成后，程序从 main 包中的 main 函数开始执行。

Go 中的所有字符串，都是 UTF-8 编码。所有的 Go 语言源文件也都是采用 UTF-8 编码。在 Go 语言中，语句末尾的分号可以省略不写。

2.2 基本类型

Go 对整数进行了更明确的规划，清晰明了。Go 里的基本类型如下表：

类型	长度（字节）	说明
bool	1	true, false。不能把非零值当作 true
byte	1	uint8 别名
rune	4	int32 别名。代表一个 Unicode 字符。
int/uint	4	依据所运行的平台，可能是 32bit 或 64bit。
int8/uint8	1	1 -128 ~ 127; 0 ~ 255
int16/uint16	2	-32768 ~ 32767; 0 ~ 65535
int32/uint32	4	-21 亿 ~ 21 亿, 0 ~ 42 亿
complex64	8	复数类型，即 32 位实数+32 位虚数
complex128	16	复数类型，即 64 位实数+64 位虚数
uintptr		能够保存指针的 32 位或 64 位整数
array		数组，值类型，如：[2] int
struct		结构体，值类型
string		值类型
slice		引用类型 如： []int
map		引用类型
channel		引用类型

Interface		接口类型
function		函数类型

2.2 定义变量

Go 语言里面定义变量有多种方式。

使用 var 关键字是 Go 最基本的定义变量方式，最常见的语法如下：

```
var n int      /*定义变量 n*/
var i int=3    /*定义变量 i 并赋值 3*/
var(//多变量的定义
    aa  int=3
    str string="abcd")
var i1,i2,i3 int=1,2,3/*定义多个变量并赋值*/
var strName="张三";/*Go 会自动检测变量的类型*/
strSex="男";/*:=定义变量，并给变量赋值，可以省略 var 关键字*/
```

下面是完整的代码

```
package main
import    "fmt"
func main() {
    var b bool
    var n int      /*定义变量 n*/
    var i int = 3 /*定义变量 i 并赋值3*/
    var (          //多变量的定义
        aa  int = 3
        str string
    )
    var i1, i2, i3 int = 1, 2, 3 /*定义多个变量并赋值*/
    var strName = "张三"          /*Go 会自动检测变量的类型*/
    strSex := "男"                /*:=定义变量，并给变量赋值，可以省略 var 关键字*/
    /*
    fmt.Println("n=", n)
    fmt.Println("b", b)
    fmt.Println("i=", i)
    fmt.Println("aa=", aa)
    fmt.Println("str=", str)
    fmt.Println("i1=", i1, ",i2=", i2, ",i3=", i3)
    fmt.Println("strName=", strName)
    fmt.Println("strSex=", strSex)
    */
}
```

编译并运行程序，你会发现 b=false,n=0；对于未赋值的变量，Go 会自动初始化，数值类型初始化为0，布尔类型初始值为 false，字符串初始值为空。

2.3 array,slice,map

2.3.1 数组

array 是固定长度的数组，这个和 C 语言中的数组是一样的，使用前必须确定数组长度。但是和 C 中的数组相比，又是有一些不同的：

1 Go 中的数组是值类型，换句话说，如果你将一个数组赋值给另外一个数组，那么，实际上就是将整个数组拷贝一份

2 如果 Go 中的数组作为函数的参数，那么实际传递的参数是一份数组的拷贝，而不是数组的指针。这个和 C 要区分开。因此，在 Go 中如果将数组作为函数的参数传递的话，那效率就肯定没有传递指针高了。

3 array 的长度也是 Type 的一部分，这样就说明[10]int 和[20]int 是两个不同的类型。

array 的结构用图示表示是这样的：

len	int	int
-----	-----	-----

len 表示数组的长度，后面的 int 储存的是实际数据

数组的定义如下：

```
//声明一个2个元素的数组，名字为 arr_1,因为是 int 型数组，所以初值为0，即[0,0]
```

```
var arr_1 [2]int
```

```
/*声明一个2个元素的数组，名字为 arr_2，并同时赋初值，{}里为空，说明没有赋初值，  
等同于上面*/
```

```
arr_2:= [2]int{}
```

```
//声明一个2个元素的数组，名字为 arr3, arr3_1, arr3_2，并同时赋初值，结果均为[1,2]
```

```
arr3:= [2]int{1,2}
```

```
//{}里的冒号左边是下标，右边是值
```

```
arr3_1 :=[2]int{0:1,1:2}
```

```
arr3_2 := [2]int{1:2,0:1}
```

```
/*不指定数组长度，自动计算长度, [...],声明一个2个（自动计算而来）元素的数组，名  
字为 arr4，并同时赋初值，结果为[1,2]*/
```

```
arr4:= [...int{1,2}
```

```
/*声明一个4个（自动计算而来）元素的数组，名字为 arr5，并同时赋初值，结果为  
[0,0,0,9]*/
```

```
arr5:= [...int{3:9}
```

完整代码如下：

```

package main
import (
    "fmt"
)
func main() {
    //声明一个2个元素的数组，名字为 arr_1,因为是 int 型数组，所以初值为0，即[0,0]
    var arr_1 [2]int
    /*声明一个2个元素的数组，名字为 arr_2，并同时赋初值，{}里为空，说明没有赋初值，等同于上面*/
    arr_2 := [2]int{}
    //声明一个2个元素的数组，名字为 arr3, arr3_1, arr3_2，并同时赋初值，结果均为[1,2]
    arr3 := [2]int{1, 2}
    //{}里的冒号左边是下标，右边是值
    arr3_1 := [2]int{0: 1, 1: 2}
    arr3_2 := [2]int{1: 2, 0: 1}
    *不指定数组长度，自动计算长度,[...],声明一个2个（自动计算而来）元素的数组，名字为 arr4，并同时赋初值，结果为[1,2]*/
    arr4 := [...]int{1, 2}
    /*声明一个4个（自动计算而来）元素的数组，名字为 shuzu5，并同时赋初值，结果为[0,0,0,9]*/
    arr5 := [...]int{3: 9}
    fmt.Println(arr_1)
    fmt.Println(arr_2)
    fmt.Println(arr3)
    fmt.Println(arr3_1)
    fmt.Println(arr3_2)
    fmt.Println(arr4)
    fmt.Println(arr5)
}

```

2.3.2 切片 Slice

Slice 是引用类型，有点像指向数组的指针。Slices 和数组的声明语法类似，但是不像数组那样要指定元素的个数；它在内部引用特定的空间，或者其它数组的空间。在 Go 语言中 Slices 比数组使用的更为普遍，因为它更有弹性，引用的语法也使得它效率很高。

[] T 是一个 T 类型的片，切片不需要指定长度，指定长度就成了数组。

切片可以被重新分片。创建一个指向同一数组的指针。

s[lo:hi] 代表的是原片的 lo 到 hi-1 位置的元素。

s[lo:lo] 是空的。

s[lo:lo+1] 只有一个元素 lo。

示例代码如下：

```

package main
import "fmt"
import "reflect"
func main() {
    p := [...]int{2, 3, 5, 7, 11, 13} //定义一个数组
    s1 := p[1:3]                      //定义切片，包含3,5两个元素
    fmt.Println(s1)
    fmt.Println(reflect.TypeOf(p)) //用反射得到变量的类型 p 是数组类型[6]int
    fmt.Println(reflect.TypeOf(s1)) //s1是切片类型[]int
    ChangeArrayValue(p) //ChangeArrayValue 函数将第一个值改为100
    fmt.Println(p)      //数组 p 的值并没有改变，因为数组是值类型
    ChangeSliceValue(s1) //ChangeSliceValue 将切片的第一个值改为100
    fmt.Println(s1)     //切片 s1 的值被改变，因为切片是引用类型
    fmt.Println(p)      //切片是引用的数组 p 第一个和第二个元素，所以数组 p 的值被改变
}
func ChangeArrayValue(arr [6]int) {
    arr[0] = 100
}
func ChangeSliceValue(slice []int) {
    slice[0] = 100
}

```

2.3.3 Map

map 是一个 key-value 的 hash 结构，类似其它语言中的 Hashtable、Dictionary 。map 的 key 必须是支持比较运算符 (==、!=) 的类型。如 number、string、pointer、array、struct、interface (接口实现类型必须支持比较运算符)，不能是 function、map、slice。

Map 用 make 来分配内存空间，mak (map[TK]TV)，TK 是 key 的类型，TV 是值的类型。

```

package main
import (
    "fmt"
)
func main() {
    mp := make(map[string]string) //key 是字符串类型，值也是字符串类型
    mp["a"] = "1"
    mp["b"] = "2"
    mp["pi"] = "3.1415926"
    mp["sh"] = "上海"
    v, ok := mp["sh"] //sh 存在，v 存放的是 value 值，ok 值为 true
}

```



```

if ok {
    fmt.Println(v)
} else {
    fmt.Println("key 'sh' 不存在")
}
v, ok = mp["bj"] //bj 不存在, ok 为 false
if ok {
    fmt.Println(v)
} else {
    fmt.Println("key 'bj' 不存在")
}
}

```

2.3.4 range 遍历数组

Range 可以对 string 、 array、 slice、 map、 channel 进行迭代器操作。

```

package main
import (
    "fmt"
)
func main() {
    arr := [3]int{1, 2, 3}
    var mp = map[int]string{1: "a", 2: "b", 3: "c"}
    for k, v := range mp {
        fmt.Println(k, "=", v)
    }
    for _, v := range arr {
        fmt.Println(v)
    }
}

```

map[int]string{1: "a", 2: "b", 3: "c"}，是另一种定义 map 的方式，1,2,3 是 key;a,b,c 是 value。

2.4 常量

常量必须是编译期能确定的，常量的定义使用 const，常量的类型可以是 char string bool 和数字常量。因为编译态的限制，定义它们的表达式必须是常量表达式，可以被编译器求值。例如，1<<3 是常量表达式，math.Sin(math.Pi/4) 不是，因为 math.Sin 的函数调用发生在运行态。

```
const PI=3.1415926
const y="Hello"
const(
z=false
a=123)
```

可以在函数内部定义局部常量。

```
func main() {
    const UserName, Sex = "张三", "男"
    fmt.Println("Hello World!", UserName)
}
```

Go 没有枚举类型，可以用常量模拟。可以用 `iota` 生成从 0 开始的自动增长的枚举值。按行递增，可以省略后续行的 `iota` 关键字。

```
const (
    Sundy = iota
    Monday
    Tuesday
)
/*Sundy= 0 Monday= 1 Tuesday= 2*/
fmt.Println("Sundy=", Sundy, "Monday=", Monday, "Tuesday=", Tuesday)
```

可以在同一行使用多个 `iota`，它们各自增长。

```
func main() {
    const (
        U, V = iota, iota
        W, X
        Y, Z
    )
    /*U= 0 V= 0 W= 1 X= 1 Y= 2 Z= 2*/
    fmt.Println("U=", U, "V=", V, "W=", W, "X=", X, "Y=", Y, "Z=", Z)
}
```

如果某行不想递增，可单独提供初始值。不过想要恢复递增，必须再次使用 `iota`。

```
func main() {
    const (
        A1 = iota //0
        A2
        str = "Hello" //独立值
    )
}
```

```

    s    //没有赋值，跟上一行一样，要想恢复自增，需再次赋值 iota
    A3   = iota
    A4
)
/*A1= 0 A2= 1 str= Hello s= Hello A3= 4 A4= 5*/
fmt.Println("A1=", A1, "A2=", A2, "s=", s, "str=", str, "A3=", A3, "A4=", A4)
}

```

2.5 控制流

2.5.1 if else

Go 的 if 与 C 和 java 中的是相似的，区别在于没有小括号。

```

a := 2
if a == 2 {
    fmt.Println("OK")
}

```

在 if 和条件之间可以包括一个初始化表达式，上面的代码可以写成：

```

if a := 2; a == 2 {
    fmt.Println("OK")
}

```

在 if 条件里初始化的变量，作用域是这个 if 语句块，如上面代码中的变量 a，只能在 if 里使用。if 条件中声明的变量，在 else 中也可以用。

```

if a := 2; a < 2 {
    fmt.Println("a<2")
} else {
    fmt.Println("a=", a)
}

```

下面的代码是错误的，因为 if 和条件之间只能有一个初始化表达式

```
if a := 2; b := 100; a == 2 {  
    fmt.Println("OK")  
}
```

2.52 Switch

Go 的 switch 非常灵活，表达式不必是常量或整数，执行的过程从上至下，直到找到匹配项；而如果 switch 没有表达式，它会匹配 true。

Go 里面 switch 默认相当于每个 case 最后带有 break，匹配成功后不会自动向下执行其他 case，而是跳出整个 switch，但是可以使用 fallthrough 强制执行后面的 case 代码。

```
func main() {  
    i := 5  
    switch i {  
        case 1:  
            fmt.Println("i is equal to 1")  
        case 2:  
            fmt.Println("i is equal to 2")  
        case 3, 4, 5, 6: //case 可以有多个值  
            fmt.Println("i is equal to 3,4,5 or 6")  
            fallthrough //添加后，相当于去掉默认的 break  
        default:  
            fmt.Println("others")  
    }  
}
```

不指定 switch 条件表达式，或直接为 true 时，可用于替代 if...else...if...else。

```
func main() {  
    result := 0  
    if result < 0 {  
        fmt.Println("小于零")  
    } else if result > 0 {  
        fmt.Println("大于零")  
    } else {  
        fmt.Println("等于零")  
    }  
}
```

上面的代码可改写为 switch 看起来更加清晰明了。

```
func main() {
    result := 0
    switch {
    case result < 0:
        fmt.Println("小于零")
    case result > 0:
        fmt.Println("大于零")
    default:
        fmt.Println("等于零")
    }
}
```

2.5.3 for 循环

Go 只有一个关键字用于引入循环。但它提供了除 do-while 外 C 语言当中所有可用的循环方式。

Go 的 for 循环有如下三种形式：

```
for init;condition;post{} ←和 C 的 for 一样
for condition{}           ←和 while 一样
for{}                     ←死循环
```

每次循环都会重新检查条件表达式，如果表达式包含函数调用，将会被多次调用。建议用初始化表达式一次性计算。在 for 中初始化的变量，作用域为该 for 语句块。如下面的代码中，i, j 只在 for 语句中有效。

```
func main() {
    str := "hello world"
    for i, j := 0, len(str); i < j; i++ {
        fmt.Println(string(str[i]))
    }
}
```

用 break 可以提前终止当前循环。

```
func main() {
    i := 0
    for {
        if i > 10 {
            break
        }
        fmt.Println(i)
        i++
    }
}
```

嵌套循环时，可以在 break 后面指定标签，用来指定要终止哪几个循环。

```
func main() {
oute:
    for i := 0; i < 5; i++ {
        for k := 0; k < 5; k++ {
            if i > 0 {
                break oute
            }
            fmt.Println(k)
        }
    }
}
```

continue 用于终止本次循环体的执行继续执行下一个循环。下面是打印非空格符。

```
str := "hello world"
for i, j := 0, len(str); i < j; i++ {
    if string(str[i]) == " " {
        continue
    }
    fmt.Println(string(str[i]))
}
```

与 break 相同，对于嵌套循环，可以用标签，来指定要继续哪一层循环。

```
func main() {
oute:
    for i := 0; i < 5; i++ {
        for k := 0; k < 100; k++ {
            if k > 0 {
                continue oute
            }
            fmt.Println(i)
        }
    }
}
```

第三章函数

3.1 函数定义

Go 是面向过程的编程语言，函数是 Go 程序的基本部件。

```
func Add(a, b int) int {  
    return a + b  
}
```

func 是定义函数的关键字，Add 是函数名；int 是返回值。可以随意安排函数定义的顺序，Go 在编译时会扫描所有的文件。

```
package main  
  
import (  
    "fmt"  
)  
  
func A() {  
    B()  
}  
func B() {  
    fmt.Println("OK")  
}  
func main() {  
    A()  
}
```

如上面的例子中，函数 A,B 的定义顺序可以任意。

函数也是一种类型。

3.2 多值返回

Go 中的函数支持多值返回，可以返回任意数量的返回值。多值返回在 Go 语言中是经常被用到的，比如，一个函数同时返回结果和异常。例如打开文件的函数：func Open(name string) (file *File, err error)。

下面我们看一个例子，divide 函数是计算 a/b 的结果，并返回商和余数。

```
package main
```



```

import (
    "fmt"
)
func divide(a, b int) (int, int) {
    quotient := a / b
    remainder := a % b
    return quotient, remainder
}
func main() {
    q, r := divide(5, 3)
    fmt.Println(q, "", r)
}

```

Go 语言中，可以给函数的返回值命名，就像函数的输入参数那样。命名的返回值，在函数开始的时候被初始化为空。在函数结尾，使用不带参数的 `return` 语句，命名的返回值变量将被用于返回。

上面的 `divide` 函数，可以使用命名的返回值，如下：

```

func divide(a, b int) (quotient, remainder int) {
    quotient = a / b
    remainder = a % b
    return
}

```

3.3 变参函数

Go 中的函数支持变参，变参就是说函数可以有任意数量的参数。变参本质上就是一个 slice，且必须是最后一个参数。将 slice 传递给变参函数时，注意用 `...` 展开，否则就当作单个参数处理了。

```

package main
import (
    "fmt"
)
func sum(aregs ...int) int {
    s := 0
    for _, number := range aregs {
        s += number
    }
    return s
}

```

```
func main() {
    total := sum(1, 2, 3, 4)
    fmt.Println(total)
    slice := []int{1, 2, 3, 4, 5, 6, 7, 8, 9} //定义一个切片
    /*将切片传 sum 时，要用...展开否则将做为一个参数处理
    等价于 sum(1,2,3,4,5,6,7,8,9)
    */
    total = sum(slice...)
    fmt.Println(total)
}
```

3.4 defer

defer 是 Go 语言所特有的，defer 的作用是延迟执行，在函数返回前，按照后进先出的原则依次执行每一个 defer 注册的函数。这样可以保证，函数在返回前被调用，通常且来进行资源释放，错误的处理，清理数据等。下面是一个读文件的例子：

```
package main
import (
    "fmt"
    "os"
)
func ReadFile(strFileName string) (string, error) {
    f, err := os.Open(strFileName)
    if err != nil {
        fmt.Println()
        return "", err
    }
    defer f.Close() //在函数返回前关闭文件
    buf := make([]byte, 1024)
    var strContent string = ""
    for {
        n, _ := f.Read(buf)
        if n == 0 {
            break
        }
        strContent += string(buf[0:n])
    }
    return strContent, nil
}
func main() {
    str, err := ReadFile("main.go")
```

```
if err != nil {  
    fmt.Println(err.Error())  
    return  
}  
fmt.Println(str)  
}
```

