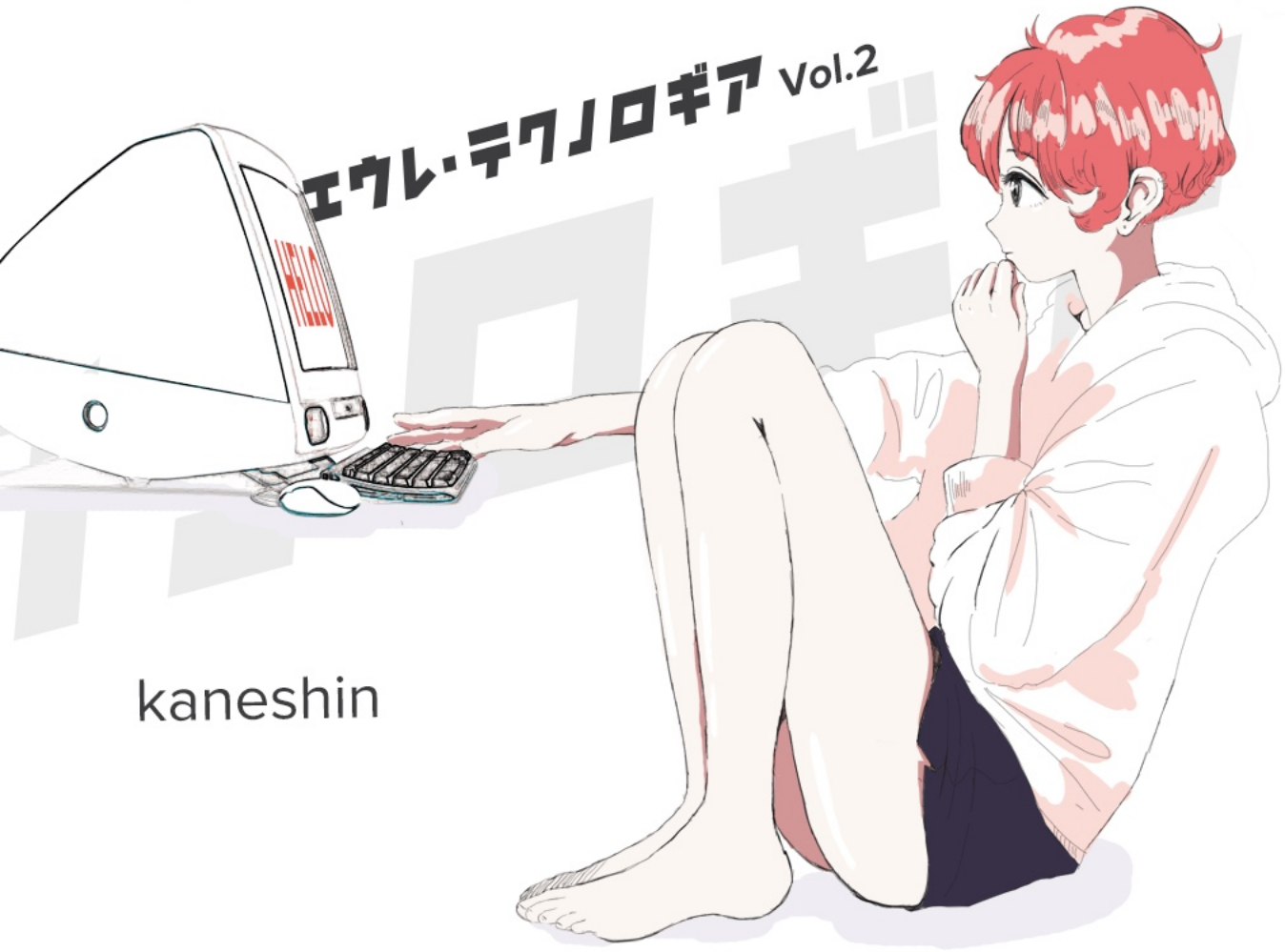
Go言語らしく Goコードを 実装するための 手法と思想

エウレ・テクノロギア Vol.2



kaneshin

Go 言語らしく Go コードを実装するための手法と思想

kaneshin　著

2018-10-08 版　　エウレ・テクノロギア 発行

はじめに

Go 言語らしく Go コードを実装するための手法と思想を手に取っていただき、ありがとうございます。

本書の内容について

本書は、実践的な Go 言語の開発方法や設計思想を解説した本です。対象読者は、Go 言語でツールやライブラリの開発経験はあるが、「Go 言語らしく実装するにはどのように記述したらいいんだろう」と悩んでいる初級者や中級者の方、Go 言語のノウハウが貯まっていないチームを対象としています。

日本では、Go 言語の本はあまり多く出版されておらず、また、「Go 言語らしさ」という言葉を考えすぎて悩んでいる人も多いと思います。「Go 言語らしさ」を意識している人は Go 言語の公式の情報や設計思想から知識を得ていますが、世の中に拡散されている記事では、それらと紐づされて Go のコードを解説しているものは少ないです。Go 言語で開発をしている方に「Go 言語らしさ」を意識して開発をしてもらうためには Go 言語の設計思想や背景を深く理解することが重要です。本書では、この「Go 言語らしさ」について深く吟味して Go のコードを解説をしています。

