Go 言語入門

tsuji

2019 年 **07** 月 **07** 日

基本事項

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

演算子

https://golang.org/ref/spec#Operators

Go では以下の演算子が定義されています。

```
Expression = UnaryExpr | Expression binary_op Expression .
UnaryExpr  = PrimaryExpr | unary_op UnaryExpr .

binary_op  = "||" | "&&" | rel_op | add_op | mul_op .
rel_op     = "==" | "!=" | "<" | "<=" | ">" | ">=" .
add_op     = "+" | "-" | "|" | "^" .
mul_op     = "*" | "/" | "%" | "<<" | ">>" | "&" | "&^" .

unary_op   = "+" | "-" | "!" | "^" | "*" | "&" | "<-" .
```

演算子の優先順位は以下のとおりです。数字が大きいほど優先順位が高いです。

```
Precedence    Operator
    5             *  /  %  <<  >>  &  &^
    4             +  -  |  ^
    3             ==  !=  <  <=  >  >=
    2             &&
    1             ||
```

第 3 章

データ型

3.1 基本型

3.1.1 数値型

整数値型

浮動小数点型

複素数型

3.1.2 文字列型

3.1.3 真偽値型

bool 型は論理値を表す型です。true または false の値を取ります。

```
b := true
```

3.2 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)
    }
```

(次のページに続く)

(前のページからの続き)

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

3.2.1 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}
```

3.3 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)
```

(次のページに続く)

(前のページからの続き)

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

3.4 インターフェース型

Go には interface{} 型があります。Java でいうところの Object 型に近い概念です。interface{} 型は任意の値の代入が可能です。

```
func main() {
    var x interface{}
    x = "hello world"
    x = 111
    x = 3.14
    fmt.Println(x)
    fmt.Printf("%T", x)
}
// 3.14
// float64
```

interface{}型は元の型で定義されている演算は定義されません。以下のコードはコンパイルエラーになります。

```
func main() {
    var x, y interface{}
    x, y = 1, 2
    fmt.Println(x + y) // コンパイルエラー
}
```

初期値は nil です。

```
func main() {
    var x interface{}
    fmt.Println(x)
    fmt.Printf("%T\n", x)
}
// <nil>
// <nil>
```

3.4.1 型アサーション

型アサーション は、インターフェースの値の基になる具体的な値を利用する手段を提供します。

以下は interface{} 型の x が T 型を保持していることをチェックします。

```
x.(T)
```

interface{} 型から int 型に代入することで演算が可能になります。

```
func main() {
    var x, y interface{}
    x, y = 1, 2
    a, b := x.(int), y.(int)
    fmt.Println(a + b)
}
```

型アサーションができない場合は panic が起こります。

```
func main() {
    var x, y interface{}
    x, y = 1, 2
    a, b := x.(float64), y.(float64)
    fmt.Println(a + b)
}
// panic: interface conversion: interface {} is int, not float64 [recovered]
//     panic: interface conversion: interface {} is int, not float64
//
// goroutine 19 [running]:
// testing.tRunner.func1(0xc0000ba100)
//     C:/Go/src/testing/testing.go:830 +0x399
// panic(0x5237a0, 0xc000066540)
//     C:/Go/src/runtime/panic.go:522 +0x1c3
// github.com/d-tsuji/goSample/types.Do()
//     C:/Users/Dai/go/src/github.com/d-tsuji/goSample/types/interface.go:8 +0x4c
// github.com/d-tsuji/goSample/types.TestDo(0xc0000ba100)
//     C:/Users/Dai/go/src/github.com/d-tsuji/goSample/types/interface_test.go:6 +0x27
// testing.tRunner(0xc0000ba100, 0x554ab0)
//     C:/Go/src/testing/testing.go:865 +0xc7
// created by testing.(*T).Run
//     C:/Go/src/testing/testing.go:916 +0x361
//
// Process finished with exit code 1
```

3.5 参照型

Go には参照型というデータ型があります。

- slice(スライス)
- map(マップ)
- channel(チャネル)

が標準で参照型のデータ型になります。

参照型の生成に関しては make によって生成します。

呼び出し形式	型 T	意味
make(T, n)	slice	要素数と容量が n である T 型のスライスを生成
make(T, n, m)	slice	要素数が n で容量が m である T 型のスライスを生成
make(T)	map	T 型のマップを生成
make(T, n)	map	T 型のマップを要素数 n をヒントにして生成
make(T)	channel	バッファのない T 型のチャネルを生成
make(T, n)	channel	バッファサイズ n の T 型のチャネルを生成

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]
```

要素の操作について、値型と参照型の違いを確認してみます。

リスト 1　NG 例：値型である配列の値を操作

```
func main() {
    a := [3]int{1, 2, 3}
    pow(a)
    fmt.Println(a)
}

func pow(a [3]int) {
    for i, v := range a {
```

(次のページに続く)