3.5 函数类型

函数也是一种类型，拥有相同参数，相同返回值的函数，是同一种类型。用 `type` 来定义函数类型。下面的例子中 `Display` 函数输出大于 5 的数值。

```
package main  
import (  
    "fmt"  
)  
//MyFuncType 是一个接受 int 类型的参数，返回 bool 值的函数类型  
type MyFuncType func(int) bool  
func IsBigThan5(n int) bool {  
    return n > 5  
}  
func Display(arr []int, f MyFuncType) {  
    for _, v := range arr {  
        if f(v) {  
            fmt.Println(v)  
        }  
    }  
}  
func main() {  
    arr := []int{1, 2, 3, 4, 5, 6, 7, 8, 9}  
    Display(arr, IsBigThan5)  
}
```

在上面的例子中，`type MyFuncType func(int) bool` 定义了一个函数类型，将其命名为 `MyFuncType`，接受一个 `int` 类型的参数，并返回一个 `bool` 类型的结果。`IsBigThan5` 是 `MyFuncType` 类型的函数。函数类型，跟 C 里的函数指针有点像。

3.6 错误处理

Go 语言中没有 `try...catch...finally` 这种结构化异常处理，而是用 `panic` 代替 `throw` 抛出异常。使用 `recover` 函数来捕获异常。`Recover` 仅在 `defer` 函数中使用才能捕获异常，此时函数的执行流程已经中断，无法恢复到后续位置继续执行。

```
package main
import (
    "fmt"
)
func Test() {
    defer func() {
        if err := recover(); err != nil {
            fmt.Println(err)
        }
    }()
    divide(5, 0)           //程序出错，中断执行
    fmt.Println("end of test") //该语句不会被执行
}
func divide(a, b int) int {
    return a / b
}
func main() {
    Test()
}
```

第四章 面向对象编程

Go 是面向过程的语言，Go 中没有类，但 Go 支持面向对象的编程，Go 中的 struct 就像其它语言中的类；Go 里没有继承，但可以用组合来实现。

4.1 struct

结构体是一种自定义类型，是不同数据的集合体 struct 是值类型。通常用来定义一个抽象的数据对象，比如学生，可以有姓名、年龄、班级等数据构成，struct 是值类型。结构体的定义格式如下：

```
type Student struct {  
    成员名字1  类型1  
    成员名字2  类型2  
    成员名字3  类型3  
}
```

可以用 New 来创建结构体，然后对各字段进行赋值。

```
type Student struct {  
    Name  string  
    Age   int  
    class string  
}  
  
func main() {  
    s1 := new(Student)  
    s1.Name = "张三"  
    s1.Age = 12  
    s1.class = "21班"  
    fmt.Println(s1)  
}
```

在 Go 中，默认只有大写字母开头的变量，函数，Struct,Struct 里的成员，才可以在包外访问，相当于 public。小写开头的相当于 private。

除了上面这种方式，还可以使用以下两种方式定义，初始化结构体

1) 按照顺序提供初始化值

```
s1:=Student{"张三",12,"2 班"}
```

2) 通过 field:value 的方式初始化

```
s1:=Student{Name:"张三",Age:12,class:"2 班"}
```

Go 语言没有 class,不支持面向对象。但支持面向 struct 结构体的成员函数，使用方法跟

面向对象编程方式相似。定义格式如下：

func (变量名 结构体类型) 函数名(参数列表) (返回值类型列表)

```
type Student struct {
    Name  string
    Age   int
    class string
}
func (this Student) getName() string {
    return this.Name
}
//结构体可以传指针类型
func (this *Student) getAge() int {
    return this.Age
}
func main() {
    s := Student{Name: "张三", Age: 15, class: "32班"}
    fmt.Println(s.getName(), s.getAge())
}
```

4.2 继承

Go 没有 class, 但可以把 struct 当 class 来看待。虽然 Go 不能像 class 那样继承, 但 struct 可以通过匿名字段来实现继承。

```
package main
import "fmt"
type Student struct {
    Name  string
    Age   int
    class string
}
//结构体可以传指针类型
func (this *Student) Display() {
    fmt.Println(this.Name, ",", this.Age)
}
//定义一个大学生类, 继承 Student
type CollegeStudent struct {
    Student
    Profession string
}
```

```
func main() {
    s1 := CollegeStudent{Student: Student{Name: "李四", Age: 23, class: "2004(2)班"},
    Profession: "物理"}
    s1.Display()
    fmt.Println(s1.Student.Name) //可以通过 student 访问 Name
    fmt.Println(s1.Name)         //也可以直接通过 name 访问
}
```

CollegeStudent 将继承 Student 的所有字段和方法。CollegeStudent 也可以重写继承的方法。

```
func (this *CollegeStudent) Display() {
    fmt.Println(this.Name, ",", this.Profession)
}
```

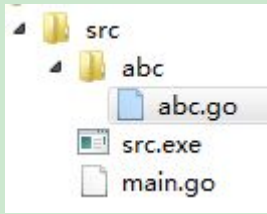
上面说过，大写开头的相当于 public，小写开头的是 private，包外是不可以访问的。所以，如果包 a 中的一个结构，去继承 b 包中结构体，只有大写开头的才能被继承。如果在同一个包中，不受此限制。

```
package abc /*在 abc 包中定义结构体 Student*/
type Student struct {
    Name string
    Age int
    class string /*小写开头的包外不可见，包外的结构体也无法继承该字段*/
}
```

下面我们在 main 包中继承 Student

```
package main
import (
    "abc"
)
type MyStudent struct {
    abc.Student
}
func main() {
    s := MyStudent{}
    s.Student.class = "aaa"
}
```

编译时将报错，s.Student.class undefined (cannot refer to unexported field or method class)。程序目录结构如下：



包名必须跟所在目录名一至。

Go 中虽然可以重写继承来的函数，但 Go 毕竟不支持继承，是以组合的方式模拟了继承。所以在有些时候会出现问题。我们看下面的一个实例。

```
package main
import (
    "fmt"
)
type Fruit struct {
}

func (this *Fruit) DisplayName() {
    fmt.Println(this.GetName())
}
func (this *Fruit) GetName() string {
    return "水果"
}

type Apple struct {
    Fruit
}

func (this *Apple) GetName() string {
    return "苹果"
}
func main() {
    fruit := Fruit{}
    fruit.DisplayName()
    apple := Apple{}
    apple.DisplayName()
}
```

上面代码运行结果为：

```
水果
水果
```

在上面的例子中，Apple 继承了 Friut，并且重载了 GetName 函数，我们期望的结果是，apple.DisplayName() 输出苹果。但实际结果是水果。当然了，这里可以再重写 Display 函数，这样做是绝对没有问题。但通常 Display 里是一些通用的业务逻辑，我不想在每个类中都去重写这个函数，这样不利于代码的维护。要想实现我们要的结果，可以使用3.4节中的函数类型来实现。我们修改一下代码，如下：

```
package main

import (
    "fmt"
)
//定义一个函数类型
type FruitName func() string

type Fruit struct {
    GetFruitName FruitName
}

func (this *Fruit) DisplayName() {
    fmt.Println(this.GetFruitName())
}
func (this *Fruit) GetName() string {
    return "水果"
}
func NewFriut() *Fruit {
    f := new(Fruit)
    f.GetFruitName = f.GetName
    return f
}

type Apple struct {
    Fruit
}

func (this *Apple) GetName() string {
    return "苹果"
}
func NewApple() *Apple {
    a := new(Apple)
    a.GetFruitName = a.GetName
    return a
}
func main() {
    fruit := NewFriut()
```

```
fruit.DisplayName()  
apple := NewApple()  
apple.DisplayName()  
}
```

上面程序的运行结果为：

```
水果  
苹果
```

上面例子中先用 `type FruitName func() string` 定义了一个 `FruitName` 函数类型，然后在 `Fruit` 结构体中定义了一个 `FruitName` 函数类型的成员 `GetFruitName`。在 `NewApple` 和 `NewFruit` 中对 `GetFruitName` 进行了赋值，在 `DisplayName` 函数中调用了 `GetFruitName`。

4.3 Interface

接口是一系列操作的集合，是一种约定。我们可以把它看作与其它对象通讯的协议。任何非接口类型只要拥有某接口的全部方法，就表示它实现了该接口，Go 中无需显式在该类上添加接口声明。

```
type Student struct {  
    Name  string  
    Age   int  
    class string  
}  
type IStudent interface {  
    GetName() string  
    GetAge() int  
}  
func (this *Student) GetName() string {  
    return this.Name  
}  
func (this *Student) GetAge() int {  
    return this.Age  
}  
func main() {  
    var s1 IStudent = &Student{"李四", 23, "2004(2)班"}  
    fmt.Println(s1.GetName())  
}
```

`Interface{}` 没有定义任何方法，称为空接口。任何类型默认都实现了 `interface{}`，相当

于 C 语言中的 `void*` 指针。

第五章多线程

5.1 多线程

线程是 CPU 调度的最小单位，只有不同的线程才能同时多核 CPU 上同时运行。但线程太占资源，线程调度开销大。Go 中的 goroutine 是一个轻量级的线程，执行时只需要 4-5K 的内存，比线程更易用，更高效，更轻便，调度开销比线程小，可同时运行上千万个并发。Go 语言中来启一个 goroutine 非常简单，Go 函数名(), 就开启了个线程。

默认情况下，调度器仅使用单线程，要想发挥多核处理器的并行处理能力，必须调用 runtime.GOMAXPROCS(n) 来设置可并发的线程数，也可以通过设置环境变量 GOMAXPROCS 达到相同的目的（在第一章，安装 Go 中有介绍）。

Runtime 包中提供了几个与 goroutine 有关的函数。Gosched() 让当前正在执行的 goroutine 放弃 CPU 执行权限。调度器安排其它正在等待的线程运行。如下面的程序，开启两个 goroutine，一个输出 Hello，一个输出 world。

```
package main
import (
    "fmt"
    "runtime"
    "time"
)
func SayHello() {
    for i := 0; i < 10; i++ {
        fmt.Print("Hello ")
        runtime.Gosched()
    }
}
func SayWorld() {
    for i := 0; i < 10; i++ {
        fmt.Println("World!")
        runtime.Gosched()
    }
}
func main() {
    go SayHello()
    go SayWorld()
    time.Sleep(5 * time.Second)
}
```

上面程序的运行结果是输出几行 Hello World!，首先启动了一个 SayHello 线程，接着启动了 SayWorld 线程，在 SayHello 线程中输出 Hello 后，调用 Gosched 函数，释 CPU 权限；之后 SayWorld 获得 CPU 权限，输出 World 两个线程交替获得 CPU 权限。但我们是多线程，

并发运行的,就是说 SayHello,SayWorld 在多核 CPU 上是同时运行的,我的 **GOMAXPROCS** 环境变量设置为了 8, 输出结果也是 Hello World!, 没有出现混乱。

NumCPU()返回 CPU 核数, NumGoroutine()返回当前进程的 Goroutine 线程数。即便我们没有开启新的 goroutine, NumGoroutine()也是2, 这是因为除了主线程 main, go 还会启动一个 GC Heap 用来对内存进行管理及垃圾回收。

```
import (  
    "fmt"  
    "runtime"  
)  
func main() {  
    fmt.Println(runtime.NumCPU())  
    fmt.Println(runtime.NumGoroutine())  
}
```

我的 CPU 是 I7, 四核,NumCPU(), 返回的结果是 8, 因为 intel 的超线程技术, 可以在一个实体处理器中, 运行两个逻辑线程(具体可以去百度一下超线程)。虽然我是四核, 但是有 8 个逻辑内核。

runtime.Goexit()函数用于终止当前 goroutine,但 defer 函数将会被继续调用。

```
package main  
import (  
    "fmt"  
    "runtime"  
)  
func test() {  
    defer func() {  
        fmt.Println("in defer!")  
    }()  
    for i := 0; i < 10; i++ {  
        fmt.Println(i)  
        if i > 5 {  
            runtime.Goexit()  
        }  
    }  
}  
func main() {  
    go test()  
    var str string  
    fmt.Scan(&str)  
}
```

GOMAXPROCS(n int) int 用来设置可同时运行的线程数, 并返回当前设置的值, 如果 n<1 将不会改变当前的设置。通常这样使用 runtime.GOMAXPROCS(runtime.NumCPU())。

```

import (
    "fmt"
    "runtime"
)
func main() {
    n:=runtime.GOMAXPROCS(runtime.NumCPU())
    fmt.Println(n)
}

```

运行程序，输出结果为 4，在第一章中提到我的 GOMAXPROCS 环境变量设置值为 4。

5.2 chan

Goroutine 之间通过 channel 来通讯，可以认为 channel 是一个管道或者先进先出的队列。你可以从一个 goroutine 中向 channel 发送数据，在另一个 goroutine 中取出这个值。生产者/消费者是最经典的 channel 使用示例。生产者 goroutine 负责将数据放入 channel，消费 goroutine 从 channel 中取出数据进行处理。

```

package main
import "fmt"
func producer(c chan int) {
    defer close(c) //关闭 channel
    for i := 0; i < 10; i++ {
        c <- i //阻塞，直到数据被消费者取走后才能发送下一条数据
    }
}
func consumer(c, f chan int) {
    for {
        if v, ok := <-c; ok {
            fmt.Println(v) //阻塞，直到生产者放入数据后继续取数据
        } else {
            break
        }
    }
    f <- 1 //向 F 发一个数据，告诉 main 数据已接收完成
}
func main() {
    buf := make(chan int)
    flg := make(chan int)
    go producer(buf)
}

```

```

go consumer(buf, flg)
<-flg //等待数据接收完成
}

```

可以使用 range 从 channel 中取数据，直到遇到 channel close 才停止。可以把上面的 consumer 改成 range。

```

func consumer(c, f chan int) {
    for v := range c {
        fmt.Println(v)
    }
    f<- 1 //向 F 发一个数据，告诉 main 数据已接收完成
}

```

可以将 channel 指定为单向通信。比如”<-chan int”仅能接收，”chan<-int”仅能发送。生产者消费者可改成下面的方式

```

func producer(c chan<- int) {
    defer close(c) //关闭 channel
    for i := 0; i < 10; i++ {
        c <- i //阻塞，直到数据被消费者取走后才能发送下一条数据
    }
}
func consumer(c <-chan int, f chan<- int) {
    for v := range c {
        fmt.Println(v)
    }
    f<- 1 //向 F 发一个数据，告诉 main 数据已接收完成
}

```

Channel 可以是带缓冲的。Make 第二个参数做为缓冲长度来初始化一个带缓冲的 channel:

```

c:=make(chan int,10)

```

向带缓冲的 channel 发送数据时，只有缓冲区满时，发送操作才会被阻塞。当缓冲区空时，接收操作才会阻塞。

```

func main() {
    c := make(chan int, 2)
    c <- 1
    c <- 2 //此时若再向 c 发送数据，将会阻塞，运行时报错
    fmt.Println(<-c)
    fmt.Println(<-c) //此时若再从 c 中取数据，将出现阻塞，运行时报错
}

```


如果有多个 channel 需要监听，可以考虑用 select，随机处理一个可用的 channel。

```
func fibonacci(c, quit chan int) {
    x, y := 0, 1
    for {
        select {
        case c <- x:
            x, y = y, x+y
        case <-quit:
            fmt.Println("quit")
            return
        }
    }
}

func main() {
    c := make(chan int)
    quit := make(chan int)
    go func() {
        for i := 0; i < 10; i++ {
            fmt.Println(<-c)
        }
        quit <- 0
    }()
    fibonacci(c, quit)
}
```

当一个 channel 被 read/write 阻塞时，会一直被阻塞下去，直到 channel 关闭,产生一个异常退出程序。Channel 内部没有超时的定时器。但我们可以用 select 来实现 channel 的超时机制。

```
import (
    "fmt"
    "time"
)

func main() {
    c := make(chan int)
    select {
    case <-c: //因为没有向 c 发送数据，所以会一直阻塞
        fmt.Print("收到数据")
    case <-time.After(5 * time.Second):
        fmt.Println("超时退出")
    }
}
```

5.3 进程同步

假设现在我们有两个线程，一个线程写文件，一个线程读文件。由于程序是多线程的，如果在读文件的同时，写文件的线程向文件中写数据，就会出现数据不一致的问题。为了保证能够正确地读写文件，在读文件的时候，不能进行写入的操作，在写入时，不能进行读的操作。这就是互斥锁。互斥锁是线程间同步的一种机制，用来保证在同一时刻只有一个线程访问共享资源。Go 中的互斥锁在 `sync` 包中。下面是一个线程安全的 map

```
package main
import (
    "errors"
    "fmt"
    "sync"
)
type MyMap struct {
    mp    map[string]int
    mutex *sync.Mutex
}
func (this *MyMap) Get(key string) (int, error) {
    this.mutex.Lock()
    i, ok := this.mp[key]
    this.mutex.Unlock()
    if !ok {
        return i, errors.New("不存在")
    }
    return i, nil
}
func (this *MyMap) Set(key string, v int) {
    this.mutex.Lock()
    defer this.mutex.Unlock()
    this.mp[key] = v
}
func (this *MyMap) Display() {
    this.mutex.Lock()
    defer this.mutex.Unlock()
    for k, v := range this.mp {
        fmt.Println(k, "=", v)
    }
}
func SetValue(m *MyMap) {
    var a rune
```

```

a = 'a'
for i := 0; i < 10; i++ {
    m.Set(string(a+rune(i)), i)
}
}

func main() {
    m := &MyMap{mp: make(map[string]int), mutex: new(sync.Mutex)}
    go SetValue(m) /*启动一个线程向 map 写入值*/
    go m.Display() /*启动一个线程读取 map 的值*/
    var str string /*这里主要是等待线程结束*/
    fmt.Scan(&str)
}

```

虽然我们赋值时，是按 123 进行的，但多次运行的结果，发现，map 的顺序是不一样的，因为 map 是无序的。

再以文件操作为例，不能允许两个线程，同时读写，但如果两个线程同时读是没有问题的。只要在读的时候不要有写的线程。这就是读写锁。读写锁允许多个线程同时读，所以并行性更好。读写锁具有以下特性：

