



Go Hands-on #2

Nov 27, 2018 #techdo #golang #mediado #redish

written by @kentfordev

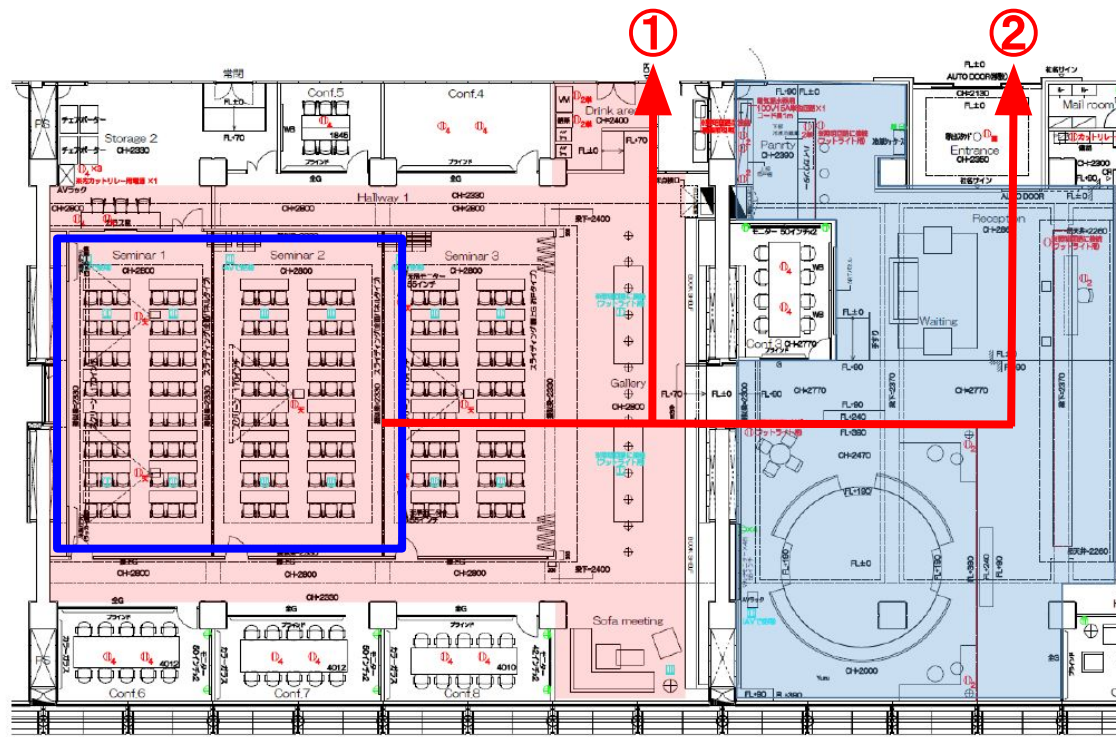
@yukpiz

proofread @ariaki4dev

避難経路

2

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



<http://bit.ly/techdo-slack>
#support-201811



#techdo

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、譲り合ってください

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

6

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	



kentfordev

@k_h_sissp



yukpiz

@yukpiz



ariaki

@ariaki4dev

座学

19:40 ~ 20:20 (40 min)



本日の学習内容



Goコマンド



Go標準パッケージ



レシーバー



インターフェース



並列処理／ゴルーチン



並列処理／チャネル



コーディング規約



ユニットテスト

Goコマンド

command-line parameters

コマンド一覧

get	run	test	fmt	build
bug	clean	doc	env	fix
generate	install	list	mod	tool
version	vet			

今回紹介するコマンド

get	run	test	fmt	build
bug	clean	doc	env	fix
generate	install	list	mod	tool
version	vet			

get

- ライブラリをダウンロードして、プロジェクトで使えるようにする
- ダウンロード先は、環境変数の \$GOPATH 配下が対象となる

get

dateライブラリをgetする

\$GOPATH/src/配下に、github.com/rickb777/dateがダウンロードされる

```
1 $ go get -u github.com/rickb777/date
```

