Go 言語入門

tsuji

2019 年 **06** 月 **24** 日

目次

注釈: 本ドキュメントは、もともと Java をメインに使用していた筆者が Go 言語のキャッチアップのために作成されました。誤り、Typo 等ありましたら、ご連絡いただけると助かります。

第 1 章

はじめに

1.1 Go 言語とは

Go 言語は 2009 年に Google にいた Robert Griesemer、Rob Pike、Ken Thompson によって生み出されたプログラミング言語です。Go 言語は以下のような特徴を持ちます。

- C 言語、Pascal、CSP を起源とした言語
- 静的型付けのコンパイル言語
- 自動メモリ管理で、GC の機構を持つ
- CSP スタイルの並行性
- シンプルな言語仕様
- 継承がない

1.2 セットアップ

課題: セットアップの手順を記載する

1.3 環境変数

Go 言語で設定されている環境変数を確認します。以下の go env コマンドで確認することができます。

```
$ go env
set GOARCH=amd64
```

(次のページに続く)

(前のページからの続き)

```
set GOBIN=
set GOCACHE=C:\Users\Dai\AppData\Local\go-build
set GOEXE=.exe
set GOHOSTARCH=amd64
set GOHOSTOS=windows
set GOOS=windows
set GOPATH=C:\Users\Dai\go
set GORACE=
set GOROOT=C:\Go
set GOTMPDIR=
set GOTOOLDIR=C:\Go\pkg\tool\windows_amd64
set GCCGO=gccgo
set CC=gcc
set CXX=g++
set CGO_ENABLED=1
set CGO_CFLAGS=-g -O2
set CGO_CPPFLAGS=
set CGO_CXXFLAGS=-g -O2
set CGO_FFLAGS=-g -O2
set CGO_LDFLAGS=-g -O2
set PKG_CONFIG=pkg-config
set GOGCCFLAGS=-m64 -mthreads -fmessage-length=0 -fdebug-prefix-
↪map=C:\Users\Dai\AppData\Local\Temp\go-build095523442=/tmp/go-build -gno-record-gcc-
↪switches
```

重要な環境変数は以下の 2 つです。

- GOROOT
- GOPATH

GOROOT は、Go がインストールされているディレクトリです。GOPATH は、外部のパッケージなどのソースを取りまとめておくディレクトリを指定するものになります。$HOME/go や $HOME/.go にするのが一般的でしょう。

1.4 コンパイル／実行

まずは Hello world を出力するプログラムを書いてみます。

リスト 1　sample.go

```go
package main

import "fmt"

func main() {
    fmt.Println("Hello world.")
}
```

main パッケージはライブラリなしで動作する実行可能なプログラムを定義します。main パッケージの main 関数からプログラムが実行されます。まずはこのプログラムを動かしてみます。

```
$ go run sample.go
```

Hello world. がコンソールに出力されたことがわかります。このプログラムは import として Go 言語に内包されている fmt パッケージを用いて標準出力しています。go run した場合はコンパイラがコンパイルしたプログラムが実行されますが、実行ファイルは生成されません。

Go 言語のビルドは以下のように実行できます。

```
$ go build sample.go
```

ビルドすると sample.exe という実行可能ファイルが生成されることがわかります。ビルドすると go run せずとも実行可能ファイルを実行することでプログラムを実行できます。

```
$ sample.exe
```

1.5 パッケージのインストール

外部パッケージを利用する場合は、go get コマンドを用います。たとえば以下のように実行すると $GOPATH/src 配下に golang/lint パッケージがインストールされることがわかります。

```
$ go get -v github.com/golang/lint
```

第 2 章

文法

2.1 基本型

Go 言語の基本型は以下です。

- bool
- string
- int (※ int 型は bit 数や符号なしなどいろいろありますが、基本は int です)
- float64 (,float32)
- complex128 (,complex64)

2.2 変数宣言

変数の定義は var でできます。

```
var i int
```

初期値を与えることができます。

```
var i int = 1
```

初期値を与えずに宣言すると、ゼロ値が与えられます。

```
func main() {
    var i int
    var f float64
    var b bool
    var s string
```

(次のページに続く)