- 1) 多个读操作可以同时进行
- 2) 写必须互斥，不允许两个写操作同时进行，也不能读、写操作同时进行。

3) 写优先于读。在当前线程以读模式加锁后，其它线程进行读模式加锁，可以获得读的权限;在当前线程以读模式加锁后，其它线程加写锁时，将会堵塞，并且后继的读锁将会堵塞。这样可以避免读模式锁长期占用，导致写操作一直阻塞。下面的例子可以改成读写锁的实现。下面仅给出有改动的部分代码：

```

type MyMap struct {
    mp    map[string]int
    mutex *sync.RWMutex
}

func (this *MyMap) Get(key string) (int, error) {
    this.mutex.RLock()
    i, ok := this.mp[key]
    this.mutex.RUnlock()
    if !ok {
        return i, errors.New("不存在")
    }
    return i, nil
}

func (this *MyMap) Display() {
    this.mutex.RLock()
    defer this.mutex.RUnlock()
    for k, v := range this.mp {
        fmt.Println(k, "=", v)
    }
}

```

```
    }  
}  
  
func main() {  
    m := &MyMap{mp: make(map[string]int), mutex: new(sync.RWMutex)}  
    .....  
}
```

第六章 日期与定时器

6.1 日期的获取与计算

Time 包定义了所有时间相关的函数。获取当前时间用 `time.Now()`。

```
import (  
    "fmt"  
    "time"  
)  
func main() {  
    fmt.Println(time.Now())  
}
```

Format 函数把一个时间格式化为字符串，定义格式为 `func (t Time) Format(layout string)`，layout 的格式有点怪，不同于其它语言中的格式。在 Go 中 2006 代表年，01 代表月，02 代表日，15 代表时，04 代表分，05 代表秒。不知道 2006-01-02 是什么重要的日子，Go 要用这种格式表。

```
import (  
    "fmt"  
    "time"  
)  
func main() {  
    fmt.Println(time.Now().Format("2006-01-02 15:04:05"))  
}
```

`func Parse(layout, value string) (Time, error)`，用来把一个字符串转换成日期。layout 与 Format 里的格式一样，2006 代表年，01 代表月，02 代表日，15 代表时，04 代表分，05 代表秒。

```
import (  
    "fmt"  
    "time"  
)  
func main() {  
    d, err := time.Parse("01-02-2006", "06-17-2013")  
    if err != nil {  
        fmt.Println(err.Error())  
    }  
    fmt.Println(d)  
}
```

type Duration int64 表示一个持续的时间，单位是纳秒。多用于时间的加减、定时等操作需要传 Duration 做为参数。时间相加用 Add,相减 Sub ,时间的比函数有 After, Equal, Before。

```
import (
    "fmt"
    "time"
)
func main() {
    t := time.Now()
    t2 := t.Add(24 * time.Hour) //当前时间加24小时，即明天的这个时间
    d := t2.Sub(t)              //t2-t1,相差24小时
    fmt.Println(t)
    fmt.Println(t2)
    fmt.Println(d)
    if t.Before(t2) { //t 小 t2
        fmt.Println("t<t2")
    }
    if t2.After(t) { //t2大于 t
        fmt.Println("t2>t")
    }
    if t.Equal(t) { //判断两个时间是否相等
        fmt.Println("t=t")
    }
}
```

6.2 定时器

Go 里的定时器相关的函数有 time.After,time.AfterFunc,time.Tick 等，下面我们分别介绍。

func After(d Duration) <-chan Time 等待一个时间段 d，然后把当前时间发送到 channel 中。与 NewTimer 等价。

```
import (
    "fmt"
    "time"
)
func main() {
    fmt.Println(time.Now())
    c := time.After(10 * time.Second) //返回 channel 类型,10秒后向 channel 发送当前时间
    t := <-c
    fmt.Println(t)
    tm := time.NewTimer(10 * time.Second) //NewTimer 返回 Timer 类型
    t = <-tm.C /*Timer 结构中有一个 channel C,10秒后，把当前时间发送到 C*/
    fmt.Println(t)
}
```

`func AfterFunc(d Duration, f func()) *Timer`，等待时间 `d`，然后调用函数 `f`。注意这里的函数 `f` 是不带任何参数和返回值的。

```
import (
    "fmt"
    "time"
)
func Test() {
    fmt.Println("Hello world!", time.Now())
}
func main() {
    fmt.Println(time.Now())
    time.AfterFunc(10*time.Second, Test)
    var str string
    fmt.Scan(&str) /*这里主要是等待用户输入，不让进程结束，进程结束定时器也就无
效了。*/
}
```

`Tick` 与 `After` 有点类似，唯一的区别是 `After` 等待时间到期后，定时器就结束了。`Tick` 是每隔一段时间 `d` 都会向 `channel` 发送当前时间。

```
import (
    "fmt"
    "time"
)
func main() {
    c := time.Tick(10 * time.Second)
    for t := range c {
        fmt.Println(t)
    }
}
```

在当前版本，也就是 Go1.1 及之前的版，定时器是有 BUG 的。把系统时间提前，然后再改回来。会导至定时器失效。

第 7 章文件操作

7.1 路径

在 `path` 包中封装了一些路径相关的操作，在开始接触文件操作之前，我们先看看路径相关的函数。在 Linux 中，路径的格式为 `/user/bin` 路径中的分隔符是 `/`；Windows 中的路径格式为 `C:\Windows` 路径中的分隔符是 `\`。而在 Go 中，只认 `/`，不知道怎么回事，也许是对 Windows 的支持不够好。所以要想能够正常的使用 `path` 中的函数需要把 `\` 转换成 `/`。

`func Base(path string) string` 返回路径的最后一部分。

```
import (
    "fmt"
    "path"
    "strings"
)
func main() {
    fmt.Println(path.Base("/usr/bin"))      //输出了 bin
    fmt.Println(path.Base(""))              //输出.
    fmt.Println(path.Base("C:\\Windows"))   /*无法识别 Windows 下的
    路径分隔符，将会把 C:\\Windows 做为一个路径*/
    fmt.Println(path.Base(strings.Replace("C:\\Windows", "\\ ", "/",
-1))) /*把\转换成*/
}
```

在上面的例子中，我们使 `strings.Replace` 对路径分隔符进行了转换，最后一个参数，用来指定替换次数，`-1` 表示替换所有。

`func Clean(path string) string` 返回一个跟 `path` 等价的短路径。一般在路径中出现 `./` 或 `../` 时可以使用本函数，返回一个等价的路径。如 `path.Clean("/a/b/../c")` 的结果为 `/a/c`

```
import (
    "fmt"
    "path"
)
func main() {
    fmt.Println(path.Clean("/a/b/../c"))    /*/a/c*/
    fmt.Println(path.Clean("/a/b/../../c")) /*/a/c*/
}
```

`func Dir(path string) string` 返回路径中的目录部分。也就是最后一个 `/` 前面的部分。


```
import (
    "fmt"
    "path"
)
func main() {
    fmt.Println(path.Dir("/a/b/./c/d/e")) /*a/c/d*/
    fmt.Println(path.Clean("/a/b/"))      /*a/b*/
}
```

func Ext(path string) string 用来取文件的扩展名。

```
import (
    "fmt"
    "path"
)
func main() {
    fmt.Println(path.Ext("/a/b/./c/d./e")) /*没有扩展名*/
    fmt.Println(path.Ext("/a/b/test.txt")) /*.txt*/
}
```

func IsAbs(path string) bool 用来判断路径是否绝对路径。在 Linux 下如果路径是以/开头的是绝对路径，如/user/bin，否则是相对路径；Windows 下，以盘符开头的是绝对路径，如 C:\Windows\system,而 Windows\system 是相对路径。

```
import (
    "fmt"
    "path"
    "strings"
)
func main() {
    fmt.Println(path.IsAbs("/a/b/c"))
    fmt.Println(path.IsAbs(strings.Replace("C:\\Windows\\system",
        "\\ ", "/", -1))) /*Go 只识别/所以需要转换一下*/
}
```

上面代码，第一个输出 true，/a/b/c 是绝对路径；第二个输出 false，没错，C:\Windows\system 是绝对路径，但 Go 好像只支持 Linux 格式的路径，所以此处判断错误。

func Join(elem ...string) string 用来进行路径的连接。如将 a/b，和 c 连接成 a/b/c。

```
import (
    "fmt"
    "path"
)
func main() {
    fmt.Println(path.Join("a/b", "c"))/*a/b/c*/
}
```

```

    fmt.Println(path.Join("C:\\Windows",
        "System"))/*C:\Windows\System*/
}

```

func Split(path string) (dir, file string)把路径分割成目录和文件两部分。

```

package main
import (
    "fmt"
    "path"
)
func main() {
    fmt.Println(path.Split("/a/b/test.txt")) /*a/b/ test.txt*/
    fmt.Println(path.Split("/a/b/c/"))       /*a/b/c/ */
}

```

在上面的例子中 path.Split("/a/b/c/") 只有目录部分/a/b/c/，没有文件部分，所以文件部分为空。

func Abs(path string) (string, error)用来把相对路径转换成绝对路径，该函数位于 path/filepath 包中。

```

import (
    "fmt"
    "path/filepath"
)
func main() {
    fmt.Println(filepath.Abs("."))
}

```

func Walk(root string, walkFn WalkFunc) error，用来遍历 root 目录下的所有文件和子目录。WalkFunc 是一个函数类型，定为 type WalkFunc func(path string, info os.FileInfo, err error) error，path 为当前文件或文件夹的完整路径，info 是 os.FileInfo 结构的表示。

```

import (
    "fmt"
    "os"
    "path/filepath"
)
func DispFile(path string, info os.FileInfo, err error) error {
    fmt.Println(path, "-----", info.Name(), "-----", info.IsDir())
    return nil
}
func main() {
    filepath.Walk(".", DispFile)
}

```

7.2 文件读写

在 `io` 包中提供了一些文件操作的函数。

`func Create(name string) (file *File, err error)`

创建新文件，如果文件已存在，将被截断。新建的文件是可读写的。默认权限为 0666(Linux 下文件的权限设置格式)。

`func Open(name string) (file *File, err error)`

打开已经存在的文件，用来读取文件内容。`Open` 打开的文件是只读的。不能写。

`func OpenFile(name string, flag int, perm FileMode) (file *File, err error)`

`OpenFile` 是一个通用的函数，可以用来创建文件，以只读方式打开文件，以读写方式打开文件等。`Name` 是要打开或创建的文件名；`flag` 是打开文件的方式，以只读式或读写方式，`flag` 可以是下面的取值：

O_RDONLY 以只读方式打开文件。

O_WRONLY 以只写方式打开文件。

O_RDWR 以读写方式打开文件

O_APPEND 以追加方式打开文件，写入的数据将追加到文件尾。

O_CREATE 当文件不存在时创建文件。

O_EXCL 与 `O_CREATE` 一起使用，当文件已经存在时 `Open` 操作失败。

O_SYNC 以同步方式打开文件。每次 `write` 系统调用后都等待实际的物理 I/O 完成后才返回，默认(不使用该标记)是使用缓冲的，也就是说每次的写操作是写到系统内核缓冲区中，等系统缓冲区满后才写到实际存储设备。

O_TRUNC 如果文件已存在，打开时将会清空文件内容。必须于 `O_WRONLY` 或 `ORDWR` 配合使用。截断文件，需要有写的权限。

`FileMode` 参数是文件的权限，只有在文件不存在，新建文件时该参数才有效。用来指定新建的文件的权了。必须跟 `O_CREATE` 配合使用。

```
import (
    "fmt"
    "io"
    "os"
)
func main() {
    /*打开 D:\\新建文本文档.txt 文件，如果文件不存在将会新建，如果已
    存在，新写入的内容将追加到文件尾*/
    f, err := os.OpenFile("D:\\ 新 建 文 本 文 档 .txt",
os.O_RDONLY|os.O_APPEND|os.O_CREATE, 0666)
    if err != nil {
        fmt.Println(err.Error())
        return
    }
    f.WriteString("\r\n 中国\r\n")
    buf := make([]byte, 1024)
    var str string
```

```

/*重置文件指针，否则读不到内容的。*/
f.Seek(0, os.SEEK_SET)
for {
    n, ferr := f.Read(buf)
    if ferr != nil && ferr != io.EOF {
        fmt.Println(ferr.Error())
        break
    }
    if n == 0 {
        break
    }
    fmt.Println(n)
    str += string(buf[0:n])
}
fmt.Println(str)
f.Close()
}

```

上面的例子中，写完文件后，要想读取文件内容，需要调 `Seek` 重置文件指针，否则是读不到文件内容的。因为我们写入操作完成后，当前文件指针是在文件的末尾。所以要想读到内容需要重置文件指针开文件的开头儿。

`func (f *File) Seek(offset int64, whence int) (ret int64, err error)`

`Seek` 用来设置文件指针的位置，`offset` 是偏移量，`whence` 的取值可以是下面的三个：

SEEK_SET `offset` 是相对文件开始位置的偏移量。

SEEK_CUR `offset` 是相对文件指针当前位置的偏移量。

SEEK_END `offset` 是相对文末尾的偏移量。

上面的例子中 `f.Seek(0, os.SEEK_SET)`，就是把文件指针，移动到文件开头。

在 `ioutil` 包中封装了一些函数，让 IO 操作更简单，方便。

`func ReadAll(r io.Reader) ([]byte, error)`

从 `r` 中读取所有内容。在上面的例子中我们用 `f.Read` 来循环读取文件中的内容。可以使用 `ReadAll` 来代替，使代码变得简单。

```

import (
    "fmt"
    "io/ioutil"
    "os"
)
func main() {
    f, err := os.OpenFile("D:\\新建文本文档.txt",
os.O_RDONLY|os.O_APPEND|os.O_CREATE, 0666)
    if err != nil {
        fmt.Println(err.Error())
    }
}

```

```

        return
    }
    defer f.Close()
    buf, err1 := ioutil.ReadAll(f)
    if err1 != nil {
        fmt.Println(err1.Error())
        return
    }
    fmt.Println(string(buf))
}

```

ReadAll 简化了 IO 操作，但要读取文件内容还有更简单的方法，看下面这个函数。

`func ReadFile(filename string) ([]byte, error)`

从文件 filename 中读取内容，一次性读取整个文件，成功 `error=nil`。

```

import (
    "fmt"
    "io/ioutil"
)
func main() {
    buf, err := ioutil.ReadFile("D:\\新建文本文档.txt")
    if err != nil {
        fmt.Println(err.Error())
        return
    }
    fmt.Println(string(buf))
}

```

`func WriteFile(filename string, data []byte, perm os.FileMode) error`

向文件中写数据，如果文件不存在，将以 perm 权限创建文件。

```

import (
    "fmt"
    "io/ioutil"
)
func main() {
    err := ioutil.WriteFile("e:\\a.txt", []byte("abcdefg"), 0777)
    if err != nil {
        fmt.Println(err.Error())
    } else {
        fmt.Println("OK")
    }
}

```

7.3 遍历目录下的文件

OpenFile 除了可以打开文件,还可以打开一个目录。在 File 对像有一个 Readdir 函数,用来读取某目录下的所有文件和目录信息,位于 OS 包中。定义如下:

```
func (f *File) Readdir(n int) (fi []FileInfo, err error)
```

n>0 最多返回 n 个文件。如个小于等零, 返回所有的。

```
import (
    "fmt"
    "os"
)
func main() {
    f, err := os.OpenFile("C:\\Windows", os.O_RDONLY, 0666)
    if err != nil {
        fmt.Println(err.Error())
        return
    }
    arrFile, err1 := f.Readdir(0)
    if err1 != nil {
        fmt.Println(err1.Error())
        return
    }
    for k, v := range arrFile {
        fmt.Println(k, "\t", v.Name(), "\t", v.IsDir())
    }
}
```

前面提到的 filepath.Walk 也是用 Readdir 来实现的。D 在 ioutil 包中还提供了一个 ReadDir 函数,定义如下:

```
func ReadDir(dirname string) ([]os.FileInfo, error)
```

读取目录下所有的文件和子目录。是对 File.Readdir 的封装。

```
import (
    "fmt"
    "io/ioutil"
)
func main() {
    arrFile, err := ioutil.ReadDir("C:\\Windows")
    if err != nil {
        fmt.Println(err.Error())
        return
    }
}
```

```
for k, v := range arrFile {
    fmt.Println(k, "\t", v.Name(), "\t", v.IsDir())
}
}
```

7.4 gob 序列化

序列化就是将对象的状态信息转换为可以存储或传输的形式过程。在序列化期间，对象将其当前状态写入到临时或持久性存储区。之后，可以通过从存储区中读取或反序列化对象的状态，重新创建该对象。

在本节中我们仅介绍 gob 序列化的方法，xml,json 的序列化将放在下一章中讨论。

Gob 是 Go 中所特有的序列化技术，它支持除 interface,function,channel 外的所有 Go 数据类型。序列化使用 Encoder，反序列化使用 Decoder。我们可以把一个结构体序列化到文件中。然后再反序列化。

```
import (
    "encoding/gob"
    "fmt"
    "os"
)
type Student struct {
    Name string
    Age  int
}
func main() {
    s := &Student{"张三", 19}
    f, err := os.Create("data.dat")
    if err != nil {
        fmt.Println(err.Error())
        return
    }
    defer f.Close()
    //创建 Encoder 对象
    encode := gob.NewEncoder(f)
    //将 s 序列化到 f 文件中
    encode.Encode(s)
    //重置文件指针到开始位置
    f.Seek(0, os.SEEK_SET)
    decoder := gob.NewDecoder(f)
    var s1 Student
    //反序列化对象
    decoder.Decode(&s1)
```

```
fmt.Println(s1)  
}
```


第 8 章 JSON 与 XML 解析

8.1 XML 序列化与解析

Xml 做为一种平台无关的数据交换和信息传递技术应用十分广泛。如在 WebService 中使用 XML 将数据编码成 SOAP 消息。很多接口也使用 XML 来传递数据。Go 中提供 XML 序列化的文法，位于 encoding/xml 包中。

func (enc *Encoder) Encode(v interface{}) error 可以把一个对像直接序列化到 io.Writer 对像中。

func (d *Decoder) Decode(v interface{}) error 从 io.Reader 中，返序列化 xml。

```
package main
import (
    "encoding/xml"
    "fmt"
    "os"
)
type Student struct {
    Name string
    Age  int
}
func main() {
    f, err := os.Create("data.dat")
    if err != nil {
        fmt.Println(err.Error())
        return
    }
    defer f.Close()
    s := &Student{"张三", 19}
    //创建 encode 对像
    encoder := xml.NewEncoder(f)
    //将 s 序列化到文件中
    encoder.Encode(s)
    //重置文件指针到开始位置
    f.Seek(0, os.SEEK_SET)
    decoder := xml.NewDecoder(f)
    var s1 Student
    //从文件中反序列化成对像
    decoder.Decode(&s1)
    fmt.Println(s1)
}
```