※-uオプションは、対象リポジトリの最新版を常に指定してダウンロードする

run

- コマンド実行時に指定した、対象のプログラムを実行する
- コンパイル結果のファイルは生成せず、即時実行する

run

先程のdateを使った、サンプルコード

```
1 import (  
2     "fmt"  
3     "time"  
4  
5     "github.com/rickb777/date"  
6 )  
7  
8 func main() {  
9     date := date.New(2018, time.November, 27)  
10    fmt.Println(date)  
11 }
```


run

実行方法

```
1 $ go run main.go  
2 $ go run .  
3 $ go run ./
```

※実行結果として、「2018-11-27」が出力される

fmt

- ソースコードを整形する
- パスの指定方法により、整形対象をファイル単体か、ディレクトリ全体かを変えることができる

fmt

整形前

```
1 package main
2 import (
3     "fmt"
4     "github.com/rickb777/date"
5     "time"
6 )
7 func main(){
8     date := date.New(2018, time.November, 27); fmt.Println(date)
9 }
```

fmt

実行方法

```
1 # 対象のファイルを整形する
2 $ go fmt main.go
3
4 # カレントディレクトリ直下のファイルを整形する
5 $ go fmt .
6 $ go fmt ./
7
8 # カレントディレクトリ配下のファイル、ディレクトリを再帰的にたどり全て整形する
9 $ go fmt ./...
```

fmt

整形後

```
1 import (  
2     "fmt"  
3     "time"  
4  
5     "github.com/rickb777/date"  
6 )  
7  
8 func main() {  
9     date := date.New(2018, time.November, 27)  
10    fmt.Println(date)  
11 }
```

test

- `_test.go`というサフィックスを持つファイルに対しテストを実行
- 詳細は後述のテストの項目で説明

build

- ソースコードをコンパイルし、実行ファイルを出力する
- 開発環境とは別の環境に向けた実行ファイルでも、オプションで指定すれば、コンパイルすることができる

build

実行方法

```
1 # 対象ファイルをビルド
2 $ go build main.go
3
4 # 出力ファイルの名前を指定して対象ファイルをビルド
5 $ go build -o output main.go
6
7 # 対象ファイル名は省略可能
8 $ go build .
9 $ go build ./
```

※-oオプション無しの場合、goファイルの拡張子なしの名前が出力ファイル名になる

build

Windows向けのバイナリを出力する例

```
1 GOOS=windows GOARCH=386 go build main.go
```

※開発環境がLinux、macOS等であっても、main.exeという名前で出力される

Go標準パッケージ

standard packages

標準パッケージとは

- Go自身に含まれるライブラリで、go getせずに使える
- 入出力周りやJSONパース、サーバ起動などの機能が提供されており、これだけでもある程度の処理を作ることができる

紹介するパッケージ

fmt	標準入力・標準出力を扱う
io	データの入出力を扱う (Writer / Reader)
log	ログ出力機能を扱う

fmt

標準入力された文字列を標準出力する

```
1 fmt.Println("入力してください")
2
3 var s string
4 fmt.Scan(&s)
5
6 fmt.Printf("入力された値は %s です", s)
7 fmt.Println()
```

io

先程の標準入力、標準出力のコードを、ReaderとWriterに置き換えてみる

```
1 fmt.Println("入力してください")
2 var reader io.Reader = os.Stdin
3 var writer io.Writer = os.Stdout
4 for i := 0; i < 5; i++ {
5     var s string
6     fmt.Fscan(reader, &s)
7     fmt.Fprintf(writer, "入力された文字=%s\n", s)
8 }
```

