

# Golang Hands-on

## #3

written by @yukpiz  
@kentfordev

proofread @ariaki4dev

Feb 18, 2019

#techdo

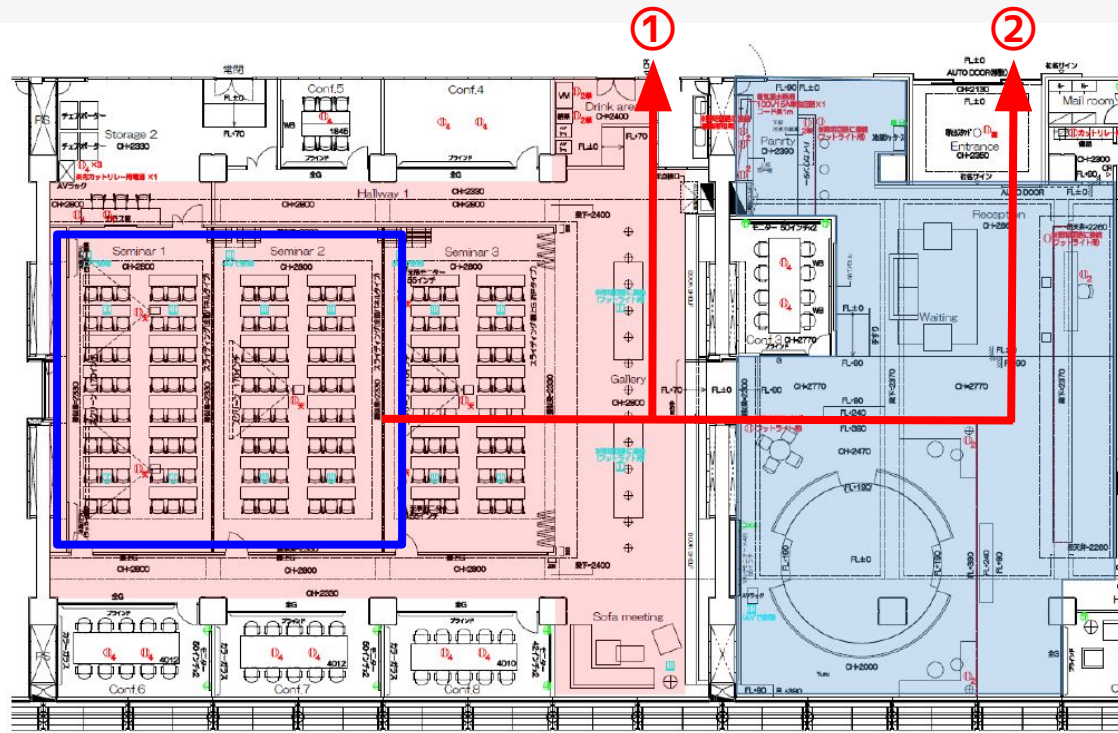
#golang

#mediado

#redish



会場前方



①会場後方勝手口  
(ベンチスペース脇)  
②中央エントランス



**非常時は係員による避難誘導指示に従ってください**

# コミュニケーション

---



<http://bit.ly/techdo-slack>  
#support-201902



#techdo

# 設備

---

## ゲスト用 Wi-Fi

- SSID                      md-guest
- Password                12345678

## 電源

- 足元の銀色カバーを開けるとコンセントがあります
- 数に限りがありますので、譲り合ってご利用ください

# その他



---

## ご案内とお願い

- 🐾 イベント内容は後日任意の媒体にて公開させて頂くことがあります
- 🐾 イベントレポート作成のため、写真および動画を撮影いたします
- 🐾 イベント中は**自由にご飲食**いただけます
- 🐾 体調不良等ございましたらスタッフまでお申し付けください

# タイムスケジュール

---

19:30	オープニング ( 10 min )	
19:40	座学 ( 40 min )	
20:20	課題 / 出題 ( 10 min )	
20:30	課題 / 解答 ( 50 min )	
21:20	課題 / 説明 & クロージング ( 10 min )	
21:30	懇親会	
22:00	撤収	