用记事本打开 data.dat，可以看到文件内容是一段 XML，如下：

```
<Student><Name>张三</Name><Age>19</Age></Student>
```

xml 包的 Marshal 函数可以把一个对象直接序列化字符。

```
import (
    "encoding/xml"
    "fmt"
)
type Student struct {
    Name string
    Age  int
}
func main() {
    s := &Student{"张三", 19}
    result, err := xml.Marshal(s)
    if err != nil {
        fmt.Println(err.Error())
        return
    }
    fmt.Println(string(result))
}
```

上面的程序结果为：

```
<Student><Name>张三</Name><Age>19</Age></Student>
```

UnMarshal 可以将一个 xml 反序列化为对象

```
import (
    "encoding/xml"
    "fmt"
)
type Student struct {
    Name string
    Age  int
}
func main() {
    str := `<?xml version="1.0" encoding="utf-8"?>
        <Student>
            <Name>张三</Name>
            <Age>19</Age>
        </Student>`
    var s Student
    xml.Unmarshal([]byte(str), &s)
    fmt.Println(s)
}
```

在反序列化 XML “<Student><Name>张三</Name><Age>19</Age></Student>” 时，结构体名称跟<Student>对应，字段名 Name，与<Name>对应。结构体，结构体中的字段必须是公有的，即大写字母开头的。如果要解析的 xml 是小的，可以使用 tag 来指定 Struct 的字段与 xml 标记的对应关系。

```
import (
    "encoding/xml"
    "fmt"
)

type Student struct {
    XMLName xml.Name `xml:"student"`
    Name     string  `xml:"name"`
    Age      int    `xml:"age"`
}

type ABC string
func main() {
    str := `<?xml version="1.0" encoding="utf-8"?>
        <student>
            <name>张三</name>
            <age>19</age>
        </student>`

    var s Student
    xml.Unmarshal([]byte(str), &s)
    fmt.Println(s)
}
```

XMLName xml.Name `xml:"student"` 指定该结构体对应的 xml 标记，`xml:"name"` 指定了该字段对应的 xml 标记，这样 xml 就可以正确解析了。结构体的字段除了可以跟 Xml 的子结点绑定，如 phones>phone。

```
import (
    "encoding/xml"
    "fmt"
)

type Student struct {
    XMLName xml.Name `xml:"student"`
    Name     string  `xml:"name,attr"`
    Age      int    `xml:"age,attr"`
    Phone    []string `xml:"phones>phone",`
}

type ABC string
func main() {
    str := `<?xml version="1.0" encoding="utf-8"?>
        <student name="张三" age="19">
            <phones>
                <phone>12345</phone>
            </phones>
        </student>`

    var s Student
    xml.Unmarshal([]byte(str), &s)
    fmt.Println(s)
}
```

```

        <phone>67890</phone>
    </phones>
</student>`

var s Student
xml.Unmarshal([]byte(str), &s)
fmt.Println(s)
}

```

除上面的方法,xml包还提供了其它的解析xml的文法。在.net,java中都提供XMLReader类来解析xml,在Go中也有类似的方法。我们看下面的一个例子:

```

package main
import (
    "encoding/xml"
    "fmt"
    "strings"
)
type Student struct {
    Name string
    Age  int
}
func main() {
    str := `<?xml version="1.0" encoding="utf-8"?><Student><Name>张三</Name><Age>19</Age></Student>`
    decoder := xml.NewDecoder(strings.NewReader(str))
    var strName string
    for {

        token, err := decoder.Token()
        if err != nil {
            break
        }
        switch t := token.(type) {
        case xml.StartElement:
            stelm := xml.StartElement(t)
            fmt.Println("Start ", stelm.Name.Local)
            strName = stelm.Name.Local
        case xml.EndElement:
            endelem := xml.EndElement(t)
            fmt.Println("End ", endelem.Name.Local)
        case xml.CharData:
            data := xml.CharData(t)

```

```

        str := string(data)
        switch strName {
        case "Name":
            fmt.Println("姓名: ", str)
        case "Age":
            fmt.Println("年龄: ", str)
        default:
            fmt.Println("other:", str)
        }
    }
}

```

在上面这几种方法中 Token 解析是最快的。对于大文件解析，或对性能有要求时，这种方法是最优选择。

8.2 JSON 序列化与反序列化

Json 是一种比 XML 更轻量级的数据交换格式，易于人们阅读和编写，也易于程序解析和生成。是较理想的、跨平台的、跨语言的数据交换语言，应用十分广泛。Go 提供了对 Json 解/编码的支持。

```

package main
import (
    "encoding/json"
    "fmt"
    "os"
)
type Student struct {
    Name string
    Age  int
}
func main() {
    f, err := os.Create("data.dat")
    if err != nil {
        fmt.Println(err.Error())
        return
    }
    s := &Student{"张三", 19}
    //创建 encode 对象
    encoder := json.NewEncoder(f)
    //将 s 序列化到文件中
    encoder.Encode(s)
}

```

```

//重置文件指针到开始位置
f.Seek(0, os.SEEK_SET)
decoder := json.NewDecoder(f)
var s1 Student
//从文件中反序列化成对像
decoder.Decode(&s1)
fmt.Println(s1)
}

```

同样 Json 也提供了 Marshal, Unmarshal。对于结构体, 可以使用`json:"JsonName"`, 来指定解/编码时对应的 json 名称。

```

package main
import (
    "encoding/json"
    "fmt"
)
type Student struct {
    Name string `json:"userName"`
    Age  int
}
func main() {
    s := &Student{"张三", 19}
    //将 s 编码为 json
    buf, err := json.Marshal(s)
    if err != nil {
        fmt.Println(err.Error())
        return
    }
    fmt.Println(string(buf))
    //将 json 字符串转换成 Student 对像
    var s1 Student
    json.Unmarshal(buf, &s1)
    fmt.Println(s1)
}

```

程序运行结果为:

```

{"userName": "张三", "Age": 19}
{张三 19}

```

Unmarshal 最大的特点就是, 可以把 json 解析到一个 `map[string]interface{}` 里。

```

import (
    "encoding/json"
    "fmt"
)

func main() {
    str := `{"userName":"张三","Age":19}`
    var m map[string]interface{}
    json.Unmarshal([]byte(str), &m)
    for k, v := range m {
        switch v.(type) {
        case float64:
            fmt.Println(k, " 是 int 类型, 值为:", v)
        case string:
            fmt.Println(k, " 是 string 类型, 值为: ", v)
        default:
            fmt.Println(k, "无法误用别的类型")
        }
    }
}

```

在上面的代码中 Age 明明是 int 解析后成了 float64。这是因为 Go 中规定，Json 中的布尔值会被解析为布尔值，Json 中的所有数字(整型，浮点型)将被解析为 float64，Json 中的 string，被解析为 string 类型，Json 中的数组被解析为 interface{} 数组，Json 中的空值解为 nil。

第 9 章 MySQL 数据库操作

9.1 安装 MySQL 驱动

在实际应用中数据库操作是经常用到的。Go 提供了 `database/sql`, `database/driver` 两个包。`database/driver` 定义了一些标准的接口, 这些接口由具体的数据库驱动程序实现, Go 官方没有提供具体的驱动程序, 仅提供了这么一个接口, 驱动程序由第三方实现。`database/sql` 包提供了一些通用的方法, 这里的函数, 多是调用了 `database/driver` 接口来实现的。接下来我们介绍几个常用的数据操作的函数。这里我们仅以 Mysql 为例, 在开始之前, 我们要先安装 Mysql 的驱动程序。我用的是 <https://github.com/go-sql-driver/mysql> 驱动。安装方式如下:

```
go get github.com/go-sql-driver/mysql
go install github.com/go-sql-driver/mysql
```

9.2 MySQL 数据库操作

下面我们介绍一下 `database/sql` 包的几个常用函数

`func Open(driverName, dataSourceName string) (*DB, error)`

根据 `driverName` 打开指定的数据库。`driverName` 驱动的名称, `dataSourceName` 通常包含了数据库名, 和连接信息, 如服务器地址、用户名、密码等。不同的驱动程序, 会有不同的格式。

- `func (db *DB) Exec(query string, args ...interface{}) (Result, error)`

执行一个 SQL 查询, 不返回任何行。通常用来执行数据的插入, 更新操作。`query` 是要执行的 SQL 语句, `args` 是参数。执行成功 `error` 为 `nil`。Result 是一个接口, 定义如下:

```
type Result interface {
    LastInsertId() (int64, error)
    RowsAffected() (int64, error)
}
```

`LastInsertId` 返回最后一次自动长列的值。`RowsAffected`, 返回所影响的行。

- `func (db *DB) Query(query string, args ...interface{}) (*Rows, error)`

执行 SQL, 并返回数据行, 一般用来获取数据。此方法相当于 .Net 里的 `ExecuteReader`。`Rows` 用来读取返回的数据, 相当于 .Net 里的 `SqlDataReader`, 需要在数据库连接的情况下读取数据。也就是在我们关闭连接前读取数据。

```
func (r *Row) Scan(dest ...interface{}) error
```

用来从返回的数据中, 取数据。

```
var id int
var name string
row.Scan(&id, &name)
```


- `func (db *DB) QueryRow(query string, args ...interface{}) *Row`

与 `Query` 类似，唯一的区别是，该函数只返回一条数，就算你 SQL 语句执的结果是多行数据，该函数也只能返回第一条。

下面我们新建一张表，然后进行插入，读取操作。表结构如下：

```
DROP TABLE IF EXISTS `person`;
CREATE TABLE `person` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(255) DEFAULT NULL,
  `age` int(11) DEFAULT NULL,
  `IsBoy` tinyint(4) DEFAULT NULL,
  PRIMARY KEY (`id`)
) DEFAULT CHARSET=utf8;
```

下面是完整的程序。

```
package main
import (
    "database/sql"
    "fmt"
    _ "github.com/go-sql-driver/mysql"
)
func main() {
    db, err := sql.Open("mysql",
"root:123@tcp(127.0.0.1:3306)/test?charset=utf8")
    if err != nil {
        fmt.Println(err)
        return
    }
    defer db.Close()
    var result sql.Result
    //向数据库中插入一条数据
    result, err = db.Exec("insert into person(name, age, IsBoy)
values(?, ?, ?)", "张三", 19, true)
    if err != nil {
        fmt.Println(err)
        return
    }
    lastId, _ := result.LastInsertId()
    fmt.Println("新插入的数据 ID 为", lastId)
    var row *sql.Row
    //返回一行数据
    row = db.QueryRow("select * from person")
    var name string
```

```

var id, age int
var isBoy bool
//取数据进行显示
err = row.Scan(&id, &name, &age, &isBoy)

if err != nil {
    fmt.Println(err)
    return
}
fmt.Println(id, "\t", name, "\t", age, "\t", isBoy)
//再插入一条数据
result, err = db.Exec("insert into person(name, age, IsBoy)
values(?, ?, ?)", "王红", 18, false)
fmt.Println("=====")
var rows *sql.Rows
rows, err = db.Query("select * from person")
if err != nil {
    fmt.Println(err)
    return
}
for rows.Next() {
    var name string
    var id, age int
    var isBoy bool
    rows.Scan(&id, &name, &age, &isBoy)
    fmt.Println(id, "\t", name, "\t", age, "\t", isBoy)
}
rows.Close()
// 最后，清空表
db.Exec("truncate table person")
}

```

注意，Query 返回的 rows，取完数据后需要调用 Close 来释放资源。

- func (db *DB) Prepare(query string) (*Stmt, error)

对 SQL 语句进行预处理，并返回 *Stmt 类型。通情况下我们用 Query,或 Exec 就可以了。但对行重复性的操，比如，循环向数据库中插入 10000 条数据，这时就要用 Prepare 了，可以提高程序的性能。

```

package main
import (
    "database/sql"
    "fmt"
    _ "github.com/go-sql-driver/mysql"

```

```

"math/rand"
"time"
)

func main() {
db, err:=sql.Open("mysql", "root:123@tcp(127.0.0.1:3306)/test?charset=utf8")
    if err != nil {
        fmt.Println(err)
        return
    }
    defer db.Close()
    var smt *sql.Stmt
    smt, err = db.Prepare("insert into person(name, age, IsBoy)
values(?, ?, ?)")
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println("开始插入数据....", time.Now())
    r := rand.New(rand.NewSource(time.Now().UnixNano()))
    for i := 0; i < 10000; i++ {
        _, err = smt.Exec(fmt.Sprintf("张%d", r.Int()), r.Intn(50),
r.Intn(100)%2)
        if err != nil {
            fmt.Println(err)
            return
        }
    }
    fmt.Println("数据插入完成!", time.Now())
}

```

插入一万条，我这里用了 22 分钟，你们可以改成 100，只要知道在什么情况下使用就行了。

9.3 事务

事务是编程中最小的执行单元，它的代码要么全部成功，要么全部失败，不能部分成功，部分失败。如果单元的所有操作都成功，则认为事务成功，即使只有一个操作失败，事务也不成功。如果所有操作完成，提交事务。如果有一个操作失败，回滚事务，该事务所有操作都将取消。

```
func (db *DB) Begin() (*Tx, error)
```

开始一个事务。

```
func (tx *Tx) Commit() error
```

提交事务。

```
func (tx *Tx) Rollback() error
```

用来回滚一个事务。Tx 还有 Query,QueryRow,,Prepare,Exec，这些用法跟 DB 对像的一样。

```
import (
    "database/sql"
    "fmt"
    _ "github.com/go-sql-driver/mysql"
)
func main() {

db, err:=sql.Open("mysql", "root:123@tcp(127.0.0.1:3306)/test?charset=utf8")
    if err != nil {
        fmt.Println(err)
        return
    }
    defer db.Close()
    var trans *sql.Tx
    trans, err = db.Begin()
    if err != nil {
        fmt.Println(err)
        return
    }
    trans.Exec("insertintoperson(name, age, IsBoy) values('    张    三',99,false)")
    trans.Rollback()
}
```

trans.Rollback()回滚事务，insert 插入的数据不会真正写到数据里。改成 trans.Commit() 可以将数据写入到数据库中。通常是根据 SQL 执行的结果，来判断是提交还是回滚，如果执行成功，则提交，如果执行失败，则回滚。所以通常这样写：

```
_, err = trans.Exec("insertintoperson(name, age, IsBoy) values(' 张三',99,false)")
    if err != nil {
        trans.Rollback()
    } else {
        trans.Commit()
    }
}
```

9.4 标准驱动的不足与改进

从前面的例子中我们看到，取数据时只能用 Scan 来按顺序一次来取出所有的数据，这样很不方便。通常情况下我更习惯按列名来取数据。Scan(dest ...interface{}), 从定义可以看出来，Scan 接受的是一个 interface{} 变量，如果我们像他传 interface{} 接口会怎么样呢？

```
package main

import (
    "database/sql"
    "fmt"
    _ "github.com/go-sql-driver/mysql"
)

func main() {
    db, err := sql.Open("mysql", "root:123@tcp(127.0.0.1:3306)/test?charset=utf8")
    if err != nil {
        fmt.Println(err)
        return
    }
    defer db.Close()
    var row *sql.Rows
    row, err = db.Query("select * from person limit 1")
    if row.Next() {
        //返回所有的列名，列的顺序跟 SCan 一至
        fmt.Println(row.Columns())
        var id, name, age, isBoy interface{}
        err = row.Scan(&id, &name, &age, &isBoy)
        if err != nil {
            fmt.Println(err)
        }
        fmt.Println("id", id, string(id.([]byte)))
        fmt.Println("name", name, string(name.([]byte)))
    }
}
```

上面程序的运行结果为：

```
[id name age IsBoy] <nil>
id [49] 1
name [229 188 160 228 184 137] 张三
```

row.Columns() 返回了所有的列，列的顺序，跟我们 SCan 的顺序是一至，这

样看来，列名，可以跟 Scan 里的值一一对应了。按列名返回字段值，应该可以实现。关键是返回的数据，interface{} 里存的到底是什么？里面存的是 []byte 字节。字段 id 的值，为 49，转换成字符串后为 1, 49 就是字符 1 对应的 ASCII 码。问题的关键是，怎么样把他转换成数字呢？我们来看一下 Scan 是怎么实现的。

```
func (rs *Rows) Scan(dest ...interface{}) error {
    .....
    for i, sv := range rs.lastcols {
        err := convertAssign(dest[i], sv)
        if err != nil {
            return fmt.Errorf("sql: Scan error on column index %d: %v", i, err)
        }
    }
    return nil
}
```

Rs.lastcols 是当前行，所有列的值。Dest 是我们传进来的参数。在 2.6 节中我们说过，变参其实就是一个 slice, dest 就是一个 slice, 里面存放的是我们传进来的参数。convertAssign 用来把列中的数据转换成我们需要的类型。该函数位于 go/src/pkg/database/sql/convert.go 文件中。列名可以得到，每一列所对应的数据也可以得到，按列返回数据，应该是可以实现了。我们有两种实现方式，一种是直接改 database/sql 包中的 Rows，给他加一个函数。另一种是在自己创建的程序中进行一层封装，来提供这个功能。我们用后一种实现方，通常情况下，不要改系统文件。下面我们实现一个自己的 Rows 结构体。

```
typeMyRowsstruct{
    *sql.Rows
    valuesmap[string]interface{}
    columnName[]string
}
```

valuesmap 用来存放当前行的数据，key 是列名，区分大小写；value 是列对应的数据。columnName 是列名。接下来我们重写 Next 函数，在这里把每列的数据取出来放到 map 中。

```
func (this *MyRows) Next() bool {
    bResult := this.Rows.Next()
    if bResult {
        //如果成功，取所有列的数据到 values 里
        if this.columnName == nil || len(this.columnName) == 0 {
            this.columnName, _ = this.Rows.Columns()
        }
        if this.values == nil {
            this.values = make(map[string]interface{})
        }
        var arr []interface{}
        for i := 0; i < len(this.columnName); i++ {
```

```

        var inf interface{}
        arr = append(arr, &inf)
    }
    //将数据接收到 interface{} 变量里
    this.Rows.Scan(arr...)

    for i := 0; i < len(this.columnName); i++ {
        this.values[this.columnName[i]] = reflect.ValueOf(arr[i]).Elem().Interface()
    }
}
return bResult
}

```

下面是完整的代码:

```

import (
    "database/sql"
    "errors"
    "reflect"
)
type MyRows struct {
    ...//上面已经有这段代码，此处省略
}
/*根据字段名来取字段的值
name:字段名，区分大小写
value:用来接收字段值的变量，需传变量的地址,如&a
*/
func (this *MyRows) GetValue(name string, value interface{}) error {
    if this.values == nil || len(this.values) == 0 {
        return errors.New("没有调用 Next,或没有可用的行")
    }
    i, ok := this.values[name]
    if ok {
        err := ConvertAssign(value, i)
        if err != nil {
            return err
        }
        return nil
    }
    return errors.New("字段不存在，请注意大小写")
}
func (this *MyRows) Scan(dest ...interface{}) error {

