

Go Web プログラミング

astaxie

2021 年 3 月 16 日

1 Introduction

Web プログラミングが好きで Go web プログラミングを書きました。皆さんに気にいってもらえれば幸いです。

2 Go の環境設定

Go の世界へようこそ、さっそく初めてみましょう！

Go は新しい言語です、並列処理、ガベージコレクションを備え、軽快にコンパイルできる言語です。以下のような特徴を持っています：

- 一台のコンピュータ上であつという間に大型の Go プログラムをコンパイルすることができます。
- Go はソフトウェアの構造にモデルを与えます。分析をより簡単にこなすことができ、ファイルやライブラリの include といった C スタイルの書き出しにありがちな部分を大幅に省くことができます。
- Go は静的型付け言語です。型に階層の概念が無いのでユーザはその関係に気をとられることもなく、典型的なオブジェクト指向言語よりももっとライトに感じるくらいです。
- Go は完全なガベージコレクションタイプの言語です。また、基本的な並列処理とネットワークをサポートしています。
- Go はマルチプロセッサ対応のソフトウェアを作成できるようデザインされています。

Go はコンパイラ型言語の一種です。インタプリタ型言語の軽い身のこなしと動的型付け言語の開発効率、それに静的型付け言語の安全性を兼ね備えています。また、今風のネットワークとマルチプロセッサもサポートしています。これらを実現する為には、表現力豊かで且つ軽いクラスシステム・並列処理とガベージコレクション・厳格な依存定義などを言語レベルで満たしていなければなりません。どれもライブラリやツールでは解決しきれないものです。Go はその要望に応えます。

この章では Go のインストール方法と設定についてご紹介します。

2.1 Go のインストール

2.1.1 3つのインストール方法

Go にはいくつかのインストール方法があります。どれでも好きなものを選んでかまいません。ここでは3つのよくあるインストール方法をご紹介します：

- ソースコードのインストール：標準的なインストール方法です。Unix 系システムをよく使うユーザ、特に開発者であれば、設定を好みに合わせて変更できます。
- 標準パッケージのインストール：Go は便利なインストールパッケージを用意しています。Windows, Linux, Mac などのシステムをサポートしています。とりあえずさっとインストールするにはうってつけでしょう。システムの bit 数に対応したインストールパッケージをダウンロードして、“Next”をただでインストールできます。おすすめ
- サードパーティツールによるインストール：現在便利なサードパーティパッケージも多くあります。たとえば Ubuntu の apt-get、Mac の homebrew などです。これらのシステムに慣れたユーザにはぴったりのインストール方法です。

最後に同じシステムの中で異なるバージョンの Go をインストールする場合は、GVM(<https://github.com/moovweb/gvm>) が参考になります。どうすればよいか分からない場合一番うまくやれます。

2.1.2 Go ソースコードのインストール

Go のソースコードには Plan 9 C と AT&T コンパイラを使って書かれている部分があります。もしソースコードからインストールしたい場合は、あらかじめ C のコンパイルツールをインストールしておく必要があります。

Mac では、Xcode に適切なコンパイラが含まれています。

Unix では、gcc などのツールをインストールする必要があります。例えば Ubuntu ではターミナルで `sudo apt-get install gcc libc6-dev` を実行することでコンパイラをインストールすることができます。

Windows では、MinGW をインストールする必要があります。その後 MinGW で gcc をインストールして、適切な環境変数を設定します。

直接オフィシャルサイトからソースコードをダウンロードできます。対応する `goVERSION.src.tar.gz` のファイルをダウンロードし、`$HOME` ディレクトリに解凍してから以下のコマンドを実行します。

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

`all.bash` を実行後“ALL TESTS PASSED”が表示されると、インストール成功です。

上記は Unix スタイルのコマンドです、Windows もインストール方法は似ており、`all.bat` を実行するだけです。コンパイラは MinGW の gcc を使います。

もし Mac または Unix ユーザであればいくつかの環境変数を設定する必要があります。再起動しても有効にしたい場合は以下のコマンドを `.bashrc` や `.zsh` に書いておきます。

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

ファイルに書き込んだ場合は、`bash .bashrc` や `bash .zshrc` を実行してすぐに設定を有効にします。

Windows システムの場合は、環境変数を設定する必要があります。path に go が存在するディレクトリを追加し、gopath 変数を設定します。

設定が終わり、コマンドプロンプトで go を入力すると、下図のような画面が表示されるはずです。

```
apblematoMacBook-Pro-3:~ apple$ go
Go is a tool for managing Go source code.

Usage:

    go command [arguments]

The commands are:

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

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

Additional help topics:

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

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

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

図 1 ソースコードインストール後 Go コマンドを実行

Go の Usage 情報が表示されれば、Go のインストールは成功です；もしこのコマンドが存在しない場合は、PATH 環境変数のなかに Go のインストールディレクトリが含まれているか確認してください。

GOPATH については以降の章で詳しくご説明します

2.1.3 Go 標準パッケージのインストール

Go はさまざまなプラットフォームでインストールパッケージを提供しています、これらのパッケージはデフォルトで以下のディレクトリにインストールします: /usr/local/go (Windows : c:\ Go)。当然これらのインストール場所を変更することもできます、ただし変更後はあなたの環境変数を以下のように設定する必要があります:

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

これらのコマンドは Mac や Unix ユーザであれば .bashrc や .zshrc ファイルに入れておくべきでしょう。Windows ユーザであれば当然環境変数に入れておきます。

2.1.4 自分の操作しているシステムが 32bit か 64bit か判断する方法

Go のインストールにはオペレーティングシステムの bit 数を判断する必要があるので、この章では先に自分のシステムの種類を確認しましょう。

Windows のユーザは Win+R を押して cmd を実行してください。systeminfo と入力してエンターキーを押します。しばらくするとシステムの情報が表示されます。”システムの種類”の一行に”x64-based PC”と表示されていれば 64bit システムです。もし”X86-based PC”とあれば、32bit システムです。

Mac ユーザは直接 64bit 版を使用することをおすすめします。Go がサポートしている Mac OS X のバージョンは、すでに 32bit プロセッサをサポートしていないためです。

Linux ユーザは Terminal で arch(すなわち、uname -a) を実行することでシステムの情報を確かめることができます。

64bit システムであれば以下のように表示されます。

```
x86_64
```

32bit システムの場合は以下のように表示されます。

```
i386
```

2.1.5 Mac インストール

ダウンロード URL にアクセスし、32bit システムは go1.4.2.darwin-386-osx10.8.pkg をダウンロードします。64bit システムであれば go1.4.2.darwin-amd64-osx10.8.pkg をダウンロードします。ファイルをダブルクリックし、すべてデフォルトで「次へ」ボタンをクリックします。これで go はあなたのシステムにインストールされました。デフォルトで PATH の中に適切な /go/bin が追加されています。端末を開いて go と入力します。

インストール成功の画像が表示されればインストール成功です。

もし go の Usage 情報が表示した場合は、go はすでにインストールされています。もしこのコマンドが存在しない则表示した場合は、自分の PATH 環境変数の中に go のインストールディレクトリが含まれているか確認してください。

2.1.6 Linux インストール

ダウンロード URL にアクセスし、32bit システムであれば `go1.4.2.linux-386.tar.gz` を、64bit システムであれば `go1.2.2.linux-amd64.tar.gz` をダウンロードします。

以下では Go がインストールされたディレクトリを `$GO_INSTALL_DIR` と仮定します。

`tar.gz` をインストールディレクトリに解凍します：`tar zxvf go1.4.2.linux-amd64.tar.gz -C $GO_INSTALL_DIR`

`PATH` を設定します。`export PATH=$PATH:$GO_INSTALL_DIR/go/bin`

その後、`go` を実行します。

```
[root@SNDA-172-17-12-5 ~]# go
Go is a tool for managing GO source code.

Usage:

    go command [arguments]

The commands are:

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

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

Additional help topics:

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

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

図 2 Linux システムでインストールに成功したあと `go` を実行した時に表示する情報

もし `go` の Usage 情報が表示された場合は、`go` はすでにインストールされています。もしこのコマンドが存在しないと出てきた場合は、自分の `PATH` 環境変数の中に `go` のインストールディレクトリが含まれているか確認してください。

2.1.7 Windows インストール

Google Code ダウンロードページ (<http://golang.org/dl/>) にアクセスし、32bit の場合は名前に `windows-386` を含む `msi` パッケージを、64bit であれば名前に `windows-amd64` を含むものをダウンロードします。ダ

ダウンロード後実行しますが、デフォルトのインストールフォルダである `C:\Go\` を変更してはいけません。他の場所にインストールしてしまうと、あなたが書いた Go コードが実行できなくなってしまうかもしれません。インストールが終わるとデフォルトで環境変数 `Path` に Go のインストールフォルダの下にある `bin` フォルダ `C:\Go\bin\` が追加され、Go のインストールフォルダである `C:\Go\` の値が環境変数 `GOROOT` に追加されます。

インストールが成功しているか確認する「ファイル名を指定して実行」に `cmd` を入力し、コマンドラインツールを開きます。プロンプトで `go` と入力することで `Usage` 情報が確認できるか確かめることができます。`cd %GOROOT%` を入力すると、Go のインストールフォルダに入れるか確認できます。どちらも成功していれば、インストールに成功しています。

インストールに成功していなければ、環境変数 `Path` と `GOROOT` の値を確認してください。もし存在しなければアンインストールの上再インストールし、存在していればコンピュータを再起動し、上の手順を再度試してください。

2.1.8 サードパーティツールのインストール

`GVM` `gvm` はサードパーティが開発した Go のバージョン管理ツールです。`ruby` の `rvm` ツールに似ています。相当使い勝手がいいです。`gvm` をインストールするには以下のコマンド実行します：

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

インストールが完了したあと、`go` をインストールすることができます：

```
gvm install go1.4.2
gvm use go1.4.2
```

下のコマンドで、毎回 `gvm use` をコールする手間を省くことができます：`gvm use go1.4.2 --default`
上のコマンドを実行したあと、`GOPATH`、`GOROOT` などの環境変数が自動的に設定されます。これで、直接利用することができます。

`apt-get` `Ubuntu` は現在最も多く利用されている Linux デスクトップシステムです。`apt-get` コマンドでソフトウェア・パッケージを管理します。下のコマンドで `Go` をインストールすることができます、今後のため `git` と `mercurial` もインストールしておくべきでしょう：

```
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
```

`homebrew` `homebrew` は Mac で現在最も使用されているソフトウェア管理ツールです。現在 `Go` をサポートしており、以下のコマンドで `Go` を直接インストールすることができます。今後のため `git` と `mercurial` もインストールしておくべきでしょう：

```
brew update && brew upgrade
brew install go
brew install git
brew install mercurial
```

2.2 GOPATH とワーキングディレクトリ

さきほど Go をインストールする際は GOPATH 変数を設定する必要があるとご説明しました。Go はバージョン 1.1 から必ずこの変数を設定するようになっており、Go のインストールディレクトリと同じにはできません。このディレクトリは、Go のソースコードや Go の実行可能ファイル、並びにコンパイル済みのパッケージファイルを保存する為に使用します。そのためこのディレクトリには3つのサブディレクトリが存在します：src、bin、pkg です。

2.2.1 GOPATH 設定

go コマンドは、ある重要な環境変数に依存しています：\$GOPATH

Windows システムにおいて環境変数の形式は %GOPATH% です。この本の中では主に Unix 形式を使用します。Windows ユーザは適時置き換えてください。

(注：これは Go のインストールディレクトリではありません。以下では筆者のワーキングディレクトリで説明します。もし異なるディレクトリを使用する場合は GOPATH をあなたのワーキングディレクトリに置き換えてください。)

Unix に似た環境であれば大体以下のような設定になります：

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

上のディレクトリを新たに作成し、上の一行を .bashrc または .zshrc もしくは自分の sh の設定ファイルに加えます。

Windows では以下のように設定します。新しく GOPATH と呼ばれる環境変数を作成します：

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

GOPATH は複数のディレクトリを許容します。複数のディレクトリがある場合、デリミタに気をつけてください。複数のディレクトリがある場合 Windows はセミコロン、Linux はコロンを使います。複数の GOPATH がある場合は、デフォルトで go get の内容が第一ディレクトリとされます。

上の \$GOPATH ディレクトリには3つのディレクトリがあります：

- src にはソースコードを保存します (例えば：.go .c .h .s 等)
- pkg にはコンパイル後に生成されるファイル (例えば：.a)
- bin にはコンパイル後に生成される実行可能ファイル (このまま \$PATH 変数に加えてもかまいません。もしいくつも gopath がある場合は、\${GOPATH//:/bin:}/bin を使って全ての bin ディレクトリを追加してください)

以降私はすべての例で mygo を私の gopath ディレクトリとします。

2.2.2 ソースコードディレクトリ構成

GOPATH 内の src ディレクトリはこれから開発するプログラムにとってメインとなるディレクトリです。全てのソースコードはこのディレクトリに置くことになります。一般的な方法では一つのプロジェクトが一つのディレクトリが割り当てられます、例えば：\$GOPATH/src/mymath は mymath というアプリケーションパッケージが実行アプリケーションになります。これは package が main かそうでないかによって決定しま

す。main であれば実行可能アプリケーションで、そうでなければアプリケーションパッケージになります。これに関しては package を後ほどご紹介する予定です。

新しくアプリケーションやソースパッケージを作成するときは、src ディレクトリの中にディレクトリを作るところから始めます。ディレクトリ名は一般的にソースパッケージ名になります。もちろんネストしたディレクトリも可能です。例えば src の中に \$GOPATH/src/github.com/astaxie/beedb というディレクトリを作ったとすると、このパッケージパスは "github.com/astaxie/beedb" になり、パッケージ名は最後のディレクトリである beedb になります。

以下では mymath を例にどのようにアプリケーションパッケージをコーディングするかご説明します。以下のコードを実行します。

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

```
// $GOPATH/src/mymath/sqrt.go コードは以下の通り :
package mymath

func Sqrt(x float64) float64 {
    z := 0.0
    for i := 0; i < 1000; i++ {
        z -= (z*z - x) / (2 * x)
    }
    return z
}
```

このように私のアプリケーションパッケージディレクトリとコードが作成されました。注意：一般的に package の名前とディレクトリ名は一致させるべきです。

2.2.3 コンパイルアプリケーション

上のとおり、我々はすでに自分のアプリケーションパッケージを作成しましたが、どのようにコンパイル/インストールすべきでしょうか？ 2 種類の方法が存在します。

1. 対応するアプリケーションパッケージディレクトリに入り、go install を実行すればインストールできます。
2. 任意のディレクトリで以下のコード go install mymath を実行します。

インストールが終われば、以下のディレクトリに入り

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

\$GOPATH/src/mathapp/main.go コード：

```
package main

import (
    "mymath"
```



```

    "fmt"
)

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

```

このパッケージは main であることが分かります。import においてコールするパッケージは mymath であり、これが \$GOPATH/src のパスに対応します。もしネストしたディレクトリであれば、import の中でネストしたディレクトリをインポートします。例えばいくつもの GOPATH があった場合も同じで、Go は自動的に複数の \$GOPATH/src の中から探し出します。

さて、どのようにプログラムをコンパイルするのでしょうか？このアプリケーションディレクトリに入り、go build を実行すれば、このディレクトリの下に mathapp の実行可能ファイルが生成されます。

```
./mathapp
```

以下のように出力されます。

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

どのようにアプリケーションをインストールするのでしょうか。このディレクトリに入り、go install を実行すると、\$GOPATH/bin/の下に実行可能ファイル mathapp が作成されます。\$GOPATH/bin が我々の PATH に追加されていることを思い出して下さい、コマンドラインから以下のように入力することで実行することができます。

```
mathapp
```

この場合も以下のように出力されます。

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

ここではどのように実行可能アプリケーションをコンパイル/インストールし、ディレクトリ構造を設計するかについてご紹介しました。

2.2.4 リモートパッケージの取得

go 言語はリモートパッケージを取得するツール go get を持っています。現在 go get は多数のオープンソースリポジトリをサポートしています (github、googlecode、bitbucket、Launchpad)

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

go get -u オプションはパッケージの自動更新を行います。また、go get 時に自動的に当該のパッケージの依存する他のサードパーティパッケージを取得します。

このコマンドでふさわしいコードを取得し、対応するオープンソースプラットフォームに対し異なるソースコントロールツールを利用します。例えば github では git、googlecode では hg。そのためこれらのコードを取得したい場合は、先に対応するソースコードコントロールツールをインストールしておく必要があります。

上述の方法で取得したコードはローカルの以下の場所に配置されます。

```

$GOPATH
src
|--github.com
    |--astaxie
        |--beedb

pkg
|--対応プラットフォーム
    |--github.com
        |--astaxie
            |--beedb.a

```

go get は以下のような手順を踏みます。まずはじめにソースコードツールでコードを src の下に clone します。その後 go install を実行します。

コードの中でリモートパッケージが使用される場合、単純にローカルのパッケージと同じように頭の import に対応するパスを添えるだけです。

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

2.2.5 プログラムの全体構成

上記で作成したローカルの mygo のディレクトリ構造は以下のようになっています。

```

bin/
  mathapp
pkg/
  プラットフォーム名/ 例：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

```

上述の構成から明確に判断できるのは、bin ディレクトリの下にコンパイル後の実行可能ファイルが保存され、pkg の下に関数パッケージが保存され、src の下にアプリケーションのソースコードが保存されているということです。