Reader/Writerはインターフェースとして定義されており、よく使われる

log

標準出力の対象をLoggerに変更し、標準入力の内容をログ出力する

```
1 writer := os.Stdout
2 var logger *(log.Logger)
3 logger = log.New(writer, "[HANDSON_LOG]", log.LstdFlags|log.Lshortfile)
4
5 var reader io.Reader = os.Stdin
6 for i := 0; i < 5; i++ {
7     fmt.Print("入力してください:")
8     var s string
9     fmt.Fscan(reader, &s)
10    logger.Printf("%s\n", s)
11 }
12
13 logger.Fatal("!!!error exit!!!")
```

log

実行結果

```
1 入力してください:test
2 [HANDSON_LOG]2018/11/22 14:41:20 main.go:54: test
3 入力してください:test
4 [HANDSON_LOG]2018/11/22 14:41:21 main.go:54: test
5 入力してください:test
6 [HANDSON_LOG]2018/11/22 14:41:21 main.go:54: test
7 入力してください:test
8 [HANDSON_LOG]2018/11/22 14:41:22 main.go:54: test
9 入力してください:test
10 [HANDSON_LOG]2018/11/22 14:41:23 main.go:54: test
11 [HANDSON_LOG]2018/11/22 14:41:23 main.go:58: !!!error exit!!!
12 exit status 1
```


レシーバー

receivers

レシーバーとは

- 定義した構造体に対して、関数を実装することができる
- 構造体のデータ取出／変更を、処理とセットにする事ができる
- 構造体の値に定義する「値レシーバー」
- 構造体のポインタ型に定義する「ポインタレシーバー」

レシーバーの定義

サンプルとしてPerson構造体を作成

```
1 type Person struct {  
2     Age int  
3     Name string  
4 }
```

レシーバーの定義

AgeとNameに対するレシーバーを作成

```
1 // 値レシーバ
2 func (p Person) MyNameIs() {
3     fmt.Println("My name is " + p.Name)
4 }
5
6 // ポインタレシーバー
7 func (p *Person) afterYear() {
8     p.Age++
9 }
```

レシーバーの定義

定義したレシーバーの呼び出し

```
1 p := &Person{
2     Age: 29,
3     Name: "kent",
4 }
5
6 p.MyNameIs()           // kentが出力される
7
8 fmt.Println(p.Age)    // 29が出力される
9 p.afterYear()
10 fmt.Println(p.Age)   // 30が出力される
```

値レシーバーとポインタレシーバー

- メモリ効率と用途によって、主にポインタレシーバーが使われる
- 値レシーバーでは呼び出し時に構造体自体を一旦コピーし、呼び出し元の構造体とは別のものが処理対象になる
- ポインタレシーバーなら呼び出し元の値を変更することができる

値レシーバーとポインタレシーバー

下記のように値レシーバーで呼び出してしまうと、出力結果は29のまま

```
1 // afterYear()と同じ処理だけど、こちらは値レシーバー
2 func (p Person) afterYearDummy() {
3     p.Age++
4 }
```

インターフェース

interfaces

インターフェースとは

- 型が持つべき関数を定義することで、パッケージ間でやり取りする型を抽象化することができる
- これにより構造体の型を隠蔽し、処理の抽象度を上げることができる
- ここではインターフェースの定義方法と、基礎的な使い方を説明する

インターフェースの定義

前章のPersonを踏襲する形でインターフェースを定義

```
1 type Human interface {  
2     MyNameIs()  
3 }
```

インターフェースを受け取る

インターフェースを引数に取る関数を作る

```
1 func whoAreYou(h Human) {  
2     h.MyNameIs()  
3 }
```

インターフェースを使う

インターフェースの定義を満たす構造体を定義する、下記は前章のPerson

```
1 type Person struct {  
2     Age int  
3     Name string  
4 }  
5  
6 func (p Person) MyNameIs() {  
7     fmt.Println("My name is " + p.Name)  
8 }
```

インターフェースを使う

Person構造体はHumanインターフェースの定義を満たすため、whoAreYou()を実行できる

```
1 p := Person{  
2     Name: "kent",  
3     Age:  29,  
4 }  
5  
6 whoAreYou(p)
```