また、本書では Go 言語でツール開発をする過程についても解説し、そのツールの開発を通して設計思想もそうですし、Go 言語でのコードやテストの記述についても解説をします。

本書をひととおり読み終えたとき、Go 言語の知識と開発におけるテクニック、設計思想が理解できているはずです。

エウレ・テクノロギアとは

エウレ・テクノロギアは株式会社エウレカ[*1]の有志によって結成された技術サークルです。

お問い合わせ先

本書に関するお問い合わせ先：https://github.com/kaneshin/contact/issues

免責事項

本書に記載された内容は、情報の提供のみを目的としています。したがって、本書を用いた開発、制作、運用は必ずご自身の責任と判断によって行ってください。これらの情報による開発、制作、運用の結果について、著者はいかなる責任も負いません。

[*1] 株式会社エウレカ https://eure.jp/

目次

第 1 章

Go 言語の思想

Go は Google によって開発されたことは有名ですが、「なぜ Google は Go が必要だったのか」という問いに答えられる人はあまりいません。Go が開発された背景については Go at Google[*1] に記載されています。

> The Go programming language was conceived in late 2007 as an answer to some of the problems we were seeing developing software infrastructure at Google. The computing landscape today is almost unrelated to the environment in which the languages being used, mostly C++, Java, and Python, had been created. The problems introduced by multicore processors, networked systems, massive computation clusters, and the web programming model were being worked around rather than addressed head-on. Moreover, the scale has changed: today's server programs comprise tens of millions of lines of code, are worked on by hundreds or even thousands of programmers, and are updated literally every day. To make matters worse, build times, even on large compilation clusters, have stretched to many minutes, even hours.
>
> Go was designed and developed to make working in this environment more productive. Besides its better-known aspects such as built-in concurrency and garbage collection, Go's design considerations include rigorous dependency management, the adaptability of software architecture as systems grow, and robustness across the boundaries between components.

Go at Google には、Google ではソフトウェアの開発やさまざまな環境下で増え続ける問題への対処のために開発されたと書かれていたり、また、開発の利便性を捨てて、大規

[*1] Go at Google: `https://talks.golang.org/2012/splash.article`

模ソフトウェアの問題に対処することにフォーカスをしています。

つまり、Go は開発のしやすさではなく、大規模ソフトウェア開発にまつわる問題を解決するために開発されたといえるでしょう。そのため、プログラミング言語を深く探求する人には言語仕様上、物足りなさを感じると思います。Go は問題解決のために作られたプログラミング言語ということが思想としてあります。

1.1 UNIX 哲学

プログラミングをしている人ならば、UNIX 哲学という単語は一度は聞いたことがあると思います。UNIX 哲学では、プログラムやソフトウェア開発における、文化的な規範や哲学的なアプローチの集合であり、UNIX やソフトウェアの開発者たちの経験に基づいて創出されたものです。

Go を開発した方たちは、UNIX や C 言語の開発者でもあるため、彼らが積み重ねてきた成功や失敗の経験を活かしています。それらを言語化された Go が暗黙的に UNIX 哲学にも結びついていると筆者は考えています。これから、ふたつの UNIX 哲学をもとに、どのように Go へ活かしていくかを解説します。

1.1.1 マキルロイの UNIX 哲学

マルコム・ダグラス・マキルロイ（Malcolm Douglas McIlroy）は、UNIX 創始者の一人であり、プロセス間通信を標準入出力を介して接続するパイプの発明者でもあります。マキルロイは UNIX 哲学をつぎのように形にしています。

> これが UNIX の哲学である。
> 一つのことを行い、またそれをうまくやるプログラムを書け。
> 協調して動くプログラムを書け。
> 標準入出力（テキスト・ストリーム）を扱うプログラムを書け。標準入出力は普遍的インターフェースなのだ。

"Do One Thing and Do It Well"、「ひとつのことを、うまくやれ」と要約されるこの哲学において、Go でツールを開発するとき、無駄な責務を与えないように開発をします。特に意識しているのはつぎの点です。

1. ひとつの責務に集中させる
2. 他のプログラムに連携可能にする

ひとつの責務に集中させる

プログラムをひとつの責務に集中させていれば複雑にならず、直感的にそのプログラムを使用することが可能です。たとえば、パッケージやコマンドラインツールにおいて、クエリやオプション、引数の渡し方によってプログラムの振る舞いを変えるという実装を多々見てきましたが、それを一度許してしまうと、「またひとつ、またひとつ」のようにオプションが増えていき、プログラムが複雑化していきます。

プログラムの「責務」を最初に定義し、そこからはみ出してしまうこと、つまり、振る舞いを無理に変えてしまうことを避けて開発をすることでプログラムの複雑度を増加させないようにします。これによって、美しい、キレイなコードを保つこともできるでしょう。

他のプログラムと連携可能にする

さて、ひとつの責務に集中させることに成功したら、別のプログラムと連携できる I/F を提供します。ひとつの責務に集中させることは、別の責務については別のプログラムに責任を委譲していることになるため、その責務の受け渡しをスムーズに行う必要があります。

