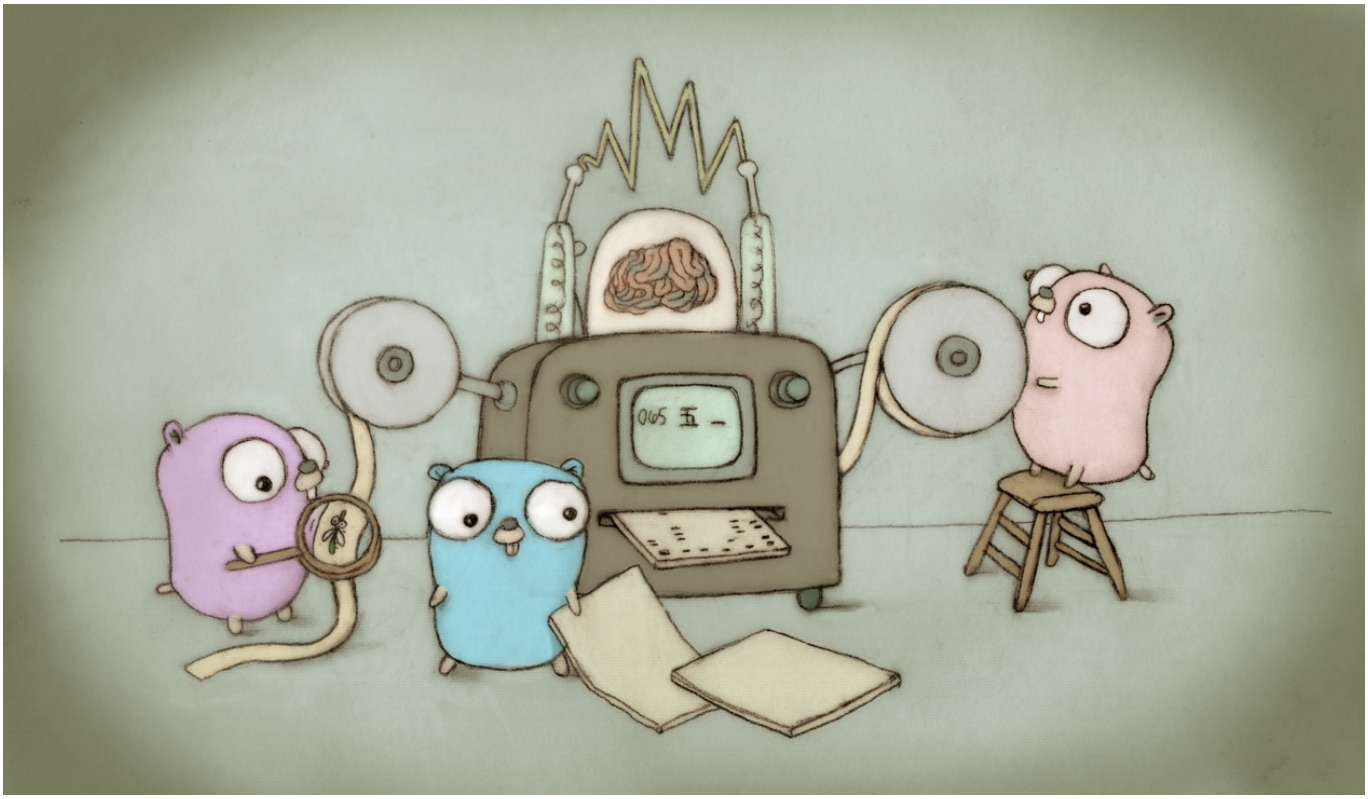
Go-lang Tutorial

英語 - The Go Programming Language

日本語 - Go言語入門



インストール

```
brew install go
```

コンパイル

```
go build hello.go
go ./hello
```

実行

```
go run hello.go
```

変数

定義

- 「`var 変数名 型名`」の形式で変数を定義
- 初期値を指定しない場合は0や空文字列などで初期化される
- 初期値により型名が明白な場合は型名を省略できる
- `var`の代わりに`:=`を使うことができる
- 1 文字目が 小文字 の場合は、そのパッケージだけで見える変数
- 1 文字目が 大文字 の場合は、他のパッケージからも見える変数

```
1. var a1 int = 123
2. a1 := 123
3. var (
        a1 int = 123
        a2 int = 456
    )
```