2.3 Go のコマンド

2.3.1 Go のコマンド

Go 言語は完全なコマンド操作ツールセットを持つ言語です。コマンドラインで `go` を実行することでそれらを確認することができます：

```
aplematoMacBook-Pro-3:~ apple$ go
Go is a tool for managing Go source code.

Usage:

    go command [arguments]

The commands are:

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

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

Additional help topics:

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

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

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

図 3 Go コマンドで詳細情報を表示

これらのコマンドは我々が普段コードを書いている時に非常に役立つものです。次に普段使用するコマンドを理解していきましょう。

2.3.2 go build

このコマンドは主にソースコードのコンパイルに用いられます。パッケージのコンパイル作業中、もし必要であれば、同時に関連パッケージもコンパイルすることができます。

- もし普通のパッケージであれば、我々が 1.2 章で書いた mypath パッケージのように、go build を実行したあと、何のファイルも生成しません。もし \$GOPATH/pkg の下に対応するファイルを生成する必要があるれば、go install を実行してください。
- もしそれが main パッケージであれば、go build を実行したあと、カレントディレクトリの下に実行可能ファイルが生成されます。もし \$GOPATH/bin の下に対応するファイルを生成する必要があるれば、go install を実行するか、go build -o パス/a.exe を実行してください。
- もしあるプロジェクトディレクトリに複数のファイルがある場合で、単一のファイルのみコンパイルしたい場合は、go build を実行する際にファイル名を追加することができます。例えば go build a.go です。go build コマンドはデフォルトでカレントディレクトリにある全ての go ファイルをコンパイルしようと試みます。
- コンパイル後に出力されるファイル名を指定することもできます。1.2 章の mathapp アプリケーションでは go build -o astaxie.exe と指定できます。デフォルトは package 名 (main パッケージではない) になるか、ソースファイルのファイル名 (main パッケージ) になります。
(注：実際は package 名は Go 言語の規格においてコード中の "package" に続く名前になります。この名前はファイル名と異なっても構いません。デフォルトで生成される実行可能ファイル名はディレクトリ名。
- go build はディレクトリ内の "." または "." で始まる go ファイルを無視します。
- もしあなたのソースコードが異なるオペレーティングシステムに対応する場合は異なる処理が必要となります。ですので異なるオペレーティングシステムの名称にもとづいてファイルを命名することができます。例えば配列を読み込むプログラムがあったとして、異なるオペレーティングシステムに対して以下のようなソースファイルがあるかもしれません。

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

go build の際、システム名の末尾のファイルから選択的にコンパイルすることができます (Linux、Darwin、Windows、Freebsd)

引数の紹介

- o 出力するファイル名を指定します。パスが含まれていても構いません。例えば go build -o a/b/c
- i パッケージをインストールします。コンパイル + go install
- a すでに最新であるパッケージを全て更新します。ただし標準パッケージには適用されません。
- n 実行が必要なコンパイルコマンドを出力します。ただし、実行はされません。これにより低レイヤーで一体何が実行されているのかを簡単に知る事ができます。
- p n マルチプロセスで実行可能なコンパイル数を指定します。デフォルトは CPU 数です。
- race コンパイルを行う際にレースコンディションの自動検出を行います。64bit マシンでのみ対応しています。
- v 現在コンパイル中のパッケージ名を出力します。

-work コンパイル時の一時ディレクトリ名を出力し、すでに存在する場合は削除しなくなります。

-x 実行しているコマンドを出力します。-nの結果とよく似ていますが、この場合は実行します。

-ccflags 'arg list' オプションを 5c, 6c, 8c に渡してコールします。

-compiler name コンパイラを指定します。gccgo か、または gc です。

-gccgoflags 'arg list' オプションを gccgo リンカに渡してコールします。

-gcflags 'arg list' オプションを 5g, 6g, 8g に渡してコールします

-installsuffix suffix デフォルトのインストールパッケージと区別するため、このサフィックスを利用して依存するパッケージをインストールします。-race をオプションに指定した場合はデフォルトで-installsuffix race が有効になっています。-n コマンドで確かめることができますよ。

-ldflags 'flag list' オプションを 5l, 6l, 8l に渡してコールします。

-tags 'tag list' コンパイル時にこれらの tag をつけることができます。tag の詳細な制限事項に関しては Build Constraints(<http://golang.org/pkg/go/build/>) を参考にして下さい。

2.3.3 go clean

このコマンドは現在のソースコードパッケージと関連するソースパッケージのなかでコンパイラが生成したファイルを取り除く操作を行います。これらのファイルはすなわち：

_obj/	旧 objectディレクトリ、MakeFilesが作成する。
_test/	旧 testディレクトリ、Makefilesが作成する。
_testmain.go	旧 gotestファイル、Makefilesが作成する。
test.out	旧 testログ、Makefilesが作成する。
build.out	旧 testログ、Makefilesが作成する。
*.[568ao]	objectファイル、Makefilesが作成する。
DIR(.exe)	go buildが作成する。
DIR.test(.exe)	go test -cが作成する。
MAINFILE(.exe)	go build MAINFILE.goが作成する。
*.so	SWIG によって生成される。

私は基本的にこのコマンドを使ってコンパイルファイルを掃除します。ローカルでテストを行う場合これらのコンパイルファイルはシステムと関係があるだけで、コードの管理には必要ありません。

```
$ 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
```

引数紹介

-i go install がインストールするファイル等の、関係するインストールパッケージと実行可能ファイルを取り除きます。

-n 実行する必要がある削除コマンドを出力します。ただし実行はされません。これにより低レイヤで何が実行されているのかを簡単に知ることができます。

-r import によってインポートされたパッケージを再帰的に削除します。

-x 実行される詳細なコマンドを出力します。-n 出力の実行版です。

2.3.4 go fmt

読者に C/C++ の経験があればご存知かもしれませんが、コードに K&R スタイルを選択するか ANSI スタイルを選択するかは常に論争となっていました。go では、コードに標準のスタイルがあります。すでに培われた習慣やその他が原因となって我々は常に ANSI スタイルまたはその他のより自分にあったスタイルでコードを書いて来ましたが、これは他の人がコードを閲覧する際に不必要な負担を与えます。そのため go はコードのスタイルを強制し（例えば左大括弧はかならず行末に置く）、このスタイルに従わなければコンパイラが通りません。整形の時間の節約するため、go ツールは go fmt コマンドを提供しています。これはあなたの書いたコードを整形するのに役立ちます。あなたの書いたコードは標準のスタイルに修正されますが、我々は普段このコマンドを使いません。なぜなら開発ツールには一般的に保存時に自動的に整形を行ってくれるからです。この機能は実際には低レイヤでは go fmt を呼んでいますが。この次の章で 2 つのツールをご紹介します。この 2 つのツールはどれもファイルを保存する際に go fmt 機能を自動化させます。

go fmt コマンドを使うにあたって実際には gofmt がコールされますが、-w オプションが必要になります。さもなければ、整形結果はファイルに書き込まれません。gofmt -w -l src、ですべての項目を整形することができます。

go fmt は gofmt の上位レイヤーのパッケージされたコマンドです。より個人的なフォーマットスタイルが欲しい場合は gofmt(<http://golang.org/cmd/gofmt/>) を参考にしてください。

gofmt の引数紹介

- l フォーマットする必要のあるファイルを表示します。
- w 修正された内容を標準出力に書き出すのではなく、直接そのままファイルに書き込みます。
- r “a[b:len(a)] -> a[b:]” のような重複したルールを追加します。大量に変換を行う際に便利です。
- s ファイルのソースコードを簡素化します。
- d ファイルに書き込まず、フォーマット前後の diff を表示します。デフォルトは false です。
- e 全ての文法エラーを標準出力に書き出します。もしこのラベルを使わなかった場合は異なる 10 行のエラーまでしか表示しません。
- cpuprofile テストモードをサポートします。対応する cpuprofile 指定のファイルに書き出します。

2.3.5 go get

このコマンドは動的にリモートコードパッケージを取得するために用いられます。現在 BitBucket、GitHub、Google Code と Launchpad をサポートしています。このコマンドは内部で実際には 2 ステップの操作に分かれます：第 1 ステップはソースコードパッケージのダウンロード、第 2 ステップは go install の実行です。ソースコードパッケージのダウンロードを行う go ツールは異なるドメインにしたがって自動的に異なるコードツールを用います。対応関係は以下の通りです：

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

そのため、go get を正常に動作させるためには、あらかじめ適切なソースコード管理ツールがインストールされていると同時にこれらのコマンドがあなたの PATH に入っていないとなりません。実は go get は

カスタムドメインの機能をサポートしています。具体的な内容は `go help remote` を参照ください。

引数紹介：

- d ダウンロードするだけでインストールしません。
- f -u オプションを与えた時だけ有効になります。-u オプションは `import` の中の各パッケージが既に取り得されているかを検証しなくなります。ローカルに `fork` したパッケージに対して特に便利です。
- fix ソースコードをダウンロードするとまず `fix` を実行してから他の事を行うようになります。
- t テストを実行する為に必要となるパッケージも同時にダウンロードします。
- u パッケージとその依存パッケージをネットワークから強制的に更新します。
- v 実行しているコマンドを表示します。

2.3.6 go install

このコマンドは実際には内部で 2 ステップの操作に分かれます。第 1 ステップはリザルトファイルの生成（実行可能ファイルまたは a パッケージ）、第 2 ステップはコンパイルし終わった結果を `$GOPATH/pkg` または `$GOPATH/bin` に移動する操作です。

引数は `go build` のコンパイルオプションをサポートしています。みなさんは `-v` オプションだけ覚えていただければ結構です。これにより低レイヤーの実行状況をいつでも確認することができます。

2.3.7 go test

このコマンドを実行すると、ソースコードディレクトリ以下の `*_test.go` ファイルが自動的にロードされ、テスト用の実行可能ファイルが生成/実行されます。出力される情報は以下のようなものになります

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

デフォルトの状態で、オプションを追加する必要はありません。自動的にあなたのソースコードパッケージ以下のすべての `test` ファイルがテストされます。もちろんオプションを追加しても構いません。詳細は `go help testflag` を確認してください。

ここでは良く使われるオプションについてご紹介します：

- bench regexp 指定した benchmarks を実行します。例えば `-bench=.`
- cover テストカバレッジを起動します。
- run regexp regexp にマッチする関数だけを実行します。例えば `-run=Array` とすることで名前が `Array` から始まる関数だけを実行します。
- v テストの詳細なコマンドを表示します。

2.3.8 go tool

`go tool` にはいくつかのコマンドがあります。ここでは 2 つだけご紹介します。fix と vet です。

- `go tool fix .` は以前の古いバージョンを新しいバージョンに修復します。例えば、`go1` 以前の古い

バージョンのコードを go1 に焼き直したり、API を変化させるといったことです。

- `go tool vet directory|files` はカレントディレクトリのコードが正しいコードであるか分析するために使用されます。例えば `fmt.Printf` をコールする際のオプションが正しくなかったり、関数の途中で `return` されたことによって到達不可能なコードが残っていないかといった事です。

2.3.9 go generate

このコマンドは Go1.4 になって初めてデザインされました。コンパイル前にある種のコードを自動で生成する目的に使用されます。`go generate` と `go build` は全く異なるコマンドです。ソースコード中の特殊なコメントをを分析することで、対応するコマンドを実行します。これらのコマンドは明確に何の依存も存在しません。この機能を使用する場合には必ず次の事を念頭に置いてください。`go generate` はあなたの為に存在します。あなたのパッケージを使用する誰かの為のものではありません。これはある一定のコードを生成するためにあります。

簡単な例をご紹介します。例えば我々が度々 `yacc` を使ってコードを生成していたとしましょう。その場合以下のようなコマンドをいつも使用することになります：

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

`-o` は出力するファイル名を指定します。`-p` はパッケージ名を指定します。これは単独のコマンドであり、もし `go generate` によってこのコマンドを実行する場合は当然ディレクトリの任意の `xxx.go` ファイルの任意の位置に以下のコメントを一行追加します。

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

注意すべきは、`//go:generate` に空白が含まれていない点です。これは固定のフォーマットで、ソースファイルを舐める時はこのフォーマットに従って判断されます。

これにより以下のようなコマンドによって、生成・コンパイル・テストを行うことができます。もし `gopher.y` ファイルに修正が発生した場合は、再度 `go generate` を実行することでファイルを上書きすればよいことになります。

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

2.3.10 godoc

Go1.2 バージョンより以前は `go doc` コマンドがサポートされていましたが、今後は全て `godoc` コマンドに移されました。このようにインストールします `go get golang.org/x/tools/cmd/godoc`

多くの人が `go` にはサードパーティのドキュメントが必要無いと謳っています。なぜなら例えば `chm` ハンドブックのように（もっとも私はすでに `chm` マニュアルを作っていますが）この中にはとても強力なドキュメントツールが含まれているからです。

どのように対応する `package` のドキュメントを確認すればよいのでしょうか？ 例えば `builtin` パッケージであれば、`go doc builtin` と実行します。もし `http` パッケージであれば、`go doc net/http` と実行してください。パッケージの中の関数を確認する場合は `godoc fmt Printf` としてください。対応するコードを確認する場合は、`godoc -src fmt Printf` とします。

コマンドラインでコマンドを実行します。godoc -http=:ポート番号 例えば godoc -http=:8080 として、ブラウザで 127.0.0.1:8080 を開くと、golang.org のローカルの copy 版を見ることができます。これを通して pkg ドキュメントなどの他の内容を確認することができます。もしあなたが GOPATH を設定されていれば、pkg カテゴリの中で、標準パッケージのドキュメントのみならず、ローカルの GOPATH のすべての項目に関連するドキュメントをリストアップすることができます。これはグレートファイアーウォールの中にいるユーザにとっては非常にありがたい選択です。

2.3.11 その他のコマンド

go は他にも様々なツールを提供しています。例えば以下のツール