パッケージでは関数のシグネチャとして、入力となる引数、出力となる戻り値、そして、関数の名前を適切に設計して公開しなければなりません。アンチパターンとして、入力の引数に関数の振る舞いを変えるための boolean を提供することは避けるべきです。

```
type Address struct {
    HomeTel string
    WorkTel string
}

func (a *Address) SetTel(tel string, isHome bool) {
    if isHome {
        a.HomeTel = tel
    } else {
        a.WorkTel = tel
    }
}
```

SetTel の第二引数に isHome という boolean 値を渡す必要がありますが、呼び出し側では tel を設定するだけなのに、なぜ、真偽値が必要になるかがコードを読まないと把握できません。このように、関数名から直感的に使えない I/F を提供するのは避けるべきです。

コマンドラインツールについては、後述するツール作成の章にてお話します。

1.1.2 ガンカーズの UNIX 哲学

マイク・ガンカーズ（Mike Gancarz）は、彼自身の経験を踏まえて、次の 9 つを UNIX 哲学として形にしています。

1. 小さいものは美しい。
2. 各プログラムが一つのことをうまくやるようにせよ。
3. できる限り早く原型（プロトタイプ）を作れ。
4. 効率よりも移植しやすさを選べ。
5. 単純なテキストファイルにデータを格納せよ。
6. ソフトウェアを梃子（てこ）として利用せよ。
7. 効率と移植性を高めるためにシェルスクリプトを利用せよ。
8. 拘束的なユーザーインターフェースは作るな。
9. 全てのプログラムはフィルタとして振る舞うようにせよ。

こちらは、「マキルロイの UNIX 哲学」と似ているところがありますが、より具体的に書かれていることや、ガンカーズの経験が活かされた哲学になっています。

動くものを作る

3. できる限り早く原型（プロトタイプ）を作れ。
5. 単純なテキストファイルにデータを格納せよ。

DRY の法則を知らないプログラマはいません（要出典）。しかし、DRY の法則を正確に理解している人は少ないです。

とあるプログラマはコードの重複を避けるために、あらゆる手段を用いてコードの重複を避けたが、逆に、複雑度が増加してしまい、メンテナンスコストが爆発的に増加してしまった。というおとぎ話のような話もあります。達人プログラマで DRY の法則について言及していた Dave Thomas も「DRY のことを、コードを重複させてはいけない、という意味にとっている人が多いようですが、そうではありません。DRY の背景にある考えは、それよりもっと大きなものなのです。」と明言[*2]しています。

すべて Go で書くな

4. 効率よりも移植しやすさを選べ。

[*2] DRY 原則の利用: コードの重複と密結合の間: https://www.infoq.com/jp/news/2012/05/DRY-code-duplication-coupling

7. 効率と移植性を高めるためにシェルスクリプトを利用せよ。

効率と移植を考えると、バイナリやコードのロジックの最適化、また、パッケージ設計の話に向かってしまう場面もありますが、そうではありません。

Go に慣れてくると、すべてを Go で実装してしまいたくなるのがエンジニアの性質ですが、すでに世の中にあるツールをうまく組み合わせて、最速で問題を解決へ導くのが事業会社にいるエンジニアには必須なスキルです。たとえば、ただ単に Web API を叩くだけのツールを作るのであれば `curl` とシェルスクリプトで十分に実現が可能です。

「そこ、本当に Go で書く必要ある？」のように自問し、本来必要である責務にのみ集中させるプログラムを記述することで、複雑度を減少させることも重要です。

1.2 早すぎる最適化は諸悪の根源である

「動くものを作る」、「すべて Go で書くな」はドナルド・エルビン・クヌース（Donald Ervin Knuth）が述べている「早すぎる最適化は諸悪の根源である」にも関連しています。まずはプロトタイプとなる動くものを作ることにフォーカスし、動いてから無駄なものを省いていく作業に移行することも重要です。YAGNI の法則でもあるように、機能は必要となるときに追加することとし、それまではシンプルに保ち続けることが重要です。そして、それがまだ不確実性の高い不透明なプロダクトであるなら、最適化にも手を出さずに待ち続けることが大事です。

Go では、`testing` パッケージにてベンチマークも公式から提供されているので、最適化をする前に、まずは計測を行い、必要なところを対処していくようにします。ベンチマークの検証方法については、後述するテストの章にて解説します。

第 2 章

標準パッケージから学ぶ Go 言語

Go 界隈では、「標準パッケージを参考にすべき」と語る人は、筆者も含めて少なくないと思います。Go も昔と比べれば、GitHub にも多くのライブラリが公開されています。しかし、GitHub にホスティングされている、「このリポジトリのパッケージ構成を参考にした方がよい」、というのはほとんど聞いた試しがありません。

筆者は、標準パッケージでなければ Google が開発をした go-github[*1]や go-cloud[*2]のような設計を参考にするなど、なるべく、「Go 言語らしさ」を理解している人がいるコミュニティのパッケージを参考にするようにしています。