代入

```
a1 = 456
name, age = "Yamada", 26
```

定数

- 定数の場合型名は省略可能
- 定数で iota を使えば、簡単に連番の定数を作成できる

```
1. const foo = 100
2. const (
    foo = 100
    bar = 200
)
3. const (
    sun = iota // 0
    mon // 1
    tue // 2
    )
```

型

型名	
bool	真偽値(true or false)

型名	
int8/int16/int32/int64	nビット整数
uint8/uint16/uint32/uint64	nビット非負整数
float32/float64	nビット浮動小数点数
complex64/complex128	nビット虚数
byte	1バイトデータ(uint8と同義)
rune	1文字(int32と同義)
uint	uint32 または uint64
int	int32 または int64
uintptr	ポインタを表現するのに十分な非負整数
string	文字列

- 下記の様に型に別名をつけることができる

```
1. type UtcTime string
   type JstTime string

2. type (
        UtcTime string
        JstTime string
    )

var t1 UtcTime = "00:00:00"
var t2 JstTime = "09:00:00"

t1 = t2
// Error: 型の異なる値を代入することはできません。
```

型変換

- 型名()で型変換を行う

```
var a1 uint16 = 1234
var a2 uint32 = uint32(a1)
```

演算子

演算子	
x + y	加算 (文字列の連結にも利用)

演算子

x - y	減算
x * y	乗算
x / y	除算
x % y	除算の余り
x & y	論理積(AND)
x | y	論理和(OR)
x ^ y	排他的論理和(XOR)
x &^ y	x AND (NOT y)
x << y	yビット左にシフト
x >> y	yビット右にシフト
x = y	xにyを代入
x := y	xにyを代入(初期化の使用可能)
x++	x = x + 1 と同義
x--	x = x - 1 と同義
x += y	x = x + y と同義
x -= y	x = x - y と同義
x *= y	x = x * y と同義
x /= y	x = x / y と同義
x %= y	x = x % y と同義
x &= y	x = x & y と同義
x |= y	x = x
x ^= y	x = x ^ y と同義
x &^= y	x = x &^ y と同義
x <<= y	x = x << y と同義
x >>= y	x = x >> y と同義
x && y	xかつy(AND)
x || y	xまたはy(OR)
!x	xがtrueの場合false/falseの場合true(NOT)
x == y	xとyが等しければ
x != y	xとyが等しくなければ

演算子	
x < y	yがxより大きければ
x < y	yがx以上であれば
x > y	yがxより小さければ
x >= y	yがx以下であれば
ch <- x	chチャネルにxを送信

リテラル・値

リテラル	
nil	無しを示す特別な値
true	真偽値の真
false	真偽値の偽
1234	整数
1_234	整数(カンマ区切りの代わりに_を使用。_は無視される)
0777	8進数
0o755	8進数(0Oも可)
0x89ab	16進数(0Xも可)
0b1111	2進数(0Bも可)
123.4	小数
1.23e4	浮動小数点数(1.23E4も可)
1.23i	虚数
"ABC"	文字列
'A'	文字(rune)

エスケープシーケンス

\a	ベル(U+0007)
\b	バックスペース(U+0008)
\t	タブ(U+0009)
\n	改行(U+000A)
\v	垂直タブ(U+000B)
\f	フォームフィード(U+000C)

\r	キャリッジリターン(U+000D)
"	ダブルクォート(U+0022)
'	シングルクォート(U+0027)
\\	バックスラッシュ(U+005C)
\x42	ASCII文字(U+0000～U+00FF)
\u30A2	Unicode(U+0000～U+FFFF)
\U0001F604	Unicode(U+0000～U+10FFFF)

コンテナ型

配列

- コンパイル時に個数が決まっていて変更不可のものを 配列 と言います。型名の前に [個数] をつけて宣言します。配列のインデックスは 0 からはじまります。途中で個数を変更することはできませんが、メモリ効率や性能の点で優れています。

```
1.  a1 := [3]string{}
    a1[0] = "Red"
    a1[1] = "Green"
    a1[2] = "Blue"
    fmt.Println(a1[0], a1[1], a1[2])

// 初期化時に値を設定することもできます。
2.  a1 := [3]string{"Red", "Green", "Blue"}

// 初期化によって個数が決まる場合は、個数を ... と省略することができます。
3.  a1 := [...]string{"Red", "Green", "Blue"}
```

スライス

- メモリ効率や速度は若干落ちますが、個数を変更可能なものを スライス と呼びます。
- 型名の前に [] をつけて宣言します。
- スライスには append() を用いて要素を追加します。

```
a1 := []string{}            // スライス。個数不定
a1 = append(a1, "Red")
a1 = append(a1, "Green")
a1 = append(a1, "Blue")
fmt.Println(a1[0], a1[1], a1[2])
```