# 登壇者のご紹介

---



**kentfordev**

@k\_h\_sissp



**yukpiz**

@yukpiz



**ariaki**

@ariaki4dev

# 座学

---


(40 min)



# 目次

---

## 本日の座学内容

 Go言語について

 型と変数

 条件分岐と繰り返し

 構造体

 レシーバー

 インターフェース

 テスト

 Webフレームワーク

# Go言語について

---

# Go言語について

---

## Go言語とは？

- 2012年にGoogleからリリース
- **静的型付け言語**（JavaとかC#、Swiftのような）
- コンパイルが早く、実行速度も速い言語
- **構文がシンプル**で読みやすい

# 開発環境

---

## IDEやエディタを選ぼう

- Go言語はIDEやエディタに依存しない
- 使い慣れた環境があれば、それを使うのがベスト

	IntelliJ IDEA	GoLand	VSCode	Vim	Emacs
価格	有償	有償	無償	無償	無償
プラグイン	有	専用	有	有	有

# 開発環境のインストール

---

## Go言語のインストール

- 最新は**バージョン1.11.5**（2019.2.18時点）
- **goenv** というツールがある
- rbenvやpyenvと同じように、簡単にGoのバージョンを切り替え
- <https://github.com/syndbg/goenv>

# はじめてのコーディングと実行

---

## main.goの準備

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     fmt.Println("Hello golang!")
7 }
```

mainパッケージの宣言  
(go run で実行可能)

import宣言  
外部パッケージの利用

main関数の定義  
(go run 実行時に呼び出し)

# はじめてのコーディングと実行

---

書いたコードを実行してみよう

- **go run main.go** で実行
- コンパイルに問題がなければ、実行される
- 実行時にも問題がなければ Hello golang! と出力される

```
$ go run main.go  
Hello golang!
```

# 型と変数

---



# 型と変数

---

論理値型	bool
数値型	uint, int, float, complex, byte, rune
文字列型	string
配列、スライス型	[3]string, []string
構造体型	struct
ポインタ型	*int, *string
関数型	func
インターフェース型	interface{}
マップ型	map[string]int
チャンネル型	chan

# 型と変数

---

## 変数の宣言

- varを使って**変数の宣言**ができる
- 初期化をしなくても**ゼロ値**が自動的に入る
- varは**省略可能**

```
1 var i1 int
2 var i2 int = 12
3
4 i3 := 123
```

var 変数名 型  
初期化はしなくてもゼロ値が入る  
var を省略した形

# 条件分岐と繰り返し

---

# 条件分岐と繰り返し

## 条件分岐 (if)

- **if** が利用できる

```
1 if sum == 30 {  
2    // ...  
3 } else if sum == 50 {  
4    // ...  
5 } else {  
6    // ...  
7 }
```

if 条件式 { ... }  
条件式に括弧は不要

{ ... } else if 条件式 { ... }

{ ... } else { ... }

# 条件分岐と繰り返し

---

## 条件分岐（switch）

- **switch** 利用できる
- 式switchと型switchがある

# 条件分岐と繰り返し

## 条件分岐（式switch）

```
1 switch sum {
2   case 30, 50, 70:
3     // ...
4   default:
5     // ...
6 }
7
8 switch {
9   case true:
10    // ...
11 }
12
13 x := 40
14 switch {
15   case x < 30:
16     // ...
17   case x < 50:
18     // ...
19 }
```

switch 式 { case 条件式: ... }

break は不要

式を省略すると true になる

条件式は定数である必要はない

# 条件分岐と繰り返し

## 条件分岐（型switch）

```
1 x := interface{}("yukpiz")
2
3 switch x.(type) {
4 case int:
5     // ...
6 case float64:
7     // ...
8 case bool, string:
9     // ...
10 }
```

... x はインターフェース型にする

switch 式.(type) { case 型: ... }  
x の型によって条件分岐する

# 条件分岐と繰り返し

## 繰り返し

- **for** のみ利用可能 (while はない)