また、筆者が、CLI ツールやパッケージを開発する場合、使用するライブラリの API をレビューをした上で、Go らしくない実装でなく API が提供されていれば、ソースコードのライセンスを踏まえた上で、ソースコードをそのままプロジェクトに含めてしまい、改修してしまうこともあります。Go ではベンダリングの機能も拡充していきていますが、開発以外の、依存パッケージのバージョン管理や後方互換性の検証という、問題解決をすることとは違うことに時間を掛けてしまうのは、Go が開発された意図に反していると考えます。

2.1 sort パッケージから学ぶ汎用例

Go でソート処理を実行する場合、sort パッケージの `sort.Interface` を満たし、`sort.Sort` を実行するか、`sort.Slice` 関数を実行してソートしているでしょう。この sort パッケージの実装を十分に理解することによって、振る舞いを内包した汎用的なパッケージを作成することができます。

[*1] github.com/google/go-github: `https://github.com/google/go-github`
[*2] github.com/google/go-cloud: `https://github.com/google/go-cloud`

sort パッケージについて、Go の公式ページからの引用ですが、サンプルコードから解説します。

```
package main

import (
    "fmt"
    "sort"
)

type Person struct {
    Name string
    Age  int
}

func (p Person) String() string {
    return fmt.Sprintf("%s: %d", p.Name, p.Age)
}

type ByAge []Person

func (a ByAge) Len() int           { return len(a) }
func (a ByAge) Swap(i, j int)      { a[i], a[j] = a[j], a[i] }
func (a ByAge) Less(i, j int) bool { return a[i].Age < a[j].Age }

func main() {
    people := []Person{
        {"Bob", 31},
        {"John", 42},
        {"Michael", 17},
        {"Jenny", 26},
    }

    fmt.Println(people)

    // ascending
    sort.Sort(ByAge(people))
    fmt.Println(people)

    // descending
    sort.Slice(people, func(i, j int) bool {
        return people[i].Age > people[j].Age
    })
    fmt.Println(people)

}
```

シンプルに記述されていますが、これを実行することによって、people のスライスを Person.Age によって昇順と降順のソートを行っています。昇順では sort.Interface を満たし、降順では sort.Slice をそれぞれ利用して、ソート処理を実現しています。

2.1.1 sort.Interface、sort.Sort

sort.Interface は、interface に３つの関数を定義しており、それらの関数を定義した型を提供するだけで、sort.Sort を実行することができます。

```
// A type, typically a collection, that satisfies sort.Interface can be
// sorted by the routines in this package. The methods require that the
// elements of the collection be enumerated by an integer index.
type Interface interface {
    // Len is the number of elements in the collection.
    Len() int
    // Less reports whether the element with
    // index i should sort before the element with index j.
    Less(i, j int) bool
    // Swap swaps the elements with indexes i and j.
    Swap(i, j int)
}
```

Len() int、Less(int, int) bool、Swap() (int, int) の３つは、ソート処理を行うときに、「これらの情報さえ知っておけば、ソート処理は実現可能」ということを示しています。

- Len() int: ソートするコレクションの要素数
- Less(int, int) bool: ソートにおける要素の比較
- Swap() (int, int): ソートにおける要素の入れ替え

これらを満たし、sort.Sort の引数に変数を渡して呼び出すことにより、ソート処理を実行します。

```
type Person struct {
    Name string
    Age  int
}
type ByAge []Person

func (a ByAge) Len() int           { return len(a) }
func (a ByAge) Swap(i, j int)      { a[i], a[j] = a[j], a[i] }
func (a ByAge) Less(i, j int) bool { return a[i].Age < a[j].Age }

func main() {
    people := []Person{
        {"Bob", 31},
        {"John", 42},
        {"Michael", 17},
        {"Jenny", 26},
```

```
    }

    sort.Sort(ByAge(people))
    fmt.Println(people)
}
```

簡略化しましたが、この例では、Person.Age をもとにソート可能にするため、sort.Interface を満たす ByAge を定義しています。このように、ByAge のように別で型を提供し、sort.Sort(ByAge(people)) のように実行することで、ソート処理の責務のみとなり、シンプルさが際立ちます。

2.1.2 sort.Slice

func Slice(slice interface\{}, less func(i, j int) bool) のシグネチャで用意されている sort.Slice 関数ですが、ソート対象のスライスと、ソートをする際の要素比較を関数として渡すことにより、sort.Interface を満たす型を提供せずとも、ソート処理が実現可能です。

```
type Person struct {
    Name string
    Age  int
}

func main() {
    people := []Person{
        {"Bob", 31},
        {"John", 42},
        {"Michael", 17},
        {"Jenny", 26},
    }

    sort.Slice(people, func(i, j int) bool {
        return people[i].Age > people[j].Age
    })
    fmt.Println(people)
}
```

sort.Interface と比べれば、コード量が削減されていますし、ByAge のような型も定義せずに、ソート処理が可能です。こちらの方が実用性が高いかもしれませんが、時には、ソート処理を満たした型を明示した方が可読性があがることももちろんあります。可読性、保守性の観点で、必要に応じて使い分けることが重要です。