```
go version は go の現在のバージョンを確認します。  
go env は現在の go の環境変数を確認します。  
go list は現在インストールされている全ての package をリストアップします。  
go run は Go プログラムのコンパイルと実行を行います。
```

これらのツールはまだ多くのオプションがあり、ひとつひとつはご紹介しませんが、ユーザは go help コマンドで更に詳しいヘルプ情報を取得することができます。

2.4 Go の開発ツール

本章ではいくつかの開発ツールをご紹介します。これらはすべて自動化を備えており、fmt 機能を自動化します。なぜならこれらはすべてクロスプラットフォームであり、そのためインストール手順といったものはすべて同じものです。

2.4.1 LiteIDE

LiteIDE は Go 言語の開発に特化したクロスプラットフォームの軽量統合開発環境 (IDE) です。visualfc で書かれています。

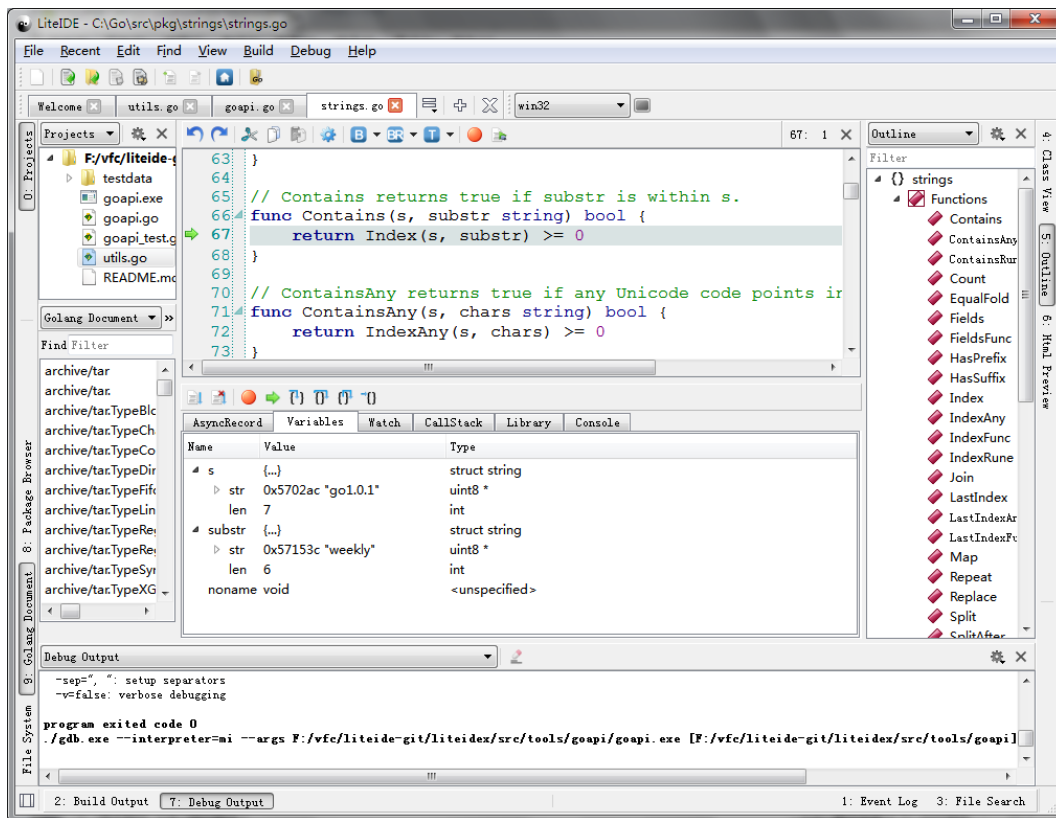


図 4 LiteIDE のメイン画面

LiteIDE の主な特徴

- 主なオペレーティングシステムのサポート
 - Windows
 - Linux
 - MacOS X
- Go コンパイル環境の管理と切り替え
 - 複数の Go コンパイル環境の管理と切り替え
 - Go 言語のクロスコンパイルのサポート
- Go 標準と同じ項目管理方式
 - GOPATH に基づいたパッケージブラウザ
 - GOPATH に基づいたコンパイルシステム
 - GOPATH に基づいたドキュメント検索
- Go 言語の編集サポート
 - クラスブラウザとアウトライン表示
 - Gocode(コード自動作成ツール) の完全なサポート
 - Go 言語ドキュメントと Api 高速検索
 - コード表現情報の表示 F1

- ソースコード定義とジャンプのサポート F2
- Gdb ブレークポイントとテストサポート
- gofmt 自動整形のサポート
- その他の特徴
 - 多言語メニューのサポート
 - 完全にプラグブルな構成
 - エディタのカラーリングサポート
 - Kate に基づいた文法表示サポート
 - 全文に基づく単語の自動補完
 - キーボードショートカットのバインディングサポート
 - Markdown ドキュメントの編集サポート
 - * リアルタイムプレビューと表示の同期
 - * カスタム CSS 表示
 - * HTML 及び PDF ドキュメントのエクスポート
 - * HTML/PDF ドキュメントへの変換とマージ

LiteIDE インストール設定

- LiteIDE インストール
 - ダウンロード <http://sourceforge.net/projects/liteide/files/>
 - ソースコード <https://github.com/visualfc/liteide>
- コンパイル環境設定

自身のシステムの要求にしたがって LiteIDE が現在使用している環境変数を切り替えまたは設定します。

Windows オペレーティングシステムの 64bit Go 言語の場合、ツール欄の環境設定のなかで win64 を選択し、‘編集環境‘をクリックして LiteIDE から win64.env ファイルを編集します。

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

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

この中の ‘GOROOT=c:\go’ を現在の Go のインストールパスに修正し、保存するだけです。もし MinGW64 があれば、 ‘c:\MinGW64\bin’ を PATH の中に入れて、go による gcc のコールで CGO コンパイラのサポートを利用することができます。

Linux オペレーティングシステムで 64bit Go 言語の場合、ツール欄の環境設定の中から linux64 を選び、‘編集環境‘をクリックし

LiteIDEからlinux64.envファイルを編集します。

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

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

この中の‘GOROOT=\$HOME/go’を現在のGoのインストールパスに修正して保存します。

- GOPATH 設定

Go言語のツールキーはGOPATH設定を使用します。Go言語開発のプロジェクトのパスリストです。コマンドライン(LiteIDEでは‘Ctrl+,’を直接入力できます)で‘go help gopath’を入力するとGOPATHドキュメントを素早く確認できます。

LiteIDEでは簡単に確認でき、GOPATHを設定することができます。
‘メニュー-確認-GOPATH’設定を通じて、システム中に存在するGOPATHリストを確認することができます。
同時に必要な追加項目にそってカスタムのGOPATHリストに追加することができます。

2.4.2 Sublime Text

ここでは Sublime Text 2 (以下「Sublime」) + GoSublime の組み合わせをご紹介します。なぜこの組み合わせなのでしょう？

- コード表示の自動化、以下の図の通り

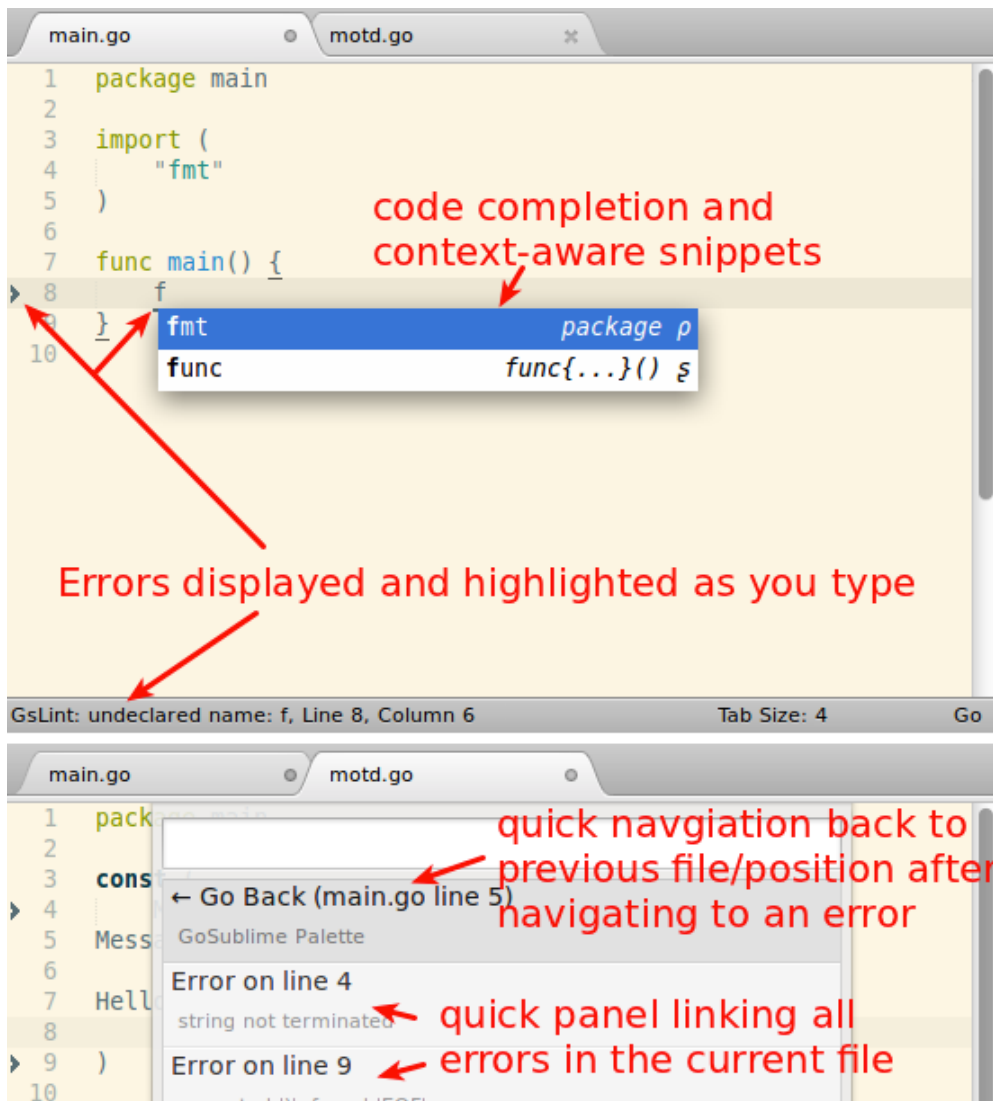


図5 sublime コードの自動化画面

- 保存した時にはコードが自動的に整形されています。あなたの書いたコードをより美しく Go の標準に合うよう仕上げてくれます。
- プロジェクト管理のサポート

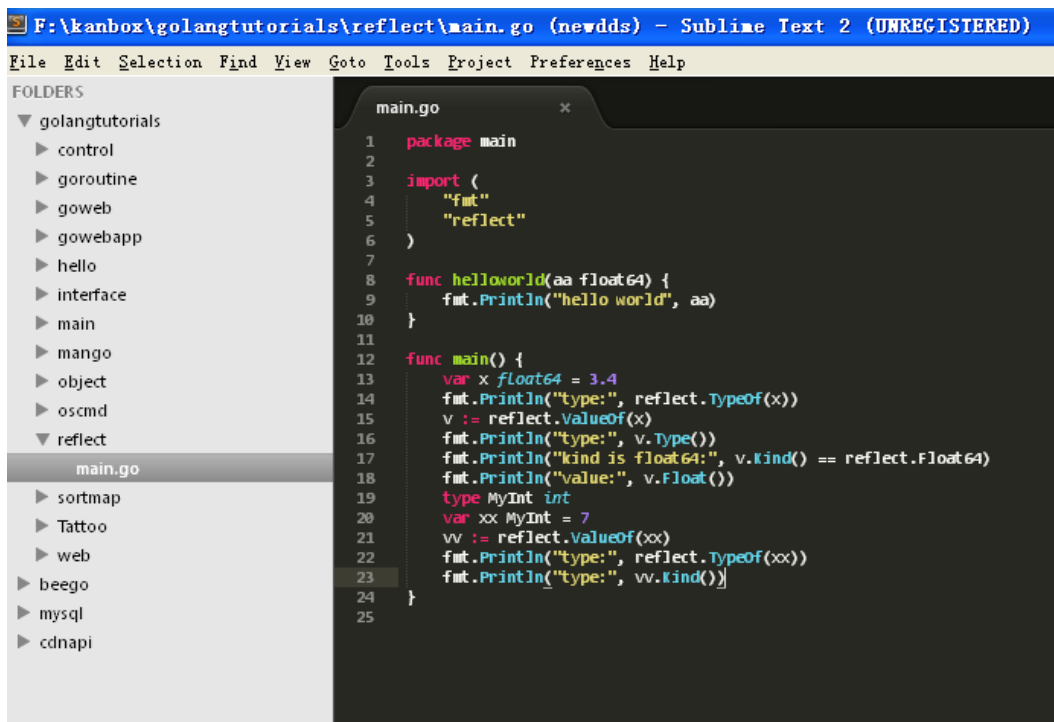


図 6 sublime プロジェクト管理画面

- 文法のハイライトサポート
- Sublime Text 2 はフリーで使用できます。保存回数が一定の量を超えると購入するか dialog が現れるので、継続利用をキャンセルするをクリックします。正式登録版とは何の変わりもありません。

次はどのようにインストールするかご説明します。Sublime ダウンロードします。

自分のシステムに合わせて対応するバージョンをダウンロードし、Sublime を開きます。Sublime に詳しくない方はまず Sublime Text 2 入門とテクニック (<http://lucifr.com/139225/sublime-text-2-tricks-and-tips/>) の文章を読んでみてください。

1. 開いた後、Package Control をインストールします。Ctrl+‘でコマンドラインを開き、以下のコードを実行します：

```
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'
```

この時 Sublime を再度開き直してください。メニュー欄に一つ項目が増えているのがお分かりいただけるかと思います。これで Package Control が正しくインストールされました。

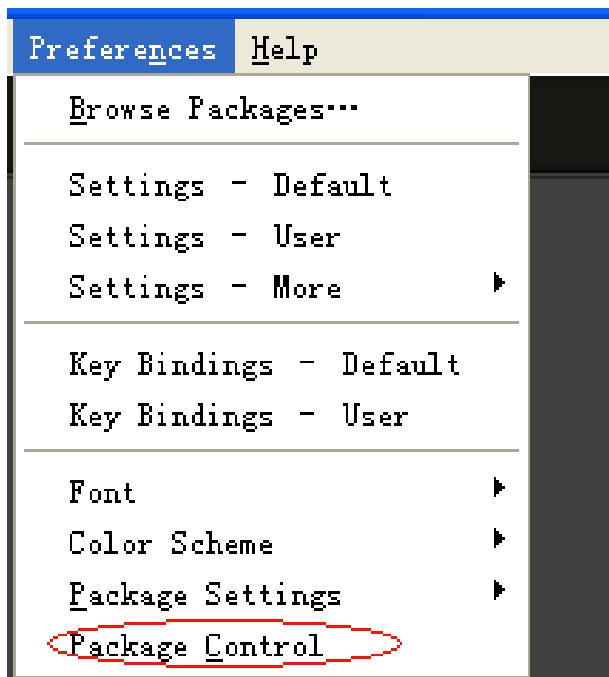


図 7 sublime パッケージ管理

2. インストールが完了すると Sublime のプラグインをインストールできます。GoSublime, SidebarEnhancements と Go Build をインストールする必要があるので、プラグインをインストールしたあと Sublime を再起動させて有効にしてください。Ctrl+Shift+p で Package Control を開き、pcip を入力します。(これは”Package Control: Install Package”と省略されます)。この時、左下のコーナーに現在読み込んでいるパッケージデータが表示されます。完了すると下のよう画面になります。

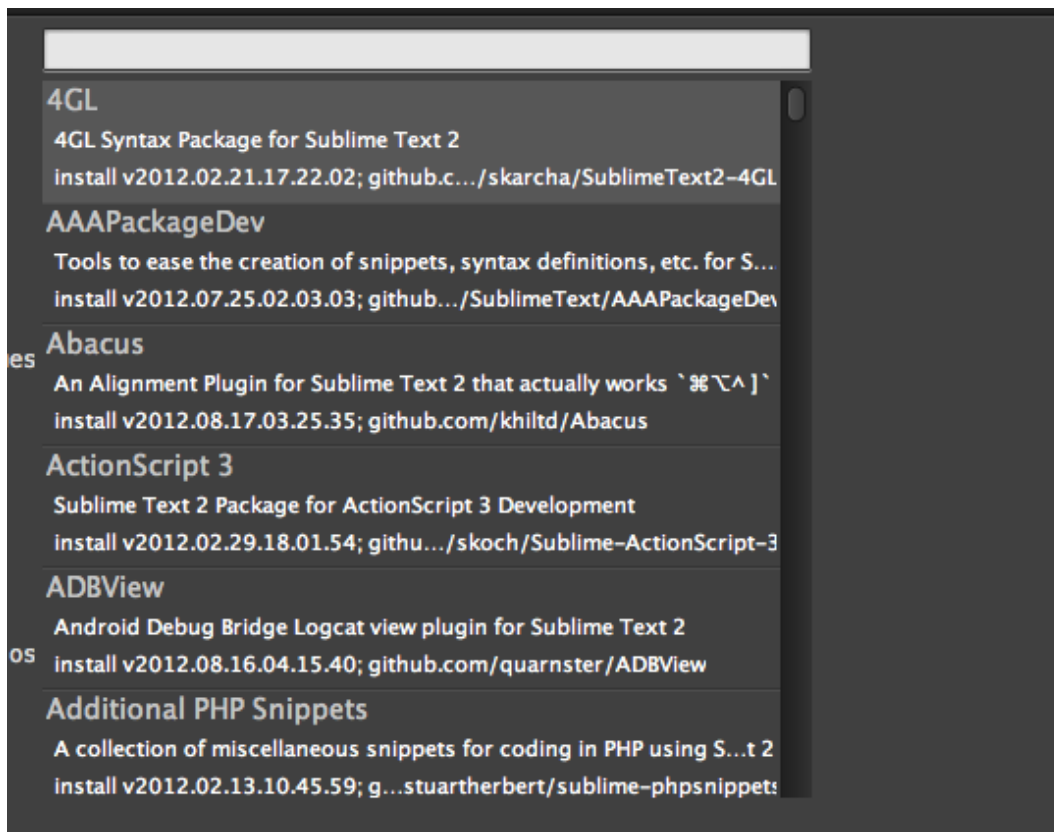


図 8 sublime プラグインのインストール画面

この時、GoSublime と入力し、「確認」をクリックするとインストールが始まります。同じように SidebarEnhancements と Go Build にも行います。

3. インストールが成功したかテストします。Sublime を開き、main.go を開いて文法がハイライトされているのをご確認ください。import を入力してコードの自動表示がされます。import "fmt"のあとに fmt. を入力すると自動的に関数の候補が現れます。

もしすでにこのような表示がされる場合は、インストールが成功しており、自動補完が完了しています。もしこのような表示がなされない場合、あなたの\$PATH が正しく設定されていないのかもしれませんが。ターミナルを開き、gocode を入力して、正しく実行できるか確認してください。もしダメであれば\$PATH が正しく設定されていません。(XP 向け) たまたまターミナルでの実行が成功することもあります。しかし sublime は何も知らせてくれないかデコードエラーが発生します。sublime text3 と convert utf8 プラグインを試してみてください。

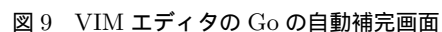
4. MacOS ではすでに\$GOROOT, \$GOPATH, \$GOBIN が設定されていても自動的にはどうすればよいかわ教えてくれません。

sublime にて command + 9 を押し、env を入力して\$PATH, \$GOROOT, \$GOPATH, \$GOBIN といった変数を確認します。もしなければ、以下の手順に従ってください。

まず下のシンボリックリンクを作成し、Terminal で直接 sublime を起動します

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


Vim は vi から発展したテキストエディタです。コード補完、コンパイルまたエラージャンプなど、プログラミングに特化した機能が豊富です。広くプログラマに使用されています。



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

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

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

4. Gocode を設定します。

```
~ 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 の2つのパラメータの意味を説明します：

propose-builtins：は Go のビルトイン関数を補完するかです。タイプは定数です。デフォルトは false で、表示しません。

lib-path:デフォルトで、gocode は\$GOPATH/pkg/\$GOOS.\$GOARCH と\$GOROOT/pkg/\$GOOS.\$GOARCH ディレクトリのパッケージを検索するだけです。当然この設定には私達の外側の lib を検索できるようなパスを設定することができます。

5. おめでとうございます。インストール完了です。あなたは今から :e main.go で Go で開発する面白さを体験することができます。

より多くの VIM の設定は、リンク (<http://monnand.me/p/vim-golang-environment/zhCN/>) をご参照ください。