(前のページからの続き)

```
        a[i] = v * v
    }
}
// [1 2 3]
```

pow(a) として関数に配列を渡したものの、関数としては引数の配列の値が **コピー** されて操作されているので、main から渡した引数には影響を及ぼしていません。配列型の要素に対して正しく操作するには以下のように記述する必要があります。

リスト 2　OK 例：値型である配列の値をポインタで渡して操作

```
func main() {
    a := &[3]int{1, 2, 3}
    pow(a)
    fmt.Println(a)
}

func pow(a *[3]int) {
    for i, v := range a {
        a[i] = v * v
    }
}

// &[1 4 9]
```

同じようでもスライス型の場合は参照型なので、関数の引数として渡した時に参照に対して関数が操作します。そのため、想定どおりスライスの値が変更されることがわかります。

リスト 3　OK 例：スライスを操作

```
func main() {
    a := []int{1, 2, 3}
    pow(a)
    fmt.Println(a)
}

func pow(a []int) {
    for i, v := range a {
        a[i] = v * v
    }
}
// [1 4 9]
```

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"])
}
// 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
```

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

(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)

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

    for {
        time.Sleep(1 * time.Second)
        msg, ok := <-message
        if !ok {
```

(次のページに続く)

(前のページからの続き)

```
            break
        }
        fmt.Print(msg)
    }
}
// Ping Pong Ping Pong Ping func() finished.
```

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

任意の位置に移動する goto 文が Go には定義されています。

```
func main() {
    fmt.Println("A")
    goto Jump
    fmt.Println("B")
Jump:
    fmt.Println("C")
}
// A
// C
```

B が出力されておらず、スキップされていることがわかります。

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")
    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
```

注意しないといけない点として、 `os.Exit(1)` とようにプロセスが終了した場合には `defer` の結果を返さずに終了することになります。以下が `defer` の結果が得られず終了する例です。

```
func main() {
    for i := 1; i < 10; i++ {
        defer fmt.Print(i)
        log.Fatal("Error occured.")
    }
}
```

結果

```
// 2019/07/06 17:08:09 Error occured.
```

上記のように `defer fmt.Print(i)` の結果が return されることはありません。

エラー発生時などは defer による処理がなされずに終了するので、 main 関数以外では error を上位の関数に戻して、一番最後に main 関数の中で終了することが多いのではないでしょうか。

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!")
}

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() {
```

(次のページに続く)

(前のページからの続き)

```
    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()
    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!
```

(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)

```
// 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())
}
```

6.1.1 可視性

課題: メソッドの可視性について記載する

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()
}
```

(次のページに続く)

(前のページからの続き)

```
type T struct {
    S string
}

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

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)
```

(次のページに続く)

(前のページからの続き)

```
    i.M()
}

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

(次のページに続く)

(前のページからの続き)

```
type Plane struct {
}

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

func main() {
    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)
```

(次のページに続く)

(前のページからの続き)

```
    }
}
// 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()
}
```

ゴルーチンからゴルーチンを呼び出すことも可能です。

```
func main() {

    fmt.Printf("[%v] main() start.\n", time.Now())

    go Do2()
    time.Sleep(5 * time.Second)

    fmt.Printf("[%v] main() stopped.\n", time.Now())

}

func Do2() {
    fmt.Printf("[%v] Do2() start.\n", time.Now())
    func() {
        go heavyProcess()
    }()
    fmt.Printf("[%v] Do2() completed.\n", time.Now())
}

func heavyProcess() {
    fmt.Printf("[%v] heavyProcess() start.\n", time.Now())
    time.Sleep(3 * time.Second)
    fmt.Printf("[%v] heavyProcess() completed.\n", time.Now())
}
// [2019-07-01 07:53:19.1271209 +0900 JST m=+0.002000001] main() start.
// [2019-07-01 07:53:19.1631137 +0900 JST m=+0.037992801] Do2() start.
// [2019-07-01 07:53:19.1631137 +0900 JST m=+0.037992801] Do2() completed.
// [2019-07-01 07:53:19.1631137 +0900 JST m=+0.037992801] heavyProcess() start.
// [2019-07-01 07:53:22.1634087 +0900 JST m=+3.038287801] heavyProcess() completed.
// [2019-07-01 07:53:24.1636904 +0900 JST m=+5.038569501] main() stopped.
```

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。

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

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 "
            } 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)
    }
```

(次のページに続く)

(前のページからの続き)

```
}
// 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 "
            }
        }
        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.
```

バッファチャネルを使った別の例を見てます。次の例は同時実行数を制限するようなチャネルの使い方です。いわゆるセマフォです。

```
const concurrency = 3
const total = 10

func main() {

    semaphore := make(chan int, concurrency)

    go consume(semaphore)

    for i := 0; i < total; i++ {
        semaphore <- i
        fmt.Printf("[%v] Add Semaphore: num -> %d\n", time.Now(), i)
    }
}

func consume(semaphore chan int) {
    time.Sleep(3 * time.Second)
    for i := 0; i < total; i++ {
        _ = <-semaphore
        time.Sleep(500 * time.Millisecond)
    }
}
```

結果は以下のように最初にチャネルのバッファ上限である 3 つまで追加され、その後は 0.5 秒ごとに処理するゴルーチンの結果を待って逐次処理されます。ゴルーチンは最大 3 並列で処理されていることがわかります。