2.1.3 sort.Sort の内部処理

パッケージの API となる I/F の設計ですが、sort.Sort では sort.Interface を引数に渡して呼び出す I/F となっています。しかし、処理の内部では、スライスの長さによって、アルゴリズムを変更するなど、振る舞いを変更しています。このような設計であれば、場合によって、処理実行における、前処理や後処理を行うことができます。このように、パッケージが提供している API の I/F を変更せず、内部アルゴリズムを変更することによって、使い手側は内部を知らずとも、最適なアルゴリズムで処理してくれることを担保できます。

```
func quickSort(data Interface, a, b, maxDepth int) {
    for b-a > 12 { // Use ShellSort for slices <= 12 elements
        if maxDepth == 0 {
            heapSort(data, a, b)
            return
        }
        maxDepth--
        mlo, mhi := doPivot(data, a, b)
        if mlo-a < b-mhi {
            quickSort(data, a, mlo, maxDepth)
            a = mhi // i.e., quickSort(data, mhi, b)
        } else {
            quickSort(data, mhi, b, maxDepth)
            b = mlo // i.e., quickSort(data, a, mlo)
        }
    }
    if b-a > 1 {
        for i := a + 6; i < b; i++ {
            if data.Less(i, i-6) {
                data.Swap(i, i-6)
            }
        }
        insertionSort(data, a, b)
    }
}

func Sort(data Interface) {
    n := data.Len()
    quickSort(data, 0, n, maxDepth(n))
}
```

sort パッケージの sort.go では、quickSort の処理をスライスの長さによって heapSort か insertionSort、また、quickSort を再帰的に呼び出す、ということをしており、実に興味深い実装になっています。

2.2 sort パッケージを模倣する

sort パッケージを模倣し、スライスの要素をシャッフルする shuffle パッケージというのを作成します。やり方は、sort パッケージと似たように、`shuffle.Interface` と `shuffle.Shuffle` という型と関数を定義します。

```
package shuffle

import (
    "math/rand"
    "time"
)

type Interface interface {
    Seed() int64
    Len() int
    Swap(int, int)
}

func Shuffle(data Interface) {
    rand.Seed(data.Seed())
    n := data.Len()
    for i := n - 1; i >= 0; i-- {
        j := rand.Intn(i + 1)
        data.Swap(i, j)
    }
}
```

`shuffle.Interface` は、シャッフルするのに必要な情報だけを取得できるよ、最低限の 3 つの関数を定義しています。

- `Seed() int64`: シャッフルするときのシードを設定する
- `Len() int`: シャッフルするコレクションの要素数
- `Swap() (int, int)`: シャッフルする要素の入れ替え

このような `interface` を提供することで、使用先では次のように使用することができます。[*3]

```
package main

import (
```

[*3] サンプルコード: https://github.com/kaneshin/playground/blob/master/shuffle/example/main.go

```
    "fmt"
    "os"

    "github.com/kaneshin/playground/shuffle"
)

type Dice []int

func (d Dice) Seed() int64   { return int64(os.Getpid()) }
func (d Dice) Len() int      { return len(d) }
func (d Dice) Swap(i, j int) { d[i], d[j] = d[j], d[i] }

var dice = Dice([]int{1, 2, 3, 4, 5, 6})

func main() {
    fmt.Printf("%v\n", dice)
    shuffle.Shuffle(dice)
    fmt.Printf("%v\n", dice)
}
```

第 3 章

Go 言語でのテストとベンチマーク

Go は標準でテストコードとベンチマークで計測するための testing パッケージが用意されています。しかし、そのパッケージを理解した上で、よいテストを書くことは簡単なことではありません。本章ではテストコードやベンチマークの一般的な書き方をサンプルコードとともに解説していきます。

3.1 テーブル駆動型テスト - Table-Driven Testing

Go では、標準パッケージも含めて、多くの状況でテーブル駆動型によるテストで記述されていることがあります。テーブルと呼ばれる、テストに必要な入力と期待する結果を含んだテストケースを準備し、テスト実行の出力にフォーカスした重複コードの無いテストを記述することができます。

テーブル駆動型テストにすることによって、テストコードの可読性や保守性を向上させることができます。Go の標準パッケージに点在するテストコードも参照するとほとんどがテストケースとしてのテーブルを準備しており、それに対してテストを繰り返し実行することで、コードの検証を行っています。

```
package main

import (
    "strings"
    "testing"
)

func Test_ToUpper(t *testing.T) {
    tests := []struct {
        input, expected string
    }{
        {"foo", "FOO"}, {"bar", "BAR"},
```

```
    }

    for _, tt := range tests {
        result := strings.ToUpper(tt.input)
        if tt.expected != result {
            t.Errorf("expected %v, but %v", tt.expected, result)
        }
    }
}
```

テーブルは、入力と期待する結果を構造体の配列として準備し、そのテーブルのテストアイテムについて繰り返しテストを実行します。このテーブル駆動型テストは、次に解説するサブテストと組み合わせることによって、効率的なテストを記述することができます。

3.2 サブテストによるテストの並行処理

Go1.7 からサブテストというテストの機能が追加になりました。このサブテストは、テストをテーブル駆動形テストとは別の方法で効率的かつ簡潔に記述することができます。テストコードを記述するときに問題になる、テストコードの煩雑化を軽減することができます。また、サブテストを並行で実行することもサポートしているため、テスト実行についても時間に対して効率的に実行することができます。

```
func Test_ToUpper(t *testing.T) {
    tests := []struct {
        input, expected string
    }{
        {"foo", "FOO"}, {"bar", "BAR"},
    }

    for _, tt := range tests {
        tt := tt
        // サブテストでの実行
        t.Run(tt.input, func(t *testing.T) {
            result := strings.ToUpper(tt.input)
            if tt.expected != result {
                t.Errorf("expected %v, but %v", tt.expected, result)
            }
        })
    }
}
```