「My name is kent」が出力される

インターフェースを使う

Personとは別に、新しく構造体を作る

```
1 type Robot struct {  
2     Id string  
3 }  
4  
5 func (r Robot) MyNameIs() {  
6     fmt.Println("My id is " + r.Id)  
7 }
```

インターフェースを使う

Humanインターフェースの定義を満たすので、RobotもwhoAreYou()を実行できる

```
1 r := Robot{  
2     Id: "abcd",  
3 }  
4  
5 whoAreYou(r)
```

「My id is abcd」が出力される

引数、返却値がある関数の定義

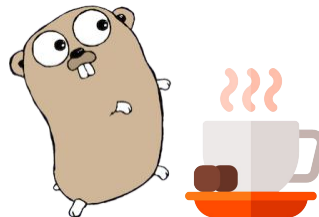
インターフェースの関数に引数、返却値を定義する

```
1 type Duck interface {  
2     Walk() int  
3     Bark(string)  
4     StringBark(string) string  
5 }
```

※引数は変数名を書いておくこともできる

Short break

(5 min)

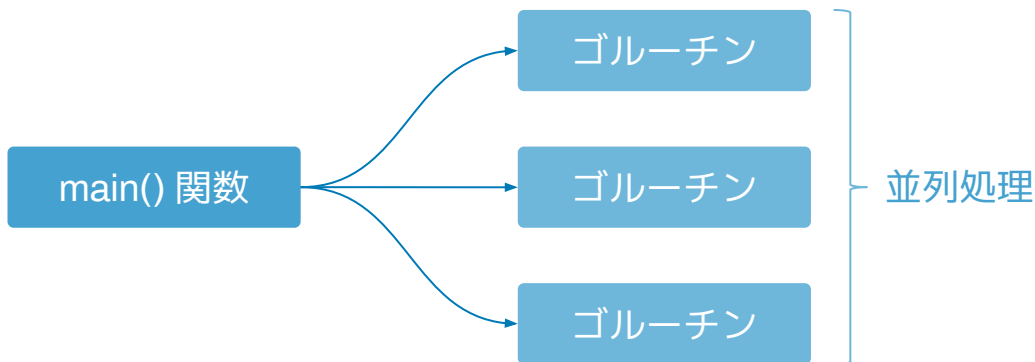


並列処理 ゴルーチン

go routines

ゴルーチンとは

- ゴルーチンはGo言語で並行処理を表すものです
- 手続き実行ではなく、並列での関数の実行に使うことができます
- ゴルーチンは呼び出し元の処理をブロックせずに実行されます



ゴルーチンの使い方

実行方法

```
1 func main() {  
2     go SayHello()  
3  
4     go func() {  
5         fmt.Println("Hello golang!")  
6     }()  
7 }  
8  
9 func SayHello() {  
10     fmt.Println("Hello golang!")  
11 }
```