```
[2019-06-30 22:56:24.1686604 +0900 JST m=+0.002497701] Add Semaphore: num -> 0
[2019-06-30 22:56:24.2056768 +0900 JST m=+0.039514101] Add Semaphore: num -> 1
[2019-06-30 22:56:24.2056768 +0900 JST m=+0.039514101] Add Semaphore: num -> 2
[2019-06-30 22:56:27.1689285 +0900 JST m=+3.002765801] Add Semaphore: num -> 3
[2019-06-30 22:56:27.6691217 +0900 JST m=+3.502959001] Add Semaphore: num -> 4
[2019-06-30 22:56:28.1695923 +0900 JST m=+4.003429601] Add Semaphore: num -> 5
[2019-06-30 22:56:28.6700202 +0900 JST m=+4.503857501] Add Semaphore: num -> 6
[2019-06-30 22:56:29.1704991 +0900 JST m=+5.004336401] Add Semaphore: num -> 7
[2019-06-30 22:56:29.6706213 +0900 JST m=+5.504458601] Add Semaphore: num -> 8
[2019-06-30 22:56:30.1715697 +0900 JST m=+6.005407001] Add Semaphore: num -> 9
```

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
```

(次のページに続く)

(前のページからの続き)

```
    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++ {
            // 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() {
```

(次のページに続く)

(前のページからの続き)

```
        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

ABS(AtCoder Beginners Selection)

簡単な文法のおさらいとして、プログラミングコンテストサイトのオンラインジャッジを用いて理解度を確認してみましょう。ここでは ABS(AtCoder Beginners Selection) を用いて確認してみます。

8.1 1. ABC086A - Product

【問題概要】

二つの正整数 a, b が与えられます。 a と b の積が偶数か奇数か判定してください。

【制約】

- $1 \leq a, b \leq 10000$
- a, b は整数

```
func main() {
    var a, b int
    fmt.Scan(&a, &b)

    if a*b%2 == 0 {
        fmt.Println("Even")
    } else {
        fmt.Println("Odd")
    }
}
```

8.2 2. ABC081A - Placing Marbles

【問題概要】

0 と 1 のみから成る 3 桁の番号 s が与えられます。1 が何個含まれるかを求めてください。

```
func main() {
    s := ""
    fmt.Scan(&s)
    ans := 0
    for _, v := range s {
        if v == '1' {
            ans++
        }
    }
    fmt.Println(ans)
}
```

8.3 3. ABC081B - Shift only

【問題概要】黒板に N 個の正の整数 $A_1, \ldots, A_N$ が書かれています。すぬけ君は，黒板に書かれている整数がすべて偶数であるとき，次の操作を行うことができます。

- 黒板に書かれている整数すべてを，2 で割ったものに置き換える。

すぬけ君は最大で何回操作を行うことができるかを求めてください。

【制約】

- $1 \leq N \leq 200$

```
func main() {
    n := 0
    fmt.Scan(&n)
    a := make([]int, n)
    for i := 0; i < n; i++ {
        fmt.Scan(&a[i])
    }

    ans := 1 << 30
    for _, v := range a {
        cnt := 0
        for v%2 == 0 {
            cnt++
            v /= 2
        }
        ans = min(ans, cnt)
    }
    fmt.Println(ans)
}

func min(x, y int) int {
```

(次のページに続く)

(前のページからの続き)

```
    if x < y {
        return x
    } else {
        return y
    }
}
```

8.4 4. ABC087B - Coins

【問題概要】500 円玉を A 枚、100 円玉を B 枚、50 円玉を C 枚持っています。これらの硬貨の中から何枚かを選び、合計金額をちょうど X 円にする方法は何通りあるでしょうか？

【制約】

- $0 \leq A, B, C \leq 50$
- $A + B + C \geq 1$
- $50 \leq X \leq 20000$
- A, B, C は整数である
- X は 50 の倍数である

```
func main() {
    var a, b, c, x int
    fmt.Scan(&a, &b, &c, &x)
    ans := 0
    for i := 0; i <= a; i++ {
        for j := 0; j <= b; j++ {
            for k := 0; k <= c; k++ {
                if 500*i+100*j+50*k == x {
                    ans++
                }
            }
        }
    }
    fmt.Println(ans)
}
```

8.5 5. ABC083B - Some Sums

【問題概要】*1* 以上 N 以下の整数のうち、*10* 進法で各桁の和が A 以上 B 以下であるものについて、総和を求めてください。

【制約】

- $1 \leq N \leq 10^4$
- $1 \leq A \leq B \leq 36$
- 入力はすべて整数