(前のページからの続き)

```
    fmt.Printf("%v %v %v %q\n", i, f, b, s)
}
// 0 0 false ""
```

関数の中では暗黙的な型宣言ができます。変数 i の型が int 型になっていることがわかります。シンプルでかつ柔軟であることからローカル変数の宣言にはこの省略変数宣言によって初期化されることが多いです。

```
func main() {
    i := 2
    fmt.Println(i)
    fmt.Println(reflect.TypeOf(i))
}
// 2
// int
```

定数は Const キーワードを用いて宣言します。

```
const Pi = 3.141592
```

定数を上書きしようとするとコンパイラエラーが発生します。

```
const Pi = 3
const Pi = 3.141592
```

第 3 章

演算子とデータ型

3.1 演算子

課題: 演算子の説明を記載する

3.2 データ型

以下の基本的な型については省略。

3.3 基本型と複合型

3.4 型宣言

3.5 数値型

3.6 文字列型

3.7 真偽値型

課題: 省略した型の説明を記載する。

3.8 構造体型

struct はフィールドの集まりです。

```
type Vertex struct {
    x, y int
}

func main() {
    fmt.Println(Vertex{1, 2})
}
// {1 2}
```

フィールド値を指定して初期化することができます。

```
type Vertex struct {
    x, y int
}

func main() {
    fmt.Println(Vertex{y: 2, x: 1})
}
// {1 2}
```

3.9 ポインタ型

Go はポイントを扱います。ポインタの値はメモリアドレスを示します。

変数 T 型のポイントは &T 型で、ゼロ値は nil です。

- オペレータはポインタの指し示す変数を示します。

```
func main() {
    i := 1
    p := &i
    fmt.Println(p)
    fmt.Println(*p)

    var q *int
    fmt.Println(q)
}
// 0xc0420080b8
// 1
// <nil>
```

3.10 配列型

[n]T として T 型の n 個の配列を宣言することができます。配列の長さは型の一部分なので配列のサイズを変えることはできません。

```
func main() {
    var strs [10]string
    for i := 0; i < 10; i++ {
        strs[i] = strconv.Itoa(i)
    }

    for _, v := range strs {
        fmt.Print(v)
    }
}
// 0123456789
```

3.11 スライス型

配列型では長さは固定長でしたが、スライスを用いると可変長の長さを扱うことができます。

[]T は T 型のスライスです。

```
func main() {
    var strs []string
    for i := 0; i < 10; i++ {
        strs = append(strs, strconv.Itoa(i))
    }
    fmt.Println(strs)
}
// [0 1 2 3 4 5 6 7 8 9]
```

3.12 マップ型

map[T]S で T 型と S 型のキーバリューを関連付ける map を生成することができます。

make 関数は指定された型の、初期化され使用できるようにしたマップを返します。

```
func main() {
    var m = make(map[string]int)
    m["hoge"] = 1
    m["fuga"] = 2
    fmt.Println(m["hoge"])
```

(次のページに続く)

(前のページからの続き)

```
}
// 1
```

マップ (m) に対して以下の操作ができます。

要素の挿入や更新

```
m[key] = value
```

要素の取得

```
value = m[key]
```

要素の削除

```
delete(m, key)
```

キーの存在チェック
 要素を取得したときの第 2 引数に bool の要素で return されます。

```
value, ok = m[key]
```

```
func main() {
    m := make(map[string]int)

    m["Answer"] = 42
    fmt.Println("The value:", m["Answer"])

    m["Answer"] = 48
    fmt.Println("The value:", m["Answer"])

    delete(m, "Answer")
    fmt.Println("The value:", m["Answer"])

    v, ok := m["Answer"]
    fmt.Println("The value:", v, "Present?", ok)
}
// The value: 42
// The value: 48
// The value: 0
// The value: 0 Present? false
```

以下は省略

- その他の型
- リテラル

課題:

- その他の型
- リテラル

省略した基本型について記載する。

3.13 nil

nil は値がないことを示す特殊な値です。Java でいうところの null と同じと考えて良いでしょう。nil は比較演算子 !=, == を用いて比較することができます。