```

```
if this.values == nil || len(this.values) == 0 {  
    return errors.New("没有调用 Next,或没有可用的行")  
}  
  
for i := 0; i < len(dest); i++ {  
    err := ConvertAssign(dest[i], this.values[this.columnName[i]])  
    if err != nil {  
        return err  
    }  
}  
return nil  
}
```

这里同样给出了 Scan 的实现，以增加灵活性，因为该函数可以一次把所有的数据取出来，在需要一次取所有字段的情况下，也是比较方便的。上面的 ConvertAssign 直接复制 go\src\pkg\database\sql\convert.go 的。详细的可以参考 <https://github.com/male110/SimpleDb>

第 10 章 反射

10.1 反射基础

反射是审查元数据并收集关于它的类型信息的能力。元数据就是一大堆的表，当编译程序集或者模块时，编译器会创建一个类定义表，一个字段定义表，和一个方法定义表等。我们可以利用反射获取某一对像类型，值，或成员变量，方法等。

```
func TypeOf(i interface{}) Type
```

返回 i 的类型信息。如果 i 为 nil，返回 nil。返回值类型为 Type。Type 是一个接口定义如下：

```
type Type interface {  
    .....  
}
```

在 Type 接口中定义有以下几个函数，这里只列常用的。

Name() string: 返回类型名称

PkgPath() string: 返回类型所在包的路径。

Kind() Kind: 返回 Type 的类型，是下列值之一。

```
const (  
    Invalid Kind = iota  
    Bool  
    Int  
    Int8  
    Int16  
    Int32  
    Int64  
    Uint  
    Uint8  
    Uint16  
    Uint32  
    Uint64  
    Uintptr  
    Float32  
    Float64  
    Complex64  
    Complex128  
    Array  
    Chan  
    Func  
    Interface  
    Map  
    Ptr
```

```
Slice
String
Struct
UnsafePointer
)
```

NumMethod() int: 返回类型的函数个数。

Method(n int) Method: 返回类型的第 n 个函数。

MethodByName(string) (Method, bool): 根据名称返回 Type 的指定函数。

NumOut() int: 返回函数类型的返回值的数量，如果不是函数类型，将会产生一个错误。

Out(i int) Type: 返回函数类型的第 i 个返回值，如果不是函数类型，将产生一个错误。

NumIn() int: 返回函数类型的输入参数数量。

In(i int) Type: 返回函数类型的第 i 个输入参数，如果不是函数类型将产生一个错误。

Elem() Type: 通常在我们反射的对像是指针类型时，使用该函数返回指针所指向的对像的类型。

NumField() int: 返回结构体类型的字段数量。

Field(i int) StructField: 返回结构体类型的第 i 个字段。

FieldByName(name string) (StructField, bool): 按名称返回结构体的字段

下面我们看一个列子，用反射来取结构体的字段，和函数：

```
import (
    "fmt"
    "reflect"
)
type Student struct {
    Name string
    Age  int
}
func (this *Student) PrintName() {
    fmt.Println(this.Name)
}
func (this *Student) GetAge() int {
    return this.Age
}
func main() {
    s := Student{Name: "abc", Age: 19}
    rt := reflect.TypeOf(s)
    //判断是否指针类型，如果是，取指针所指向的元素的类型
    if rt.Kind() == reflect.Ptr {
        rt = rt.Elem()
    }
    //输出类型所在的包的路径
    fmt.Println(rt.PkgPath())
}
```

```

//反射取所有字段
fmt.Println(rt.Name(), "共有", rt.NumField(), "个字段")
for i, j := 0, rt.NumField(); i < j; i++ {
    rtField := rt.Field(i)
    fmt.Println(rtField.Name)
}
/*因为我们的函数定义是在*Student 类型上的，所以这里转换为指针类型，否则反射会取不到函数*/
rt = reflect.PtrTo(rt)
//反射取所有函数
fmt.Println(rt.Name(), "共有", rt.NumMethod(), "个函数")
for i, j := 0, rt.NumMethod(); i < j; i++ {
    mt :=rt.Method(i)
    fmt.Println(mt.Name)
    //输入参数的数量
    numIn := mt.Type.NumIn()
    //输出参数的数量
    numOut := mt.Type.NumOut()
    //输出输入参数
    if numIn > 0 {
        fmt.Println("\t 共", numIn, "个输入参数")
        for k := 0; k < numIn; k++ {
            in := mt.Type.In(k)
            fmt.Println("\t", in.Name(), "\t", in.Kind())
        }
    }
    //输出输出参数
    if numOut > 0 {
        fmt.Println("\t 共", numOut, "个输出参数")
        for k := 0; k < numOut; k++ {
            out := mt.Type.Out(k)
            fmt.Println("\t", out.Name(), "\t", out.Kind())
        }
    }
}
}

```

TypeOf 只能取到字段名，字段类型，取不到字段值；要取字段值，需要用 ValueOf。

```

import (
    "fmt"
    "reflect"
)
type Student struct {
    Name string
    Age  int
}
func main() {
    s := Student{Name: "abc", Age: 19}
    rv := reflect.ValueOf(s)
    //判断是否指针类型，如果是，取指针所指向的元素的类型
    if rv.Kind() == reflect.Ptr {
        rv = rv.Elem()
    }
    rvField := rv.FieldByName("Name")//取 Name 字段的值
    fmt.Println(rvField.String())
}

```

因为我们知道，Name 是字符串所以可以用 String()来取字符串值，如果 rvField 不是字符串将会报错。除了 String 还有 Bool(), Bytes(), Int(), Float() 等函数来取反射变量的值。在我们不知道类型时，可以用 Interface() 来取值。

SetString(), SetBool(), SetInt() 等用来设置反射对像的值。反射时必须是对指针进行反射，因为值类型的参数，在函数内被改变时不会改外函数外的值。

```

import (
    "fmt"
    "reflect"
)
type Student struct {
    Name string
    Age  int
}
func main() {
    s := Student{Name: "abc", Age: 19}
    //这里传的是&s 因为要修改字段的地址，否在会报错。
    rv := reflect.ValueOf(&s)
    //判断是否指针类型，如果是，取指针所指向的元素的类型
    if rv.Kind() == reflect.Ptr {
        rv = rv.Elem()
    }
    //取 Name 字段的值
    rvField := rv.FieldByName("Name")
    fmt.Println(rvField.String())
}

```

```
rvField.SetString("已改名")
    fmt.Println(s.Name) //输出已改名
}
```

10.2 反射调用函数

.TypeOf, ValueOf 都可以对函数进行调用, 区别在于, 使用 TypeOf 时, 函数的第一个参数是结构体本身, 需要把结构体自身做为输入参数传递, 而 ValueOf 不需要这样。

```
import (
    "fmt"
    "reflect"
)
type Student struct {
    Name string
    Age  int
}
func (this *Student) PrintName() {
    fmt.Println(this.Name)
}
func (this *Student) GetAge() int {
    return this.Age
}
func main() {
    s := Student{Name: "abc", Age: 19}
    rt := reflect.TypeOf(&s)
    rv := reflect.ValueOf(&s)
    fmt.Println("typeof 调用函数")
    rtm, ok := rt.MethodByName("PrintName")
    if ok {
        var parm []reflect.Value
        //函数默认第一个参数是结构体本身即*Student
        parm = append(parm, rv)
        rtm.Func.Call(parm)
    }
    //valueof 调用函数
    fmt.Println("valueof 调用函数")
    rvm := rv.MethodByName("GetAge")
    //用 valueof 调用函数时不需要把 Struct 本身做为参数传递过去
    ret := rvm.Call(nil)
    //显示返回值
    fmt.Println("返回值")
}
```

```

    ShowSlice(ret)
}
func ShowSlice(s []reflect.Value) {
    if s != nil && len(s) > 0 {
        for _, v := range s {
            fmt.Println(v.Interface())
        }
    }
}

```

10.3 反射取 Struct 的 Tag 信息

在 Go 还可以给结构体的字段设置一些附加信息，可以在对结构体进行反射时取出这些附加信息，来使用。如我们在 xml 序列化一章中定义的一个结构体

```

type Student struct {
    XMLName xml.Name `xml:"student"`
    Name     string  `xml:"name"`
    Age      int     `xml:"age"`
}

```

在字段名后面的字符串就是字段的 Tag 信息，在 XML 序列化时，根据这些 Tag 信息来对应到 XML 标签。下面我们看一下如何来取 Tag 值。

```

package main
import (
    "fmt"
    "reflect"
)
type Student struct {
    Name string "学生姓名"
    Age  int   `a:"1111"b:"3333"` //这个不是单引号，而是~键上的符号
}
func main() {
    s := Student{}
    rt := reflect.TypeOf(s)
    fieldName, ok := rt.FieldByName("Name")
    //取 tag 数据
    if ok {
        fmt.Println(fieldName.Tag)
    }
    fieldAge, ok2 := rt.FieldByName("Age")
}

```

/*可以你 JSON 一样，取 TAG 里的数据，注意，设置时，两个之间无逗号, 键名无引号*/

```
if ok2 {  
    fmt.Println(fieldAge.Tag.Get("a"))  
    fmt.Println(fieldAge.Tag.Get("b"))  
}  
}
```

第 11 章 实现一个自己的 ORM

11.1 实现自己的 ORM

Orm 是什么？这个可以百度一下，简单来说，就是对数据库操作的一个抽象。使用 ORM，可以让数据库操作更加简单，方便。对 ORM 的态度，有的人喜欢，有的人不喜欢，个人感觉看个人习惯了。我一般使用 ORM 时，只使用他的 Insert, Update, Delete, Load 等几个方法，理由是可以少写代码。对于其它的操作多数我还是写 SQL 的。ORM 虽然简单，但需要花时间去学习 ORM 的用法，而且，有时程序出错，你还很难查出是什么错误，不如 SQL 来得直接。

在这节，我们将实现一个简单的 ORM，只实现 Insert, Update, Delete, Load 几个方法。通常向数据库插入数据时，只要 Insert(model)就可以了，不需要写 SQL 代码。model 是 Struct 结构体，在 Insert 的内部，利用反射，来取结构体的名称做表名，结构体的字段名做为数据表的字段名，结构体字段的值，做为数据表的字段值。然后拼 SQL 语句，执行 SQL 语句进行插入操作，然后取自动增长 ID，赋给结构体标记为自增的字段。

但有时，我们的结构体名称不一定是数据库名，结构体的字段名，也不一定跟数据表的字段名一至。这时怎么处理呢？还记得 XML 序列化的那节吗？当 XML 结点名称，跟结构体字段名不一至时可以使用 tag 来设置一些附加信息。当结构体的字段名称，跟数据表的字段名称不一至时，我们也用同样的方式解决。我们定义一个 TableName 类型，用该字段的 tag 来设置表名。就像下面这样来定义一个结构体，来跟数据表相对应。

```
type Person struct {
    /*TableName 类型只是用来设置表名。如果结构体名跟表名相同可以省略*/
    TableName SimpleDb.TableName "person"

    /*name 是表名,PK 用来设置是否主键，true 主键，false 非主键*/
    Id int `name:"id"PK:"true"Auto:"true"`

    Name    string "name" //tag 里的 name 表是对应的字段名
    Age     int    "age"  //tag 里的 age 表是对应的字段名
    IsBoy   bool
    NotUse  string "-" //不会保存到数据库中
}
```

上面的说明已经很详细了， SimpleDb.TableName 类型的字段，只用来在 tag 中定义结构体对应的表名，如果没有该字段，认为表名跟结构体名相同。PK:"true"表示是主键，Auto:"true"表示该字段是自动增长的列，name:"id",来指定该字段对应的数据表中的字段名，如不指定认为跟结构体字段名相同。当只需要指定数据表字段名时，可以直接写在 tag 中，如："name"、"age"。tag 为 "-" 表示不对应数据表中的任何列。

在规定了上面的规则后，如何来实现 Insert 呢？首先用反射来取得表名，字段名，然后拼 Insert 的 SQL 语句。一张数据表，有表名，和字段组成，一个字段包括字段名，字段类型，是否主键，是否自动增长等信息。所以我们定义下面两个结构体来存放反射取得的数据

表信息。

```
type TableInfo struct {
    Name    string    //表名
    Fields []FieldInfo //字段
}

type FieldInfo struct {
    Name          string //字段名
    IsPrimaryKey  bool   //是否主键
    IsAutoGenerate bool   //是否自动生成(增长)
    Value         reflect.Value
}
```

我们定义一个 TableName 类型，该类型的 tag 信息用来存放表名，字段本身没有意义。所以下面我们定义一个 TableName 类型。

```
type TableName string
var typeTableName TableName
var tableNameType reflect.Type = reflect.TypeOf(typeTableName)
```

接下来我们要先用反射，来取结构体对应的数据表的信息。

```
func getTableInfo(model interface{}) (tbinfo *TableInfo, err error) {
    {
        defer func() {
            if e := recover(); e != nil {
                tbinfo = nil
                err = e.(error)
            }
        }()
        err = nil
        tbinfo = &TableInfo{}
        rt := reflect.TypeOf(model)
        rv := reflect.ValueOf(model)
        //默认是结构体名
        tbinfo.Name = rt.Name()
        if rt.Kind() == reflect.Ptr {
            rt = rt.Elem()
            rv = rv.Elem()
        }
        for i, j := 0, rt.NumField(); i < j; i++ {
            rtf := rt.Field(i)
            rvf := rv.Field(i)
```

```

//tableNameType 类型，只是用该字段的 tag 来设置表名
if rtf.Type == tableNameType {
    tbinfo.Name = string(rtf.Tag)
    continue
}
if rtf.Tag == "-" {
    continue
}
//如果字段没有 tag
var f FieldInfo
if rtf.Tag == "" {
    f = FieldInfo{Name: rtf.Name, IsPrimaryKey: false,
IsAutoGenerate: false, Value: rvf}
} else {
    //判断 tag 中有没有:有的话，说明设置了主键自增等参数，否则认为 tag 中存的是数据库的字段名
    strTag := string(rtf.Tag)
    if strings.Index(strTag, ":") == -1 {
        f = FieldInfo{Name: strings.TrimSpace(strTag), IsPrimaryKey:
false, IsAutoGenerate: false, Value: rvf}
    } else {
        //取字段名
        strName := rtf.Tag.Get("name")
        if strName == "" {
            strName = rtf.Name
        }
        //取主键
        isPk := false
        str := rtf.Tag.Get("PK")
        if str == "true" {
            isPk = true
        }
        str = rtf.Tag.Get("Auto")
        //获取是否自增的值
        isAuto := false
        if str == "true" {
            isAuto = true
        }
        f = FieldInfo{Name: strName, IsPrimaryKey: isPk,
IsAutoGenerate: isAuto, Value: rvf}
    }
}
}

```

```

        tbinfo.Fields = append(tbinfo.Fields, f)
    }
    return
}

```

下一步，我们要用取得的 TableInfo 信息，来生成 Insert Sql 语句。

```

//生成插入的 SQL 语句，和对应的参数
func generateInsertSql(model interface{}) (string, []interface{},
*TableInfo, error) {
    tbinfo, err := getTableInfo(model)
    if err != nil {
        return "", nil, nil, err
    }
    //如果结构体中没有字段，抛出异常
    if len(tbinfo.Fields) == 0 {
        return "", nil, nil, errors.New(tbinfo.Name + "结构体中没有
字段")
    }
    strSql := "insert into " + tbinfo.Name
    strField := ""
    strValue := ""
    var param []interface{}
    for _, v := range tbinfo.Fields {
        if v.IsAutoGenerate { //跳过自动增长的自段
            continue
        }
        strField += v.Name + ","
        strValue += "?,"
        param = append(param, v.Value.Interface())
    }
    if strField == "" {
        return "", nil, nil, errors.New(tbinfo.Name + "结构体中没有
字段,或只有自增自段")
    }
    strField = strings.TrimRight(strField, ",")
    strValue = strings.TrimRight(strValue, ",")
    strSql += " (" + strField + ") values(" + strValue + ")"
    return strSql, param, tbinfo, nil
}

```

在数据插入到数据表后，如果有自动增长列，会返回最后自增的值，我们要把这个数值赋给结构体的字段。所以还需要一个设置自增字段值的函数。

```

//将插入时得到自动增长的 ID 赋值给 Struct 对应字段
func setAuto(result sql.Result, tabinfo *TableInfo) (err error) {
    defer func() {
        if e := recover(); e != nil {
            err = e.(error)
        }
    }()
    id, err := result.LastInsertId()
    if id == 0 {
        return
    }
    if err != nil {
        return
    }
    for _, v := range tabinfo.Fields {
        if v.IsAutoGenerate {
            v.Value.SetInt(id)
            break
        }
    }
    return
}

```

11.2 Insert 函数的实现及所有源码

这个 ORM 是按我的使用习惯实现的，只实现了 Insert, Update, Delete 几个方法，读者可以参考其它的 ORM 框架，根据自己的使用习惯进行修改，下面是 Insert 函数的实现。基本原理就是反射取各字段名称和值，拼接 SQL 语句，执行 SQL 语句，最后判断是否有自动增长的字段，有的话设置自动增长字段的值。

```

/*将一个 Struct 结构体插入到数据库, 如有自增自段,
会把自增值赋给 model 中的对应字段, model 必须是可修改的, 即传地址如,
&m*/
func (this *MyDb) Insert(model interface{}) error {
    strSql, param, tabinfo, err := generateInsertSql(model)
    if err != nil {
        return err
    }
    var result sql.Result
    result, err = this.Exec(strSql, param...)
}

```

```
        if err != nil {  
            return err  
        }  
        setAuto(result, tabinfo)  
        return nil  
    }
```

Update, Delete 的实现，基本上类似，完整的代码可以参考这里：

<https://github.com/male110/SimpleDb>

第 12 章 TCP 与 UDP 网络编程

12.1 TCP 编程

TCP 即传输控制协议/网间协议，是一种面向连接（连接导向）的、可靠的、基于字节流的一个端到端（Peer-to-Peer）的传输层协议，在实际中应用十分广泛。如文件传送协议 FTP、网络终端协议 TELNET、SMTP、POP3、HTTP 协议等。Go 的 net 包，提供了对 Tcp 操作的支持。下面我们看一下相关的一些函数

- `func InterfaceAddrs() ([]Addr, error)`

返回本机的网络地址列表。

```
package main
import (
    "fmt"
    "net"
)
func main() {
    addr, err := net.InterfaceAddrs()

    if err != nil {
        fmt.Println(err)
    }
    fmt.Println(addr)
}
```