- 関数/無名関数を扱える
- 呼出元の処理をブロックしない

ゴルーチンの処理順序

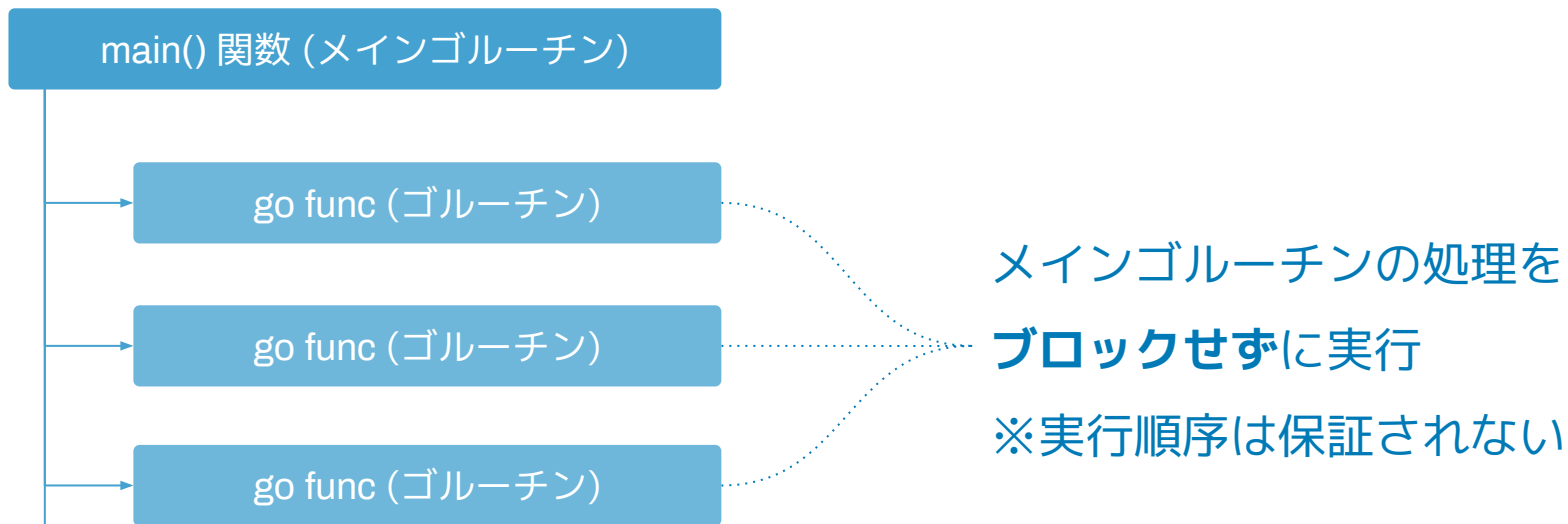
実行内容

```
1 fmt.Println("Hello golang1!")
2
3 go func() {
4     fmt.Println("Hello golang2!")
5 }()
6
7 fmt.Println("Hello golang3!")
8 time.Sleep(3 * time.Second)
```

ゴルーチンの処理順序

実行結果

```
Hello golang1!  
Hello golang3!  
Hello golang2!
```



main() 関数が終了すると、実行中の全てのゴルーチンを中断します

※実行中のゴルーチンがあっても、完了を待ちません

並列処理 チャンネル

channels

チャンネルとは

- チャンネルはゴルーチン間のメッセージのやり取りをするものです
- メッセージは型を指定できます
- チャンネルは容量 / 送信 / 受信 / クローズで構成されています

チャンネルの使い方

実行方法

```
1 ch := make(chan int, 5) //容量5のチャンネルを初期化
2
3 go func() {
4     ch <- 99 //送信文
5     close(ch) //チャンネルを閉じる
6 }()
7
8 v, closed := <-ch //受信の2番目の返り値でcloseかどうかの判定
9
10 fmt.Println(v) //99
11 fmt.Println(closed) //true
```

チャンネルのキャパシティ

- チャンネルにはキャパシティ(容量)があります
- チャンネルのキャパシティは送信された値を保管できる容量です
- 受信側が待機している場合は、バッファは使用されません
- 容量が足りなくなると、チャンネルへの送信はブロックされます

ゴルーチンの使い方

実行方法

```
1 func main() {  
2     go SayHello()  
3  
4     go func() {  
5         fmt.Println("Hello golang!")  
6     }()  
7 }  
8  
9 func SayHello() {  
10     fmt.Println("Hello golang!")  
11 }
```

- 関数/無名関数を扱える
- 呼出元の処理をブロックしない

送信チャンネルのブロック

サンプルコード

```
1 ch := make(chan int) //容量0のチャンネル
2
3 go func() {
4     //受信がないので、送信側はバッファに入れようと思いますが、
5     //容量が0なのでブロックされます
6     ch <- 99
7     fmt.Println("送信できない!")
8 }()
```

チャンネルバッファを使わない送受信

サンプルコード

```
1 ch := make(chan int) //容量0のチャンネル
2
3 go func() {
4     //受信があるので、バッファは使われずに送信できる
5     ch <- 99
6     fmt.Println("送信できる!")
7 }()
8 fmt.Println(<-ch)
```