```
func main() {
    var n, a, b int
    fmt.Scan(&n, &a, &b)
    ans := 0
    for i := 1; i <= n; i++ {
        sum := dsum(i)
        if a <= sum && sum <= b {
            ans += i
        }
    }
    fmt.Println(ans)
}

func dsum(x int) int {
    ret := 0
    for x > 0 {
        ret += x % 10
        x /= 10
    }
    return ret
}
```

8.6 6. ABC088B - Card Game for Two

【問題概要】N 枚のカードがあり、i 枚目のカードには a_i という数が書かれています。Alice と Bob はこれらのカードを使ってゲームを行います。ゲームでは 2 人が交互に 1 枚ずつカードを取っていきます。Alice が先にカードを取ります。2 人がすべてのカードを取ったときゲームは終了し、取ったカードの数の合計がその人の得点になります。2 人とも自分の得点を最大化するように最適戦略をとったとき、Alice は Bob より何点多くの得点を獲得できるかを求めてください。

【制約】

- N は 1 以上 100 以下の整数

```
func main() {
    var n int
    fmt.Scan(&n)
    a := make([]int, n)
```

(次のページに続く)

(前のページからの続き)

```
    for i := 0; i < n; i++ {
        fmt.Scan(&a[i])
    }
    sort.Sort(sort.Reverse(sort.IntSlice(a)))
    alice, bob := 0, 0
    for i := 0; i < n; i++ {
        if i%2 == 0 {
            alice += a[i]
        } else {
            bob += a[i]
        }
    }
    fmt.Println(alice - bob)
}
```

8.7 7. ABC085B - Kagami Mochi

【問題概要】N 個の整数 $d[0], d[1], \ldots, d[N-1]$ が与えられます。この中に何種類の異なる値があるでしょうか？

【制約】

- $1 \leq N \leq 100$
- $1 \leq d[i] \leq 100$
- 入力値はすべて整数

```
func main() {
    var n int
    fmt.Scan(&n)
    d := make([]int, n)
    m := make(map[int]bool)
    for i := 0; i < n; i++ {
        fmt.Scan(&d[i])
        m[d[i]] = true
    }
    fmt.Println(len(m))
}
```

8.8 8. ABC085C - Otoshidama

【問題概要】10000 円札と、5000 円札と、1000 円札が合計で N 枚あって、合計金額が Y 円であったという。このような条件を満たす各金額の札の枚数の組を 1 つ求めなさい。そのような状況が存在し得ない場合には -1 -1 -1 と出力しなさい。

【制約】

- $1 \leq N \leq 2000$
- $1000 \leq Y \leq 2 * 10^7$
- N は整数
- Y は 1000 の倍数

```
func main() {
    var n, y int
    fmt.Scan(&n, &y)

    for i := 0; i <= 2000; i++ {
        for j := 0; j <= 2000; j++ {
            k := n - i - j
            if k >= 0 {
                if 10000*i+5000*j+1000*k == y {
                    fmt.Printf("%d %d %d\n", i, j, k)
                    return
                }
            }
        }
    }
    fmt.Println("-1 -1 -1")
}
```

8.9 9. ABC049C - 白昼夢 / Daydream

【問題概要】英小文字からなる文字列 S が与えられます。T が空文字列である状態から始めて、以下の操作を好きな回数繰り返すことで $S = T$ とすることができるか判定してください。

- T の末尾に "dream", "dreamer", "erase", "eraser" のいずれかを追加する。

【制約】

- $1 \leq |S| \leq 10^5$
- S は英小文字からなる

```
func main() {
    s := ""
    fmt.Scan(&s)
    ts := []string{"dream", "dreamer", "erase", "eraser"}
    i := len(s)
    for i > 0 {
        c := false
```

(次のページに続く)

(前のページからの続き)

```
        for _, t := range ts {
            if i-len(t) >= 0 && s[i-len(t):i] == t {
                i -= len(t)
                c = true
                break
            }
        }
        if i > 0 && !c {
            fmt.Println("NO")
            return
        }
    }
    fmt.Println("YES")
}
```

8.10 10. ABC086C - Traveling

【問題概要】

シカの AtCoDeer くんは二次元平面上で旅行をしようとしています。AtCoDeer くんの旅行プランでは、時刻 0 に点 $(0,0)$ を出発し、1 以上 N 以下の各 i に対し、時刻 t_i に 点 (x_i,y_i) を訪れる予定です。

AtCoDeer くんが時刻 t に 点 (x,y) にいる時、 時刻 $t+1$ には 点 $(x+1,y),(x-1,y),(x,y+1),(x,y-1)$ のうちいずれかに存在することができます。その場にとどまることは出来ないことに注意してください。AtCoDeer くんの旅行プランが実行可能かどうか判定してください。

【制約】

- $1 \leq N \leq 10^5$
- $0 \leq x_i, y_i \leq 10^5$
- $1 \leq t_i \leq t_{i+1} \leq 10^5$
- 入力はすべて整数

```
func main() {
    var n int
    fmt.Scan(&n)
    t, x, y := make([]int, n+1), make([]int, n+1), make([]int, n+1)
    for i := 1; i < n+1; i++ {
        fmt.Scan(&t[i], &x[i], &y[i])
    }

    for i := 1; i < n+1; i++ {
        dist := abs(x[i]-x[i-1]) + abs(y[i]-y[i-1])
```

(次のページに続く)

(前のページからの続き)