3.14 型キャスト

課題: 省略した型キャストについて記載する。

第 4 章

制御文

4.1 条件分岐

if 文は以下のように書けます。(,) は不要です。

```
func main() {
    rand.Seed(time.Now().UnixNano())
    i := rand.Int()
    fmt.Println(i)
    if i % 2 == 1 {
        fmt.Println("odd")
    } else {
        fmt.Println("even")
    }
}
```

if 文内のみのスコープの変数と合わせて使うことができます。

```
rand.Seed(time.Now().UnixNano())
if i := rand.Int(); i % 2 == 1 {
        fmt.Println("odd")
} else {
        fmt.Println("even")
}
```

switch 文は以下のように書けます。

```
rand.Seed(time.Now().UnixNano())
i := rand.Int()
switch {
case i % 15 == 0:
        fmt.Println("FizzBuzz")
case i % 3 == 0:
```

(次のページに続く)

(前のページからの続き)

```
        fmt.Println("Fizz")
case i % 5 == 0:
        fmt.Println("Buzz")
default:
        fmt.Println(i)
}
```

4.2 繰り返し

ベーシックな for 文は以下のように記述します。

```
sum := 0
for i := 0; i <= 10; i++ {
        sum += i
}
```

Java の *while* キーワードはなく、*while* ループは for 文で記述することになります。

```
sum, i := 0, 0
for i <= 10 {
        sum += i
        i++
}
```

これは無限ループです。

```
for {
}
```

for 文でループする際に range を用いることができます。スライスやマップをひとつずつ反復処理する場合に用いられます。

```
func main() {
    var pow []int
    for i, j := 0, 1; i < 10; i++ {
        pow = append(pow, j)
        j *= 2
    }
    for i, v := range pow {
        fmt.Printf("2**%d = %d\n", i, v)
    }
}
// 2**0 = 1
// 2**1 = 2
```

(次のページに続く)

(前のページからの続き)

```
// 2**2 = 4
// 2**3 = 8
// 2**4 = 16
// 2**5 = 32
// 2**6 = 64
// 2**7 = 128
// 2**8 = 256
// 2**9 = 512
```

ループの index が不要であれば _ で捨てることができます。

```
func main() {
    var pow []int
    for i, j := 0, 1; i < 10; i++ {
        pow = append(pow, j)
        j *= 2
    }
    for _, v := range pow {
        fmt.Printf("%d ", v)
    }
}
// 1 2 4 8 16 32 64 128 256 512
```

4.3 goto

課題: goto の説明を記載する

4.4 例外処理

Go のプログラムは、エラーの状態を error 値で表現します。以下の組み込みのインターフェースがあります。

```
type error interface {
    Error() string
}
```

エラーハンドインターフェースとも関連しますが、標準関数を用いるときの例外処理は以下のように err の変数が nil であるかの判定をすることが一般的です。

```
func main() {
    file, err := os.Open("test.txt")
```

(次のページに続く)

(前のページからの続き)

```
    if err != nil {
        fmt.Fprint(os.Stderr, err.Error())
        os.Exit(1)
    }
    defer file.Close()
    fmt.Println("ok")
}
// open test.txt: The system cannot find the file specified.
```

第 5 章

関数

5.1 関数定義

Go の関数のシグネチャは最初に変数でその後に型となります。以下のようになります。

```
func add(x int, y int) int {
    return x + y;
}
```

型が同じ場合は、型を省略することができます。

```
func add(x, y int) int {
    return x + y;
}
```

関数の戻り値として複数の値を返すことができます。

```
func swap(s, t string) (string, string) {
    return t, s
}
```

5.2 関数型

関数も変数です。よって変数のように関数を渡すことができます。

```
func add(fn func(int, int) int) int {
    return fn(3, 4)
}

func main() {
    vFunc := func(x, y int) int {
```

(次のページに続く)

(前のページからの続き)

```
        return x*x + y*y
    }
    fmt.Println(vFunc(2, 3))
    fmt.Println(add(vFunc))
}
```