```
1 for i := 0; i < 10; i++ {  
2     // ...  
3 }
```

条件式が true の間、繰り返す

```
4  
5 i := 0  
6 for i < 10 {  
7     i++  
8 }
```

条件式が true の間、繰り返す

```
9  
10 for idx, itm := range []string{"a", "b", "c"} {  
11     // ...  
12 }
```

range式で指定された値を、繰り返し左辺に代入  
range式は配列、スライス、  
文字列、マップ、チャンネルが使用可能



# 構造体

---

# 構造体

---

## 構造体とは

- 構造体は**名前と型**を持ったフィールドの**集合**
- 多種多様なデータを塊として扱える為、Go言語の開発では必須
- 構造体型は**値型**なので、代入や引数でコピーされる

# 構造体

## 構造体を作る

```
1 type User struct {  
2  
3     ID      int  
4     Name    string  
5     FavoriteFoods []string  
6     Entries []Entry  
7 }  
8  
9 type Entry struct {  
10     ID    int  
11     Name  string  
12     Body  string  
13 }
```

type 構造体名 { ... }  
Userという名前の構造体を定義

フィールド名と型  
別の構造体を入れ子にできる

# 構造体

## 構造体を使用する

```
1 user1 := User{} ..... User構造体を初期化する
2                                     各フィールドはゼロ値で初期化される
3 user2 := User{
4     ID: 1,
5     Name: "yukpiz",
6     FavoriteFoods: []string{"生ハム", "チーズ", "寿司"},
7     Entries: []Entry{
8         Entry{
9             ID: 1,
10            Name: "Goハンズオン",
11            Body: "hello, go!ang!",
12        },
13    },
14 }
```

# 構造体

## 構造体タグ（JSON）

- 構造体はタグと呼ばれる機能を利用できる
- JSON変換やデータ検証、ORMマッパーなどでよく取り扱われる

```
1 type User struct {  
2     ID    int    `json:"id"`  
3     Name  string `json:"name"`  
4 }  
5  
6 user := User{ID: 1, Name: "yukpiz"}  
7 jsonBytes, _ := json.Marshal(user)  
8  
9 json.Unmarshal(jsonBytes, &user)
```

jsonタグの例

User構造体を  
JSON文字列に変換

JSON文字列を  
User構造体に変換

# 構造体

---

## 構造体タグ（その他）

- CSVタグなら [github.com/gocarina/gocsv](https://github.com/gocarina/gocsv) で構造体をCSVに変換
- XMLタグなら [encoding/xml](#) で構造体をXMLに変換
- 構造体タグは **reflect**パッケージで取得可能

```
1 type User struct {  
2     ID    int    `json:"id" csv:"id" xml:"id"`  
3     Name string `json:"name" csv:"id" xml:"id"`  
4 }
```

... jsonタグ、csvタグ、xmlタグを設定した例

# Break

---

(5 min)



レシーバー

---



# レシーバー

---

## レシーバーとは

- Go言語では型に対して定義した関数を、**メソッド**と呼ぶ
- メソッドの定義された型をレシーバーと呼ぶ
- **値レシーバー**と**ポインターレシーバー**がある

# レシーバー

## 値レシーバーにメソッドを定義する

```
1 type User struct {  
2     ID      int  
3     LastName string  
4     FirstName string  
5 }  
6  
7 func (u User) GetFullName() string {  
8  
9     return u.LastName + u.FirstName  
10 }
```

User構造体に GetFullName関数を定義  
これは値レシーバーと呼ばれる

関数内から構造体にアクセス可能  
ここでUser構造体を書き換えても、  
呼び出し元に影響しない

# レシーバー

## ポインタレシーバーにメソッドを定義する