2.4.4 Emacs

Emacs は伝説の神器です。彼女はエディタであるだけでなく、統合環境でもあります。または開発環境の集大成と呼んでもよいかもしれません。これらの機能はユーザの身を万能のオペレーティングシステムに置きます。

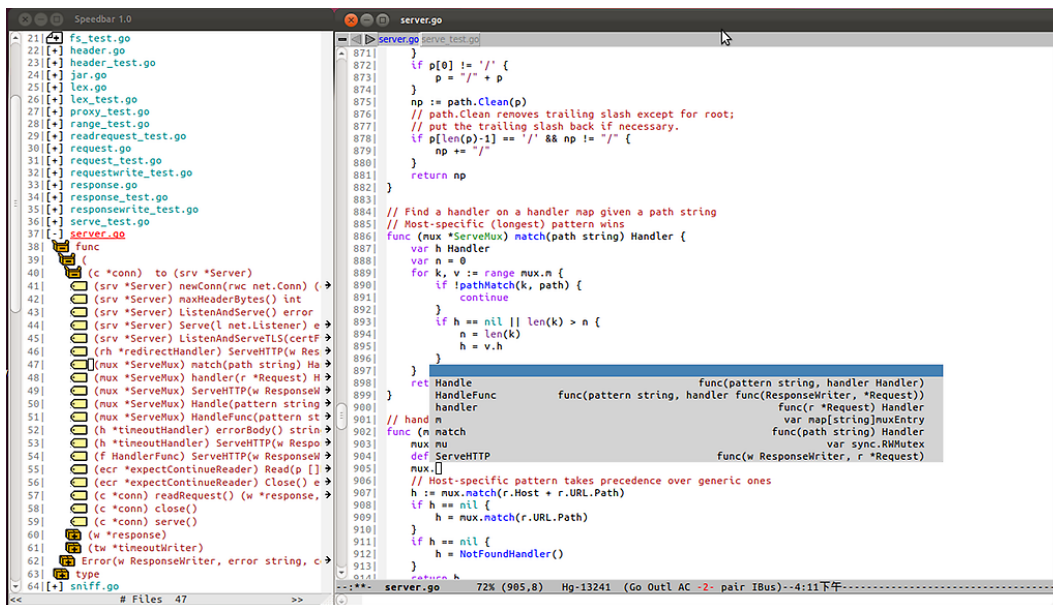


図 10 Emacs で Go を編集するメイン画面

1. Emacs のハイライト表示設定

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

2. Gocode をインストール

```
go get -u github.com/nsf/gocode
gocodeはデフォルトで '$GOBIN' の下にインストールされます。
```

3. Gocode を設定

```
~ 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"
// あなたのパスに置き換えてください。
lib-path "/home/border/gocode/pkg/linux_amd64"
~ gocode set
propose-builtins true
lib-path "/home/border/gocode/pkg/linux_amd64"
```

4. Auto Completion をインストールする必要があります。

AutoComplete をダウンロードして解凍します。

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

/.emacs ファイルを設定します。

```
;; 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)
```

詳細情報はこちらを参考にしてください: <http://www.emacswiki.org/emacs/AutoComplete>

5. .emacs を設定します。

```
;; go lang 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\\.\\|ty
(outline-minor-mode 1)
(local-set-key "\M-a" 'outline-previous-visible-heading)
(local-set-key "\M-e" 'outline-next-visible-heading)
;; Menu bar
(require 'easymenu)
(defconst go-hooked-menu
  '("Go tools"
    ["Go run buffer" go t]
    ["Go reformat buffer" go-fmt-buffer t]
    ["Go check buffer" go-fix-buffer t]))
(easy-menu-define
  go-added-menu
  (current-local-map)
  "Go tools"
  go-hooked-menu)

;; Other
(setq show-trailing-whitespace t)
))

;; helper function
(defun go ()
  "run current buffer"
  (interactive)
  (compile (concat "go run " (buffer-file-name))))

;; helper function
(defun go-fmt-buffer ()
  "run gofmt on current buffer"
  (interactive)
  (if buffer-read-only
    (progn
      (ding)
      (message "Buffer is read only"))
    (let ((p (line-number-at-pos))
          (filename (buffer-file-name))
          (old-max-mini-window-height max-mini-window-height))
      (show-all)
      (if (get-buffer "*Go Reformat Errors*")
        (progn
          (delete-windows-on "*Go Reformat Errors*")
          (kill-buffer "*Go Reformat Errors*"))
        (setq max-mini-window-height 1)
        (if (= 0 (shell-command-on-region
                  (point-min) (point-max)
                  "gofmt" "*Go Reformat Output*" nil "*Go Reformat Errors*" t))
          (progn

```

```

        (erase-buffer)
        (insert-buffer-substring "*Go Reformat Output*")
        (goto-char (point-min))
        (forward-line (1- p)))
(with-current-buffer "*Go Reformat Errors*"
(progn
  (goto-char (point-min))
  (while (re-search-forward "<standard input>" nil t)
    (replace-match filename))
    (goto-char (point-min))
    (compilation-mode))))
(setq max-mini-window-height old-max-mini-window-height)
(delete-windows-on "*Go Reformat Output*")
(kill-buffer "*Go Reformat Output*"))))
;; helper function
(defun go-fix-buffer ()
  "run gofix on current buffer"
  (interactive)
  (show-all)
  (shell-command-on-region (point-min) (point-max)
    "go tool fix -diff"))

```

6. おめでとうございます。今からあなたはこの神器を使って Go 開発の楽しみを体験できます。デフォルトの speedbar は閉じています。もし開く場合は ;; (speedbar 1) の前のコメントを取り去るか、M-x speedbar を手動で起動してください。

2.4.5 Eclipse

Eclipse も非常によく使われる開発ツールです。以下では Eclipse を使ってどのように Go プログラムを編集するかご紹介します。

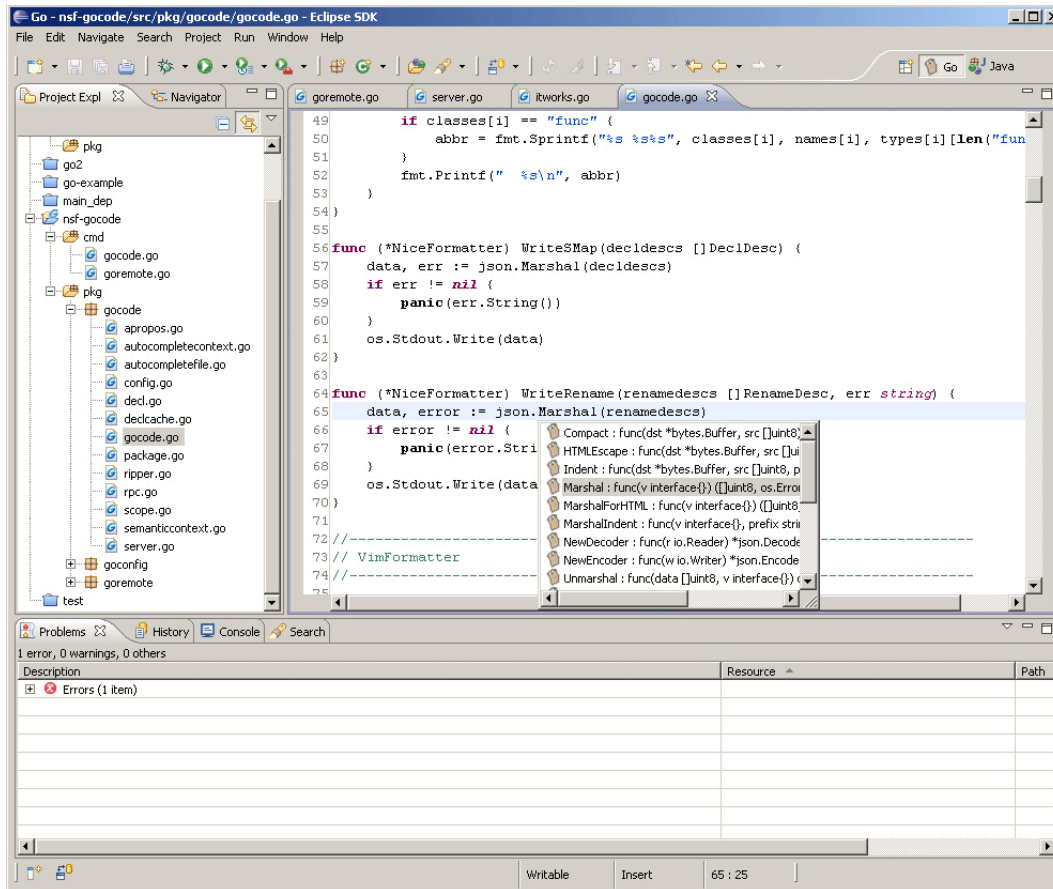


図 11 Eclipse での Go 編集のメイン画面

1. まず Eclipse をダウンロードしてインストールします。
2. goclipse プラグインをダウンロードします。
<http://code.google.com/p/goclipse/wiki/InstallationInstructions>
3. gocode をダウンロードして、go のコード補完を表示させます。

gocode の github アドレス :
<https://github.com/nsf/gocode>
 windows では git をインストールする必要があります。通常は
 [msysgit](<https://code.google.com/p/msysgit/>) を使います。
 cmd でインストールを行います :
 go get -u github.com/nsf/gocode
 以下のコードをダウンロードし、直接 go build でコンパイルしても
 かまいません。
 この場合は gocode.exe が生成されます。