5.3 クロージャ

Go の関数はクロージャ[*1] として定義することができます。

```
func adder() func(int) int {
    // main 関数で呼ばれた関数は adder() 関数内の sum を参照している
    sum := 0
    return func(x int) int {
        sum += x
        return sum
    }
}

func main() {
    pos, neg := adder(), adder()
    for i := 1; i <= 10; i++ {
        fmt.Printf("%d %d\n", pos(i), neg(-2*i))
    }
}
```

Exercise: Fibonacci closure の実装の一例は以下のようになります。

```
// fibonacci is a function that returns
// a function that returns an int.
func fibonacci() func() int {
    a1, a2 := 0, 1
    return func() int {
        ret := a1
        a1 = a2
        a2 += ret
        return ret
    }
}

func main() {
    f := fibonacci()
    for i := 0; i < 10; i++ {
```

(次のページに続く)

[*1] https://ja.wikipedia.org/wiki/%E3%82%AF%E3%83%AD%E3%83%BC%E3%82%B8%E3%83%A3

(前のページからの続き)

```
        fmt.Println(f())
    }
}
```

5.4 defer 文

defer 文は呼び出し元の関数が return するまで処理を遅延させることができます。defef は通常さまざまなクリーンアップ処理を実行する機能を単純化するために用いられます。

```
func helloworld() {
    defer fmt.Println("world")
    fmt.Print("hello ")
}

func main() {
    helloworld()
}
// hello world
```

defer 文は stack です。

```
func main() {
    for i := 1; i < 10; i++ {
        defer fmt.Print(i)
    }
}
// 987654321
```

5.5 パニック

致命的なエラーとして呼び出し元でエラーハンドリングする必要のない、もしくはエラーが起きても本当にリカバリが必要なときだけエラーハンドリングさせたい場合に panic 関数を呼び出してパニックという状態を作ることができます。

Go 言語に組み込まれている panic 関数は以下です。

```
func panic(interface{})
```

panic を起こしてみましょう。

```
func f() {
    panic("panic occured!")
```

(次のページに続く)

(前のページからの続き)

```
}

func main() {
    f()
}
// panic: panic occured!

// goroutine 1 [running]:
// main.f()
//  C:/Users/testUser/go/src/sample/sample.go:4 +0x40
// main.main()
//  C:/Users/testUser/go/src/sample/sample.go:8 +0x27
```

配列外参照のなどの実行時エラーの場合もパニックが発生します。

```
func main() {
    array := [...]int{1, 2, 3}
    for i := 0;; i++ {
        fmt.Println(array[i])
    }
}
```

パニック発生時でも defer 関数は実行されます。

```
func f() {
    panic("panic occured!")
}

func main() {
    defer fmt.Println("do something...")
    f()
}
// do something...
// panic: panic occured!

// goroutine 1 [running]:
// main.f()
//  C:/Users/testUser/go/src/sample/sample.go:6 +0x40
// main.main()
//  C:/Users/testUser/go/src/sample/sample.go:11 +0xae
```

recover 関数を呼び出して処理することでパニックを中断させることができます。参考[*2] の内容になります。

```
func main() {
    f()
```

(次のページに続く)

[*2] https://blog.golang.org/defer-panic-and-recover

(前のページからの続き)

```
    fmt.Println("Returned normally from f.")
}

func f() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("Recovered in f", r)
        }
    }()
    fmt.Println("Calling g.")
    g(0)
    fmt.Println("Returned normally from g.")
}

func g(i int) {
    if i > 3 {
        fmt.Println("Panicking!")
        panic(fmt.Sprintf("%v", i))
    }
    defer fmt.Println("Defer in g", i)
    fmt.Println("Printing in g", i)
    g(i + 1)
}
// Calling g.
// Printing in g 0
// Printing in g 1
// Printing in g 2
// Printing in g 3
// Panicking!
// Defer in g 3
// Defer in g 2
// Defer in g 1
// Defer in g 0
// Recovered in f 4
// Returned normally from f.
```

第 6 章

メソッドとインターフェース

6.1 メソッド

Go は型にメソッドを定義することができます。メソッドはレシーバを引数にとる関数となります。

Abs メソッドは Vertex 型のレシーバを持つことを意味しています。

```
type Vertex struct {
    X, Y float64
}

func (v Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func main() {
    v := Vertex{X: 3, Y: 4}
    fmt.Println(v.Abs())
}
// 5
```