チャンネルバッファを使った送信

サンプルコード

```
1 ch := make(chan int, 1) //容量1のチャンネル
2
3 go func() {
4     //バッファに容量があり、送信するとバッファに入る為、
5     //受信がなくてもブロックされない
6     ch <- 99
7     fmt.Println("送信できる!")
8 }()
```

並列処理の完了を検知する

- 並列処理の完了を検知したい時、チャンネルは有効です
- チャンネルは使い方によっては問題を起こしてしまう事もあります
- 以下の例では受信側が処理の完了を待ち続けてしまいます

完了検知のアンチパターン

サンプルコード

```
1 done := make(chan bool)
2
3 go func() {
4     //... 何かの処理
5
6     //ここで例外などでゴルーチンを抜けてしまった場合、
7     //受信側がdoneへの送信を待ち続けてしまう
8     done <- true
9 }()
10
11 fmt.Println(<-done)
12 fmt.Println("処理完了!")
```

deferとcloseで完了を検知する

サンプルコード

```
1 done := make(chan bool)
2
3 go func() {
4     defer close(done)
5     //... 何かの処理
6     //何が起きても最後にcloseされる
7 }()
8
9 fmt.Println(<-done)
10 fmt.Println("処理完了!")
```

コーディング規約

coding standards

コード整形とコードスタイル

- Go言語は標準のgofmtを使ってコード整形をサポートしています
- Go言語の実装コードで挙げたレビューコメントが公開されています
- これらを厳密に守る必要はないですが、コードスタイルに悩んだ時の参考にできます

 <https://qiita.com/knsh14/items/8b73b31822c109d4c497>

ユニットテスト

unit tests

標準パッケージによるユニットテスト

- Go言語は標準のtestingパッケージでユニットテスト、ベンチマークテストをサポートしています
- `_test.go`をつけて、テストとして実行します

テストの実行

- 実際に上記のテストを実行してみましょう
- `go test`で実行ができます
- `go test -v`で詳細な実行結果を出力する事もできます
- `go test -cover`でコードカバレッジも出力する事もできます

テスト対象

サンプルコード

```
1 package example1
2
3 func Sum(x int, y int) int {
4     return x + y
5 }
```


ユニットテストコード

サンプルコード

```
1 package example1
2
3 import (
4     "testing"
5 )
6
7 func TestSum(t *testing.T) {
8     if example1.Sum(1, 2) != 3 {
9         t.Error("エラーです!")
10    }
11 }
```

テストとして実行する場合はTestから始まる関数名にする

テストの実行

実行結果

```
$ go test
PASS
ok      _/home/yukpiz/labo/repos/private/go-test/example1  0.001s

$ go test -v
=== RUN   TestSum
--- PASS: TestSum (0.00s) PASS
ok      _/home/yukpiz/labo/repos/private/go-test/example1  0.001s

$ go test -v -cover
=== RUN   TestSum
--- PASS: TestSum (0.00s)
PASS coverage: 100.0% of statements
ok      _/home/yukpiz/labo/repos/private/go-test/example1  0.001s
```


課題

20:20 ~ 21:20 (60 min)



本日の課題

 じゃんけんシミュレーターを作ろう

 会話ゲームを作ろう

じゃんけんシミュレーターを作ろう

基本問題

- グー、チョキ、パーからランダムで1つを出す2CPを作しましょう
- 結果は勝ち、負け、あいこのどれかです
- ゴルーチンやチャネルを使う必要はありません
- 100万回試行して、勝敗をカウントして表示しましょう