テーブルとサブテストを掛け合わせることにより、ここまで簡潔にテストを記述することができますし、サブテストによって、複数のテストケースの集合を扱えるようになるため、テストスイートという考えてテストを記述することが可能です。

サンプルコード中に出てくる「tt := tt」はループ内で並行処理を行う際に、変数をキャプチャしておかないと変数への代入で意図しない結果となってしまうために、ループ内で改めて変数を定義しています。

3.2.1 setup と teardown

サブテストは、テスト実行されている関数内で、階層的にテストを記述することが可能でした。その階層化されたテストで setup や teardown と呼ばれる、テスト時の前処理と後処理を行うことも可能です。

```
func Test_Foo(t *testing.T) {
    // setup code
    t.Run("Sub", func(t *testing.T) {
        // setup code
        t.Run("Sub/1", func(t *testing.T) {
            // ...
        })
        t.Run("Sub/2", func(t *testing.T) {
            // ...
        })
        // teardown code
    })
    // teardown code
}
```

この前処理と後処理をシリアルで適切に実行可能なことにより、たとえば、永続データの後始末をした上で、再テストを実行することが可能になります。

ちょっとした Tips になりますが、テストコードの前処理と後処理はほとんどの場合はセットで定義されるため、まとめてひとつの関数に記述しておくことによって、処理のセットとして、コードの理解へとつながります。

```
func setupWithTeardown() func() {
    // setup code
    return func() {
        // teardown code
    }
}

// ケース1
func Test_Foo(t *testing.T) {
    // do setup
    teardown := setupWithTeardown()
    t.Run("Foo", func(t *testing.T) {
        // ...
    })
```

```
    // do teardown
    teardown()
}

// ケース2
func Test_Bar(t *testing.T) {
    // do setup and then stacking teardown func
    defer setupWithTeardown()()
    t.Run("Bar", func(t *testing.T) {
        // ...
    })
}
```

ケース 1 のパターンは、丁寧に処理の前後に関数を呼び出しています。ケース 2 のパターンは、setup を実行し、その後、コールスタックに teardown 関数をスタックさせています。

3.2.2 サブテストの並行処理

サブテストを実行している Run 関数内で、Parallel 関数をコールすることにより、テストを並行で処理させることができます。そして、すべてのサブテストの処理が終了し次第、親のテストを終了することができます。これは、teardown のコードがすべてのサブテストが終了してから実行されることが保証されることを意味しています。

```
func Test_ToUpper(t *testing.T) {
    tests := []struct {
        input, expected string
    }{
        {"foo", "FOO"}, {"bar", "BAR"},
    }

    for _, tt := range tests {
        tt := tt
        // サブテストでの実行
        t.Run(tt.input, func(t *testing.T) {
            t.Parallel() // 並行処理実行
            result := strings.ToUpper(tt.input)
            if tt.expected != result {
                t.Errorf("expected %v, but %v", tt.expected, result)
            }
        })
    }
}
```

上記のサンプルを実行すると、全てのサブテストが並行で処理されます。teardown を実行させる場合、次のようにサブテストで囲む必要があります。

```
func setupWithTeardown() func() {
    // setup code
    return func() {
        // teardown code
    }
}

func Test_ToUpper(t *testing.T) {
    tests := []struct {
        input, expected string
    }{
        {"foo", "FOO"}, {"bar", "BAR"},
    }

    // setup
    teardown := setupWithTeardown()
    // teardown 効かせるために Run で囲む
    t.Run("parent group", func(t *testing.T) {
        for _, tt := range tests {
            tt := tt
            // サブテストでの実行
            t.Run(tt.input, func(t *testing.T) {
                t.Parallel() // 並行処理実行
                result := strings.ToUpper(tt.input)
                if tt.expected != result {
                    t.Errorf("expected %v, but %v", tt.expected, result)
                }
            })
        }
    })
    // teardown
    teardown()
}
```

ここまでがサブテストと並行処理の話でした。サブテストと同じような記述方法でサブベンチマークも記述することができます。

サブテストを使用することにより、何かが大きく変わるわけではないです。ただ、テストコードの可読性と保守性が高ければ高いほど、メンテナンスが容易なテストコードになるため、メンテナンスのしやすいテストコードとするためにも積極的に使用することをお勧めします。

3.3 ベンチマークでのパフォーマンス測定

Go の testing パッケージでは、Go コードをベンチマークするための仕組みも標準パッケージが提供しています。ベンチマークでパフォーマンスを測定するための実装はテストコードと同じくらい容易ですし、実行も `go test` コマンドで使用して行います。

さて、ベンチマークでパフォーマンス計測をするための題材に、簡単な数値解析の

ニュートン法[*1]にて、平方根を求める関数を対象としてパフォーマンス測定をします。

```
func Sqrt(x float64) float64 {
    z := 1.0
    for i := 0; i < 10; i++ {
        z = z - (z*z-x)/(2*z)
    }
    return z
}
```