```
1 type User struct {  
2     ID      int  
3     LastName string  
4     FirstName string  
5 }  
6  
7 func (u *User) SetFirstName(n string) {  
8     u.FirstName = n  
9 }  
10  
11 func (u *User) SetLastName(n string) {  
12     u.LastName = n  
13 }
```

ポインタ型にすると、  
ポインタレシーバーとなる

ポインタレシーバーでは、  
変更が元の値に適用される

# レシーバー

---

## それぞれのレシーバーのメソッド呼び出し

```
1 user := User{ID: 1}  
2  
3 user.SetLastName("yuk")  
4 user.SetFirstName("piz")  
5  
6 fmt.Println(user.GetFullName())
```

ポインタレシーバーのメソッドなので、  
user変数に変更が適用される

# インターフェース

---

# インターフェース

---

## インターフェースとは

- インターフェースは型の1つで、**メソッド定義の集まり**
- インターフェースを使い、あらゆる型を受け付ける関数を作れる
- インターフェースを使い、型によって挙動を変える処理を作れる

```
1 type Example interface {}  
2  
3 type Animal interface {  
4     Cry()  
5 }
```

Exampleという名前の  
インターフェース型を定義

Cry関数を持ったインターフェース型を  
Animalという名前で定義

# インターフェース

---

## あらゆる型を受け付ける関数を作る

```
1 func main() {  
2     f(123)  
3     f("yukpiz")  
4     f([]int{1, 2, 3})  
5 }  
6  
7 func f(a interface{}) {  
8     // ...  
9 }
```

・ a はどんな型でも受け付ける事ができる

# インターフェース

## 型によって挙動を変える処理を作る

```
1 type Animal interface {  
2     Cry()  
3 }  
4  
5 type Cat struct{}  
6 type Dog struct{}  
7  
8 func (c Cat) Cry() {  
9     fmt.Println("にゃー")  
10 }  
11 func (c Dog) Cry() {  
12     fmt.Println("わん")  
13 }
```

構造体にレシーバーでCry関数を定義すると、  
構造体はAnimalインターフェースに準拠して  
いる事となる



# インターフェース

---

## 型によって挙動を変える処理を作る

```
1 type main() {  
2     cat := Cat{}  
3     animalCry(cat)  
4  
5     dog := Dog{}  
6     animalCry(dog)  
7 }  
8  
9 func animalCry(animal Animal) {  
10     animal.Cry()  
11 }
```

Animalインターフェース型を  
受け付ける関数を作ることができる

# エラーハンドリング

---

# エラーハンドリング

---

## Go言語におけるエラーハンドリング

- try-catch構文や例外送出の仕組みがない
- 戻り値を利用して、**errorインターフェース**を返す

```
1 err := f()
2 if err != nil {
3     return err
4 }
5
6 if err := f(); err != nil {
7     return err
8 }
```

if文でエラーチェックを行う

代入if文でコンパクトにできる

# エラーハンドリング

---

## エラーの作成

- 任意のエラー作成には、**errors.New** または **fmt.Errorf** を利用

```
1 func f1() error {  
2     return errors.New("ERROR")  
3 }  
4  
5 func f2() error {  
6     emsg := "エラーメッセージ"  
7     return fmt.Errorf("ERROR: %s", emsg)  
8 }
```

どちらも、errorインターフェースを  
返す機能を持つ

テスト

---

# テスト

---

## Go言語のテスト

- Go言語では組み込みの**testingパッケージ**でテストが書ける
- **stretchr/testify**で、アサーションやモックの機能を利用
- ファイル名は末尾が **\_test.go**、関数名は先頭に **Test** をつける

example\_test.go の例

```
1 func Test_Example1(t *testing.T) {  
2     // ...  
3 }
```

# テスト

---

## テストの実行

- `go test` でテスト実行
- `go test ./{directory}` で指定ディレクトリのテストが実行可能
- `go test --help` でオプションを確認

<code>-v</code>	詳細なテスト結果を出力
<code>-cover</code>	テストカバレッジを出力
<code>-run {regexp}</code>	実行するテスト関数名
<code>-parallel {n}</code>	テストを並列実行する

# テスト

---

## 組み込みのtestingパッケージの機能

Error(...interface{})	Log(...interface{})
Errorf(string, ...interface{})	Logf(string, ...interface{})
Fail()	Skip(...interface{})
FailNow()	SkipNow()
Failed() bool	Skip(string, ...interface{})
Fatal(...interface{})	Skipped() bool
Fatalf(string, ...interface{})	Parallel()



# テスト

---

## 組み込みのtestingパッケージの利用