```
        time := t[i] - t[i-1]
        if dist > time || (time-dist)%2 == 1 {
            fmt.Println("No")
            return
        }
    }

    fmt.Println("Yes")
}

func abs(x int) int {
    if x < 0 {
        return -x
    } else {
        return x
    }
}
```

第 9 章

Java と Go の違い

課題: Java と Go の違いを書く

第 10 章

標準パッケージ構成

課題: 標準構成を記載

第 11 章

テスト

公式ドキュメントはこちら。go test コマンドによるテストがされることを想定して作成されている。

https://golang.org/pkg/testing/

11.1 規則

- テストしたい関数は大文字から始まる関数とする
- テスト関数内では Error を用いる
- テスト関数が含まれるファイル名の suffix は _test.go とする

テスト関数の例は以下

```
func TestAbs(t *testing.T) {
    got := Abs(-1)
    if got != 1 {
        t.Errorf("Abs(-1) = %d; want 1", got)
    }
}
```

第 12 章

json

12.1 JSON を読み込む

```
import (
    "encoding/json"
    "fmt"
    "io/ioutil"
    "log"
    "net/http"
)

type UserResult struct {
    Status string `json:"status"`
    Result []struct {
        ContestID               int    `json:"contestId"`
        ContestName             string `json:"contestName"`
        Handle                  string `json:"handle"`
        Rank                    int    `json:"rank"`
        RatingUpdateTimeSeconds int    `json:"ratingUpdateTimeSeconds"`
        OldRating               int    `json:"oldRating"`
        NewRating               int    `json:"newRating"`
    } `json:"result"`
}

func main() {

    response, err := http.Get("http://codeforces.com/api/user.rating?handle=tutuz")

    if err != nil {
        log.Fatal(err)
    }

    body, err := ioutil.ReadAll(response.Body)
```

(次のページに続く)

(前のページからの続き)

```
    if err != nil {
        log.Fatal(err)
    }

    var result UserResult
    if json.Unmarshal(body, &result); err != nil {
        log.Fatal(err)
    }

    for _, v := range result.Result {
        fmt.Println(v.ContestName)
    }

}
// Educational Codeforces Round 50 (Rated for Div. 2)
// Codeforces Round #510 (Div. 2)
// Codeforces Round #512 (Div. 2, based on Technocup 2019 Elimination Round 1)
// Codeforces Round #516 (Div. 2, by Moscow Team Olympiad)
// Codeforces Round #517 (Div. 2, based on Technocup 2019 Elimination Round 2)
// Educational Codeforces Round 54 (Rated for Div. 2)
// Codeforces Round #521 (Div. 3)
// Codeforces Round #524 (Div. 2)
// Codeforces Round #541 (Div. 2)
```

12.2 JSON を書き込む

```
import (
    "bytes"
    "encoding/json"
    "fmt"
)

type message struct {
    TaskId     string            `json:"taskid"`
    Parameters map[string]string `json:"parameters"`
}

func main() {
    m := "getFromToYmd"
    d := map[string]string{
        "executeHostname": "localhost",
        "fromYM":          "201801",
        "toYM":            "201905",
    }
    res := message{
```

(次のページに続く)

(前のページからの続き)

```
        TaskId:     m,
        Parameters: d,
    }

    jsonBytes, err := json.Marshal(res)
    if err != nil {
        fmt.Println("JSON Marshal error:", err)
        return
    }

    out := new(bytes.Buffer)
    json.Indent(out, jsonBytes, "", "    ")
    fmt.Println(out.String())

}
// {
//     "taskid": "getFromToYmd",
//     "parameters": {
//         "executeHostname": "localhost",
//         "fromYM": "201801",
//         "toYM": "201905"
//     }
// }
```

12.3 参考

- https://golang.org/pkg/encoding/json/
- https://qiita.com/msh5/items/dc524e38073ed8e3831b

第 13 章

database/sql

公式ドキュメント https://golang.org/pkg/database/sql/#DB[*1] を参考に DB を扱う準備をします。今回は Driver として Postgres (pure Go)[*2] を用いることにします。

```
$ go get "github.com/lib/pq"
```

init 関数を呼び出して import します。

```
_ "github.com/lib/pq"
```

13.1 レコードの SELECT

まずは DB(PostgreSQL) のテーブルを select してみます。

```
import (
    "database/sql"
    "fmt"
    "log"

    _ "github.com/lib/pq"
)

func main() {
    connStr := "postgres://dev:dev@localhost/dev?sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
        log.Fatal(err)
    }
```

(次のページに続く)

[*1] https://golang.org/pkg/database/sql/#DB
[*2] https://godoc.org/github.com/lib/pq

(前のページからの続き)