4. MinGW をダウンロードして要求に従いインストールしてください。
5. プラグイン設定
 Windows->Reference->Go
 (a) Go のコンパイラを設定します。

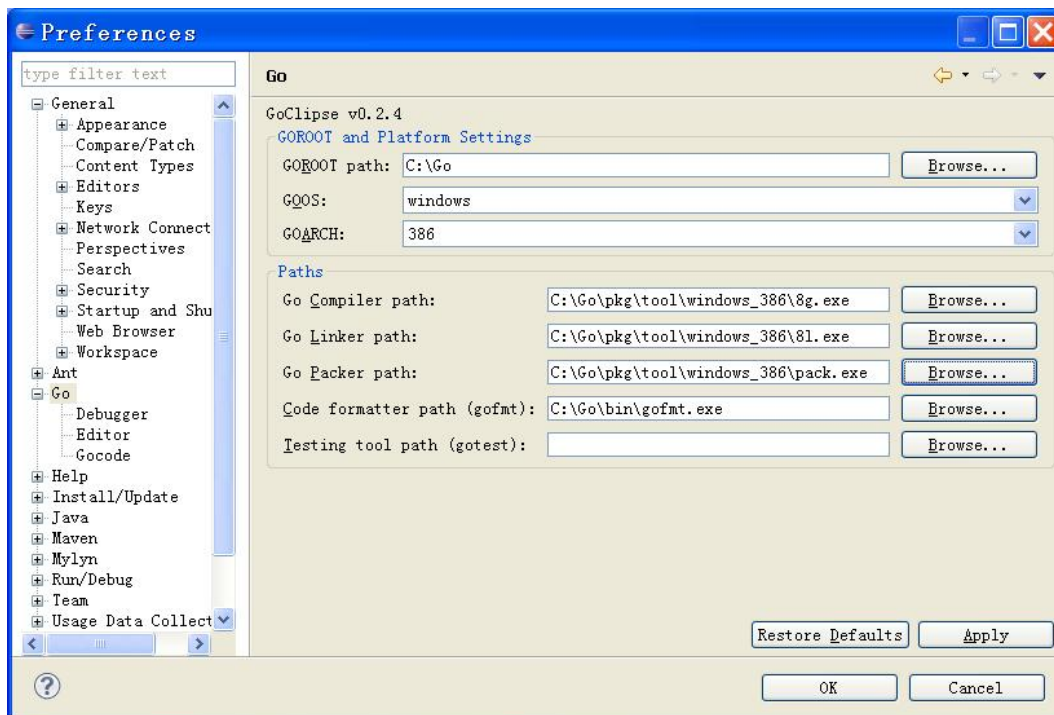


図 12 Go の基本情報を設定します。

(b) Gocode を設定します (オプション、コード補完)。Gocode のパスは事前に生成した gocode.exe ファイルを設定します。

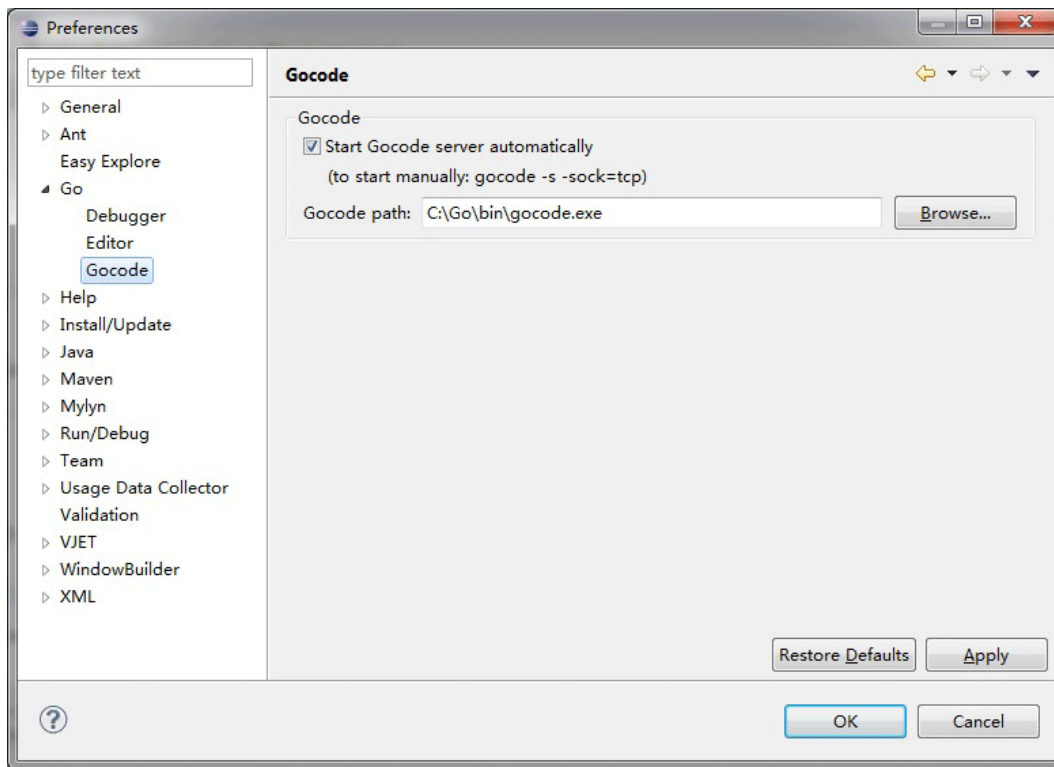


図 13 gocode 情報を設定します。

(c) GDB を設定します (オプション、テスト用) GDB のパスは MingGW のインストールディレクトリ下の gdb.exe ファイルを設定します。

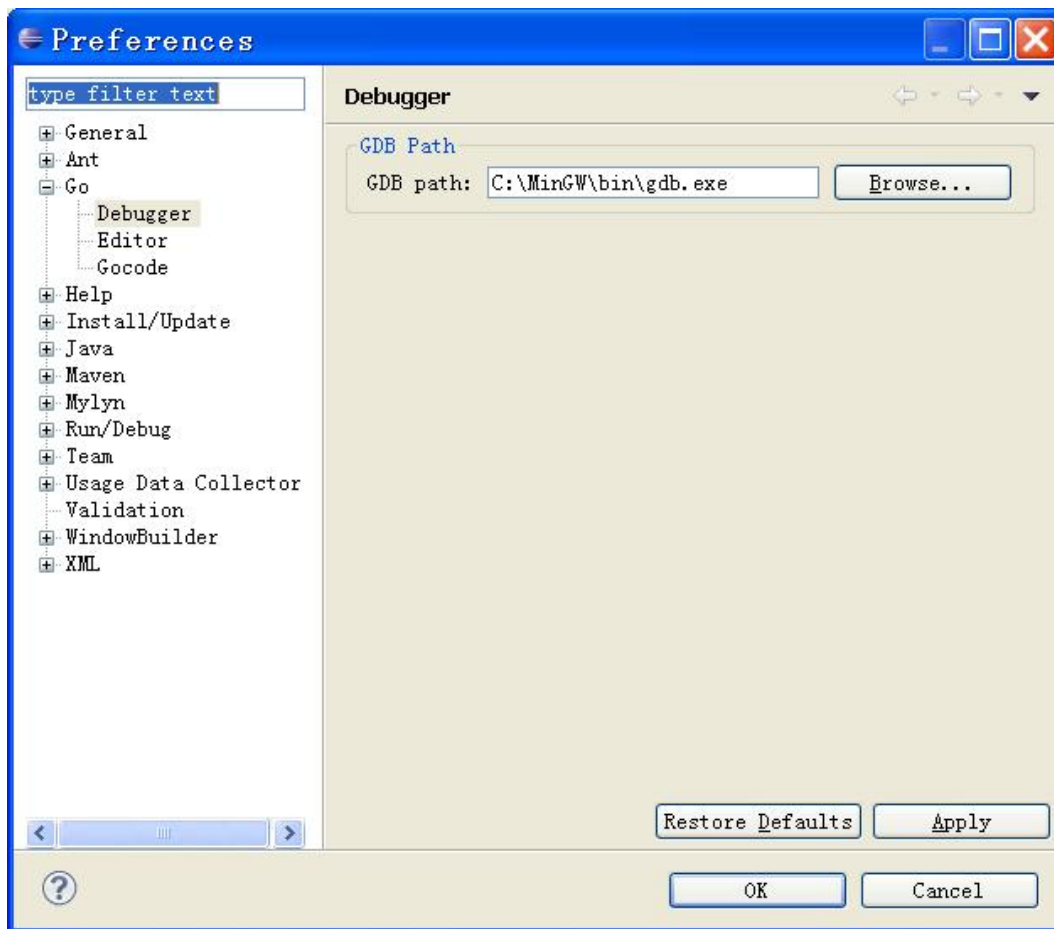


図 14 GDB 情報の設定

6. テストが成功するか

go プロジェクトを一つ新規作成して、hello.go を作成します：

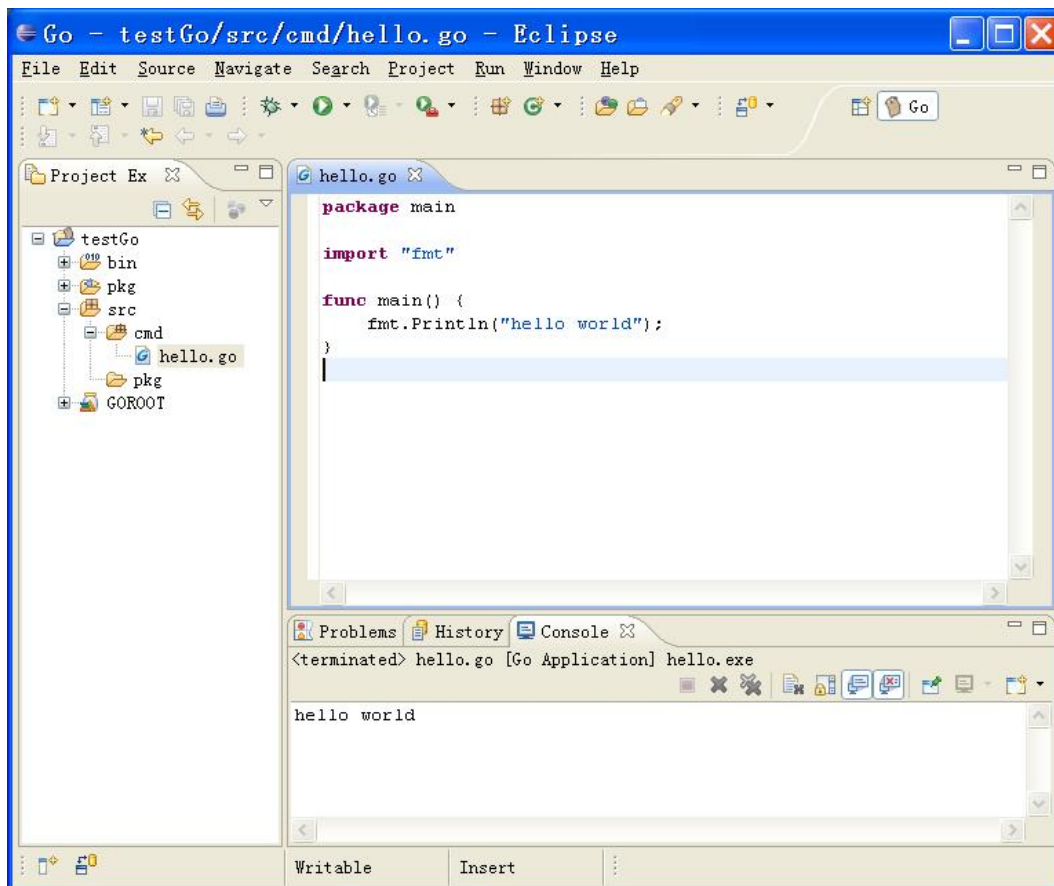


図 15 プロジェクトの新規作成とファイルの編集

7. テストの実行 (console でコマンドを入力する必要があります):

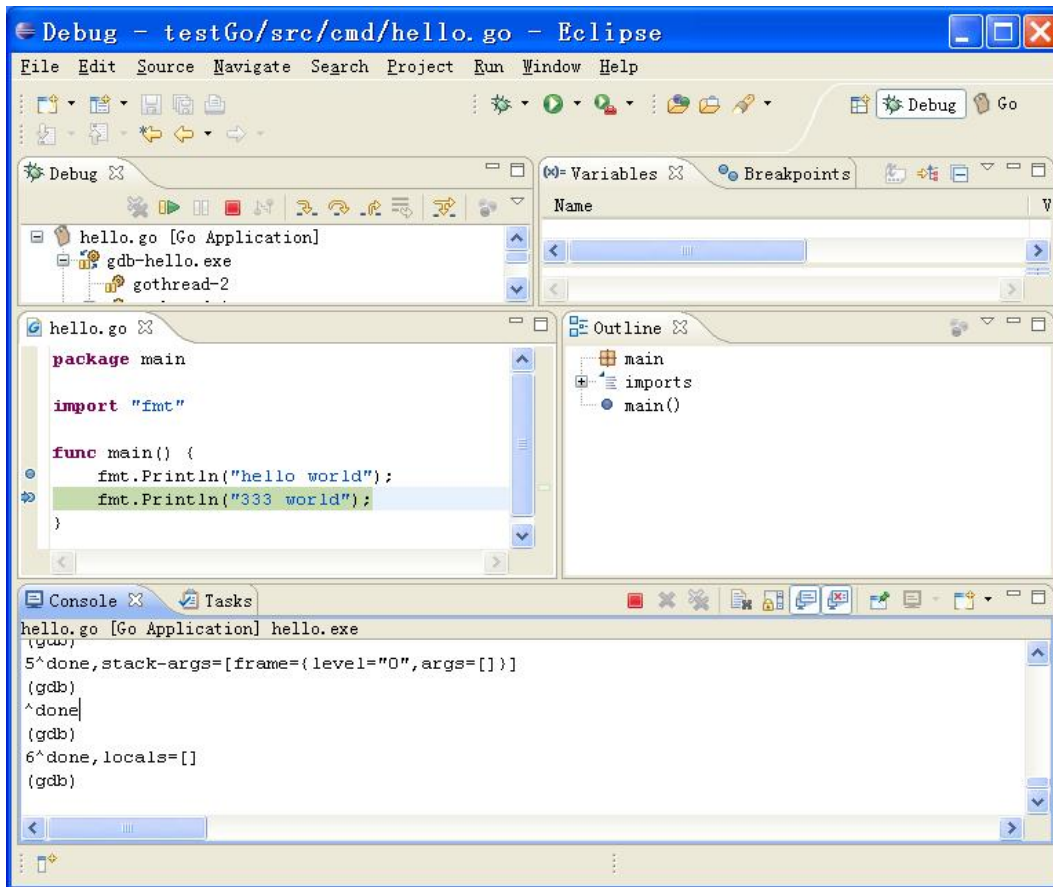
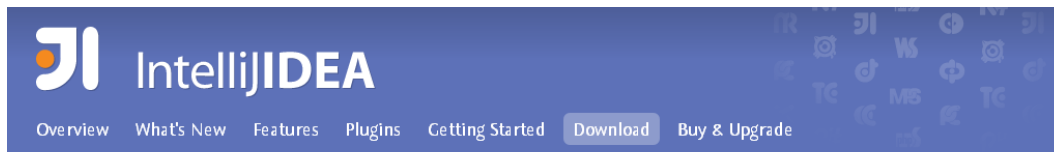


図 16 Go プログラムのテスト

2.4.6 IntelliJ IDEA

Java に親しい読者は idea に詳しいことでしょう。idea はプラグインを通して go 言語のシンタックスハイライト、コード補完およびビルドをサポートしています。

1. idea を先にダウンロードします。idea はマルチプラットフォームをサポートしています：win, mac, linux、もしお金があれば正式版を購入します、もし無ければ、コミュニティの無料版を使ってください。Go 言語を開発するだけであれば無料版で十分事足ります。



Download IntelliJ IDEA 12

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

Version: 12.0.2 Build: 123.123 Released: January 15, 2013 [System requirements](#) [Installation Instructions](#)

Ultimate Edition Free 30-day trial

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

Java EE, Spring/Hibernate and other technologies support

Deployment and debugging with most application servers

Duplicate code search, dependency structure matrix, etc.

[Download Now](#)

Community Edition FREE

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

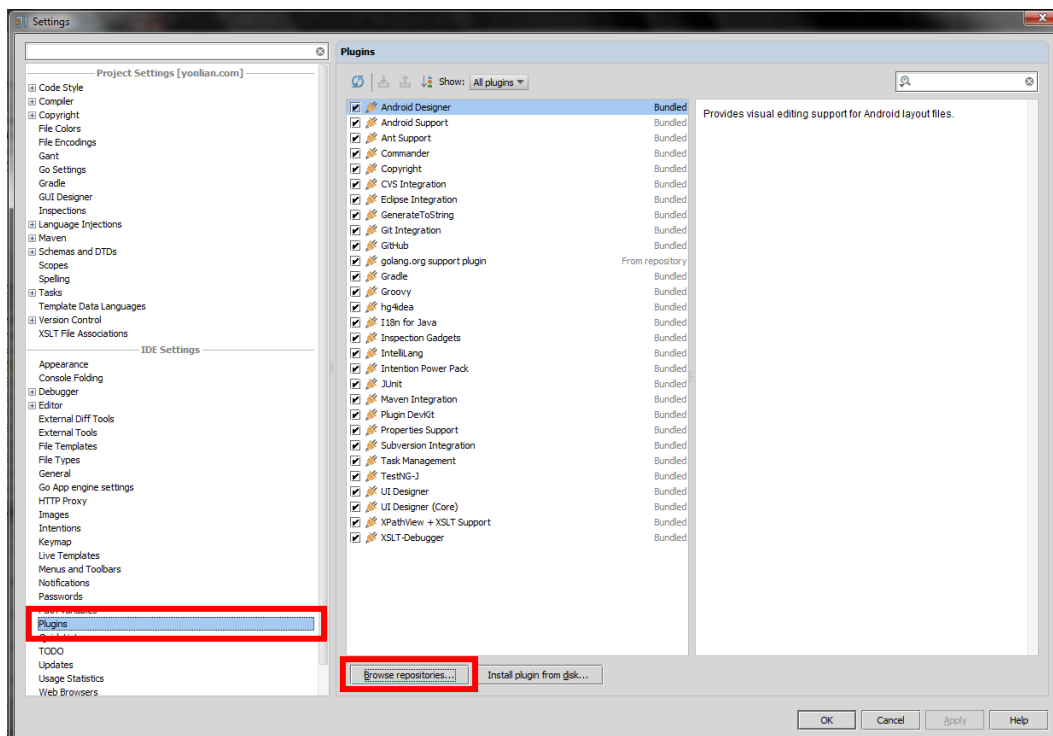
Powerful environment for building **Google Android** apps

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

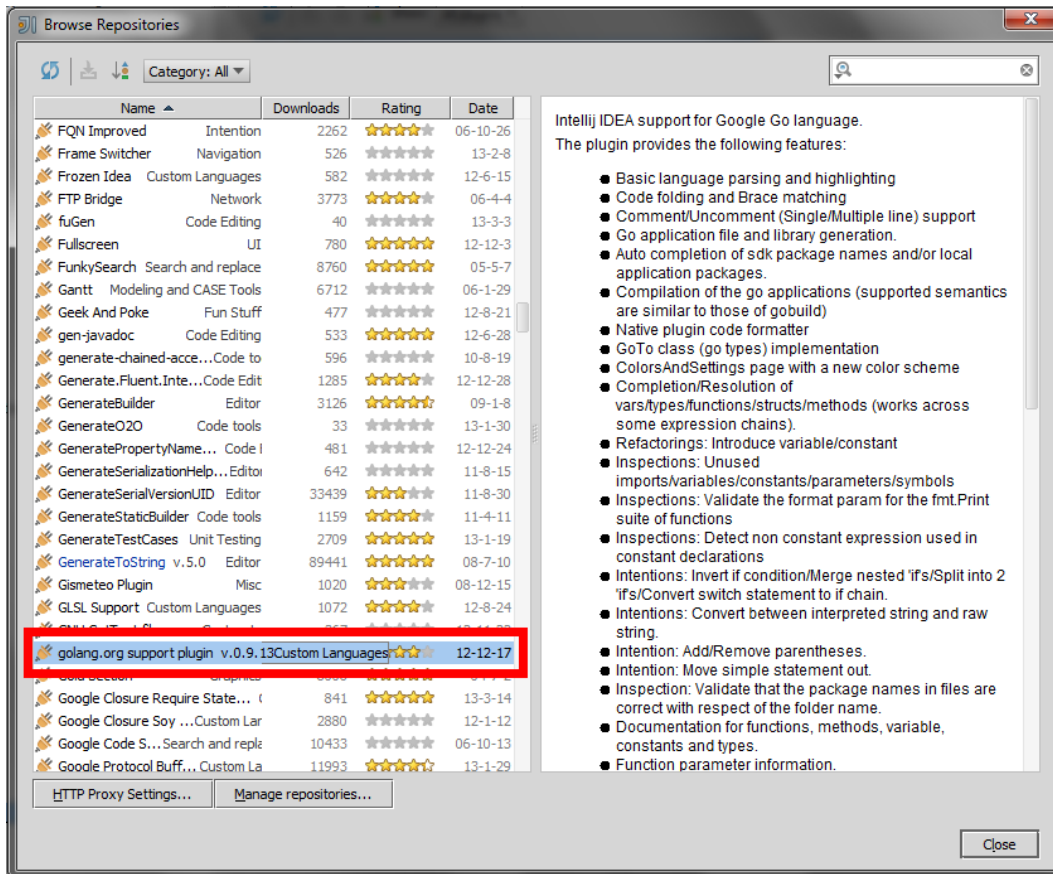
Free and open-source ([get the source code](#))

[Download Now](#)

2. Go プラグインをインストールし、File メニューの Setting をクリックします。Plugins を探したら、Browser repo ボタンをクリックします。中国国内のユーザはおそらくエラーが出るかもしれませんが、自分で解決してくれよな。

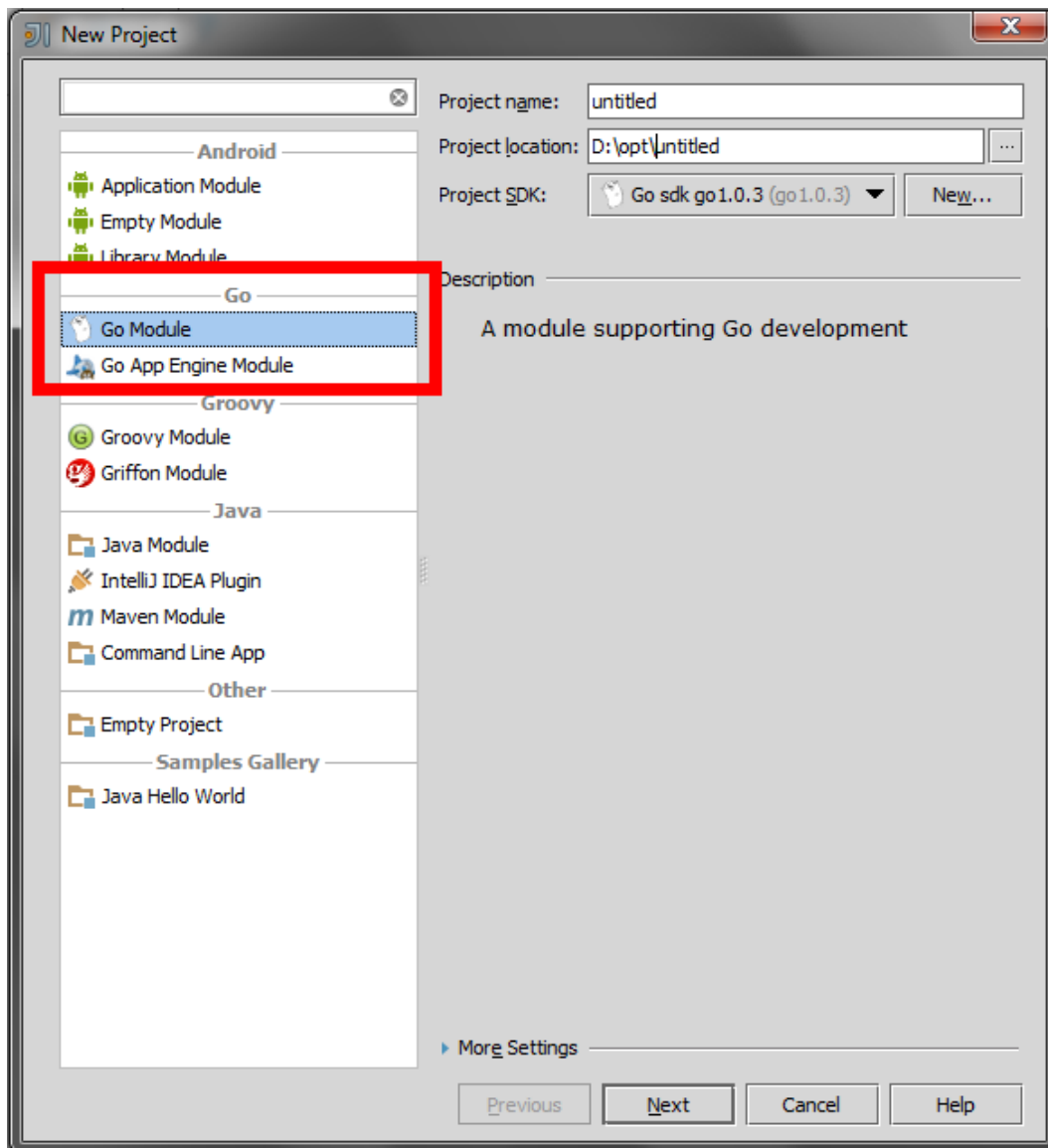


3. この時いくつかのプラグインが見つかります。Golang を検索して、download and install をダブルクリックしてください。golang の行末に Downloaded の表示が現れるのを待って、OK をクリックします。