発展問題

- 片方のCPを50%の確率でパーを出すように設定して、実行しましょう

じゃんけんシミュレーターを作ろう

実行結果の例

勝利=340000回

引分=330000回

敗北=300000回

合計=1000000回

会話ゲームを作ろう

基本問題

- ゴルーチンとチャンネルを使って、関数同士で会話をさせましょう
- 登場人物は2人です
- 各発言はそれぞれの名前がついた関数から発言します

発展問題

- 登場人物を3人に増やしましょう

会話ゲームを作ろう

実行結果の例

kent: あなたの名前を教えてください

yukpiz: 私はゆくぴずです

kent: 私はけんとです

yukpiz: よろしくおねがいします

kent: 生ハム食べに行きましょう

yukpiz: 行きましょう

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
- 解答時間中は自由にご飲食・ご休憩ください
- 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

最後に

21:20 ~ 21:30 (10 min)



解答例は以下URLよりご覧ください

 <https://qiita.com/yukpiz/items/ef15557de423a2ccb230>

ハンズオン後のサポートについて

閉会後に引き続きご質問がある場合、Slackでサポートします

以下のワークスペースにジョインし、#support-201811チャンネルでお気軽にご質問ください

 bit.ly/techdo-slack

次回開催予定のご案内

84

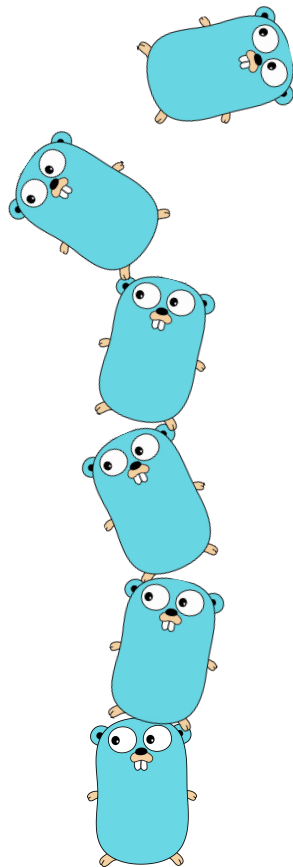
2019年2月上旬頃（予定）

さらに進化した初心者向けハンズオンがここに・・・！

 <https://techdo.connpass.com>

※詳細が決まり次第、TwitterやConnpassグループ等で告知します

また次回もご参加ください！





Twitter: @yukpiz

- 都内でエンジニアとして活動しています
- エンジニアのアウトプットと成長

#golang #vim

#小型船舶操縦士1級

#engineers_it

エンジニアの登壇を応援する会の運営メンバー

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

@k_h_sissp



スタッフに拍手を！



kentfordev

@k_h_sissp

三澤 悠介

前川 尚輝

奥野 遥



yukpiz

@yukpiz

山澤 沙也加

池田 奈美

三森 泰規

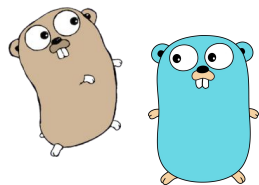


ariaki

アンケートにご協力ください

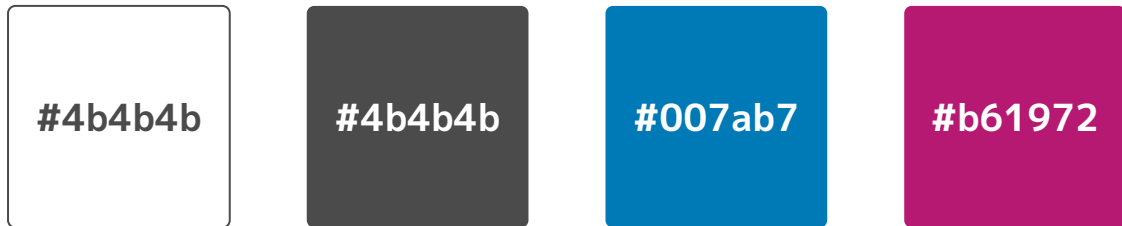


<http://bit.ly/go20181127>



The Go gopher was designed by Renée French.

Go Hands-on #2



#4b4b4b

#007ab7

#b61972

あいうえおかきくけこさしすせそ

あいうえおかきくけこさしすせそ

あいうえおかきくけこさしすせそ