- len() は配列やスライスの長さ(length)、cap() は容量(capacity)を求めます。
- 長さは実際に使用されている数、容量はメモリ上に確保されている数です。
- 容量を超えると、倍の容量のメモリが別に確保され、既存データがコピーされます。

```
a := []int{}
for i := 0; i < 10; i++ {
    a = append(a, i)
    fmt.Println(len(a), cap(a))
}
```

- スライスの場合、make(スライス型, 初期個数, 初期容量) を用いたメモリの確保ができます。
- 初期容量を省略した場合は初期個数と同じ容量が確保されます。
- 容量をあらかじめ確保しておくことで、容量超過時の再確保を減らして速度を速めることができます。

```
bufa := make([]byte, 0, 1024)
```

マップ

- 連想配列型コンテナ
- 定義：「map[キーの型]値の型」
- 参照：「map[キー]」
- 追加：「map[キー] = 値」
- 削除：「delte(map, キー)」
- 長さ：「len(map)」
- 存在：「_, ok := map[キー]」
- イテラブル

```
1.  a1 := map[string]int{
        "x": 100,
        "y": 200,           // 改行する場合はカンマ必須
    }

2.  fmt.Println(a1["x"])

3.  a1["z"] = 300

4.  delete(a1, "z")

5.  fmt.Println(len(a1))

6.  _, ok := a1["z"]
    if ok {
        fmt.Println("Exist")
    } else {
        fmt.Println("Not exist")
    }

7. for key, value := range a1 {
        fmt.Printf("%s=%d\n", key, value)
    }
```

文法

条件式

if文

- 「`if 条件 { 処理 }`」 は条件が真の時のみ処理を実行します。
- 条件を `(...)` で囲む必要はありません。
- `{ ... }` の中括弧は必須です。

```
if x > max {
    max = x
}
```

Switch文

- 「`switch 式 { ... }`」は、式の値によって処理を振り分けます。
- 「`switch { ... }`」では、case 分で条件を記述することもできます。
- 次の条件の処理も実行するには `fallthrough` を用います。

```
1.  switch mode {
        case "running":
              return "実行中"
        case "stop":
            return "停止中"
        default:
            return "不明"
    }
2.  switch {
        case x > y:
            return "Big"
        case x < y:
            return "Small"
        default:
            return "Equal"
    }
// dayOfWeek が "Sat" または "Sun" であれば "Horiday" を返します。
3.  switch dayOfWeek {
        case "sat":
            fallthrough
        case "sun":
            return "Horiday"
        default:
            return "Weekday"
    }
```

For文

- Go言語には `while`文が無く、繰り返し処理はすべて`for`を用います。
- 「`for 開始処理; 条件; 後処理 { 処理 }`」は、最初に開始処理を行い、条件が真の間、処理と後処理を繰り返し実行します。
- 条件を省略すると無限ループになります。`continue` は次のループを繰り返します。
- `break` はループを抜けます。
- 配列やスライスなどイテラブルなものに対しては `range` を用いてループ処理することができます。

```
1.  for x < y {
        x++
    }

2.  for i := 0; i < 10; i++ {
        fmt.Println(i)
    }

3.  n := 0
    for {
        n++
        if n > 10 {
            break
        } else if n % 2 == 1 {
            continue
        } else {
            fmt.Println(n)
        }
    }

4.  colors := [...]string{"Red", "Green", "Blue"}
    for i, color := range colors {
        fmt.Printf("%d: %s\n", i, color)
    }
```

Goto文

- goto 文は指定したラベルにジャンプします。Go言語には try catch raise のような例外処理構文はサポートされていませんので、似たようなことをやるとすると下記の様になります。

```
func funcA() (int, string) {
    err := nil
    filename := ""
    data := ""

    err, filename = GetFileName()
    if err != nil {
        goto Done
    }
```

```
    err, data = ReadFile(filename)
    if err != nil {
        goto Done
    }

Done:
    return err, data
}
```

関数

- 定義：「func」
- 複数の値を返却することもできます。複数の場合は型名は (...) で囲む必要があります。
- 複数の値を返却する関数などで、不要な戻り値がある場合は、ブランク変数 _ を使用することができます。
- ... を用いることで可変引数を実現することができます。