我运行的输出结果是 192.168.2.2。

- `func LookupIP(host string) (addr []IP, err error)`

用来获取主机所对应的 IP 地址。IP 是一个 []byte 类型，用来表示一个 IP 地址，定义如下：

`type IP []byte`

```
import (
    "fmt"
    "net"
)
func main() {
    ips, err := net.LookupIP("www.baidu.com")
    if err != nil {
        fmt.Println(err)
    }
    fmt.Println(ips)
}
```

程序的运行结果为: [115. 239. 210. 26 115. 239. 210. 27], 即 www.baidu.com 所对应的 IP 地址。

- **func ResolveTCPAddr(net, addr string) (*TCPAddr, os.Error)**

该函数用来创建一个 TCPAddr, 第一个参数为 tcp, tcp4 或者 tcp6, addr 是一个字符串, 由主机名或 IP 地址, 以及 ":" 后跟随着端口号组成, 例如: "www.baidu.com:80" 或 '127.0.0.1:8080'。如果地址是一个 IPv6 地址, 由于已经有冒号, 主机部分, 必须放在方括号内, 例如: "[::1]:8080"。

TCPAddr 包含一个 IP 和 Port 端口号, 定义如下。

```
type TCPAddr struct {
    IP    IP
    Port  int
}
```

下面是一个 ResolveTCPAddr 的例子:

```
import (
    "fmt"
    "net"
)
func main() {
    ip, err := net.ResolveTCPAddr("tcp", "www.baidu.com:80")
    if err != nil {
        fmt.Println(err)
    }
    fmt.Println(ip)
}
```

- **func ListenTCP(net string, laddr *TCPAddr) (*TCPLListener, error)**

TCP 程序分为服务端和客户端。服务端程序在某一端口监听客户端的连接请求, 有客户端的连接请求时, 读取客户端发来的数据, 进行相关的处理, 然后关闭链接。ListenTCP 函数就是在指定的端口监听, 等待客户端的连接。

- **func (l *TCPLListener) AcceptTCP() (*TCPConn, error)**

用来接受客户端的请求, 返回一个 Conn 链接, 通过这个 Conn 来与客户端的进行通信。

- **func (l *TCPLListener) Accept() (Conn, error)**

与 AcceptTCP 相同, 这两个方法, 用哪儿一个都可以。我们看一下他的实现。

```
// Accept implements the Accept method in the Listener interface; it
// waits for the next call and returns a generic Conn.
func (l *TCPLListener) Accept() (Conn, error) {
    c, err := l.AcceptTCP()
    if err != nil {
        return nil, err
    }
}
```

```
}  
    return c, nil  
}
```

Accept 最终是调用了 AcceptTCP，所以这两个函数是一样的效果。

- `func (c *TCPConn) Write(b []byte) (int, error)`
向 TCPConn 网络链接发送数据，b 是要发送的内容，返回值 int 为实际发送的字节数。
- `func (c *TCPConn) Read(b []byte) (int, error)`
从 TCPConn 网络链接接收数据。返回值 int 是实际接收的字节数。b 是接收的数据。
- `func DialTCP(net string, laddr, raddr *TCPAddr) (*TCPConn, error)`
用来链接远程服务器。net 可以是 tcp、tcp4、tcp6 中的一个。laddr 为本地地址，通常传 null，raddr 是要链接的远端服务器的地址。成功返回 TCPConn，用返回的 TCPConn 可以向服务器发送消息，读取服务器的响应信息。

12.2 TCP 编程实战

下面我们实现一个简单的程序，客户端向服务端发送 ls 列出当前目录下的文件，发送 cd 命令来改变当前目录。服务端收到客户端的命令后，进行相关的处理，并将结果发送给客户端。

服务端：

```
package main  
import (  
    "bytes"  
    "fmt"  
    "io/ioutil"  
    "net"  
    "os"  
)  
const (  
    LS = "LS"  
    CD = "CD"  
    PWD = "PWD"  
)  
func main() {  
    //在7070端口监听  
    tcpAddr, err := net.ResolveTCPAddr("tcp", ":7076")  
    checkError(err)  
    listener, err1 := net.ListenTCP("tcp", tcpAddr)  
    checkError(err1)  
    for {  
        //等待客户端的连接  
        conn, err2 := listener.Accept()  
    }
```



```

        if err != nil {
            /*通常服务端为一个服务，不会因为错误而退出。出错后，继续
            等待下一个连接请求*/
            fmt.Println(err2)
            continue
        }
        fmt.Println("收到客户端的请求")
        go ServeClient(conn)
    }
}

func ServeClient(conn net.Conn) {
    defer conn.Close()
    str := ReadData(conn)
    if str == "" {
        SendData(conn, "接收数据时出错")
        return
    }
    fmt.Println("收到命令：", str)
    switch str {
    case LS:
        ListDir(conn)
    case PWD:
        Pwd(conn)
    default:
        if str[0:2] == CD {
            Chdir(conn, str[3:])
        } else {
            SendData(conn, "命令错误")
        }
    }
}

func Chdir(conn net.Conn, s string) {
    err := os.Chdir(s)
    if err != nil {
        SendData(conn, err.Error())
    } else {
        SendData(conn, "OK")
    }
}

func ListDir(conn net.Conn) {
    files, err := ioutil.ReadDir(".")
    if err != nil {

```

```

        SendData(conn, err.Error())
        return
    }
    var str string
    for i, j := 0, len(files); i < j; i++ {
        f := files[i]
        str += f.Name() + "\t"
        if f.IsDir() {
            str += "dir\r\n"
        } else {
            str += "file\r\n"
        }
    }
    SendData(conn, str)
}

/*读取数据*/
func ReadData(conn net.Conn) string {
    var data bytes.Buffer
    var buf [512]byte
    for {
        n, err := conn.Read(buf[0:])
        if err != nil {
            fmt.Println(err)
            return ""
        }
        //我们的数据以0做为结束的标志
        if buf[n-1] == 0 {
            //n-1去掉结束标记0
            data.Write(buf[0 : n-1])
            break
        } else {
            data.Write(buf[0:n])
        }
    }
    return string(data.Bytes())
}

func SendData(conn net.Conn, data string) {
    buf := []byte(data)
    /*向 byte 字节里添加结束标记*/
    buf = append(buf, 0)
    _, err := conn.Write(buf)
    if err != nil {

```

```

        fmt.Println(err)
    }
}

func Pwd(conn net.Conn) {
    s, err := os.Getwd()
    if err != nil {
        SendData(conn, err.Error())
    } else {
        SendData(conn, s)
    }
}

func checkError(err error) {
    if err != nil {
        fmt.Println(err)
        os.Exit(0)
    }
}

```

在客户端，我们重用服务端的 `SendData,ReadData` 来收发数据，客户端的代码如下：

```

package main

import (
    "bufio"
    "bytes"
    "fmt"
    "net"
    "os"
    "strings"
)

const (
    LS   = "LS"
    CD   = "CD"
    PWD  = "PWD"
    QUIT = "QUIT"
)

func main() {
    reader := bufio.NewReader(os.Stdin)
    for {
        fmt.Print("请输入命令:")
        line, err := reader.ReadString('\n')
    }
}

```

```

        checkError(err)
        //去掉两端的空格
        line = strings.TrimSpace(line)
        //统一转换成大写字母
        line = strings.ToUpper(line)
        arr := strings.SplitN(line, " ", 2)
        fmt.Println(arr)
        switch arr[0] {
        case LS:
            SendRequest(LS)
        case CD:
            SendRequest(CD + " " + strings.TrimSpace(arr[1]))
        case PWD:
            SendRequest(PWD)
        case QUIT:
            fmt.Println("程序退出")
            return
        default:
            fmt.Println("命令错误！")
        }
    }
}

//发送请求
func SendRequest(cmd string) {
    tcpAddr, err := net.ResolveTCPAddr("tcp", "127.0.0.1:7076")
    checkError(err)
    conn, err := net.DialTCP("tcp", nil, tcpAddr)
    checkError(err)
    SendData(conn, cmd)
    fmt.Println(ReadData(conn))
    conn.Close()
}

/*读取数据*/
func ReadData(conn net.Conn) string {
    var data bytes.Buffer
    var buf [512]byte
    for {
        n, err := conn.Read(buf[0:])
        if err != nil {
            fmt.Println(err)
            return ""
        }
    }
    //我们的数据以0做为结束的标志

```

```

        if buf[n-1] == 0 {
            //n-1去掉结束标记0
            data.Write(buf[0 : n-1])
            break
        } else {
            data.Write(buf[0:n])
        }
    }
    return string(data.Bytes())
}

func SendData(conn net.Conn, data string) {
    buf := []byte(data)
    /*向 byte 字节里添加结束标记*/
    buf = append(buf, 0)
    _, err := conn.Write(buf)
    if err != nil {
        fmt.Println(err)
    }
}

func checkError(err error) {
    if err != nil {
        fmt.Println(err)
        os.Exit(0)
    }
}

```

TCP 协议需要通信双方约定数据的传输格式，否则接收方无法判断数据是否接收完成。在上面的例子中，SendData 用来发送数据，在数据发送前，添加结束标记，buf = append(buf, 0)，用来表示数据发送结束。接收方收到 0 说明数据接收完成。要么接收无法判断数据是否接收完。

如果发送端，发送数据后，调用 Close 关闭连接，不等待服务端的返回数据，服务端可以用 ioutil.ReadAll 来读取数据，这时可以判断出 EOF，读取结束。但如果客户端发送数据后，没有关闭，而是等待服务端的数据返回，用 ReadAll 是不行的。所以在上面的例子中，用 0 来表示数据的发送完成。

12.3 UDP 网络编程

UDP 是用户数据报协议（User Datagram Protocol，UDP）的简称，UDP 协议提供的是面向无连接的、不可靠的数据报投递服务。当使用 UDP 协议传输信息流时，用户应用程序必须负责解决数据报丢失、重复、排序，差错确认等问题。因为 UDP 不是面向链接的，所以资源消耗小，处理速度快，通常音频、视频和普通数据在传送时使用 UDP 较多。如 QQ 使用的就是 UDP 协议。UDP 适用于一次只传少量数据的环境，数据报的最大长度根据操作环

境的不同而各异。从理论上说，包含报头在内的数据报的最大长度为 65535 字节。不过，一些实际应用往往会限制数据报的大小，有时会降低到 8192 字节。

UDP 函数跟 TCP 的很像。

- `func ResolveUDPAddr(net, addr string) (*UDPAddr, error)`
把 `addr` 地址字符串，解析成 `UDPAddr` 地址。`net` 可以是`"udp"`、`"udp4"`、`"udp6"`，`addr` 是一个地址字符串，由主机名或 IP 地址，以及 `“:”` 后面跟着的端口号组成。如果是 IPv6，主机部分必须在方括内，如`[::1]8080`
- `func ListenUDP(net string, laddr *UDPAddr) (*UDPCConn, error)`
在指定的地址(`laddr`)监听，等待 UDP 数据包的到达。返回`*UDPCConn`，可以使用连接的 `ReadFrom` 函数来读取 UDP 数据，用 `WriteTo` 来向客户端发送数据。
- `func (c *UDPCConn) ReadFrom(b []byte) (int, Addr, error)`
服务端用来读取 UDP 数据。`Addr` 是发送方的地址。
- `func (c *UDPCConn) WriteTo(b []byte, addr Addr) (int, error)`
向 `addr` 发数据时用。`b` 是要发送的数据，`addr` 是接收方的地址。
- `func DialUDP(net string, laddr, raddr *UDPAddr) (*UDPCConn, error)`
连接到远端服务器 `raddr`，`net` 参数必须是`"udp"`、`"udp4"`、`"udp6"`中的一个。`Laddr` 通常为 `nil`，如果不是 `nil` 将使用 `laddr` 来连接到服务端。
- `func (c *UDPCConn) Write(b []byte) (int, error)`
用来向服务端发送数据。
- `func (c *UDPCConn) ReadFromUDP(b []byte) (n int, addr *UDPAddr, err error)`
与 `ReadFrom` 相同，用来读取发来 UDP 数据。

12.4 UDP 编程实战

下面来看一个简单的例子，客户端向服务端发送一条数据，服务端收到数据后，给客户端一个响应，告诉他，数据已到。

服务端：

```
import (
    "fmt"
    "net"
)
func main() {
    //在7070端口监听
    addr, err := net.ResolveUDPAddr("udp", ":7070")
    if err != nil {
        fmt.Println(err)
        return
    }
    conn, err := net.ListenUDP("udp", addr)

    if err != nil {
        fmt.Println(err)
        return
    }
```

```

    }
    for {
        var buf [1024]byte
        n, addr, err := conn.ReadFromUDP(buf[0:])
        if err != nil {
            fmt.Println(err)
            return
        }
        go HandleClient(conn, buf[0:n], addr)
    }
}

func HandleClient(conn *net.UDPConn, data []byte, addr *net.UDPAddr)
{
    fmt.Println("收到数据: " + string(data))
    conn.WriteToUDP([]byte("OK, 数据已到"), addr)
}

```

HandleClient 函数的{, 是在)后面的){, 跟函数名在同一行, 不是两行。

客户端:

```

import (
    "fmt"
    "net"
)

func main() {
    addr, err := net.ResolveUDPAddr("udp", "127.0.0.1:7070")
    if err != nil {
        fmt.Println(err)
        return
    }
    conn, err := net.DialUDP("udp", nil, addr)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer conn.Close()
    conn.Write([]byte("Hello Server"))
    var buf [1024]byte
    n, _, err := conn.ReadFromUDP(buf[0:])

    if err != nil {
        fmt.Println(err)
        return
    }
}

```

```
    fmt.Println(string(buf[0:n]))  
}
```

在前面 TCP 的示例里，我们有约定，一条消息的结束标记。在 UDP 里，不需要有约定结束标记，但需要约定，UDP 报文的最大长度。UDP 的数据，必须一次接收完成。比如上面的例子中，我们用，`var buf [1024]byte`，定义了 1KB 的缓冲区来接收数据。因为我们发送的数据量不大，所以可以一次读取。如果我们把上面的缓冲区改两个字节 `var buf [2]byte` 会怎么样？大家可以自己试一下，会报一个错误：

WSARecvFrom udp 0.0.0.0:7070: More data is available.

所以，UDP 通信的双方需要约定报文的最大长度。

第十三章 WEB 编程

13.1 第一个 WEB 程序

Go 可以用来开发 WEB 应用程序,但跟传统的 PHP 有些区别。Go 内置实现了一个 WEB 服务, Net/http 包提供了相应的实现。通常 Go WEB 程序以反向带理的方式发布。下面是两个基本的函数。

- `func HandleFunc(pattern string, handler func(ResponseWriter, *Request))`
用来注册 http 路由的处理函数, pattern 是 http 的地址, handler 是对应的处理函数。
- `func ListenAndServe(addr string, handler Handler) error`
在指定端口监听 HTTP 请求, 并阻塞程序, 直到退出。

下面我拉看一个例子:

```
package main

import (
    "net/http"
)

func main() {
    http.HandleFunc("/test", HandleRequest)
    http.ListenAndServe(":8888", nil)
}

func HandleRequest(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("<h1>第一个 WEB 应用</h1>"))
    w.Write([]byte(r.URL.Path))
}
```

编译并运行程序, 然后打开浏览器在地址栏中输入: `http://localhost:8888/test`, 结果如下图:



ListenAndServe(addr string, handler Handler) 函数第二个参数，Handler 其实是一个接口，定义如下。

```
type Handler interface {  
    ServeHTTP(ResponseWriter, *Request)  
}
```

第二个参数，可以传一个实现了 ServeHttp 接口的类(结构体)。

```
import (  
    "net/http"  
)  
func main() {  
    hadler := &HttpHandler{}  
    http.ListenAndServe(":8888", hadler)  
}  
type HttpHandler struct {  
}  
func (this *HttpHandler) ServeHTTP(w http.ResponseWriter, r  
*http.Request) {  
    w.Write([]byte("<h1>在 ServeHTTP 里</h1>"))  
    w.Write([]byte(r.URL.Path))  
}
```

13.2 URL 参数与 Form 表单处理

http.Request.URL.Query() 可以获取地址栏中的参数，返回 Values 类型，即 map[string][]string，比如在地址栏中，先设置 a=1，再设置 a=10，这两个值将按先后顺序存到 string 数组中，即 map["a"]=[]string{"1","2"}。

```

package main
import (
    "fmt"
    "net/http"
)
func main() {
    http.HandleFunc("/test", HandleRequest)
    http.ListenAndServe(":8888", nil)
}
func HandleRequest(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("<h1>URL 参数</h1>"))
    w.Write([]byte(fmt.Sprintf("%v", r.URL.Query())))
}

```

运行上面的程序，在地址栏中输入 `http://localhost:8888/test?a=1&a=10&b=2&c=3`，会看到输出结果为 `map[a:[1 10] b:[2] c:[3]]`。Request 的 ParseForm 函数可以对 URL 参数和表单进行处理。

- `func (r *Request) ParseForm() error`

解析 URL 请求的参数并更新 `r.Form` (`r.Form` 中存放 URL 传进来的参数和 Post 过来的数据)，对于 Post、Put 请求，还将解析 POST 的内容，将结果放到 `r.PostForm` 和 `r.Form` 中。`r.Form` 中，Post 的内容比 URL 的参数优先级高，如果 Post 和 URL 参数有相同的，将会被 Post 的值覆盖。如果没用 `MaxBytesReader` 来设置 Post 内容的大小，默认为 10MB。

`r.Form` 中存放了 URL 的参数值，和 Post 的 Form 值。

`r.PostForm` 只存放了 Post 的 Form 值。

`r.Form`、`r.PostForm` 都是 `url.Values` 类型，前面提到 `url.Values` 是 `map[string][]string` 类型。

```

package main
import (
    "fmt"
    "net/http"
)

func main() {
    http.HandleFunc("/test", HandleRequest)
    http.ListenAndServe(":8888", nil)
}

func HandleRequest(w http.ResponseWriter, r *http.Request) {
    w.Header().Add("Content-Type", "text/html; charset=utf-8")
    if r.Method == "POST" {
        r.ParseForm()
        /*username 有两个值，默认取的是第一个的*/
        w.Write([]byte("用户名: " + r.FormValue("username") + "<br/>"))
        w.Write([]byte("<hr/>"))
    }
}