その後 Apply をクリックすると、IDE が再起動を要求します。

4. 再起動が完了し、新規プロジェクトを作成すると、golang プロジェクトが作成可能であることがお分りいただけるかとおもいます：



次に、go sdk の場所を入力するよう促されるかもしれません。普段はいつも C:\Go にインストールされています。Linux と mac は自分のインストールディレクトリの設定にしたがって、ディレクトリを選択すれば大丈夫です。

2.5 まとめ

この章では主にどのようにして Go をインストールするかについてご紹介しました。Go は 3 つの種類のインストール方法があります：ソースコードインストール、標準パッケージインストール、サードパーティツールによるインストールです。インストール後開発環境を整え、ローカルの \$GOPATH を設定します。\$GOPATH 設定を通じて読者はプロジェクトを作成することができます。次にどのようにプロジェクトをコンパイルするのか説明しました。アプリケーションのインストールといった問題はたくさんの Go コマンドを使用する必要

があります。そのため、Go で日常的に用いられるコマンドツールについてもご説明しました。コンパイル、インストール、整形、テストなどのコマンドです。最後に Go の開発ツールについてご紹介しました。現在多くの Go の開発ツールには : LiteIDE、sublime、VIM、Emacs、Eclipse、Idea といったツールがあります。読者は自分が一番慣れ親しんだツールを設定することができます。便利なツールで素早く Go アプリケーションを開発できるよう願っています。

3 Go 言語の基礎

Go は C に似たコンパイル型言語です。ですが、このコンパイル速度は非常に速く、この言語のキーワードもたったの 25 個です。英文よりも少し少なく勉強するにはかなり簡単です。まずはこれらのキーワードがどのようなものか見てみることにしましょう :

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

この章では、この言語の基礎勉強にあなたを連れていきます。各章の紹介を通じて、Go の世界がどれほどまでに簡潔で絶妙にデザインされているかお分かりいただけるはずです。Go を書くことはとても楽しいことです。後から振り返ると、この 25 個のキーワードがどれだけフレンドリーか理解するはずです。

3.1 こんにちは、Go

アプリケーションを書き始める前に、まず基本となるプログラムから始めます。家を建てようとする前に建物の基礎がどういったものかわからないのと同じように、プログラムの編集もどのように始めたらよいかわからないものです。そのため、本章では、最も基本的な文法を学習し、Go プログラムを実行してみます。

3.1.1 プログラム

これは伝統なのですが、大部分の言語を学習するときは、どのようにして hello world を出力するかというプログラムを書くことを学びます。

用意はいいですか？ Let's Go!

```
package main

import "fmt"

func main() {
    fmt.Printf("Hello, world or ^^e4^^bd^^a0好, 世界
               or      μ ^^cc^^81      ^^c3^^b3      μ or こんにちははせかい\n")
}
```

```
Hello, world or ^^e4^^bd^^a0好, 世界
or      μ ^^cc^^81      ^^c3^^b3      μ or こんにちははせかい
```

3.1.2 説明

まず我々はある概念を理解する必要があります。Go プログラムは package で構成されています。

package <pkgName> (我々の例では package main) の 1 行は現在のファイルがどのパッケージの属しているかを表しています。またパッケージ main はこれが独立して実行できるパッケージであることを示しています。コンパイル後実行可能ファイルが生成されます。main パッケージを除いて、他のパッケージは最後には *.a というファイルが生成され (パッケージファイルとも呼ばれます。) \$GOPATH/pkg/\$GOOS_\$GOARCH に出力されます。(Mac では \$GOPATH/pkg/darwin_amd64 になります。)

それぞれの独立して実行できる Go プログラムは必ず package main の中に含まれます。この main パッケージには必ずインターフェースとなる main 関数が含まれます。この関数には引数がなく、戻り値もありません。

Hello, world... と出力するために、我々は Printf 関数を用います。この関数は fmt パッケージに含まれるため、我々は 3 行目でシステム固有の fmt パッケージを導入しています: import "fmt".

パッケージの概念は Python の package に似ています。これらには特別な利点があります: モジュール化 (あなたのプログラムを複数のモジュールに分けることができます) と再利用性 (各モジュールはすべて他のアプリケーションプログラムで再利用することができます)。ここではパッケージの概念を理解するにとどめ、あとで自分のパッケージを書くことにしましょう。

5 行目では、キーワード func を通じて main 関数を定義しています。関数の中身は {} (大括弧) の中に書かれます。我々が普段 C や C++、Java を書くのと同じです。

main 関数にはなんの引数もありません。あとでどのように引数があったり、0 個または複数の値を返す関数を書くか学ぶことにしましょう。

6 行目では、fmt パッケージに定義された Printf 関数をコールします。この関数は <pkgName>.<funcName> の形式でコールされます。この点は Python とよく似ています。

上述の通り、パッケージ名とパッケージが存在するディレクトリは異なっていておかまいません。ここでは <pkgName> がディレクトリ名ではなく package <pkgName> で宣言されるパッケージ名とします。

最後に、我々が出力した内容に多くの非 ASCII コードが含まれていることにお気づきかもしれません。実際、Go は生まれながらにして UTF-8 をサポートしており、いかなる文字コードも直接出力することができます。UTF-8 の中の任意のコードポイントを識別子にしても構いません。

3.1.3 結論

Go は package (Python のモジュールに似ています) を使用してコードを構成します。main.main() 関数 (この関数は主にメインパッケージにあります) は個別に独立した実行可能プログラムのインターフェースになります。Go は UTF-8 文字列と識別子 (なぜなら UTF-8 の発明者も Go の発明者と同じだからです。) を使用しますので、はじめから多言語サポートを有しています。

3.2 Go の基礎

この節では変数、定数、Go の内部クラスの定義と、Go プログラムの設計におけるテクニックをご紹介します。

3.2.1 変数の定義

Go 言語では変数は数多くの方法で定義されます。

var キーワードを使用することは Go の最も基本的な変数の定義方法です。C 言語と異なり、Go では変数の型を変数の後に置きます。

```
// "variableName" という名前で定義します。型は "type" です。  
var variableName type
```

複数の変数を定義します。

```
// すべて "type" 型の 3 つの変数を定義します。  
var vname1, vname2, vname3 type
```

変数を定義し、初期化します。

```
// "variableName" の変数を "value" で初期化します。型は "type" です。  
var variableName type = value
```

複数の変数を同時に初期化します。

```
/*  
    すべてが "type" 型となる変数をそれぞれ定義し、個別に初期化を行います。  
    vname1 は v1, vname2 は v2, vname3 は v3  
*/  
var vname1, vname2, vname3 type = v1, v2, v3
```

あなたは上述の定義が面倒だと思いますか？大丈夫、Go 言語の設計者もわかっています。少し簡単に書くこともできます。直接型の宣言を無視することができるので、上のコードはこのようにも書けます：

```
/*  
    3 つの変数を定義し、それぞれ個別に初期化する。  
    vname1 は v1, vname2 は v2, vname3 は v3  
    このあと Go は代入される値の肩に従ってそれぞれ初期化を行います。  
*/  
var vname1, vname2, vname3 = v1, v2, v3
```

これでもまだ面倒ですか？ええ、私もそう思います。更に簡単にしてみましょう。

```
/*  
    3 つの変数を定義し、それぞれ個別に初期化します。  
    vname1 は v1, vname2 は v2, vname3 は v3  
    コンパイラは初期化する値に従って自動的にふさわしい型を導き出します。  
*/  
vname1, vname2, vname3 := v1, v2, v3
```

これなら非常に簡潔になったでしょう？`:=`の記号は `var` と `type` に直接取って代わるものです。これらの形式を短縮宣言と呼びます。ただしこれにはひとつ制限があります。これらは関数の内部でしか使用できません。関数の外で使用するとコンパイルが通らなくなります。そのため、一般的には `var` 方式でグローバル変数が定義されます。

`_` (アンダースコア) は特別な変数名です。どのような値もすべて捨てられてしまいます。この例では 35 という値を `b` に与えますが、同時に 34 は失われてしまいます。

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

Go はすでに宣言されている未使用の変数をコンパイル時にエラーとして出力します。例えば下のコードはエラーを一つ生成します。`i` は宣言されましたが使用されていません。

```
package main

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

3.2.2 定数

いわゆる定数というのは、プログラムがコンパイルされる段階で値が決定されます。そのため、プログラムが実行される時には値の変更は許されません。定数には数値、`bool` 値または文字列等の型を定義することができます。

この文法は以下の通りです：

```
const constantName = value
//もし必要であれば、定数の型を明示することもできます：
const Pi float32 = 3.1415926
```

ここでは定数の宣言の例を示します：

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

Go の定数は一般的なプログラミング言語と異なり、かなり多くの小数点以下の桁を指定することができます (たとえば 200 桁など)、`float32` に自動的な 32bit への短縮を指定したり、`float64` に自動的な 64bit への短縮を指定するにはリンク (<http://golang.org/ref/spec#Constants>) をご参照ください。

3.2.3 ビルトイン基本型

Boolean Go では `bool` 値の型は `bool` です。値は `true` もしくは `false` です。デフォルト値は `false` です。

```
// コード例
var isActive bool // グローバル変数の宣言
var enabled, disabled = true, false // 型を省略した宣言
func test() {
```

```

var available bool // 通常の宣言
valid := false    // 短縮宣言
available = true   // 代入操作
}

```

数値型 整数型には符号付きと符号無しの2つがあります。Go はまた `int` と `uint` をサポートしています。この2つの型の長さは同じですが、実際の長さは異なるコンパイラによって決定されます。Go では直接 bit 数を指定できる型もあります: `rune`, `int8`, `int16`, `int32`, `int64` と `byte`, `uint8`, `uint16`, `uint32`, `uint64` です。この中で `rune` は `int32` のエイリアスです。 `byte` は `uint8` のエイリアスです。

注意しなければならないのは、これらの型の変数間は相互に代入または操作を行うことができないということです。コンパイル時にコンパイラはエラーを発生させます。

下のコードはエラーが発生します。: `invalid operation: a + b (mismatched types int8 and int32)`

```

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

```

また、`int` の長さは 32bit ですが、`int` と `int32` もお互いに利用することはできません。

浮動小数点の型には `float32` と `float64` の二種類があります (`float` 型はありません。)。デフォルトは `float64` です。

これで全てですか? No! Go は複素数もサポートしています。このデフォルト型は `complex128` (64bit 実数 + 64bit 虚数) です。もしもう少し小さいのが必要であれば、`complex64` (32bit 実数 + 32bit 虚数) もあります。複素数の形式は `RE + IMi` です。この中で `RE` が実数部分、`IM` が虚数部分になります。最後の `i` は虚数単位です。以下に複素数の使用例を示します:

```

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

```

文字列 前の章で述べた通り、Go の文字列はすべて UTF-8 コードが採用されています。文字列は一对のダブルクォーテーション (`"`) またはバッククォート (```) で括られることで定義されます。この型は `string` です。

```

//コード例
var frenchHello string // 文字列変数の宣言の一般的な方法
var emptyString string = ""
// 文字列変数を一つ宣言し、空文字列で初期化する。
func test() {
    no, yes, maybe := "no", "yes", "maybe"
    // 短縮宣言、同時に複数の変数を宣言
    japaneseHello := "Konichiwa" // 同上
    frenchHello = "Bonjour" // 通常の代入
}

```

Go の文字列は変更することができません。例えば下のコードはコンパイル時にエラーが発生します。: `cannot assign to s[0]`

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

ただし、本当に変更したくなったらどうでしょうか？ここでは以下のコードで実現します：

```
s := "hello"
c := []byte(s) // 文字列 s を []byte 型にキャスト
c[0] = 'c'
s2 := string(c) // もう一度 string 型にキャストし直す
fmt.Printf("%s\n", s2)
```

Go では + 演算子を使って文字列を連結することができます：

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

文字列の修正もこのように書けます：

```
s := "hello"
s = "c" + s[1:] // 文字列を変更することはできませんが
                // スライスは行えます。
fmt.Printf("%s\n", s)
```

もし複数行の文字列を宣言したくなったらどうでしょうか？この場合 ` で宣言することができます：

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

` で括られた文字列は Raw 文字列です。すなわち、文字列はコード内の形式がそのままプリント時の形式になります。文字列の変更はありません。改行はそのまま出力されます。例えばこの例では以下のように出力されます：

```
hello
world
```

エラー型 Go にはビルトインの error 型があります。専らエラー情報の処理に使用されます。Go の package の中にはエラー処理を行う errors というパッケージがあります。

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

3.2.4 Go データの低レイヤの保存

下の図は Russ Cox Blog の中の一文中で紹介されている Go データ構造の文章です。これらの基本型は低レイヤでメモリを分配し、対応する値を保存していることが見て取れるとおもいます。

1 byte



```
i := 1234 //type: int
```



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



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



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



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



図 17 Go データ形式の保存

3.2.5 テクニク

グループ化による宣言 Go 言語では、複数の定数・変数を宣言する場合、または複数のパッケージをインポートする場合、グループ化による方法で宣言することができます。

例えば以下のコード：

```
import "fmt"
import "os"

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

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

グループ化によって以下のような形式になります：

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

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

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

iota 列挙型 Go では `iota` というキーワードがあります。このキーワードは `enum` を宣言する際に使用されます。このデフォルト値は 0 から始まり、順次 1 が追加されます：

```
const(
    x = iota // x == 0
    y = iota // y == 1
    z = iota // z == 2
    w // 定数の宣言で値を省略した場合、デフォルト値は
    // 前の値と同じになります。ここではw = iotaと
    // 宣言していることと同じになりますので、
    // w == 3となります。実は上のyとzでも
    // この"= iota"は省略することができます。
)

const v = iota // constキーワードが出現する度に、
               // iotaは置き直されます。ここではv == 0です。

const (
    e, f, g = iota, iota, iota //e=0,f=0,g=0 iotaの同一行は同じです
)
```

他の値や `iota` に設定されているものを除いて、各 `const` グループのはじめの定数はデフォルトで 0 となります。二番目以降の定数は前の定数の値がデフォルト値となります。もし前の定数の値が `iota` であれば、直後の値も `iota` になります。

3.2.6 Go プログラムのデザインルール

Go がこのように簡潔なのは、それがいくつかのデフォルトの行為を持っているからです：

- 大文字で始まる変数はエクスポート可能です。つまり、他のパッケージから読むことができる、パブリックな変数だということです。対して小文字で始まる変数はエクスポートできません。これはプライベート変数です。
- 大文字で始まる関数も同じです。class の中で public キーワードによってパブリック関数となっているのと同じです。対して小文字で始まる関数は private キーワードのプライベート関数です。

3.2.7 array、slice、map

array array は配列です。この定義は以下のようになります：

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

[n]type の中で、n は配列の長さを表しています。type は保存する要素の型を示しています。配列に対する操作は他の言語とよく似ていて、どれも [] を通して値の取得および代入を行います。

```
var arr [10]int // int 型の配列を宣言します。
arr[0] = 42     // 配列のインデックスは0からはじまります。
arr[1] = 13     // 代入操作
fmt.Printf("The first element is %d\n", arr[0])
                // データを取得して、42を返します。
fmt.Printf("The last element is %d\n", arr[9])
                // 値が代入されていない最後の要素を返します。
                // デフォルトでは0が返ります。
```

長さも配列の一部ですので、[3]int と [4]int は異なる型になります。配列も長さを変えることはできません。配列間の代入は値渡しです。つまり、一つの配列が関数の引数となった場合、渡されるのは実はこの配列のコピーであり、ポインタではありません。もしポインタを使いたい場合は、この後にご紹介する slice 型をご利用ください。

配列はもうひとつの := で宣言することができます。

```
a := [3]int{1, 2, 3} // 長さが3のintの配列を宣言します。

b := [10]int{1, 2, 3} // 長さが10のint配列を宣言します。
                      // この中で3つの要素の初期値は1、2、3で、
                      // そのほかのデフォルトは0です。

c := [...]int{4, 5, 6} // 長さを'...'で省略することも
                      // できます。Goは自動で要素数から長さを
                      // 計算します。
```

もしあなたが「配列に配列を込めたい場合は実現できますか？」と問うならば、当然ですとも、とお応えしましょう。Go はネストした配列をサポートしています。例えば下のコードでは二次元配列を宣言しています：

```
// 二次元配列を一つ宣言します。この配列は
// 2つの配列を要素としており、各配列には4つ
```

```
// の int 型の要素が含まれます。
doubleArray := [2][4]int{[4]int{1, 2, 3, 4},
                        [4]int{5, 6, 7, 8}}

// 上の宣言は簡略化できます。直接内部の型を省略しています。
easyArray := [2][4]int{{1, 2, 3, 4}, {5, 6, 7, 8}}
```

