

# A Quick Tour of Go for Java programmers

# Goのインストール

Goの公式サイトからバイナリがダウンロードできます。

macOSならHomebrewでインストールできます。

```
$ brew install go
```

# GOPATH

Goをインストールしたらはじめに環境変数 `GOPATH` を設定する必要があります。

場所はどこでもいいですが、Goに関するファイルが以下のように配置されます。

```
$GOPATH/
|-- bin/ ... コンパイル済みの実行ファイル
|-- pkg/ ... コンパイル済みのオブジェクト
|-- src/ ... ソースコード
    |-- <import_path>/
        |-- foo.go
        |-- foo_test.go
```

こだわりが無ければ `$HOME` や `$HOME/go` でいいです。後から変更できるので悩む必要ないです。

# go get

GOPATH を設定すると go get コマンドが実行できるようになります。

とりあえず雰囲気をつかむために go get の実行例と結果を示します。

```
$ go get github.com/shrhdk/echo
```

```
# 結果
$GOPATH/
| -- bin/
|   | -- echo (ビルドされた実行ファイル)
|
| -- src/
|   | -- github.com/
|   |   | -- shrhdk/
|   |   |   | -- echo/
|   |   |   |   | -- .git/
|   |   |   |   | -- .gitignore
|   |   |   |   | -- README.md
|   |   |   |   | -- main.go
```

`go get <import_path>` と実行すると `<import_path>` の先からコードをダウンロードして `$GOPATH/src/<import_path>` に配置します。

そしてコードをビルドして、生成された実行ファイルを `$GOPATH/bin` に配置します。

ここで注意したいのは `.git` フォルダもダウンロードされていることです。 `go get` した結果はそのままGitのリポジトリとして扱えます。

`<import_path>` に関する細かい規則は次のページに書かれています。

<http://golang-jp.org/doc/code.html#remote>

# プロジェクト

Javaでは `build.xml`、`pom.xml`、`build.gradle` といったファイルを置いて、プロジェクトを単位として開発をすることが多いです。

しかし、Goでは標準ではプロジェクトファイルのような仕組みは用意されていません。

Goでは1つのワークスペース上ですべての開発を行います。`GOPATH` がそれです。

# main関数

Javaでいうところの `public static void main(String[] args)` は単に `func main()` となります。

```
public class Foo {  
    public static void main(String[] args) {  
        // Javaのエントリーポイント  
    }  
}
```

```
package main  
  
func main() {  
    // Goのエントリーポイント  
}
```

Go言語では関数を直に書けます。メソッドではなく関数です。

main関数はmainパッケージにしか書けません。

コマンドライン引数はosパッケージからとれます。

```
package main

import (
    "fmt"
    "os"
)

func main() {
    fmt.Println(os.Args[1])
}
```

Goの `import` はJavaとは意味が異なるので注意してください。

Javaではあくまでもコード中でパッケージ名の記述を省略するための宣言でしたが、Goでは別パッケージを参照するための宣言となります。

# パッケージ

Goにもパッケージの概念があります。

既にコード例が出ていましたが、fooパッケージならコードの先頭に以下のように書きます。

```
package foo  
... 以下コード ...
```

package宣言は必須です。デフォルトパッケージはありません。

1つのパッケージはファイルシステム上の1つのフォルダと密接に対応します。

```
$GOPATH/
|-- src/
    |-- foo/ ... パッケージfoo
        |-- hoge.go ... パッケージfooのコード
        |-- fuga.go ... パッケージfooのコード
        |-- piyo.go ... パッケージfooのコード
    |-- bar/ ... パッケージbar
        |-- foo/ ... 上のfooとは別のパッケージ
            |-- hoge2.go
```

フォルダ内で矛盾するパッケージ宣言がされている場合はコンパイルエラーになります。

例えば `hoge.go` では `package foo` が宣言され、 `fuga.go` では `package foofoo` が宣言されていた場合エラーです。

# インポートパス

パッケージ名と混同しやすい概念にインポートパスがあります。

インポートパスは `go get` や `import` でパッケージを指定する文字列です。

例えば `https://github.com/shrhdk/foo` に配置されたパッケージであれば、インポートパスは `github.com/shrhdk/foo` になります。

インポートパスとパッケージ名は独立したものです。  
インポートパスはソースコードが配置されているパスで、  
パッケージ名はソースコード中のpackage宣言で宣言されたものです。  
しかし、慣習的にインポートパスの末尾とパッケージ名は一致させま  
す。

インポートパスが `https://github.com/shrhdk/foo` であれば、  
そのフォルダにあるコードのパッケージ名は `foo` にします。

Javaではパッケージ名にドメイン名を使うことでの名前の衝突を避けていましたが、Goではインポートパスに必然的にドメイン名が含まれるので衝突が避けられます。

ただ、パッケージ名は衝突する可能性があります。

例えば `github.com/shrhdk/foo` と `github.com/mikan/foo` はインポートパスは衝突しませんがパッケージ名は衝突します。(どちらも `foo` )