```

```

        names := r.Form["username"]
        w.Write([]byte("username 有两个: " + fmt.Sprintf("%v", names)))
        w.Write([]byte("<hr/>r.Form 的内容: " + fmt.Sprintf("%v",
r.Form)))
        w.Write([]byte("<hr/>r.PostForm 的内容: " + fmt.Sprintf("%v",
r.Form)))

        //r.Form
    } else {
        strBody := `<form action=` + r.URL.RequestURI() + `
method="post">
用户名: <input name="username" type="text" /><br />
用户名: <input name="username" type="text" /><br />
<input id="Submit1" type="submit" value="submit" />
</form>`
        w.Write([]byte(strBody))
        r.ParseForm()
    }
}

```

在上面的例子中 `r.FormValue("username")` 得到的是第一个 `username` 的值。

12.3 文件上传

Go 的文件上传处理十分方便, `Request.FormFile` 返回一个 `multipart.File` 对像, 可以直接读取文件内容, 并保存。其定义如下:

```
func (r *Request) FormFile(key string) (multipart.File, *multipart.FileHeader, error)
```

`multipart.File` 是一个接口, 继承了 `io.Reader` 接口, 可以通过该接口读取上传文件的内容。其定义如下:

```

type File interface {
    io.Reader
    io.ReaderAt
    io.Seeker
    io.Closer
}

```

`multipart.FileHeader` 是一个结构体, 可以通过该结构体取到上传文件的名称, 文件类型, 其定义如下:

```

type FileHeader struct {
    Filename string
    Header   textproto.MIMEHeader
    // contains filtered or unexported fields
}

```

FileHeader.Header["Content-Type"]可以获取上传文件的 MIME 类型。下面是一个文件上传的例子。

```

package main

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

func HelloServer(w http.ResponseWriter, r *http.Request) {
    if "POST" == r.Method {
        file, handler, err := r.FormFile("file")
        if err != nil {
            http.Error(w, err.Error(), 500)
            return
        }
        fmt.Println(handler.Header)
        defer file.Close()
        f, err := os.OpenFile("./"+handler.Filename,
os.O_WRONLY|os.O_CREATE, os.ModePerm)
        if err != nil {
            fmt.Println(err)
            return
        }
        defer f.Close()
        size, err := io.Copy(f, file)
        if err != nil {
            fmt.Println(err)
            return
        }
        fmt.Fprintf(w, "上传文件的大小为: %d", size)
        return
    }
}

```

```

    // 上传页面
    w.Header().Add("Content-Type", "text/html")
    w.WriteHeader(200)
    html := `
<form enctype="multipart/form-data" action="/" method="POST">
    请选择要上传的文件: <input name="file" type="file" /><br/>
    <input type="submit" value="Upload File" />
</form>
`

    io.WriteString(w, html)
}

func main() {
    http.HandleFunc("/", HelloServer)
    err := http.ListenAndServe(":12345", nil)
    if err != nil {
        fmt.Println(err)
    }
}

```

12.4 HTML 模板处理

什么是 HTML 模板，简单来简就是用来展示动态 HTML 页面的东东。还是用例子来说明吧。

```

<html>
<head>
    <title>Untitled Page</title>
</head>
<body>
    Hello &nbsp;{{.UserName}}.
</body>
</html>

```

如上面的代码，`{{.UserName}}` 是一个动态参数，我们在程序里赋不同的值，就会有不同的展示结果。这就是 HTML 模板。Go 语言默认提供了一个 HTML 模板的实现，位于 `html/template` 包下。

模板展示

在 `html/template` 包中提供了 `Parse` 和 `Execute` 函数，`parse` 用来解析模板，`Execute` 用来

将结果展示出来。定义如下：

```
func (t *Template) Parse(text string) (*Template, error)
```

Text 为要解析的模板内容，返回*Template 对象，如果成功 error 为 nil。

```
func (t *Template) Execute(wr io.Writer, data interface{}) (err error)
```

将模板输出到 wr 中，data 为向模板传递的数据。

下面看一个简单的例子。

```
package main
import (
    "fmt"
    "html/template"
    "net/http"
)
func main() {
    http.HandleFunc("/", HelloServer)
    err := http.ListenAndServe(":12345", nil)
    if err != nil {
        fmt.Println(err)
    }
}
func HelloServer(w http.ResponseWriter, r *http.Request) {
    strTemplate := "<div><h3>欢迎光临! </h3></div>"
    t := template.New("test")
    //解析模板
    t, err := t.Parse(strTemplate)
    if err != nil {
        fmt.Println(err)
        return
    }
    err = t.Execute(w, nil)
    if err != nil {
        fmt.Println(err)
    }
}
```

编译并运行程序，在浏览器地址栏中输入 <http://localhost:12345/>，就可以看到模板的输出结果。通常在实际应用中，会把模板放到一个独立的文件中，这样就可以在不修改程序的情况下，修改模板，实现程序业务逻辑与展示的分隔。Template 提供了一个 ParseFiles 函数，用来解析模板文件，定义如下：

```
func ParseFiles(filenamees ...string) (*Template, error)
```

ParseFiles 创建一个新的 Template 对象，并对指定的模板文件进行解析。我们把上面的例子做些修改。把”<div><h3>欢迎光临! </h3></div>”保存到一个文件中，命名为 hello.tpl。然后修改程序如下。

```

package main

import (
    "fmt"
    "html/template"
    "net/http"
)

func main() {
    http.HandleFunc("/", HelloServer)
    err := http.ListenAndServe(":12345", nil)
    if err != nil {
        fmt.Println(err)
    }
}

func HelloServer(w http.ResponseWriter, r *http.Request) {
    t, err := template.ParseFiles("test.tpl")
    if err != nil {
        fmt.Println(err)
        return
    }
    err = t.Execute(w, nil)
    if err != nil {
        fmt.Println(err)
    }
}

```

基本语法

Go 的模板支持一些简单的语法，如 `if else`, `range`，等。下面我们一一介绍。

1) 变量的展示

在模板中，使用 `{{和}}` 来输出变量到当前位置，如 `{{.}}`、`{{.UserName}}`。下面是一个例子：

```

package main

import (
    "fmt"
    "html/template"
    "os"
)

func main() {
    strTpl := "你好, {{.}}\r\n"
    t, err := template.New("test").Parse(strTpl)
    if err != nil {

```



```

        fmt.Println(err)
        return
    }
    //将要输出的数据传到模板中
    err = t.Execute(os.Stdout, "张三丰")
    if err != nil {
        fmt.Println(err)
    }

    strTpl2 := "姓名: {{.Name}}\r\n年龄: {{.Age}}\r\n"
    user := make(map[string]interface{})
    user["Name"] = "吕洞宾"
    user["Age"] = 3000
    t, err = template.New("test2").Parse(strTpl2)
    if err != nil {
        fmt.Println(err)
        return
    }
    //这里用的是一个 map, 如果改成 struct 也是可以的
    err = t.Execute(os.Stdout, user)
    if err != nil {
        fmt.Println(err)
    }
}

```

2) if else 条件

在模板中可以使用 if else 条件语句，格式如下：

```
{{if pipeline}} T1 {{else}} T0 {{end}}
```

或

```
{{if pipeline}} T1 {{end}}
```

```

package main
import (
    "fmt"
    "html/template"
    "os"
)
func main() {
    strTpl := "{{if .IsLogin}} 已登录 {{else}} 请登录 {{end}}\r\n{{if .IsVip}}贵宾{{else}}非贵宾{{end}}\r\n"
    data := make(map[string]bool)
    data["IsLogin"] = true
    t, err := template.New("test").Parse(strTpl)
}

```

```

if err != nil {
    fmt.Println(err)
    return
}

err = t.Execute(os.Stdout, data)
if err != nil {
    fmt.Println(err)
}
}

```

在上面的例子中只有 `data["IsLogin"] = true`, `data` 中没有 `IsVip`, `{{if .IsVip}}` 的结果是 `false`, 说明不存在时为 `false`。

3) range

在模板中可以使用 `range` 来取 `array,slice,map` 中的值。定义如下:

```
{{range pipeline}} T1 {{end}}
```

或

```
{{range pipeline}} T1 {{else}} T0 {{end}}
```

只能对 `array,slice,map,channel` 使用 `range`, 如果变量不存在, 或长度为 0 将执行 `else` 部分 `T0`。

```

package main
import (
    "fmt"
    "html/template"
    "os"
)
func main() {
    strTpl := `{{range .test}}{{.}}\r\n{{end}}
{{range .test1}}{{.}}\r\n{{else}}test1 不存在。{{end}}\r\n`
    data := make(map[string]interface{})
    arr := []int{1, 2, 3, 4}
    data["test"] = arr
    t, err := template.New("test").Parse(strTpl)
    if err != nil {
        fmt.Println(err)
        return
    }
    err = t.Execute(os.Stdout, data)
    if err != nil {
        fmt.Println(err)
    }
}

```

模板函数

Go 提供了一些模板函数，如 `and`, `or`, `len` 等函数。需要注意的是 `and`, `or` 在这里是函数，用法跟语言中的 `and`, `or` 有些区别。具体可以参考 go 语言官方 `text/template` 包的文档。

```
package main
import (
    "fmt"
    "html/template"
    "os"
)
func main() {
    strTpl := "and a b 结果为 {{and .a .b}}\r\nor a b 结果为 {{or .a .b}}\r\n"
    data := make(map[string]bool)
    data["a"] = true
    data["b"] = false
    t, err := template.New("test").Parse(strTpl)
    if err != nil {
        fmt.Println(err)
        return
    }
    err = t.Execute(os.Stdout, data)
    if err != nil {
        fmt.Println(err)
    }
}
```

Go 模板本身提供的函数有限，在实际应用中可能无法满足我们的需求，这时可以使用自定义的模板函数。Go 提供了一个 `Funcs` 函数用来设置自定义的模板函数，定义如下：

```
func (t *Template) Funcs(funcMap FuncMap) *Template
```

`FuncMap` 的定义如下：

```
type FuncMap map[string]interface{}
```

下面看一个例子：

```
package main
import (
    "fmt"
    "html/template"
    "os"
)
func main() {
    strTpl := "{{SayHello}}\r\n"
    funcs := make(template.FuncMap)
    funcs["SayHello"] = SayHello
    t, err := template.New("test").Funcs(funcs).Parse(strTpl)
```

```
    if err != nil {  
        fmt.Println(err)  
        return  
    }  
    err = t.Execute(os.Stdout, nil)  
    if err != nil {  
        fmt.Println(err)  
    }  
}  
  
func SayHello() string {  
    return "你好，自定义模板函数"  
}
```

在上面的例子中我们定义了一个模板函数 SayHello，并在模板中进行了调用。

第十四章 GoMvc Web 框架

14.1 MVC 简介

MVC 是一种软件架构模式,Model-View-Controller 的缩写,也就是模型—视图—控制器的意思。它把软件系统分为三个基本部分:模型(Model),视图(View)和控制器(Controller)。MVC 的目的是实现业务逻辑(M)和数据显示(V)的分离,从而降低应用程序的复杂度,提高代码的可维护性。

模型 (Model) “数据模型”(Model)用于封装与应用程序的业务逻辑相关的数据以及对数据的处理方法。通常, model 对象从数据库中获数, 或将数据保存到数据库中。

控制器 (Controller) 控制器处理用户的输入, 与用户进行交互, 并最终选择一个视图(View)来显示数据。在 MVC 应用程序中, 视图只用来显示数据; 控制器用来处理和响应用户的输入, 与用户交互。例如一个产品查询页面, 控制器接收用户输入的产品名称, 并将该值传给模型层(Model)查询相关的产品信息。

视图 (View) 视图用来显示数据。如在 12.4 HTML 模板处理一节中, HTML 模板可以看作视图, 用来展示数据。

在早期, 业务逻辑, 用户交互, 数据展示是放在一起的, 比如 ASP 的一些程序, 多是这样的。下面我们用一个产品展示的例子, 对比 MVC 跟非 MVC 两种处理方式的区别, 以便说明问题。

新建一个产品表, 并向表中插入几条数据, 这里只有产品名称, 和价格两个字段, 主要是为了说明问题。

```
CREATE TABLE `Product` (  
  `Id` INT(10) NULL AUTO_INCREMENT,  
  `ProductName` VARCHAR(100) NULL,  
  `Price` DOUBLE NULL,  
  PRIMARY KEY (`Id`)  
)  
COLLATE='utf8_general_ci'  
ROW_FORMAT=DEFAULT;  
  
insert into `Product` (`ProductName`,`Price`) values('中医四大名著',580);  
insert into `Product` (`ProductName`,`Price`) values('伤寒论',10);  
insert into `Product` (`ProductName`,`Price`) values('本草纲目',49.9);  
insert into `Product` (`ProductName`,`Price`) values('图解中草药大全',26.8);  
insert into `Product` (`ProductName`,`Price`) values('汤头歌诀',10);
```

接下来, 我们用 Go 写一个简单的查询页面, 用户输入处理, 数据的查询, 展示, 都在一起, 以便说明问题。

```
package main  
import (  
  "database/sql"
```

```

    "fmt"
    _ "github.com/go-sql-driver/mysql"
    "html"
    "net/http"
)

func main() {
    http.HandleFunc("/", HandleRequest)
    http.ListenAndServe(":8888", nil)
}

func HandleRequest(w http.ResponseWriter, r *http.Request) {
    var strProductName string
    if r.Method == "POST" {
        //获取用户输入的产品名称
        r.ParseForm()
        strProductName = r.FormValue("txtProductName")
    }
    //查询数据
    db, err := sql.Open("mysql", "root:123@tcp(127.0.0.1:3306)/test?charset=utf8")
    if err != nil {
        fmt.Println(err)
        return
    }
    defer db.Close()
    var rows *sql.Rows
    if strProductName == "" {
        rows, err = db.Query("select ProductName,Price from product")
    } else {
        strProductName = "%" + strProductName + "%"
        rows, err = db.Query("select ProductName,Price from product where ProductName like ?", strProductName)
    }
    if err != nil {
        w.Write([]byte(err.Error()))
        return
    }
    //展示数据
    strHTML := "<html><header><title>产品查询</title></header><body>"
    strHTML += "<form action=\"/\" method=\"post\">产品名称: "
    strHTML += "<input id=\"txtProductName\" name=\"txtProductName\" type=\"text\" />"
    strHTML += "<input type=\"submit\" value=\"搜索\" /></form>"
    strHTML += "<table style='width:500pt' border='1' cellpadding='2' cellspacing='2'><tr>"
    strHTML += "<th align='left'>产品名称</th><th align='left'>价格</th></tr>"

```

```

strHTML += "<th align='left'>产品名称</th><th align='left'>价格</th></tr>"
for rows.Next() {
    var name string
    var price float64
    rows.Scan(&name, &price)
    strHTML += "<tr><td>" + html.EscapeString(name) + "</td><td>" +
fmt.Sprintf("%v", price) + "</td></tr>"
}
rows.Close()
strHTML += "</table></body></html>"
w.Write([]byte(strHTML))
}

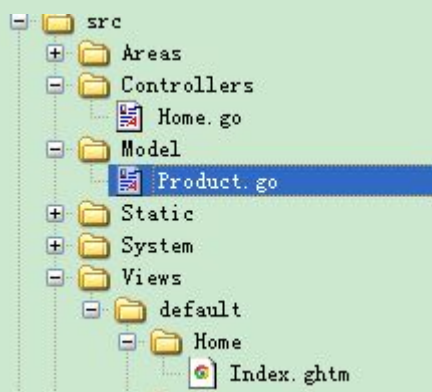
```

在地址栏输入 <http://localhost:8888/> 打开页面，会列出所有的产品，当然在实际应用中还应该分页，我们这里主要为了说明问题，所以就没做分页。这个页面非常简单，当应用程序小，并且不经常更改时，这是一种不错的实现方案。在实际开发中可能遇到下面的情况：

- 通常情况下，对页面显示部分的修改频率远比对业务逻辑的修改高，混在一起增加了修改时出错的可能性。
- 在某些情况下，应用程序需要以不同的方式来显示数据。例如，当用户通过手机访问时，跟通过 PC 访问时，展示的页面格式是不同的。
- 在一些项目中，业务逻辑开发，跟前端页面的展示，可能是有不同的人完成的。混在一起不利于两者的并行开发。
- 在其它的地方，如果也要用到从数据库中取产品数据，需要重新写或把这段代码复制过去，不利于代码的重用和维护。

使用 MVC 框架可以解决上面所遇到的问题。下在我们把上面的示例改 MVC 的代码。这里以 GoMvc 为例，GoMvc 可在 <https://github.com/male110/GoMvc> 页面进行下载。

我们首先实现 Model 层，在 Model 目录下，新建一个 Product.go 文件，如下图所示。



在 Product.go 中实现产品的查询函数，源码如下：

```

package Model
import (
    "database/sql"
    _ "github.com/go-sql-driver/mysql"
)

```

```

type Product struct {
    Id          int
    ProductName string
    Price       float32
}

func (this *Product) GetProduct(productName string) ([]Product, error) {
    //查询数据
    db, err := sql.Open("mysql", "root:123@tcp(127.0.0.1:3306)/test?charset=utf8")
    if err != nil {
        return nil, err
    }
    defer db.Close()
    var rows *sql.Rows
    if productName == "" {
        rows, err = db.Query("select Id,ProductName,Price from product")
    } else {
        productName = "%" + productName + "%"
        rows, err = db.Query("select Id,ProductName,Price from product where ProductName like ?", productName)
    }
    if err != nil {
        return nil, err
    }
    var arrPreoduct []Product
    for rows.Next() {
        var id int
        var name string
        var price float32
        rows.Scan(&id, &name, &price)
        p := Product{Id: id, ProductName: name, Price: price}
        arrPreoduct = append(arrPreoduct, p)
    }
    return arrPreoduct, nil
}

```

在 Model 层，我们定义了一个 Product 结构体，跟数据表 product 对应。结构体 Product 有一个方法 GetProduct，用来从数据库中取数据，该函数可以在任何地方调用。相比原来写在一起的代码，更加方便重和维护。

在 Controllers 目录下，新建一个 Home.go 来处理用户的输入，与用户进行交互，通常会命名为 Product.go，这里使用 Home.go 是因为框架的默认首页是 Home，所以这里 controller 的代码就写在 Home 里了，为了简单。代码如下：


```

package Controllers
import (
    "Model"
    "System/Web"
)
type Home struct {
    Web.Controller
}
//注册 Controller
func init() {
    Web.App.RegisterController(Home{})
}
func (this *Home) Index() *Web.ViewResult {
    var product Model.Product
    strProductName := ""
    if this.Request.Method == "POST" {
        strProductName = this.Form["txtProductName"]
    }
    arrProduct, err := product.GetProduct(strProductName)
    if err != nil {
        this.ViewData["errMsg"] = err.Error()
    } else {
        this.ViewData["arrProduct"] = arrProduct
    }
    this.ViewData["productName"] = strProductName
    return this.View()
}