配列の状態は以下のとおりです：

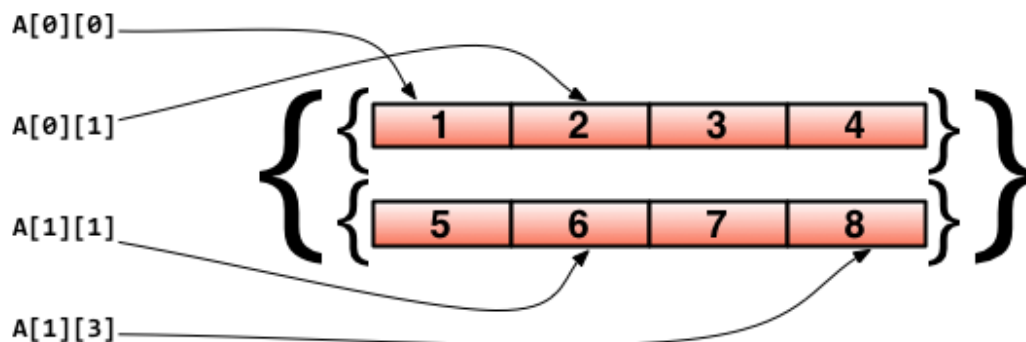


図 18 多次元配列のマッピング関係

`slice` 多くのアプリケーションでは、配列はあまりわたしたちの要求を満たしてはくれません。配列を初期化する場合、どれぐらいの大きさの配列が必要かわからないからです。そのため、“動的な配列”が必要となります。Go ではこのようなデータ構造を `slice` と呼びます。

`slice` は本当の意味での動的な配列ではありません。これは単なる参照型です。`slice` は常に低レイヤの `array` を指しています。`slice` の宣言も `array` と同様に長さを指定する必要はありません。

```
// array の宣言と同じですが、長さは必要ありません。
var fslice []int
```

次に `slice` を宣言すると同時にデータを初期化します：

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

`slice` はひとつの配列またはすでに存在する `slice` の中から宣言することができます。`slice` は `array[i:j]` で取得することができます。この中で `i` は配列の開始位置です。`j` は終了位置です。ただし `array[j]` は含みません。長さは `j-i` となります。

```
// 10 個の要素を宣言します。要素の型は byte の配列です。
var ar = [10]byte {'a', 'b', 'c', 'd', 'e',
                  'f', 'g', 'h', 'i', 'j'}

// byte を含む 2 つの slice を宣言します
var a, b []byte

// a ポインタ配列の 3 つ目の要素から始まり、5 つ目の要素まで
a = ar[2:5]
```



```
// 現在 a の持つ要素は : ar [2]、ar [3] と ar [4] です。

// b は配列 ar のもう一つの slice です。
b = ar [3:5]
// b の要素は : ar [3] と ar [4] です。
```

slice と配列は宣言時に区別されますのでご注意ください：配列を宣言するとき、中括弧の中で配列の長さを明示するかまたは... で自動的に長さを計算します。一方 slice を宣言する時は、中括弧内には文字はありません。

これらのデータ構造は以下のようになっています。

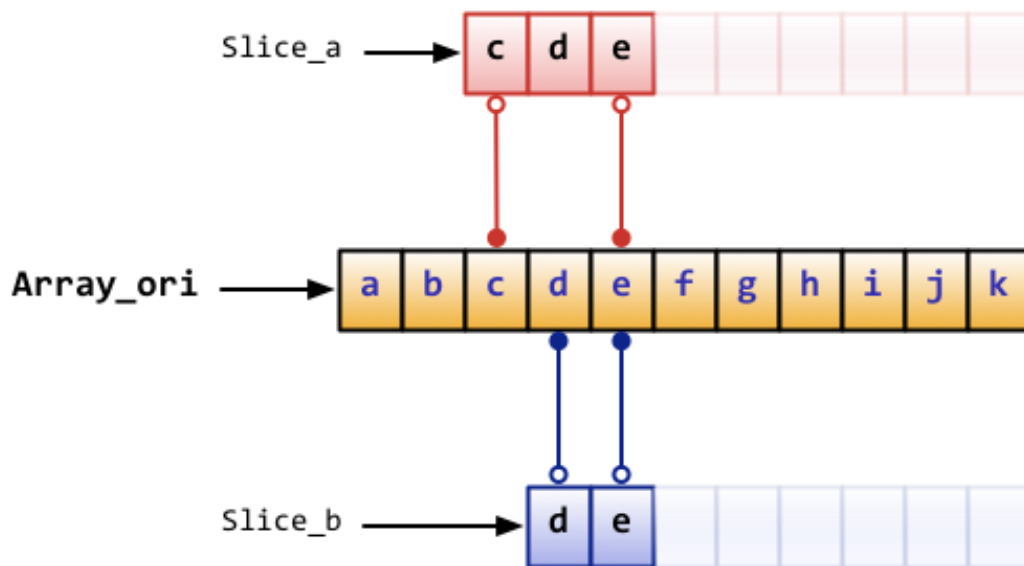


図 19 slice と array の対応関係図

slice には便利な操作があります

- slice のデフォルト開始位置は 0 です。ar[:n] などは ar[0:n] と等価です。
- slice の 2 つ目の値のデフォルトは配列の長さです。ar[n:] は ar[n:len(ar)] 等価です。
- もし配列の中から直接 slice を取り出す場合は、ar[:] というような形で指定することができます。なぜならデフォルトのはじめの値は 0 で 2 つ目は配列の長さだからです。すなわち、ar[0:len(ar)] と等価となります。

ここではより多くの slice の操作についていくつか例を挙げます：

```
// 配列を宣言
var array = [10] byte{'a', 'b', 'c', 'd', 'e',
                    'f', 'g', 'h', 'i', 'j'}

// slice を 2 つ宣言
var aSlice, bSlice [] byte
```

```
// 便利な操作のデモンストレーション
aSlice = array[:3] // aSlice = array[0:3] と同じ。
// aSliceには以下の要素が含まれます: a,b,c
aSlice = array[5:] // aSlice = array[5:10] と同じ。
// aSliceには以下の要素が含まれます: f,g,h,i,j
aSlice = array[:] // aSlice = array[0:10] と同じ。
// この場合 aSliceにはすべての要素が含まれます。

// sliceからsliceを取り出す
aSlice = array[3:7] // aSliceには以下の要素が含まれます:
// d,e,f,g, len=4, cap=7
bSlice = aSlice[1:3] // bSlice にはaSlice[1], aSlice[2] が
// 含まれそれぞれの要素は以下のとおりです: e,f
bSlice = aSlice[:3] // bSlice には aSlice[0], aSlice[1],
// aSlice[2] が含まれます。それぞれ以下のとおりです: d,e,f
bSlice = aSlice[0:5] // sliceのsliceに対してcapの
// 範囲内で拡張することができます。
// この時 bSliceには以下の要素が含まれます: d,e,f,g,h
bSlice = aSlice[:] // bSliceにはaSliceのすべての
// 要素が含まれます: d,e,f,g
```

slice は参照型ですので、この中の要素の値を変更すると、そのほかのすべての参照でも値が変更されます。たとえば上の aSlice と bSlice で、aSlice の中の要素を変更すると、bSlice の対応する値も同時に変更されます。

概念上では、slice は構造体です。この構造体には 3 つの要素が含まれます：

- 一つはポインタです。配列中の slice が示す開始位置を指しています。
- 長さ、つまり slice の長さです。
- 最大の長さ、slice の開始位置から配列の最後の位置までの長さです。

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

上のコードの正しい保存構造は下の図に示す通りです。

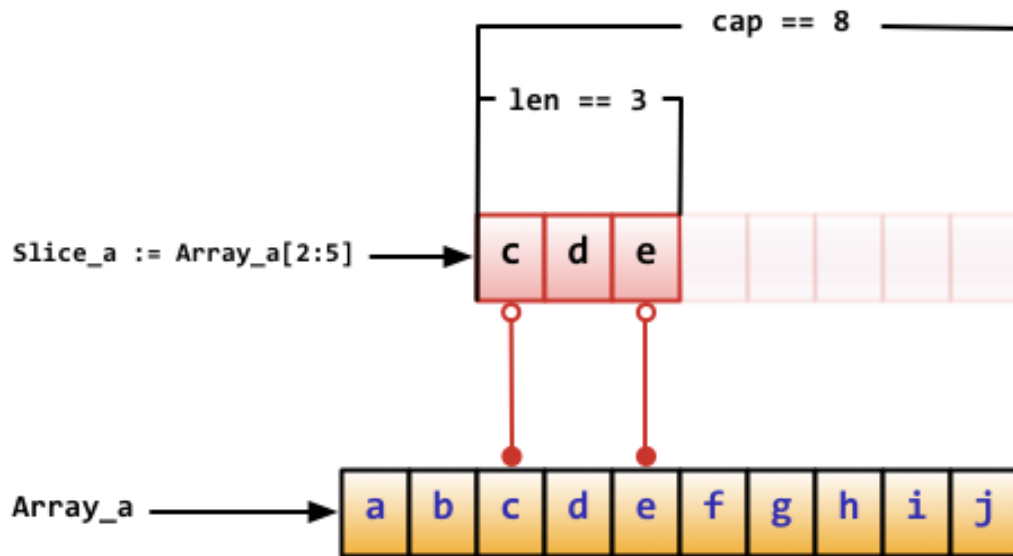


図 20 slice に対応する配列の情報

slice に対しては、いくつかの便利なビルトイン関数があります：

`len` slice の長さを取得します。

`cap` slice の最大容量を取得します。

`append` slice に対して一つまたは複数の要素を追加します。その後 slice と同じ型の slice を返します。

`copy` 関数 `copy` はもとの slice の `src` を `dst` に要素をコピーし、コピーした要素の個数を返します。

注：append 関数は slice が参照した配列の内容を変更し得ます。そのため、参照先と同一の配列の他の slice にも影響します。しかし slice の中に余分なスペースが無い (`cap-len`) == 0) 場合、動的なメモリから新たな配列空間が割り当てられます。返される slice 配列のポインタはこの空間を指しています。また、もとの配列の内容は変わりません。この配列を参照している他の slice は影響を受けません。

Go1.2 より、slice は三引数の slice をサポートするようになりました。以前まで我々は以下のような方法で slice または array から slice を取り出していました

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

この例では slice の要素数は 3 で、新しいバージョンでは以下のように要素数を指定することができます

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

上の要素数は 7-2、即ち 5 となります。このように生成された新しい slice では最後の 3 つの要素にアクセスする方法がなくなります。

もし slice が `array[:i:j]` のような形式だった場合は、第一引数は空と見なされ、デフォルトの 0 となります。

`map` `map` の概念も Python のディクショナリです。この形式は `map[keyType]valueType` です。

下のコードをご覧ください。map の読み込みと代入は slice と似ています。key を通して操作します。ただ、slice の index は int 型のみになります。map には多くの型があります。int でもかまいませんし、string や==と!=演算子が定義されている全ての型でもかまいません。

```
// keyを文字列で宣言します。値はintとなるディクショナリです。
// この方法は使用される前にmakeで初期化される必要があります。
var numbers map[string]int
// もうひとつのmapの宣言方法
numbers := make(map[string]int)
numbers["one"] = 1 // 代入
numbers["ten"] = 10 // 代入
numbers["three"] = 3

fmt.Println(" 3 つ目の数字は: ", numbers["three"]) // データの取得
// " 3 つ目の数字は: 3"という風に出力されます。
```

この map は我々が普段目にする表と同じです。左の列に key、右の列に値があります。

map を使う段階で注意しなければならないことがいくつかあります：

- map は順序がありません。毎回 map の出力は違ったものになるかもしれません。index で値を取得することはできず、かならず key を使うことになります。
- map の長さは固定ではありません。slice と同じで、参照型的一种です。
- ビルトインの len 関数を map に適用すると、map がもつ key の個数を返します。
- map の値は簡単に修正することができます。numbers["one"]=11 というように key が one のディクショナリの値を 11 に変えることができます。
- map は他の基本型と異なり、thread-safe ではありません。複数の go-routine を扱う際には必ず mutex lock メカニズムを使用する必要があります。

map の初期化では key:val の方法で初期値を与えることができます。また同時に map には標準で key が存在するか確認する方法が存在します。

delete で map の要素を削除します：

```
// ディクショナリを初期化します。
rating := map[string]float32{"C":5, "Go":4.5,
                             "Python":4.5, "C++":2 }
// mapは2つの戻り値があります。2つ目の戻り値では、
// もしkeyが存在しなければ、okはfalseに、存在すればokはtrueになります。
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") // keyがCの要素を削除します。
```

上述の通り、map は参照型の一種ですので、もし2つの map が同時に同じポインタを指している場合、一つの変更で、もう一つにも変更が行われます。

```
m := make(map[string]string)
m["Hello"] = "Bonjour"
m1 := m
m1["Hello"] = "Salut" // この時、m["hello"] の
                      // 値もすでに Salut になっています。
```

make, new 操作 make はビルトイン型 (map、slice および channel) のメモリの割り当てです。new は各型のメモリを割り当てます。

ビルトイン関数 new は本質的には他の言語で使われる同名の関数と機能が同じです：new(T) はゼロサプレスされた T 型のメモリ空間を割り当て、そのアドレスを返します。すなわち *T 型の値です。Go の専門用語で言えば、ポインタを返すということです。新たに割り当てられた型 T のゼロ値です。とても重要なことに：

new はポインタを返します。

ビルトイン関数 make(T, args) と new(T) は異なる機能を持っています。make は slice、map または channel を作成し、初期値 (非ゼロ値) を持つ T 型を返すのみで、*T ではありません。本質的には、この3つの型が異なる点はデータ構造を指し示す参照が使用される前に初期化されているということです。例えば、データ (内部 array) を指し示すポインタ、長さ、容量による3点で記述される slice の各項目が初期化される前は、slice は nil です。slice, map, channel にとって、make は内部のデータ構造を初期化し、適当な値で埋め尽くされます。

make は初期化後の (非ゼロの) 値を返します。

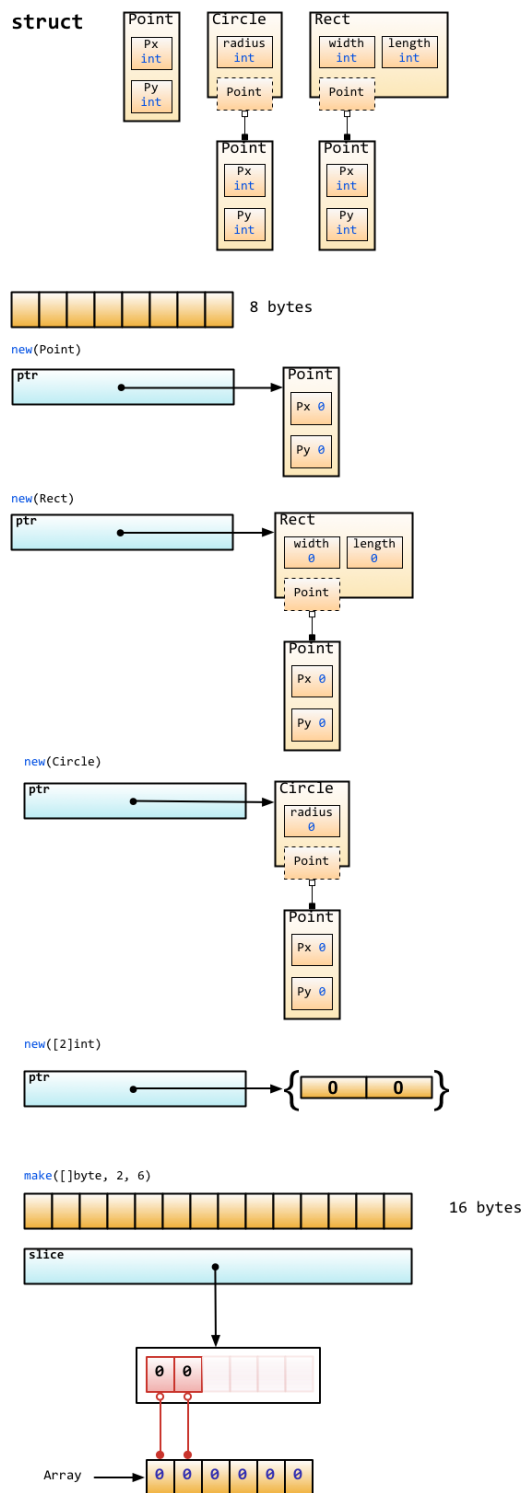


図 21 make と new の低レイヤでのメモリの割り当て

3.2.8 ゼロ値

”ゼロ値”というのは何も空の値ではありません。これは一種の”変数が埋めらる前”のデフォルト値であり、通常は0です。それぞれの型のゼロ値は以下の通りです

```
int      0
int8     0
int32    0
int64    0
uint     0x0
rune     0 // rune の実際の型は int32 です。
byte     0x0 // byte の実際の型は uint8 です。
float32  0 // 長さは 4 byte
float64  0 // 長さは 8 byte
bool     false
string   ""
```

3.3 フローと関数

この節では Go の中のフロー制御と関数操作についてご紹介します。

3.3.1 フロー制御

フロー制御はプログラム言語の中の最も偉大な発明です。なぜならこれがあるだけで、あなたはとても簡単なフローの記述でとても複雑なロジックを表現できるからです。Go ではフロー制御は3つの部分から成ります：条件判断、ループ制御及び無条件ジャンプです。

if if はあらゆるプログラミング言語の中で最もよく見かけるものかもしれません。この文法は大雑把に言えば：もし条件を満足しなければ何々を行い、そうでなければまたもう一つ別のことをやるということです。