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

    func main() {
        fmt.Println(add(5, 3))      // => 8
    }

2.  func addMinus(x int, y int) (int, int) {
        return x + y, x - y
}

3.  _, y := funcA()

4.  func funcA(a int, b ... int) {
        fmt.Printf("a=%d\n", a)
        for i, num := range b {
            fmt.Printf("b[%d]=%d\n", i, num)
        }
    }

    func main() {
        funcA(1, 2, 3, 4, 5)
    }
```

構造体(struct)

- Go言語では、クラス(class)の代わりに 構造体(struct) を使用します。
- 構造体にはメンバ変数のみを定義し、クラスメソッドに相当する関数は関数名の前に (thisに相当する変数 *構造体名) をつけて定義します。

```
type Person struct {
    name string
    age int
}

func (p *Person) SetPerson(name string, age int) {
    p.name = name
    p.age = age
}

func (p *Person) GetPerson() (string, int) {
    return p.name + "(" + p.age + ")"
}

func main() {
    var p1 Person
    p1.SetPerson("Yamada", 26)
    name, age = p1.GetPerson()
}
```

- 構造体のメンバの内、大文字で始まるものはパッケージ外からアクセス可能、小文字で始まるものはパッケージ外からアクセス不可となります。

```
type Person struct {
    Name string // 外部からアクセス可能
    Age int     // 外部からアクセス可能
    status int      // 外部からアクセス不可
}
```

- 構造体を使用する際は、下記の様にパラメータを初期化することができます。

```
a1 := Person{"Yamada", 26}          // 順序通りに初期化
a2 := Person{name: "Tanaka", age: 32}       // 名前で初期化
```

インタフェース(interface)

- インタフェース(interface) はポリモーフィズムを実装するための機能です。
- 下記の例では構造体 `Person` も、構造体 `Book` も `ToString()` というメソッドと `PrintOut()` というメソッドを実装しています。
- `ToString()` は中身が異なるので仕方ないですが、`PrintOut()` は中身も同じなので、インタフェースを用いてひとつの関数にまとめることができます。`Printable` というインタフェースは、`ToString()` というメソッドをサポートしている構造体であれば、自動的に適用することが可能となります。

```

```

```
type Person struct {
    name string
}
func (p Person) ToString() string {
    return p.name
}
// func (p Person) PrintOut() {
//     fmt.Println(p.ToString())
// }

type Book struct {
    title string
}
func (b Book) ToString() string {
    return b.title
}
// func (b Book) PrintOut() {
//     fmt.Println(b.ToString())
// }

// インターフェースによる実装
type Printable interface {
    ToString() string
}
func PrintOut(p Printable) {
    fmt.Println(p.ToString())
}

func main() {
    a1 := Person {name: "山田太郎"}
    a2 := Book {title: "吾輩は猫である"}
    a1.PrintOut()
    a2.PrintOut()
}
```

ポインタ(pointer)

- ポインタとは、変数が格納されているメモリのアドレスです。
- C言語と同様に、オブジェクトのポインタを参照するには & を、ポインタの中身を参照するには * を用います。
- 変数の値を渡すことを「値渡し」、変数のポインタを渡すことを「参照渡し」と呼びます。

```
var a1 int      // int型変数a1を定義
var p1 *int;        // intへのポインタ変数p1を定義

p1 = &a1        // p1にa1のポインタを設定
*p1 = 123       // ポインタの中身(つまりa1)に123を代入
fmt.Println(a1) // => 123
```

- 値渡しでは値のコピーしか渡していないので元の変数を変更することはできませんが、ポインタ渡しであれば関数の中で変数の値を変更することが可能となります。

```
func main() {
    var a1 int = 123
    var a2 int = 123
    fn(a1, &a2)     // a1は値渡し、a2は参照渡し
    fmt.Println(a1, a2) // => 123 456
}

func fn(b1 int, b2 *int) {
    b1 = 456
    *b2 = 456
}
```

- 演算子 . は、構造体のメンバ変数でも、ポインタが指し示す構造体のメンバ辺でもアクセスすることができます。

```
a1 := Person{"Tanaka", 26}  // 構造体Personのオブジェクトa1を確保して初期化
p1 := &a1           // 構造体a1へのポインタをp1に格納
fmt.Println(a1.name)        // メンバ変数には左記のようにアクセス
fmt.Println((*p1).name) // ポインタpの中身(後続体)のメンバ変数には左記のようにアクセス
fmt.Println(p1.name)        // ただし、Go言語ではこれを、左記のようにも記述できる
```

領域確保(new)

- new() を用いて領域を動的に確保し、その領域へのポインタを得ることができます。
- 確保した領域は参照されなくなった後にでガベージコレクションにより自動的に開放されます。