```
1 package sample
2
3 import "testing"
4
5 func Test_Example1(t *testing.T) {
6     if (1 + 1 == 1) {
7         t.Fail()
8     }
9 }
```

ここ来ると、テストの失敗

# テスト

## 外部パッケージのtestifyの利用

- `go get -u github.com/stretchr/testify` を実行

```
1 import (  
2     "testing"  
3     "github.com/stretchr/testify/assert"  
4 )  
5  
6 func Test_Example(t *testing.T) {  
7     assert.Equal(t, 123, 124, "一致しません!")  
8     assert.NotEqual(t, 123, 123, "同じです!")  
9 }
```

testify のインポート

簡単なアサーション  
Equalで一致するかどうかを判定  
NotEqualで一致しないかどうかを判定

# テスト

---

## testify のアサーション機能

Empty	IsType	HTTPBody
Equal	JSONEq	Implements
Error	DirExists	Nil
NoError	FileExists	ElementsMatch
Zero	False	Exactly
EqualError	True	Len
EqualValues	Contains	<b>...etc</b>

# Webフレームワーク

---

# Webフレームワーク

---

## Webフレームワークとは

- Webサーバーの開発をサポートする為の**ツール群**  
**(アプリケーションフレームワーク)**
- Go言語にも様々なWebフレームワークが存在している

echo	gin	beego	iris
revel	kit	martini	...etc

参考：<https://github.com/mingrammer/go-web-framework-stars>

# Webフレームワーク

---

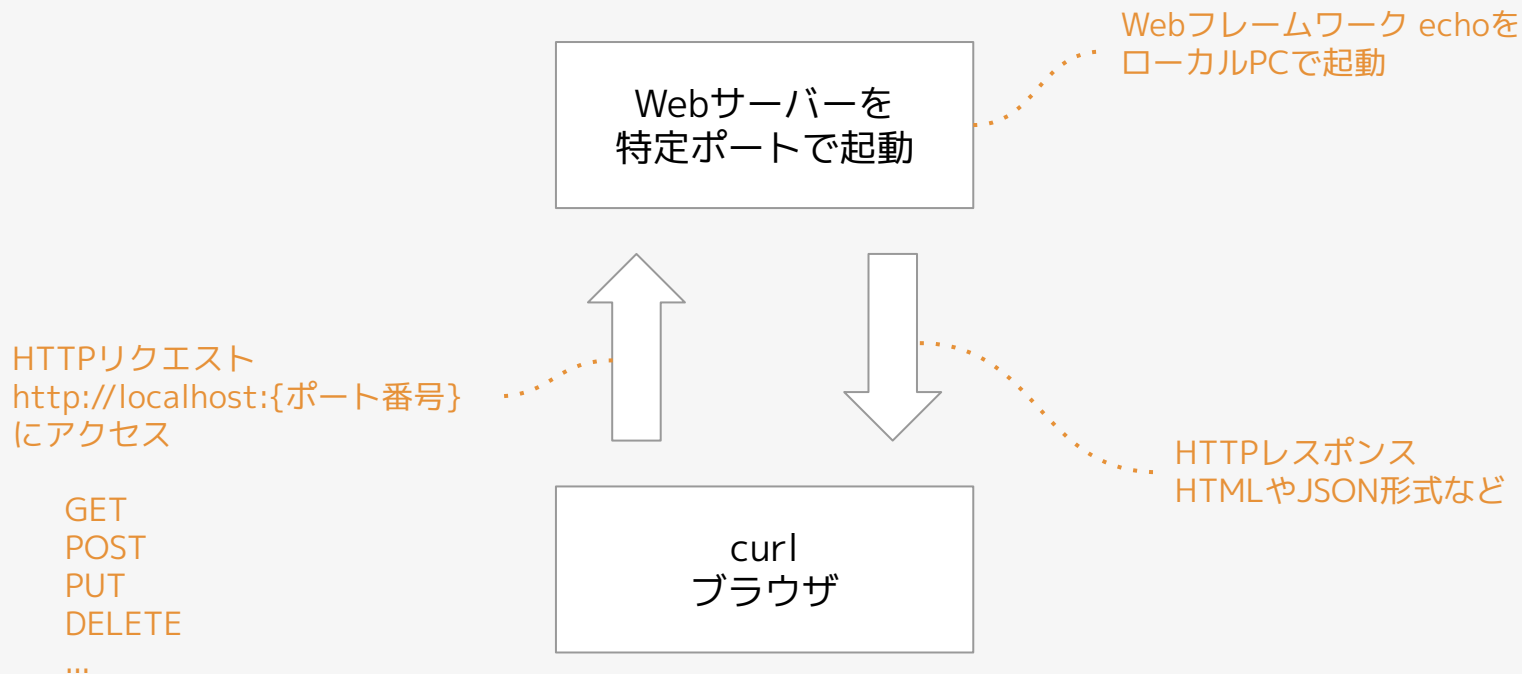
## Go言語におけるWebフレームワークの主な機能

- HTTPルーティング機能
- RESTful APIサポート
- Middlewareサポート
- データバインディング・レンダリング
- テンプレート機能

# Webフレームワーク

---

## Webサーバー開発の全体像



# 課題

---

(60 min)