このニュートン法の数値計算によって求められる Sqrt 関数と、math パッケージが容易している math.Sqrt 関数の性能を比較します。

さて、実際にベンチマークのコードを書きますが、テストコードと基本の書き方は変わりません。テストコードの場合、テストコードと認識するために、TestXXX(t *testing.T) のような関数名と引数である必要がありますが、ベンチマークでは BenchmarkXXX(b *testing.B) のような関数名と引数が必要になります。

さて、今回のベンチマークは「2 の平方根を求める」ベンチマークを行います。

```
// ユーザ定義のSqrt関数
func Benchmark_Sqrt(b *testing.B) {
    for n := 0; n < b.N; n++ {
        Sqrt(2)
    }
}

// mathパッケージのSqrt関数
func Benchmark_MathSqrt(b *testing.B) {
    for n := 0; n < b.N; n++ {
        math.Sqrt(2)
    }
}
```

コード中で急に出てくる b.N ですが、これは対象となるベンチマークが信頼に足る回数分のループをベンチマークの方で判断して測定を行ってくれます。

このベンチマークを実行すると、次のように出力されます。

```
PASS
Benchmark_Sqrt           30000000                51.8 ns/op
Benchmark_MathSqrt       2000000000               0.36 ns/op
ok      github.com/eure/go-benchmark/math       2.373s
```

[*1] Tour of Go を最後までやり通した方ならご存知のはずです。

Benchmark_Sqrt の結果は、「30,000,000 回」のループを実施し、一回のループに平均「51.8ns」の実行時間が掛かっています。他方で、BenchmarkMathSqrt の結果は、「2,000,000,000 回」のループを実施し、一回のループに平均「0.36ns」の実行時間しか掛かっておらず、圧倒的に BenchmarkMathSqrt の方が実行速度が速いことがわかります。

ループの回数は b.N が決定しているため、基本的には、1 回のループの平均時間でパフォーマンスを比較します。今回だと math パッケージの Sqrt 実装の方がかなり高速で処理を完了することができていることが判断できます。

3.3.1 ベンチマークのメモリ測定

ベンチマークではメモリの測定も行うことができますが、本書では深く触れません。コマンド実行時に各種オプションが存在していますので、そちらをご確認ください。

-benchmem オプションでベンチマークを実行すると、メモリアロケーションに関する情報も出力してくれます。速度以外のパフォーマンス・チューニングを行う場合、必ず付けてベンチマークを実行してメモリ状態も併せて確認した方がよいです。

-benchtime オプションでベンチマークを実行すると、実行する時間をこちらで指定することが可能です。デフォルトの実行時間は 1 秒なので、10 秒に変更したい場合は -benchtime 10s をコマンド実行時に付与することで実現可能です。

第 4 章

Go 言語での CLI ツール

Go で CLI ツールを開発したことがある人は多くいると思います。また、業務で CLI ツールが必要になったため、それを目的に Go を書き始めた人も少なくないと思います。また、Go に限らず、他のスクリプト言語で CLI ツールを作成した方は本書を手にとっている人なら必ず開発した経験があるはずです。

Go で CLI ツールを作成しはじめた人は、まずはパッケージ構成に悩むと思います。しかし、CLI（Command-Line Interface）の I/F（Interface）が適切に設計された CLI ツールは少ないです。

ツールの I/F こそ、UNIX 哲学を意識し、「ひとつのことをうまくやる」を徹底し、他のプログラムと協働するように開発を心がけることで、ひとつ上のツール開発者になれるでしょう。

4.1 CLI ツールを開発する

コマンドの最適な開発は、標準パッケージを参考に理解を深めることができます。参考にすべきコマンドは、Go を書いたことがある方なら必ず使用したことがある、go コマンドの設計を理解することで、Go らしさを表現することができます。しかし、go コマンドを理解するには、他の Go の機能を知っておいた方がよいため、今回は題材にしません。

今回、作成途中の拙作ではございますが、QR code を画像で生成可能なエンコード処理、また、画像から文字列を取得するためのデコード処理を実装した QR code codex ツール[*1]にて解説をします。

[*1] github.com/kaneshin/qrcode: `https://github.com/kaneshin/qrcode`

4.1.1 パッケージ構成

Go のリポジトリで実行可能ファイルを作成する場合、cmd ディレクトリの配下に、実行ファイル名のディレクトリを作成し、そのディレクトリにて実行可能ファイルを作成します。

```
qrcode/
├── cmd/
│   ├── qrcode/
│   │   └── main.go
│   └── qrfmt/
│       └── main.go
├── decode.go
├── encode.go
├── format.go
└── README.md
```