```
type Book struct {
    title string
}

func main() {
    bookList := []*Book{}

    for i := 0; i < 10; i++ {
        book := new(Book)
        book.title = fmt.Sprintf("Title#%d", i)
        bookList = append(bookLlist, book)
    }
    for _, book := range bookList {
        fmt.Println(book.title)
    }
}
```

遅延実行(defer)

- 「defer 処理」は、関数から戻る直前に処理を遅延実行します。
- リソースを忘れずに解放する際によく用いられます。

```
func funcA() {
    fp, err := os.Open("sample.txt")
    if err != nil {
        return
    }
    defer fp.Close()

    for {
        ...
    }
```

インポート(import)

- import はパッケージをインポートします。

```
1.  import "fmt"

2.  import (
        "os"
        "fmt"
    )
```

- `go get` コマンドを用いて、環境変数 `GOPATH` 配下に外部のパッケージをインストールすることができます。標準では、Goパッケージがインストールされた場所 (`/usr/local/go/src`)、`$HOME/go/src`、`$GOPATH/src` を検索対象とします。

```
export GOPATH=$HOME/go
go get github.com/google/go-cmp/cmp
```

- インポート時にパッケージの別名をつけることができます。これによりパッケージ名の重複の問題を回避することが可能です。

```
1.  import "github.com/google/go-cmp/cmp"
2.  import gcmp "github.com/google/go-cmp/cmp"
```

パッケージ(package)

```
$HOME
 └ go
   └ src
     ├ sample.go
     └ local
       └ mypkg
         └ mypkg.go
```

- `mypkg.go` ファイルを次の内容で作成します。`package` でパッケージ名を宣言します。

```
package mypkg

import "fmt"

func FuncA() {          // 大文字で始まるものは自動的にエクスポートされる
    fmt.Println("FuncA()")
}

func funcB() {          // 小文字で始まるものはエクスポートされない
    fmt.Println("funcB()")
}
```

- インポート
- `sample.go`ファイルを次の内容で作成します。大文字で始まる `FuncA()` は公開されているので使用できますが、小文字で始まる `funcB()` は非公開なので使用することができません。

```
package main

import "local/mypkg"

mypkg.FuncA()       // 呼び出せる
mypkt.funcB()       // Error
```

ゴルーチン(Goroutine)

- ゴルーチン(goroutine)はGo言語における並行処理を実現するもので、スレッドよりも高速に並行処理を実現することができます。下記の例では、メインの処理を実行しながら、並行して `funcA()` ゴルーチンを `go` により実行しています。

```
func funcA() {
    for i := 0; i < 10; i++ {
        fmt.Print("A")
        time.Sleep(10 * time.Millisecond)
    }
}
```

```
func main() {
    go funcA()
    for i := 0; i < 10; i++ {
        fmt.Print("M")
        time.Sleep(20 * time.Millisecond)
    }
}
```

- 下記の例ではチャネルを用いてゴルーチンの終了を待ち合わせる例です。
- `chan` はチャネルを生成します。`<-` はチャネルにメッセージを送受信します。

```
func funcA(chA chan <- string) {
    time.Sleep(3 * time.Second)
    chA <- "Finished"        // チャネルにメッセージを送信する
}

func main() {
    chA := make(chan string)     // チャネルを作成する
    defer close(chA)         // 使い終わったらクローズする
    go funcA(chA)        // チャネルをゴルーチンに渡す
    msg := <- chA        // チャネルからメッセージを受信する
    fmt.Println(msg)
}
```

- 次の例では `select` を用いて、ゴルーチン `funcA()` と `funcB()` 双方の待ち合わせを行います。

```
func funcA(chA chan <- string) {
    time.Sleep(1 * time.Second)
    chA <- "funcA Finished"
}

func funcB(chB chan <- string) {
    time.Sleep(2 * time.Second)
    chB <- "funcB Finished"
}

func main() {
    chA := make(chan string)
    chB := make(chan string)
    defer close(chA)
    defer close(chB)
    finflagA := false
    finflagB := false
    go funcA(chA)
    go funcB(chB)
    for {
        select {
        case msg := <- chA:
```

```
            finflagA = true
            fmt.Println(msg)
        case msg := <- chB:
            finflagB = true
            fmt.Println(msg)
        }
        if finflagA && finflagB {
            break
        }
    }
}
```