```
    relname := "pg_user"
    rows, err := db.Query("SELECT relname, relnamespace FROM pg_class WHERE relname =
↪$1", relname)

    for rows.Next() {
        var rel string
        var relnamespace int
        if err := rows.Scan(&rel, &relnamespace); err != nil {
            log.Fatal(err)
        }
        fmt.Println(rel, relnamespace)
    }
    defer rows.Close()
}
// pg_user 11
```

想定通りレコードが取得できていることがわかりました。

13.2 レコードの INSERT

```
import (
    "database/sql"
    "log"

    _ "github.com/lib/pq"
)

func main() {
    connStr := "postgres://dev:dev@localhost/dev?sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
        log.Fatal(err)
    }

    statement := "INSERT INTO item (invoiceid, item, productid, quantity, cost) VALUES␣
↪($1, $2, $3, $4, $5)"
    stmt, err := db.Prepare(statement)
    if err != nil {
        log.Fatal(err)
    }

    defer stmt.Close()
    if _, err := stmt.Exec(49, 17, 24, 18, 10.8); err != nil {
        log.Fatal(err)
    }
}
```

13.3 レコードの UPDATE

レコードの UPDATE は INSERT とほぼ同じです。

```
import (
    "database/sql"
    "log"

    _ "github.com/lib/pq"
)

func main() {
    connStr := "postgres://dev:dev@localhost/dev?sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
        log.Fatal(err)
    }

    statement := "UPDATE item SET cost = $1 WHERE invoiceid = $2 AND ITEM = $3"
    stmt, err := db.Prepare(statement)
    if err != nil {
        log.Fatal(err)
    }

    defer stmt.Close()
    if _, err := stmt.Exec(0.1, 49, 17); err != nil {
        log.Fatal(err)
    }
}
```

13.4 レコードの DELETE

DELETE も INSERT(, UPDATE) とほぼ同様です。

```
import (
    "database/sql"
    "log"

    _ "github.com/lib/pq"
)

func main() {
    connStr := "postgres://dev:dev@localhost/dev?sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
        log.Fatal(err)
```

(次のページに続く)

(前のページからの続き)

```
    }

    statement := "DELETE FROM item WHERE invoiceid = $1 AND ITEM = $2"
    stmt, err := db.Prepare(statement)
    if err != nil {
        log.Fatal(err)
    }

    defer stmt.Close()
    if _, err := stmt.Exec(49, 17); err != nil {
        log.Fatal(err)
    }
}
```

13.5 トランザクション管理

上の例では INSERT/UPDATE/DELETE を処理する際に、 *sql.Stmt を用いていました。今回はトランザクション管理をするために

```
func (db *DB) Begin() (*Tx, error)
```

を用いることにします。実装例としては[*3] が参考になります。

```
import (
    "database/sql"
    "log"

    _ "github.com/lib/pq"
)

func main() {
    connStr := "postgres://dev:dev@localhost/dev?sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
        log.Fatal(err)
    }

    tx, err := db.Begin()
    if err != nil {
        log.Fatal(err)
    }
    defer tx.Rollback()
```

(次のページに続く)

[*3] https://stackoverflow.com/questions/16184238/database-sql-tx-detecting-commit-or-rollback

(前のページからの続き)

```
    statement := "INSERT INTO item (invoiceid, item, productid, quantity, cost) VALUES␣
↪(49, 17, 24, 18, 10.8)"

    defer func() {
        if err != nil {
            tx.Rollback()
            return
        }
        err = tx.Commit()
    }()
    if _, err := tx.Exec(statement); err != nil {
        log.Fatal(err)
    }
}
```

13.6 参考

13.7 あとで読みたい

- http://go-database-sql.org/index.html

第 14 章

net/http

Go で HTTP サーバ/HTTP クライアントを試してみます。net/http パッケージにまとまっています。

https://golang.org/pkg/net/http/

14.1 HTTP サーバ

とてもシンプルに HTTP サーバになることができます。

リスト 1　Hello world を出力するサーバ

```
func main() {

    http.HandleFunc("/hello", func(writer http.ResponseWriter, request *http.Request) {
        fmt.Fprintln(writer, "Hello world!")
    })
    http.ListenAndServe(":8021", nil)

}
```

curl で HTTP サーバにリクエストしてみます。

```
D:\>curl -v localhost:8021/hello
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8021 (#0)
> GET /hello HTTP/1.1
> Host: localhost:8021
> User-Agent: curl/7.55.1
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Sun, 30 Jun 2019 11:16:20 GMT
```

(次のページに続く)

(前のページからの続き)

```
< Content-Length: 13
< Content-Type: text/plain; charset=utf-8
<
Hello world!
* Connection #0 to host localhost left intact
```

Hello world! のレスポンスが返ってくることがわかりました。HTTP/1.1 で通信していることがわかります。HTTP リクエストの中をのぞいてみます。

ブラウザから http://localhost:8021/hello2?foo=hoge&boo=fuga にリクエストしてみます。

```
func main() {

    http.HandleFunc("/hello2", func(writer http.ResponseWriter, request *http.Request)
↪{

        method := request.Method
        fmt.Printf("[Method] %v\n", method)
        for k, v := range request.Header {
            fmt.Printf("[Header] %v: %s\n", k, strings.Join(v, ","))
        }