解決方法はありますがここでは説明しません。

# パッケージのパスまわりまとめ

リポジトリURL :	<code>https://github.com/shrhdk/foo</code>
ソースコード :	<code>\$GOPATH/src/github.com/shrhdk/foo/*.go</code>
実行ファイル :	<code>\$GOPATH/bin/foo</code>
インポートパス :	<code>github.com/shrhdk/foo</code>
パッケージ名 :	<code>foo</code>

## アクセスレベル

Javaは `public` , `protected` , `private` , `package private` といった、細やかなアクセスレベルの設定が可能ですが、そのあたりGoは大雑把というかシンプルです。

Goには `public` か `package private` しかありません。

名前が大文字で始まつていれば`public`、小文字で始まつていれば`package private`です。

```
var Foo string これはpublic  
var bar string これはpackage private
```

# type

Javaでいうところのプリミティブ型に別名をつけたりメソッドを定義したりできます。

```
type Username string

func (u Username) Encode() []byte {
    ...
}
```

# defer (try-finally)

Javaでは必ず実行したい処理はfinallyブロックに書きます。

```
void bar() {  
    try {  
        FileInputStream in = new FileInputStream("bar.txt");  
        ...  
    } finally {  
        in.close(); // try-catchブロックを抜けるときに必ず実行され  
    }  
}
```

対して、Goではdefer分を使います。

defer文の関数呼び出しが関数を出るときに必ず実行されます。

Goの場合はJavaと違ってスコープが関数に限定されてしまうので少し不便です。

```
func foo() {
    f, err := os.Open("bar.txt")
    if err != nil {
        panic("Panic!")
    }
    defer f.Close() // 関数を出るときに必ず実行される
    ...
}
```

また、defer文は関数の式を書くのではなく、関数呼び出しを書きます。

上の例では `defer f.Close` と書くとエラーです。(これは混乱しやすい)

# nilは正当なレシーバー

Javaの場合、次の例では必ず例外が発生します。

```
Integer i = null;  
i.toString(); // ( `^` )< ぬるぼ
```

Goの場合、次の例ではパニックになるとは限りません。

```
var foo *Foo  
foo.Bar() // パニックになるとは限らない
```

例えば、`Bar` が以下のように実装されているとパニックになりません。

```
type Foo struct{}  
  
function (f *Foo) Bar() string {  
    return "bar" // 一切fooを参照していない  
}
```

# 例外処理

Goの例外処理は最後の返り値でerrorオブジェクトを返します。

```
func Parse(s string) (int, error) {
    ...
    if num < 0 {
        return 0, errors.New("値が0より小さいです") // エラー発生
    }
    return num, nil // 成功時
}
```

呼び出し側では最後の返り値がnilか否かを検査します。

```
func main() {
    data, err := Parse("hogehoge")
    if err != nil {
        panic("hogehogeパースでエラー")
    }
}
```

Javaでは `Exception` に `cause` を設定することで、抽象的な情報から詳細な情報まで報告しますが、Goでも同様の習慣があります。

```
func Parse(s string) (int, error) {
    num, err := strconv.Atoi(s)
    if err != nil {
        // errを含んだ新しいerrorを返す
        return 0, fmt.Errorf("%sの数値化に失敗: %v", s, err)
    }
    ...
}
```

# テスト

Goでは標準でテストの仕組みが用意されています。

テストは `hoge_test.go` という具合にサフィックス `_test` がついたファイルに書きます。 `hoge_test.go` はテスト対象のパッケージと同じフォルダに置きます。

テストの書き方の説明は割愛しますが、書いたテストは以下のように実行します。

```
$ go test <import_path>
```

# ドキュメンテーション

GoでもJavadocのようなドキュメンテーションコメントを書けます。

書き方は極めてシンプルで、対象の直前の行にコメントを書きます。

```
// Hello outputs "hello" to standard output.  
func Hello() {  
    fmt.Println("hello")  
}
```

JavaではドキュメンテーションコメントをHTMLに変換したものをまとめて、別途ホスティングしたり、アーカイブを配布する必要がありました。

しかし、これはGoでは不要です。単にコードにドキュメンテーションコメントを書いておくだけでOKです。

利用者は `go doc` コマンド一発でドキュメントを閲覧可能です。

```
$ go doc <import_path>[.<変数名|関数名|型名>]
```

io パッケージのドキュメントは次のコマンドで見られます。

```
$ go doc io  
package io // import "io"
```

Package io provides basic interfaces to I/O primitives. Its primary goal is to wrap existing implementations of such primitives, such as those in package os, into shared public interfaces that abstract the functionality plus some other related primitives.

... (省略、実際は全部表示されます) ...

```
$ go doc io.EOF  
var EOF = errors.New("EOF")
```

EOF is the error returned by Read when no more input is available. It should return EOF only to signal a graceful end of input. If EOF occurs unexpectedly in a structured data stream, the appropriate error should be ErrUnexpectedEOF or some other error giving more detail.

JavadocのようにHTML形式で閲覧することも可能です。

次のコマンドを実行するとドキュメンテーションコメントをHTML化したファイルをホストするWebサーバーが立ち上がります。

`GOROOT` および `GOPATH` 内のすべてのパッケージのドキュメントを閲覧できます。

```
$ godoc -http=:8000
```

URL: <http://localhost:8000/>