Go の中では if 分岐の文法の中は括弧で括る必要はありません。以下のコードをご覧ください。

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

Go の if はすごいことに、条件分岐の中で変数を宣言できます。この変数のスコープはこの条件ロジックブロック内のみ存在し、他の場所では作用しません。以下に示します

```
// 取得値 x を計算し、x の大きさを返します。
// 10 以上かどうかを判断します。
if x := computedValue(); x > 10 {
    fmt.Println("x is greater than 10")
} else {
    fmt.Println("x is less than 10")
}

// ここではもしこのようにコールしてしまうとコンパイルエラー
```

```
// となります。 xは条件の中の変数だからです。  
fmt.Println(x)
```

この条件の時は以下のようになります：

```
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 には goto 句があります - ぜひ賢く使ってください。goto は必ず事前に関数内で定義したタグにジャンプします。例えばこのようなループがあったと仮定します：

```
func myFunc() {  
    i := 0  
Here:    //この行の最初の単語はコロンを  
        //最後に持つてくることでタグとなります。  
    println(i)  
    i++  
    goto Here    //Hereにジャンプします。  
}
```

タグの名前は大文字小文字を区別します。

for Go にで最も強力なロジックコントロールといえば、for です。これはループでデータを読むのに使えます。while でロジックをコントロールしても構いません。イテレーション操作も行えます。文法は以下の通りです：

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

expression1、expression2 と expression3 はどれも式です。この中で expression1 と expression3 は変数宣言または関数のコールの戻り値のようなものです。expression2 は条件判断に用いられます。expression1 はループの開始前にコールされます。expression3 は毎回ループする際の終了時にコールされます。

だらだら喋るよりも例を見たほうが早いでしょう。以下に例を示します：

```
package main  
import "fmt"  
  
func main(){  
    sum := 0;  
    for index:=0; index < 10 ; index++ {  
        sum += index  
    }  
}
```



```

    fmt.Println("sum is equal to ", sum)
}
// 出力 : sum is equal to 45

```

時々複数の代入操作を行いたい時があります。Go のなかには、という演算子はないので、平行して代入することができます。i, j = i+1, j-1

時々 expression1 と expression3 を省略します：

```

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

```

この中で、は省略することができます。ですので下のようなコードになります。どこかで見た覚えはありませんか？そう、これは while の機能です。

```

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

```

ループの中では break と continue という2つのキーとなる操作があります。break 操作は現在のループから抜け出します。continue は次のループに飛び越えます。ネストが深い場合、break はタグと組み合わせて使用することができます。つまり、タグが指定する位置までジャンプすることになります。詳細は以下の例をご覧ください。

```

for index := 10; index>0; index-- {
    if index == 5{
        break // またはcontinue
    }
    fmt.Println(index)
}
// breakであれば10、9、8、7、6が出力されます。
// continueの場合は10、9、8、7、6、4、3、2、1が出力されます。

```

break と continue はタグを添えることができます。複数ネストしたループで外側のループからジャンプする際に使用されます。

for は range と組み合わせて array、slice、map、string のデータを読み込むことができます：

```

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

```

Go は”複数の戻り値”をサポートしていますが、”宣言して使用されていない”変数に対してコンパイラはエラーを出力します。このような状況では、を使って必要のない戻り値を捨てる事ができます。例えば

```

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

```

switch 時々たくさんの if-else を書くことでロジック処理を行いたくなるかもしれません。コードは非常に醜く冗長になります。またメンテナンスも容易ではなくなるので、switch を使って解決することができます。この文法は以下のようなものです

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

sExpr と expr1、expr2、expr3 の型は一致させる必要があります。Go の switch は非常に使い勝手がよく、式は必ずしも定数や整数である必要はありません。実行のプロセスは上から下まで、マッチする項目が見つかるまで行われます。もし switch に式がなければ、true とマッチします。

```
1 i := 10
2 switch i {
3 case 1:
4     fmt.Println("i is equal to 1")
5 case 2, 3, 4:
6     fmt.Println("i is equal to 2, 3 or 4")
7 case 10:
8     fmt.Println("i is equal to 10")
9 default:
10    fmt.Println("All I know is that i is an integer")
11 }
```

5 行目で、いくつかの値を case の中に集めています。また同時に、Go の switch はデフォルトで case の最後に break があることになっているので、マッチに成功した後は他の case が実行されることはなく、switch 全体から抜け出します。ただし、fallthrough を使用することであとに続く case コードを強制的に実行させることができます。

```
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")
}

```

上のプログラムは以下のように出力します

```

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

```

3.3.2 関数

関数は Go の中心的な設計です。キーワード `func` によって宣言します。形式は以下の通り：

```

func funcName(input1 type1, input2 type2)
    (output1 type1, output2 type2) {
    //ここはロジック処理のコードです。
    //複数の値を戻り値とします。
    return value1, value2
}

```

上のコードから次のようなことが分かります

- キーワード `func` で `funcName` という名前の関数を宣言します。
- 関数はひとつまたは複数の引数をとることができ、各引数の後には型が続きます。、をデリミタとします。
- 関数は複数の戻り値を持ってかまいません。
- 上の戻り値は2つの変数 `output1` と `output2` であると宣言されています。もしあなたが宣言したくないというのであればそれでもかまいません。直接2つの型です。
- もしひとつの戻り値しか存在せず、また戻り値の変数が宣言されていなかった場合、戻り値の括弧を省略することができます。
- もし戻り値が無ければ、最後の戻り値の情報も省略することができます。
- もし戻り値があれば、関数の中で `return` 文を追加する必要があります。

以下では実際に関数の例を応用しています（Max の値を計算します）

```

package main
import "fmt"

// a、bの中から最大値を返します。
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) // 関数 max(x, y) をコール
    max_xz := max(x, z) // 関数 max(x, z) をコール

    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)) // 直接コールしてもかまいません
}

```

上では max 関数に 2 つの引数があることがわかります。この型はどれも int です。第一引数の型は省略することができます（つまり、a,b int, であり a int, b int ではありません）デフォルトは直近の型です。2 つ以上の同じ型の変数または戻り値も同じです。同時に戻り値がひとつであることに注意してください。これは省略記法です。

複数の戻り値 Go 言語は C に比べ先進的な特徴を持っています。関数が複数の戻り値を持てるのもその一つです。

コードの例を見てみましょう

```

package main
import "fmt"

//A+B と A*B を返します
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)
}

```

上の例では直接 2 つの引数を返しました。当然引数を返す変数に命名してもかまいません。この例では 2 つの型のみ使っていますが、下のように定義することもできます。値が返る際は変数名を付けなくてかまいません。なぜなら関数の中で直接初期化されているからです。しかしもしあなたの関数がエクスポートされるのであれば（大文字からはじまります）オフィシャルではなるべく戻り値に名前をつけるようお勧めしています。なぜなら名前のわからない戻り値はコードをより簡潔なものにしますが、生成されるドキュメントの可読性がひどくなるからです。

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

可変長引数 Go の関数は可変長引数をサポートしています。可変長引数を受け付ける関数は不特定多数の引数があります。これを実現するために、関数が可変長引数を受け取れるよう定義する必要があります：

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

arg ...int は Go にこの関数が不特定多数の引数を受け付けることを伝えます。ご注意ください。この引数の型はすべて int です。関数ブロックの中で変数 arg は int の slice となります。

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

値渡しと参照渡し 引数をコールされる関数の中に渡すとき、実際にはこの値のコピーが渡されます。コールされる関数の中で引数に修正をくわえても、関数をコールした実引き数には何の変化もありません。数値の変化はコピーの上で行われるだけだからです。

この内容を検証するために、ひとつ例を見てみましょう

```
package main
import "fmt"

// 引数+1を行う、簡単な関数
func add1(a int) int {
    a = a+1 // aの値を変更します。
    return a // 新しい値を返します。
}

func main() {
    x := 3

    fmt.Println("x = ", x) // "x = 3"と出力するはずです。

    x1 := add1(x) // add1(x) をコールします。

    fmt.Println("x+1 = ", x1)
    // "x+1 = 4" と出力するはずです。

    fmt.Println("x = ", x)
    // "x = 3" と出力するはずです。
}
```

どうです？ add1 関数をコールし、add1 のなかで a = a+1 の操作を実行したとしても、上述の x 変数には何の変化も発生しません。

理由はとても簡単です：add1 がコールされた際、add1 が受け取る引数は x そのものではなく、x のコピーだからです。

もし本当にこの x そのものを渡したくなったらどうするの？と疑問に思うかもしれません。

この場合いわゆるポインタにまで話がつながります。我々は変数がメモリの中のある特定の位置に存在していることを知っています。変数を修正するということはとどのつまり変数のアドレスにあるメモリを修正していることになります。add1 関数が x 変数のアドレスを知ってさえいれば、x 変数の値を変更することが可能です。そのため、我々は x の存在するアドレスである &x を関数に渡し、関数の変数の型を int からポインタ変数である *int に変更します。これで関数の中で x の値を変更することができるようになりました。この時関数は依然としてコピーにより引数を受け渡しますが、コピーしているのはポインタになります。以下の例をご覧ください。

```
package main
import "fmt"

// 引数に+1を行う簡単な関数
func add1(a *int) int { // ご注意ください。
    *a = *a+1 // aの値を修正しています。
    return *a // 新しい値を返します。
}

func main() {
    x := 3

    fmt.Println("x = ", x) // "x = 3"と出力するはずです。

    x1 := add1(&x)
    // add1(&x) をコールして x のアドレスを渡します。

    fmt.Println("x+1 = ", x1)
    // "x+1 = 4"を出力するはずです。

    fmt.Println("x = ", x)
    // "x = 4"を出力するはずです。
}
```

このように x を修正するという目的に到達しました。では、ポインタを渡す長所はなんなのでしょう？

- ポインタを渡すことで複数の関数が同じオブジェクトに対して操作を行うことができます。
- ポインタ渡しは比較的軽いです（8 バイト）。ただのメモリのアドレスです。ポインタを使って大きな構造体を渡すことができます。もし値渡しを行っていたら、相対的にもっと多くのシステムリソース（メモリと時間）を毎回のコピーで消費することになります。そのため大きな構造体を渡す際は、ポインタを使うのが賢い選択というものです。
- Go 言語の channel、slice、map の 3 つの型はメカニズムを実現するポインタのようなものです。ですので、直接渡すことができますので、アドレスを取得してポインタを渡す必要はありません。（注：もし関数が slice の長さを変更する場合はアドレスを取得し、ポインタを渡す必要があります。）

defer Go 言語のすばらしいデザインの中に、遅延 (defer) 文法があります。関数の中で defer 文を複数追加することができます。関数が最後まで実行された時、この defer 文が逆順に実行されます。最後にこの関数が返ります。特に、リソースをオープンする操作を行なっているようなとき、エラーの発生に対してロールバックし、必要なリソースをクローズする必要があるかと思います。さもないととても簡単にリソースのリークといった問題を引き起こすことになります。我々はリソースを開く際は一般的に以下のようにします：

```
func ReadWrite() bool {
    file.Open("file")
    // 何かを行う
    if failureX {
        file.Close()
        return false
    }

    if failureY {
        file.Close()
        return false
    }

    file.Close()
    return true
}
```

上のコードはとても多くの重複がみられます。Go の defer はこの問題を解決します。これを使用した後、コードは減るばかりでなく、プログラムもよりエレガントになります。defer の後に指定された関数が関数を抜ける前にコールされます。

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

もし defer を多用する場合は、defer は LIFO モードが採用されます。そのため、以下のコードは 4 3 2 1 0 を出力します。

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

値、型としての関数 Go では関数も変数の一種です。type を通して定義します。これは全て同じ引数と同じ戻り値を持つ一つの型です。

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

関数を型として扱うことにメリットはあるのでしょうか？ではこの型の関数を値として渡してみましょう。以下の例をご覧ください。

```
package main
import "fmt"

type testInt func(int) bool // 関数の型を宣言します。

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
}

// ここでは宣言する関数の型を引数のひとつとみなします。

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) // 関数の値渡し
    fmt.Println("Odd elements of slice are: ", odd)
    even := filter(slice, isEven) // 関数の値渡し
    fmt.Println("Even elements of slice are: ", even)
}
```

共有のインターフェースを書くときに関数を値と型にみなすのは非常に便利です。上の例で `testInt` という型は関数の型の一つでした。ふたつの `filter` 関数の引数と戻り値は `testInt` の型と同じですが、より多くのロジックを実現することができます。このように我々のプログラムをより優れたものにすることができます。

ます。

Panic と Recover Go には Java のような例外処理はありません。例外を投げないのです。その代わり、panic と recover を使用します。ぜひ覚えておいてください、これは最後の手段として使うことを。つまり、あなたのコードにあってはなりません。もしくは panic を極力減らしてください。これは非常に強力なツールです。賢く使ってください。では、どのように使うのでしょうか？

Panic

ビルトイン関数です。オリジナルの処理フローを中断させることができます。パニックが発生するフローの中に入って関数 F が panic をコールします。このプロセスは継続して実行されます。一旦 panic の goroutine が発生すると、コールされた関数がすべて返ります。この時プログラムを抜けます。パニックは直接 panic をコールします。実行時にエラーが発生させてもかまいません。例えば配列の境界を超えてアクセスする、などです。

Recover

ビルトイン関数です。パニックを発生させるフローの goroutine を復元することができます。recover は遅延関数の中でのみ有効です。通常の実行中、recover をコールすると nil が返ります。他には何の効果もありません。もし現在の goroutine がパニックに陥ったら recover をコールして、panic の入力値を補足し、正常な実行に復元することができます。

下の関数のフローの中でどのように panic を使うかご覧ください

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

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

この関数は引数となっている関数が実行時に panic を発生するか検査します：

```
func throwsPanic(f func()) (b bool) {
    defer func() {
        if x := recover(); x != nil {
            b = true
        }
    }()
    f() // 関数 f を実行します。もし f の中で panic が出現したら、
        // 復元を行うことができます。
    return
}
```

main 関数と init 関数 Go では2つの関数が予約されています：init 関数（すべての package で使用できます）と main 関数（package main でしか使用できません）です。この2つの関数は定義される際いかなる引数と戻り値も持ちません。package のなかで複数の init 関数を書いたとしても、もちろん可読性が後々

のメンテナンス性に対してですが、package の中では各ファイルに一つだけの init 関数を書くよう強くおすすめます。

Go のプログラムは自動で init() と main() をコールしますので、どこかでこの 2 つの関数をコールする必要はありません。各 package の init 関数はオプションです。しかし package main は必ず一つ main 関数を含まなければなりません。

プログラムの初期化と実行はすべて main パッケージから始まります。もし main パッケージが他のパッケージをインポートしていたら、コンパイル時にその依存パッケージがインポートされます。あるパッケージが複数のパッケージに同時にインポートされている場合は、先にその他のパッケージがインポートされ、その後このパッケージの中にあるパッケージクラス定数と変数が初期化されます。次に init 関数が（もしあれば）実行され、最後に main 関数が実行されます。以下の図で実行過程を詳しくご説明しています。

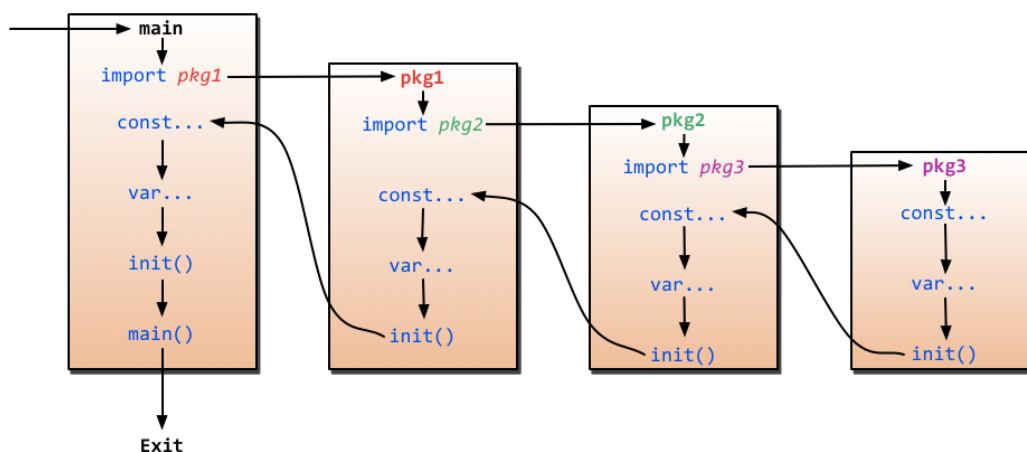


図 22 main 関数によるパッケージのインポートと初期化過程の図

import Go のコードを書いている時は、import コマンドによってパッケージファイルをインポートすることがよくあります。私達が通常使う方法は以下を参考にしてください：

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

その後コードの中では以下のような方法でコールすることができます。

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

上の fmt は Go 言語の標準ライブラリです。実は GOROOT 環境変数で指定されたディレクトリの下にこのモジュールが加えられています。当然 Go のインポートは以下のような 2 つの方法で自分の書いたモジュールを追加することができます：

1. 相対パス

import "../model" //カレントファイルと同じディレクトリにある model ディレクトリ、ただし、この方法による import はおすすしません。