        // GET
        if method == "GET" {
            request.ParseForm()
            for k, v := range request.Form {
                fmt.Print("[Param] " + k)
                fmt.Println(": " + strings.Join(v, ","))
            }
            fmt.Fprintln(writer, "Hello world! Get request recieved.")
        }
    })
    http.ListenAndServe(":8021", nil)

}
// [Method] GET
// [Header] Connection: keep-alive
// [Header] Cache-Control: max-age=0
// [Header] Upgrade-Insecure-Requests: 1
// [Header] User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36␣
↪(KHTML, like Gecko) Chrome/75.0.3770.100 Safari/537.36
// [Header] Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,
↪image/apng,*/*;q=0.8,application/signed-exchange;v=b3
// [Header] Accept-Encoding: gzip, deflate, br
// [Header] Accept-Language: ja,en-US;q=0.9,en;q=0.8
// [Param] foo: hoge
// [Param] boo: fuga
```

次に POST リクエストを扱ってみます。メソッドが POST は Body を解析してサーバ側に出力させることにします。

```
func main() {

    http.HandleFunc("/hello3", func(writer http.ResponseWriter, request *http.Request)
↪{

        method := request.Method
        fmt.Printf("[Method] %v\n", method)
        for k, v := range request.Header {
            fmt.Printf("[Header] %v: %s\n", k, strings.Join(v, ","))
        }

        if method == "POST" {
            defer request.Body.Close()
            body, err := ioutil.ReadAll(request.Body)
            if err != nil {
                log.Fatal(err)
            }

            fmt.Println("[request body row] " + string(body))
            decoded, error := url.QueryUnescape(string(body))
            if error != nil {
                log.Fatal(error)
            }
            fmt.Println("[request body decoded] ", decoded)
            fmt.Fprint(writer, "Hello world. Recieved Post(form) request!!")
        }
    })
    http.ListenAndServe(":8021", nil)

}
```

curl は以下のようにします。

```
curl -v -XPOST -d '%E3%81%93%E3%82%93%E3%81%AB%E3%81%A1%E3%81%AF+%E4%B8%96%E7%95%8C'␣
↪localhost:8021/hello3
```

結果 (サーバ側)

```
[Method] POST
[Header] Accept: */*
[Header] Content-Length: 66
[Header] Content-Type: application/x-www-form-urlencoded
[Header] User-Agent: curl/7.55.1
[request body row] '%E3%81%93%E3%82%93%E3%81%AB%E3%81%A1%E3%81%AF+%E4%B8%96%E7%95%8C'
[request body decoded]  'こんにちは 世界'
```

結果 (クライアント側)

```
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8021 (#0)
> POST /hello3 HTTP/1.1
> Host: localhost:8021
> User-Agent: curl/7.55.1
> Accept: */*
> Content-Length: 66
> Content-Type: application/x-www-form-urlencoded
>
* upload completely sent off: 66 out of 66 bytes
< HTTP/1.1 200 OK
< Date: Sun, 30 Jun 2019 11:47:21 GMT
< Content-Length: 42
< Content-Type: text/plain; charset=utf-8
<
Hello world. Recieved Post(form) request!!* Connection #0 to host localhost left intact
```

想定どおりリクエスト Body が解析されて、表示されていることがわかります。

14.2 HTTP クライアント

続いて HTTP クライアントを実装してみます。サーバは上記で作成したサーバを別プロセスで立ち上げておきます。実装方法はいくつかあるようです。 Go net/http パッケージの概要と HTTP クライアント実装例 を参考にしました。

14.2.1 GET メソッド

一番単純な http.Get(url) による GET

```
values := url.Values{}
values.Add("name", "d-tsuji")

res, err := http.Get("http://127.0.0.1:8021/hello2?" + values.Encode())
if err != nil {
        fmt.Println(err)
        return
}

defer res.Body.Close()

body, err := ioutil.ReadAll(res.Body)
if err != nil {
```

(次のページに続く)

(前のページからの続き)

```
        log.Fatal(err)
}
fmt.Println(string(body))
```

結果 (サーバ側)

```
> goSample.exe
[Method] GET
[Header] User-Agent: Go-http-client/1.1
[Header] Accept-Encoding: gzip
[Param] age: 999
[Param] name: d-tsuji
```

結果 (クライアント側)

```
Hello world! Get request recieved.
```

想定通りサーバと通信できていることがわかりました。

Client 型

Client.Do(Request) メソッド

14.2.2 POST メソッド

14.3 その他参考

- http://pppurple.hatenablog.com/entry/2018/04/26/225932

第 15 章

flag

コマンドラインのフラグを解析するパッケージです。

https://golang.org/pkg/flag/

簡単なサンプルを試してみます。