```

在 Index 函数中，获取用户输入的产品名称，然后调用 Model 层的 GetProduct 来获取产品，将展示相关的数据放到 ViewData 中。最后调用 this.View 来展示数据。视图位于 Views\default\Home 目录下，对应的文件为 Index.ghtml。Home 目录对应控制器 (Controller)Home，Index.ghtml，对应 Home.Index 函数，this.View 默认展示 Home/Index.ghtml 视图。视图部分的原码如下：

```

<html>
<head>
    <title>产品管理</title>
</head>
<body>
<form action="/" method="post">
产品名称: <input id="txtProductName" name="txtProductName" value="{{.productName}}" type="text" />
<input type="submit" value="搜索" />
</form>

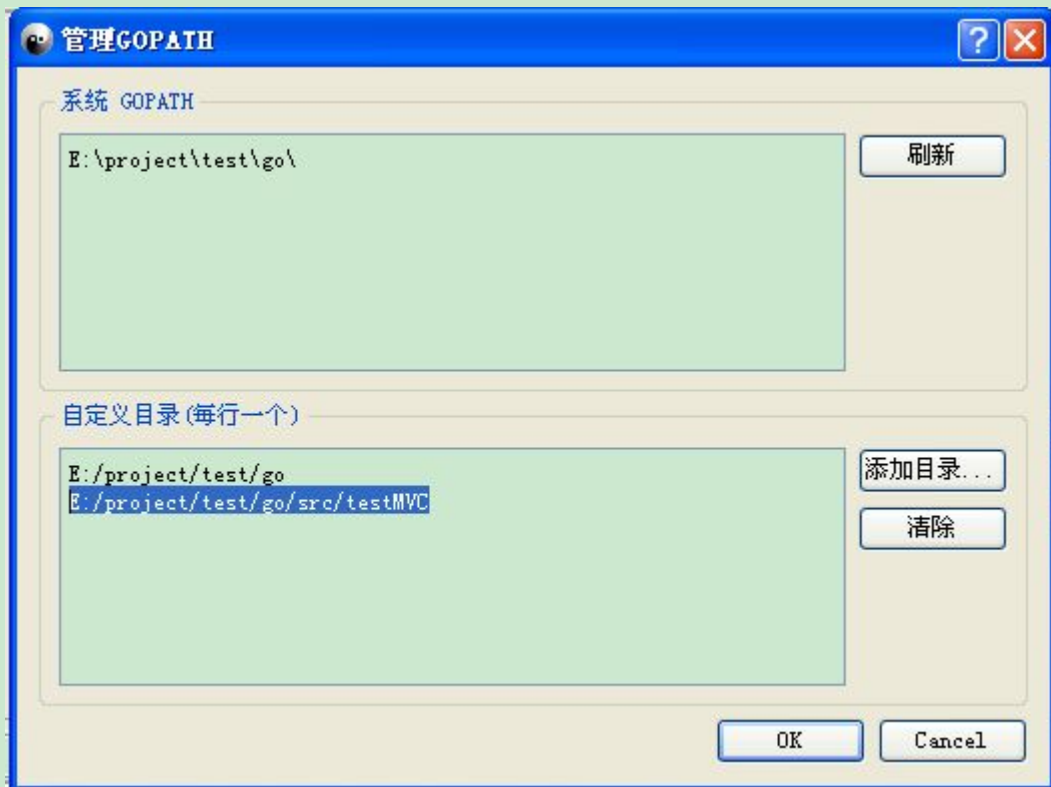
```

```

{{if .errMsg}}
  {{.errMsg}}
{{else}}
<table style='width:500pt' border='1' cellpadding='2' cellspacing='2'>
<tr><th align='left'>Id</th><th align='left'>产品名称</th><th align='left'>价格</th></tr>
  {{range .arrProduct}}
<tr><td>{{.Id}}</td><td>{{.ProductName}}</td><td>{{.Price}}</td></tr>
  {{else}}
<tr><td colspan="3">没有数据</td></tr>
  {{end}}
</table>
{{end}}
</body>
</html>

```

这里的 View 跟逻辑代码，交互处理的代码是分开的，独立的，要修改 View 更加方便，且不会影其它部分。要编译程序，需要把当前目录添加到 GOPATH 中，我用的是 LiteIDE，在“查看”=》“管理 GO PATH...”中，把程序所在路径添加进去。



修改 web.config，ListenPort 改为 8888，其它保持默认值，如下：

```
<ListenPort>8888</ListenPort>
```

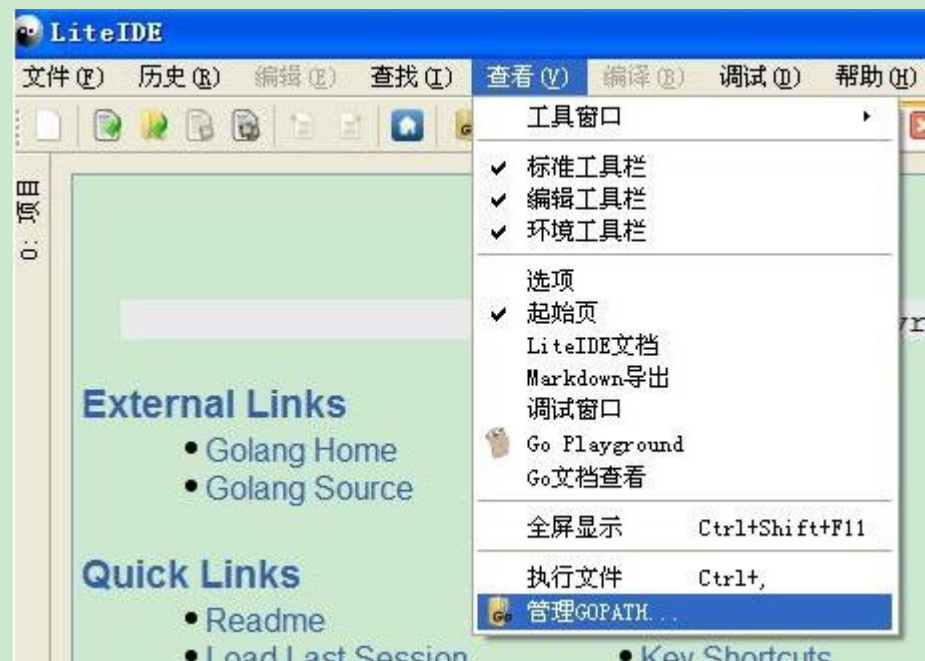
编译并运行程序。浏览器中输入 <http://localhost:8888/> 查看程序。

4.2 GoMvc 简介

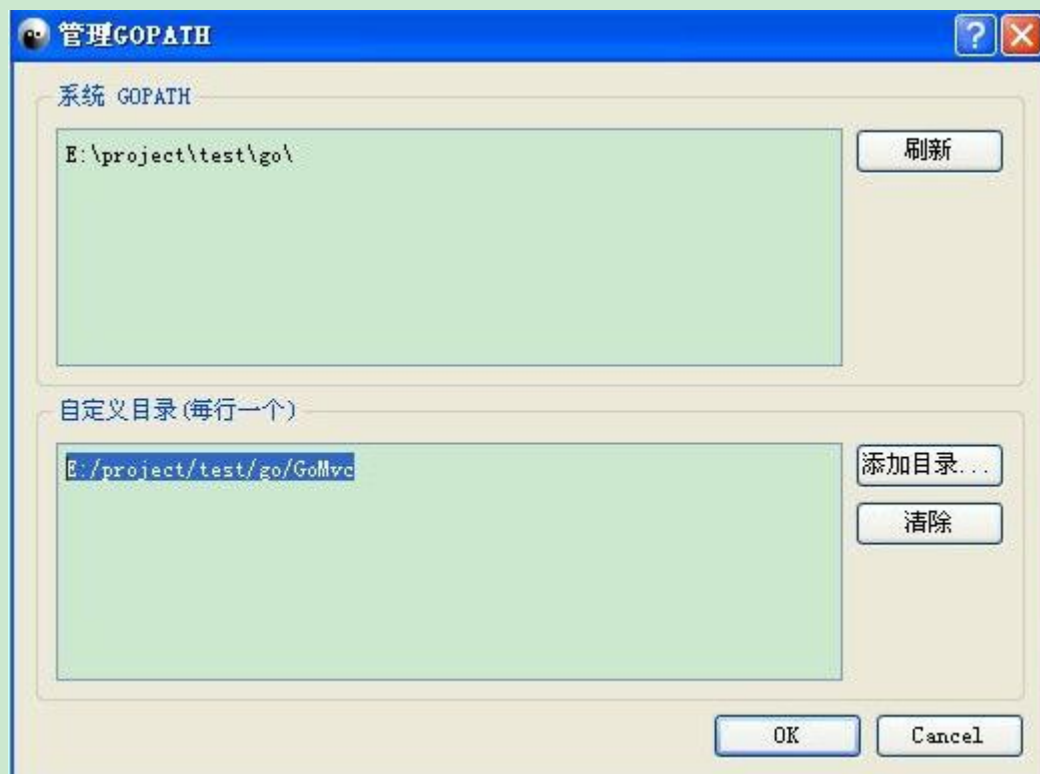
Mvc 框架已有很多, GoMvc 是我写的一个简单的 Mvc 框架, 其实现参考了 `asp.net mvc`。很适合 .net, , java , PHP 转来的, 应该会有有一种似曾相识的感觉。可在 <https://github.com/male110/GoMvc> 页面进行下载。我尽量做得简单, 让用户下载后就能运行, 尽可能的减少对其它第三方包的依赖。尽管如此, 在编译时还是需一个 `mysql` 的驱动包, 该项目位于 <https://github.com/go-sql-driver/mysql>。如果你没有安装该 `mysql` 驱动可以输入下面的命令进行安装。

```
go get github.com/go-sql-driver/mysql
```

在编译时, 需要把 GoMvc 所在的目录设置到环境变量 `GOPATH` 中。如果对 `GOPAT` 不熟可以参看第一章第二节, 安装 Go。如果你用的是 LiteIDE, 也可以在 LiteIDE 中设置。在 LiteIDE 的“查看”菜单下有一项“管理 `GOPATH...`”, 如下图:



然后在弹出的窗口中设置 GoPath 目录, 如下图所示:



14.3 GoMvc 目录结构

GoMvc 目录结构十分简单,框架本身相关的文件都放在 System 目录下;视图放在 Views 目录下;所有的 Controller 放在 Controllers 目录下;静态文件,如 JS, CSS,图片等放在 static 目录下。除 System,Views 目录是规定死的,其它的都是推荐的目录结构,可以根据习惯做些改。

└── src	
│ ├── System	// GoMvc 系统文件所在的目录
│ ├── Controllers	// 所有 Controller 都在这个目录下
│ ├── Model	//所有模型层的代码放在此处
│ ├── Static	//所有的静态文件放在此处,如 JS, CSS, 图片等静态资源,
│ └── Views	//所有的视图页面存放在这里,文件的扩展名为.ghtml

14.4 配置文件

GoMvc 是仿 Asp.net 做的,所以网站的配置文件也命名为了 web.config,格式为 XML,与 asp.net 相似,配置项的内容如下:

IsDebug: 配置为 false 时, _Global 目录下的公共模将会被缓存,一分钟后模板的修改才能

生效；配置 true 时，每次都会去检查公共模板是否有更新。建议正式环境配置为 false,开发环境配置为 true。

ShowErrors: 是否显示错误信息。true,显示；false,不显示。建议在测试时可以设置为 true,发布到正式环境后设置为 false。

CookieDomain: Cookies 的 Domain 信息，可用来共享 cookie。如 domain.com，和 sub.domain.com，可以通过把 CookieDomain 统一设置为 domain.com 来共享 cookies 信息

Theme: 网站当前使用的主题，在 Views 目录下，可以有多套网站模板。

LogPath: 日志文件的存放位置

LogFileMaxSize: 单个日志文件的大小，超过指定大小后将创建一个新的日志文件。

DriverName: 数据库的驱动名称。

DataSourceName: 数据库的连接字符串。

StaticDir: 静态目录,该目录下通常是 CSS,JS,图片等静态资源。

StaticFile: 静态文件，用来设置单个的静态文件，主要是为了提高灵活性，满足特殊的需求。

SessionType: Session 的存放类型,1:文件,2:内存,3:Mysql 数据库,修改需重启才能生效。当配置为 3 时，需要在数据库中创建一个表，来存放 session,创建表的 SQL 如下：

```
CREATE TABLE `session` (  
    `session_id` CHAR(32) NULL,  
    `session_data` BLOB NULL,  
    `lastupdatetime` DATETIME NULL,  
    PRIMARY KEY (`session_id`)  
)  
COLLATE='utf8_general_ci';
```

SessionLocation: 当 SessionType 为 1 时，该项为 Session 文件的存放路径；SessionType 为 3 时,该项为数据库连接字符串。

SessionTimeOut: Session 超时时间，单位分钟

MemFreeInterval: 程序中有定时器，定时对 Session 进行检查，删除超时的 Session，该配置项用来设置多久进行一次检查，单位秒，默认值 60。

ListenPort: 网站的端口号,该配置改后必须重启程序才能生效。

14.5 路由

GoMvc 的路由注册，跟 asp.net Mvc 的路由注册非常相似，路由部分的实现参考了 Mono 的代码，或者说是从 Mono 移植过来的。路由用 `RouteTable.AddRote` 来注册。其格式如下：

```
//注册标准路由
RouteTable.AddRote(&RouteItem{
    Name:      "default",
    Url:       "{controller}/{action}",
    Defaults: map[string]interface{}{"controller": "home", "action": "index"}})
```

Name:路由名称

Url:路由的格式

Defaults: 路由参数的默认值

除了默认值，还可以指定约束，来限制参数的类型，如下面的例子，指定 `id` 参数，只能是数字型。

```
RouteTable.AddRote(&RouteItem{
    Name: "default",
    Url:  "{controller}/{action}/{id}",
    Defaults: map[string]interface{}{"controller": "home", "action": "index", "id": 123},
    Constraints: map[string]string{"id": `^\d+$`})
```

在上面的例子中我们指定了 `id` 参数只能是数字，并设置了默认值 123。要在 Controller 中获取该参数值，可以用 `this.RouteData["id"]`。

14.6 Controller

所有的控制器都放在 `Controllers` 目录下，所有的控制器都需要用 `Web.App.RegisterController` 来注册。这是有 Go 的反射特性所决定的，因为我没办法用反射来获取包中所有的 struct。下面是 Home 控制器的示例代码：

```
package Controllers
import (
    "Model"
    "System/Web"
    "fmt"
)
type Home struct {
    Web.Controller
}
```

```
//注册 Controller
func init() {
    Web.App.RegisterController(Home{})
}
```

在上面的 init 函数中，使用 Web.App.RegisterController 对控制器进行了注册，因为在 Go 没有办法反射出 Controllers 包中的所有 struct，所以需要手动来注册 Controller。

在 GoMvc 中并没有实现 Filter(过滤器的功能)，只提供了 OnLoad，UnLoad 两个函数，OnLoad 在页面加载时，即 Action 执行之前调，该函数没有任何返回值。UnLoad，在页面结束时，即 Action 已经执行完，页面已输出展示时执行。如用户登录的判断可以放在 OnLoad 中执行。GoMvc 提供了 ResponseEnd 函数，在调用了 ResponseEnd 后，请求将结束，如在 OnLoad 中调用了 this.ResponseEnd()，Action 将得不到执行，请求结束。

Controller 的定义如下：

```
type Controller struct {
    Request      *http.Request
    Response      http.ResponseWriter
    ViewData      map[string]interface{}
    Session       map[string]interface{}
    RouteData     map[string]interface{}
    QueryString   map[string]string
    Form          map[string]string
    .....
    Cookies       map[string]string
    DefaultBinder *Binder
}
```

Form 中存放了所有的 Post 过来的表单值，QueryString 存放了地址栏参数的值。当值重复时，这里存放的是最后一次的值，如 a=1&a=2，最后保存的值是 2。Cookies 存放了所有 Cookies 的值。

14.7 Action

Action 的返回结果通常为 IActionResult 对像，有 ViewResult、JavaScriptResult、JsonResult、XmlResult。具体实现可以参考"System/Web/Controller.go"文件中的实现代码。下面分别介绍。

- **ViewResult:** 表示当前返回结果为 HTML 页面，通常用法如下：

```
func (this *Home) Index() *Web.ViewResult {
    this.ViewData["Title"] = "欢迎使用 GoMvc"
    return this.View()
}
```

this.View 接受一到两个参数，第一个参数为模板的名称，第二个为主题名称，两个参数都可以省略，第一个参数省略时或为""时，Action 做为模板名称，第二个参数省略时为默认的主题。模板引擎将会在 Views/Home 目录下去找 Index.ghtml 文件，并展示到前端。Action

通过 ViewData 来向 View 模板传送数据的。在模板中展示 ViewData 中的数据，如下所示：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
    <title>{{.Title}}</title>
```

有关模板的语法，可以参考[官方文档](http://golang.org/pkg/text/template/)，<http://golang.org/pkg/text/template/>。

- **JavaScriptResult**: 用来返回一段 JavaScript 脚本，示例代码如下：

```
func (this *Home) TestScript() *Web.JavaScriptResult {
    return this.JavaScript("alert('OK!');", "utf-8")
}
```

第一个参数为要输出的脚本，第二个参数是字符编码，可省略，默认为 utf-8。

- **JsonResult**: 用来输出 JSON 字符串，示例代码如下：

```
func (this *Home) TestJson() *Web.JsonResult {
    this.ViewData["UserName"] = "张三"
    this.ViewData["AGe"] = "30"
    return this.Json(this.ViewData, "utf-8")
}
```

第一个参数可以是 JSON 字符串，struct 或 map 对象，第二个参数是字符编码，可省略，默认为 utf-8。当第一个参数为 map[string]interface{} 类型时，map 的 key 中不能包含 Action, Controller, Request 键值，这些在展示时会被屏蔽掉。主要是因为 ViewData 中，默认会有 Action, Controller, Request 几个值，如果不屏蔽这几个值，this.Json(this.ViewData, "utf-8") 得到的将不是我们想要的结果。

XmlResult: 用来输出 XML 文档，示例代码如下：

```
type User struct {
    UserName string
    Age      int
}

func (this *Home) TestXml() *Web.XmlResult {
    u := User{"张三", 19}
    return this.Xml(u, "utf-8")
}
```

第一个参数可以是 XML 字符串或 struct 结构体，但不能是 map 对象。第二

个参数是字符编码，可省略，默认为 utf-8。

更多内容请下载 [GoMvc](#) 参看说明文档,有任何问题可以加入 QQ 群 184572648, 一起讨论解决。