メソッドはレシーバを伴う関数ですので、メソッドでない関数として記述することも可能です。

```
type Vertex struct {
    X, Y float64
}

func AbsFunc(v Vertex) float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func main() {
    v := Vertex{X: 3, Y: 4}
    fmt.Println(AbsFunc(v))
```

(次のページに続く)

(前のページからの続き)

```
}
// 5
```

ポイントレシーバでメソッドを宣言することができます。レシーバの変数を更新する必要がある場合はポイントレシーバにする必要があります。

```
type Vertex struct {
    X, Y float64
}

func (v Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func (v *Vertex) Scale(f float64) {
    v.X = v.X * f
    v.Y = v.Y * f
}

func main() {
    v := Vertex{3, 4}
    v.Scale(10)
    fmt.Println(v.Abs())
}
```

6.2 インターフェース

インターフェース型を定義できます。インターフェースはメソッドのシグネチャの集まりで定義します。構造体などの具象型が満たすべき仕様といえます。Java では implement が必要でしたが、Go では具象型が明示的にインターフェースを実装することを示す必要がありません。

以下は A Tour of Go の例です。T 型の関数 M がインターフェース型 I の関数 M を実装していることになります。

```
type I interface {
    M()
}

type T struct {
    S string
}

func (t T) M() {
    fmt.Println(t.S)
}
```

(次のページに続く)

(前のページからの続き)

```
func main() {
    var i I = T{"hello"}
    i.M()
}
```

インターフェースに定義されている関数をメソッドとして実装している型は自動的にそのインターフェースを実装していることになります。

インターフェースにある具体的な値が nil の場合はメソッドは nil レシーバーとして呼び出されます。Go では nil をレシーバーとして呼び出されても適切に処理するメソッドを記述するのが一般的です。以下は A Tour of Go の例です。(なお以下の例で nil の処理をしない場合はパニックが発生します。)

```
type I interface {
    M()
}

type T struct {
    S string
}

func (t *T) M() {
    if t == nil {
        fmt.Println("<nil>")
        return
    }
    fmt.Println(t.S)
}

func main() {
    var i I

    var t *T
    i = t
    describe(i)
    i.M()

    i = &T{"hello"}
    describe(i)
    i.M()
}

func describe(i I) {
    fmt.Printf("(%v, %T)\n", i, i)
}
// (<nil>, *main.T)
// <nil>
// (&{hello}, *main.T)
```

(次のページに続く)

(前のページからの続き)

```
// hello
```

nil インターフェースの値は nil です。以下は A Tour of Go の例です。

```
type I interface {
    M()
}

func main() {
    var i I
    describe(i)
    i.M()
}

func describe(i I) {
    fmt.Printf("(%v, %T)\n", i, i)
}
// (<nil>, <nil>)
// panic: runtime error: invalid memory address or nil pointer dereference
// [signal 0xc0000005 code=0x0 addr=0x0 pc=0x48cd4d]

// goroutine 1 [running]:
// main.main()
//  C:/Users/testUser/go/src/sample/sample.go:12 +0x3d
```

Java でいうところのポリモフィズムのような実装も可能です。

```
type Vehicle interface {
    run()
}

type Car struct {
    name string
}

func (c Car) run() {
    fmt.Println("Booom")
}

type Plane struct {
}

func (p Plane) run() {
    fmt.Println("Boooooooooooom")
}

func main() {
```

(次のページに続く)

(前のページからの続き)

```
    var vehicle Vehicle
    vehicle = &Car{}
    vehicle.run()
    vehicle = &Plane{}
    vehicle.run()
}
// Booom!
// Boooooooooooom!
```

6.2.1 エラーインタフェース

エラー型を error 値で表現します。 error 型は Go 言語に組み込まれているインターフェースです。

```
type error interface {
    Error() string
}
```

error 型を使うことで Go 言語として一貫した例外処理を記述することができます。error を実装した関数を記述した一例を紹介します。

```
type MyError struct {
    message string
}

func (err MyError) Error() string {
    return err.message
}

func throwError() (val string, err error) {
    return "hoge", MyError{"illegal statement"}
}

func main() {
    val, err := throwError()
    if err != nil {
        fmt.Fprint(os.Stderr, "error occured! val: " + val)
    }
}
// error occured! val: hoge
```