```
func main() {

    var (
        concurrency int
        environment string
        debug       bool
    )

    flag.StringVar(&environment, "e", "local", "environment")
    flag.BoolVar(&debug, "x", false, "debug mode")
    flag.IntVar(&concurrency, "n", 32, "concurrent count")
    flag.Parse()

    fmt.Printf("environment : %v\n", environment)
    fmt.Printf("debug : %v\n", debug)
    fmt.Printf("concurrency : %v\n", concurrency)
}
```

環境別になんやかんやしたい...!という場合に実行時の引数にするのはよくあるパターンかと思います。上記の場合、何もしないと local 環境で並行数 3 で実行するプログラムを起動するイメージです。

```
> main.exe
environment : local
debug : false
concurrency : 3
```

オプションを指定しないと、デフォルト値が設定されていることがわかります。

次にオプションを指定してみます。

```
> main.exe -e stg -n 20 -x true
environment : stg
debug : true
concurrency : 20
```

このように指定したオプションの値が設定されていることがわかります。

ヘルプコマンドはデフォルトで実装されていて、 -h を指定すると各オプションの内容が表示されます。

```
> main.exe -h
Usage of main.exe:
-e string
        environment (default "local")
-n int
        concurrent count (default 3)
-x    debug mode
```

間違ったオプションの値が設定された場合 (例えば int 型が指定されるオプションに文字列型の値を指定した場合など) はエラーになります。

```
> main.exe -n miss
invalid value "miss" for flag -n: parse error
Usage of main.exe:
-e string
        environment (default "local")
-n int
        concurrent count (default 3)
-x    debug mode
```

第 16 章

yaml

16.1 yaml ファイルを読み込む

簡単に以下の形式の山手線の駅名が記載された yaml ファイルを読み込んでみます。

リスト 1　test.yaml

```
-
station_id: 1
station_name: 大崎
-
station_id: 2
station_name: 五反田
-
station_id: 3
station_name: 目黒
-
station_id: 4
station_name: 恵比寿
-
station_id: 5
station_name: 渋谷
```

gopkg.in/yaml.v2 のパッケージが必要になるので、 go get -v gopkg.in/yaml.v2 しておきます。

実装例は以下の通りです。

リスト 2　yaml2struct.go

```
func ReadYaml() (*[]station, error) {

    buf, err := ioutil.ReadFile("./test.yml")
    if err != nil {
        fmt.Println(err)
    }
```

(次のページに続く)

(前のページからの続き)

```
    data := make([]station, 1)
    err = yaml.Unmarshal(buf, &data)
    if err != nil {
        fmt.Println(err)
        return nil, err
    }

    return &data, nil
}
```

本質的には go get した yaml.go の以下の関数を呼び出すことになります。

```
func Unmarshal(in []byte, out interface{}) (err error) {
    return unmarshal(in, out, false)
}
```

関数の引数であるインターフェース型の out にアンマーシャリングされた構造体のデータが write されます。

リスト 3　yaml2struct_test.go

```
func TestReadYaml(t *testing.T) {

    data, err := ReadYaml()
    if err != nil {
        fmt.Println(err)
    }

    fmt.Println(*data)
}
```

結果

```
=== RUN   TestReadYaml
 [{1 大崎} {2 五反田} {3 目黒} {4 恵比寿} {5 渋谷}]
 --- PASS: TestReadYaml (0.00s)
 PASS
```

16.2 yaml を書き込む

続いて、構造体のデータから yaml ファイルを生成してみます。

```
func WriteYaml() error {

    stations := make([]station, 0)
```

(次のページに続く)

(前のページからの続き)

```
    stations = append(stations, station{
        Id:   1,
        Name: "東京",
    })
    stations = append(stations, station{
        Id:   2,
        Name: "新橋",
    })
    stations = append(stations, station{
        Id:   3,
        Name: "品川",
    })
    stations = append(stations, station{
        Id:   4,
        Name: "横浜",
    })

    dat, err := yaml.Marshal(&stations)
    if err != nil {
        fmt.Println(err)
        return err
    }
    fmt.Println(string(dat))

    return nil
}
```

リスト 4　yaml2struct_test.go

```
func TestWriteYaml(t *testing.T) {

    if err := WriteYaml(); err != nil {
        fmt.Println(err)
    }
}
```

結果

yaml 形式で write されていることがわかります。

```
=== RUN   TestWriteYaml
- station_id: 1
station_name: 東京
- station_id: 2
station_name: 新橋
- station_id: 3
station_name: 品川
- station_id: 4
```

(次のページに続く)

(前のページからの続き)

```
station_name: 横浜

--- PASS: TestWriteYaml (0.00s)
PASS
```

16.3 参考

- https://godoc.org/gopkg.in/yaml.v2

第 17 章

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: セットアップの手順を記載する

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

課題: メソッドの可視性について記載する

(元のエントリ は、 D:\book\go-introduction\source\MethodAndInterface\index.rst の 72 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:\book\go-introduction\source\Parallel\index.rst の 312 行目です)

課題: Java と Go の違いを書く

(元のエントリ は、 D:\book\go-introduction\source\differenceJavaAndGo.rst の 4 行目です)

課題: 標準構成を記載

(元のエントリ は、 D:\book\go-introduction\source\package.rst の 4 行目です)