# 課題 Webアプリを作ろう

---

GoによるWebアプリの制作過程を体験しましょう

- HTMLテンプレートによる、Webページ出力
- REST APIによる、JSON出力
- Webフレームワークを使った実装
- ローカルソースの自動ビルド組み込み

# 課題 Webアプリを作ろう

---

## 課題内容

Qiita記事を参照してください

<https://qiita.com/kentfordev/items/f3497b8982e731a6d953>

# 課題 Webアプリを作ろう

---

## 解答時間のご案内

- 解答時間は **21:20** までの **50分間** です
- 質問がある場合はスタッフにお声がけください
  - 挙手いただければスタッフが伺います
  - Slackチャンネルで質問いただければ回答します
- 解答中も自由にご飲食ください
- 体調不良等ございましたら、スタッフにご連絡ください

# ハンズオン後のサポートについて

---

## Slackチャンネルでのサポート

- 閉会後にご質問がある場合、Slackでサポートします
- 以下のワークスペースにて、**#support-201902**チャンネルでお気軽に質問ください
- <https://bit.ly/techdo-slack>
- 終わらなかった課題は是非、持ち帰って実践してみてください

We deliver more content for everybody to enjoy

ひとつでも多くのコンテンツを  
ひとりでも多くの人へ



kentfordev

@kentfordev



すべての出会いを価値あるものに



**Marketing** Support

**Concierge** Support

**Opening** Support

for Restaurant !!

yukpiz  
@yukpiz



# スタッフに拍手を！



kentfordev

@kentfordev

@zucky\_zakizaki

三澤 悠介

奥野 遥

koki\_okazawa

miyume\_yamamoto



yukpiz

@yukpiz

saya713y

池田 奈美

三森 泰規



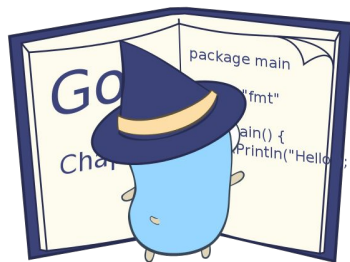
ariaki

アンケートにご協力ください



<http://bit.ly/go20190218>





goto after party!

Special Thanks!

The Go gopher was designed by Renée French.

Free Gophers: <https://github.com/egonelbre/gophers>