第 7 章

並行処理

7.1 goroutine

goroutine (ゴルーチン) は、Go のランタイムに管理される軽量なスレッドです。

go 関数 で goroutine として動作します。

並行して処理されていることがわかります。

```
func f() {
    for i := 0; i < 5; i++ {
        time.Sleep(1 * time.Second)
        fmt.Printf("%d ", i)
    }
}

func main() {
    go f()
    f()
}
// 0 0 1 1 2 2 3 3 4 4
```

なお以下のようにすると main 関数の処理が終了してしまい、goroutine が動作中に関わらず処理が終了します。

```
func f() {
    for i := 0; i < 5; i++ {
        time.Sleep(1 * time.Second)
        fmt.Printf("%d ", i)
    }
}

func main() {
    go f()
```

(次のページに続く)

(前のページからの続き)

```
    go f()
}
```

7.2 Channel

チャネル (Channel) 型は、チャネルオペレータの <- を用いて値の送受信ができる通り道です。同一プロセスのゴルーチン間での、通信・同期・値の共用、に使用します。Go のチャネルは FIFO キューに並行処理をしても正しく処理できることを保証する機能を組み合わせたものです。

Go ならわかるシステムプログラミング[*1] のよると、チャネルには以下の 3 つの性質があります。

> チャネルは、データを順序よく受け渡すためのデータ構造である
>
> チャネルは、並行処理されても正しくデータを受け渡す同期機構である
>
> チャネルは、読み込み・書き込みの準備ができるまでブロックする機能である

チャネル型は chan です。

チャネルの送受信の書式は以下のようになります。

```
chan 要素型
chan <- 送信する値 （送信専用チャネル）
<- chan （受信専用チャネル）
```

チャネルの生成は以下のように宣言します。

```
make(chan 要素型)
make(chan 要素型, キャパシティ)
```

チャネルを利用した送受信は以下のようになります。

```
チャネル <- 送信する値 // 送信
<- チャネル // 受信
```

以下は char <-string 型のチャネルを用いてメッセージをやりとりするサンプルです。

```
func main() {
    message := make(chan string, 10)

    go func(m chan<- string) {
        for i := 0; i < 5; i++ {
            if i%2 == 0 {
                m <- "Ping "
```

(次のページに続く)

[*1] https://www.lambdanote.com/products/go

(前のページからの続き)

```
            } else {
                m <- "Pong "
            }
        }
        close(m)
        fmt.Println("func() finished.")
    }(message)

    for {
        time.Sleep(1 * time.Second)
        msg, ok := <-message
        if !ok {
            break
        }
        fmt.Print(msg)
    }
}
// func() finished.
// Ping Pong Ping Pong Ping
```

チャネルは、キャパシティの有無によって動作が変わってきます。バッファつきのチャネルの場合、送信側はバッファがある限りすぐに送信を完了して、次の処理を実行できます。一方、バッファなしのチャネルの場合、送信後、受信されないとそれまで処理がブロックされます。

先程の例でチャネル生成時に以下のようにバッファありのチャネルを生成していました。

```
message := make(chan string, 10)
```

これを以下のようにバッファなしのチャネルに変更してみます。

```
message := make(chan string)
```

送信側は受信側でチャネルから受信されるまでブロッキングされるので、出力される順序が変わることがわかります。

```
func main() {
    message := make(chan string)

    go func(m chan<- string) {
        for i := 0; i < 5; i++ {
            if i%2 == 0 {
                m <- "Ping "
            } else {
                m <- "Pong "
            }
        }
```

(次のページに続く)

(前のページからの続き)

```
        close(m)
        fmt.Println("func() finished.")
    }(message)

    for {
        time.Sleep(1 * time.Second)
        msg, ok := <-message
        if !ok {
            break
        }
        fmt.Print(msg)
    }
}
// Ping Pong Ping Pong Ping func() finished.
```