今回のサンプルでは、cmd ディレクトリの中に、`qrcode` と `qrfmt` のディレクトリを作成して実行ファイルを作成します。

また、作成するツールをパッケージとして API 提供する場合、適切にディレクトリを作成して公開しておきます。今回の例では、qrcode ディレクトリ配下（`"github.com/kaneshin/qrcode"`）をそのまま使用可能なパッケージとして提供できるように設計しています。

cmd 配下で作成するファイルでは、ビジネスロジックとなる実装は極力避けています。今回の cmd/qrcode/main.go の例では、`"github.com/kaneshin/qrcode"`に QR code の Codex 処理を実装しています。その qrcode パッケージをインポートし、実行可能ファイルでは、Encode/Decode 処理を呼び出すだけの責任を負っています。

```
var decode = flag.Bool("d", false, "decode data")

func doEncode(dst io.Writer, src []byte) error {
    enc := qrcode.NewEncoder(dst)
    return enc.Encode(src)
}

func doDecode(dst io.Writer, src []byte) error {
    buf := bytes.NewBuffer(src)
    dec := qrcode.NewDecoder(buf)
    return dec.Decode(dst)
}

func do(r io.Reader) error {
```

```
    src, err := ioutil.ReadAll(r)
    if err != nil {
        return err
    }

    var buf bytes.Buffer
    if *decode {
        err = doDecode(&buf, src)
    } else {
        err = doEncode(&buf, src)
    }
    if err != nil {
        return err
    }

    os.Stdout.Write(buf.Bytes())
    return nil
}
```

このように、読み出し先となる io.Reader を do 関数に渡して呼び出すのがコマンドの責務であり、それ以降の Encode/Decode 処理は qrcode パッケージに責務を委譲させ、そちらで処理を行います。

4.2 CLI ツールの実行

今回のサンプルでは、qrcode と qrfmt の 2 種類をコマンドとして作成しています。それぞれのコマンドは、別の責務を委譲されており、適切に連携を行うような設計にしています。

* qrcode: 標準入力、もしくは、ファイルのテキストを Encode/Decode 処理し、標準出力に PNG 画像ファイルを出力する* qrfmt: QR code のフォーマットに従った文字列を標準出力に出力する

このように設計することにより、次のように、それぞれが連携した独立したツールと成り立っています。

```
qrfmt -ssid Foo -password Bar -type WPA2 | qrcode > WiFi-Foo.png
```

まさに、UNIX 哲学の章にて記載した「ひとつのことをうまくやる」を実現している I/F です。

4.2.1 標準入力とファイル名

CLI ツールにて、標準入力からテキストを入力されることにより、プロセス間でのパイプ処理が可能なツールを作成することができます。ここの設計次第で、使われるか使われないかのツールになるでしょう。また、場合によって、ファイルからテキストを入力することもあります。それら２つのケースに対応できる I/F をコードで実現します。

```
func run() error {
    var name string
    if args := flag.Args(); len(args) > 0 {
        name = args[0]
    }

    var r io.Reader
    switch name {
    case "", "-":
        // ケース：標準入力
        // $ echo 'input' | command
        //  or
        // $ command -
        r = os.Stdin
    default:
        // ケース：ファイル
        // $ command FILENAME
        f, err := os.Open(name)
        if err != nil {
            panic(err)
        }
        defer f.Close()
        r = f
    }

    // ...
}
```

標準入力での受け渡しは、os.ModeNamedPipe で検証できますが、使用する必要はありません。あまりオススメしないパターンですが、コマンドの引数にファイルではない、実行用のテキストを渡しているツールもあるので、その場合は必要になるケースがあります。

```
func main() {
    fi, err := os.Stdin.Stat()
    if err != nil {
        panic(err)
    }
```

```
    var r io.Reader
    if fi.Mode()&os.ModeNamedPipe == os.ModeNamedPipe {
        r = os.Stdin
    } else {
        args := flag.Args()
        if len(args) == 0 {
            r = os.Stdin
        } else {
            f, err := os.Open(args[0])
            if err != nil {
                panic(err)
            }
            defer f.Close()
            r = f
        }
    }
}
```

このように os.Stdin.Stat.Mode から判別することも可能です。

Go の開発思想からすれば、Go を用いてツール開発することは重要なモチベーションでもあるため、これからもツール作成者は増えてくるでしょう。読者の方は、ぜひ、標準パッケージのコマンドの作られ方と、UNIX 哲学を踏まえた I/F の設計を意識して開発をしてください。

おわりに

本書をお読みいただき、ありがとうございました。

Go 言語はシンプルであるがゆえに、小難しい設計を考えることは大規模アプリケーションとならなければ必要でないと思いますが、どの規模であっても、その「シンプルさ」を忘れて実装することは、Go の思想に背くことになり、Go 言語が本来開発された背景を忘れていることにつながります。

日本でも、盛り上がりを見せている Go 言語ですが、他の言語とは違う、「シンプルさ」という特徴を踏まえた「Go 言語らしさ」というところをより広くこれからも伝えていきたいと思います。

Go 言語らしく Go コードを実装するための手法と思想

2018 年 10 月 8 日　技術書典 5 版 v1.0.0
2018 年 10 月 9 日　電子書籍版 v1.0.1
2019 年 3 月 18 日　電子書籍版 v1.0.2
2020 年 1 月 5 日　電子書籍版 v1.1.0
2020 年 9 月 27 日　電子書籍版 v1.1.1

著　者　　kaneshin
イラスト　pekoro
編　集　　kaneshin
発行所　　エウレ・テクノロギア