7.3 Select

select は複数のチャネルに対して同時に送受信待ちを行うときに使用します。以下の例では、チャネルを通じてデータを送受信する場合に、データ送信用のチャネルとは別に、送受信が終わったことを示すチャネルを用いています。

```
func fibonacci(c, quit chan int) {
    x, y := 0, 1
    for {
        select {

        // (c) チャネルに値を送信
        case c <- x:
            x, y = y, x+y

            // (quit) チャネルから値を受信できるようになった場合は関数から return する
        case <-quit:
            fmt.Println("quit")
            return
        }
    }
}

func main() {
    c := make(chan int)
    quit := make(chan int)

    // 値の送受信をゴルーチンとして非同期処理する
    go func() {
        for i := 0; i < 10; i++ {
```

(次のページに続く)

(前のページからの続き)

```
            // チャネルから値を受信
            // チャネルが空の場合は、チャネルに値が送信されるまでブロッキング
            fmt.Printf("%v ", <-c)
            time.Sleep(500 * time.Millisecond)
        }

        // quit チャネルに値を送信することで fibonacci 関数から return させる
        quit <- 0
    }()

    fibonacci(c, quit)
}
// 0 1 1 2 3 5 8 13 21 34 quit
```

7.4 context

コンテキストの簡単な例を見てみます。

```
func main() {

    fmt.Println("start func()")
    ctx, cancel := context.WithCancel(context.Background())
    go func() {
        for i := 0; i < 5; i++ {
            fmt.Printf("%v second passes\n", i)
            time.Sleep(1 * time.Second)
        }
        fmt.Println("func() is finished")
        cancel()
    }()
    <-ctx.Done()
    fmt.Println("all tasks are finished")
}
// start func()
// 0 second passes
// 1 second passes
// 2 second passes
// 3 second passes
// 4 second passes
// func() is finished
// all tasks are finished
```

課題: コンテキストに関しては別で詳細を確認します。

第 8 章

参考

8.1 参考資料

- https://go-tour-jp.appspot.com/list/
- https://blog.golang.org/defer-panic-and-recover
- https://golang.org/ref/spec
- https://book.mynavi.jp/manatee/books/detail/id=64463
- https://www.amazon.co.jp/%E3%83%97%E3%83%AD%E3%82%B0%E3%83%A9%E3%83%9F%E3%83%B3%E3%82%B0%E7%B5%8C%E9%A8%93%E8%80%85%E3%81%8CGo%E8%A8%80%E8%AA%9E%E3%82%92%E6%9C%AC%E6%A0%BC%E7%9A%84%E3%81%AB%E5%8B%89%E5%BC%B7%E3%81%99%E3%82%8B%E5%89%8D%E3%81%AB%E8%AA%AD%E3%82%80%E3%81%9F%E3%82%81%E3%81%AE%E6%9C%AC-%E5%A4%A9%E7%94%B0%E5%A3%AB%E9%83%8E-ebook/dp/B06XJ86BFZ/
- https://www.lambdanote.com/products/go

課題: goto の説明を記載する

(元のエントリ は、 D:\workspace\book\go-introduction\source\Control\index.rst の 126 行目です)

課題: セットアップの手順を記載する

(元のエントリ は、 D:\workspace\book\go-introduction\source\Introduction\index.rst の 21 行目です)

課題: 演算子の説明を記載する

(元のエントリ は、 D:\workspace\book\go-introduction\source\OperatorsAndTypes\index.rst の 8 行目です)

課題: 省略した型の説明を記載する。

(元のエントリ は、 D:\workspace\book\go-introduction\source\OperatorsAndTypes\index.rst の 38 行目です)

課題:

- その他の型
- リテラル

省略した基本型について記載する。

(元のエントリ は、 D:\workspace\book\go-introduction\source\OperatorsAndTypes\index.rst の 208 行目です)

課題: 省略した型キャストについて記載する。

(元のエントリ は、 D:\workspace\book\go-introduction\source\OperatorsAndTypes\index.rst の 225 行目です)

課題: コンテキストに関しては別で詳細を確認します。

(元のエントリ は、 D:\workspace\book\go-introduction\source\Parallel\index.rst の 236 行目です)