ゴーファーの書

ゴーファーの書

golang.tokyo　著

2018-10-08 版　　発行

はじめに

本書の内容

本書は golang.tokyo[*1]という Go コミュニティの運営メンバーによって書かれた本です。著者たちは日頃の業務で Go を本格的に使用しています。そのため、各企業における実際のプロダクト開発で得た高いレベルのノウハウを 1 つの本に集めることができました。

読者のみなさんの Go によるプロダクト開発に本書を活かして頂くことができれば幸いです。また、まだ Go を業務で導入されてない方にとっては、導入のきっかけになればうれしい限りです。

golang.tokyo について

golang.tokyo はプログラミング言語の Go の導入企業のメンバーが集まり、Go の普及を推進するコミュニティです。トークイベント、ハンズオン、etc のイベントを開催しています。

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、製作、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、製作、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] https://golangtokyo.github.io

目次

第 1 章

Go クイズと解説

1.1　はじめに

メルペイ バックエンドエンジニアの@tenntenn[*1]です。本章では、2018 年 10 月 04 日に行われた Mercari Tech Conf 2018[*2]のメルカリ エキスパートチームブースで展示した Go クイズについて解説を書きたいと思います。

1.2　埋め込みとメソッド

1.2.1　問題

▼リスト 1.1　埋め込みとメソッド

```
package main

type Hoge struct{}

func (h *Hoge) A() string {
  return "Hoge"
}

func (h *Hoge) B() {
  println(h.A())
}

type Fuga struct {
  Hoge
}
```

[*1] https://github.com/tenntenn
[*2] https://techconf.mercari.com/2018

```
func (f *Fuga) A() string {
  return "Fuga"
}

func main() {
  var f Fuga
  f.B()
}
```

リスト 1.1 を実行した結果は次のうちどれになるでしょうか？

1. コンパイルエラー
2. Hogeと表示される
3. Fugaと表示される
4. パニックが発生する

1.2.2 解答と解説

問題のコードを実行してみる[*3]と、Hogeと表示されるため、答えは 2 となります。

構造体の埋め込みを Java などの"継承"と同じだと理解していると、この問題を間違ってしまうでしょう。構造体の埋め込みは、匿名フィールドという扱いです。

匿名フィールドはフィールド名を持たないため、アクセスする場合は`f.Hoge.B()`のように型名でアクセスします。また、`f.B()`は`f.Hoge.B()`と記述した場合と同じように動作するため、メソッド B内で呼んでいるメソッド Aは*Hoge型のものを呼び出しています。

1.3 組込み型の埋め込み

1.3.1 問題

▼リスト 1.2 組込み型の埋め込み

```
func main() {
  type Hoge struct{ int }
  var h Hoge
  fmt.Println(h)
}
```

リスト 1.2 を実行した結果は次のうちどれになるでしょうか？

[*3] https://play.golang.org/p/ShgVQOB3yXb

1. コンパイルエラー
2. {0}と表示される
3. {<nil>}と表示される
4. パニックが発生する

1.3.2 解答と解説

問題のコードを実行してみる[*4]と{0}と表示されます。

構造体には任意の名前付きの型を埋め込むことができます。そのため、int型のような組込み型であっても、名前付きの型であることには変わらないため、構造体に埋め込むことができます。

また、構造体のゼロ値はすべてのフィールドがゼロ値で初期化された値であるため、匿名フィールドである埋め込みも同様にゼロ値で初期化されています。そのため、Hoge構造体のゼロ値は匿名フィールドの intがゼロ値である 0で初期化された値となります。

1.4 スライスと引数

1.4.1 問題

▼リスト 1.3 スライスと引数

```
package main

import "fmt"

func f(ns []int) {
  ns[1] = 200
  ns = append(ns, 40, 50)
}

func main() {
  ns := []int{10, 20, 30}
  f(ns)
  fmt.Println(ns)
}
```

リスト 1.3 を実行した結果は次のうちどれになるでしょうか？

1. コンパイルエラー
2. [10 200 30]と表示される

[*4] https://play.golang.org/p/1Dp5JBiXEog

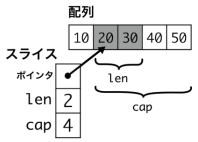
3. [10 20 30]と表示される
4. [10 200 30 40 50]と表示される

1.4.2 解答と解説

問題のコードを実行してみる[*5]と、[10 200 30]と表示されるため、答えは 2 となります。

スライスは配列の一部分を見ているウィンドウのようなものです。そのため、スライス自体は値を保持せず、図 1.1 のように配列の特定の要素へのポインタと、スライスのサイズである lenとスライスをどこまで拡張可能かを表す capを持ちます。

capはそのスライスの背後にある配列の大きさによって決まります。図 1.1 の場合は、配列の 1番目の要素へのポインタを保持しているため、そこから拡張可能な大きさは 4となります。



▲図 1.1　スライスの内部構造

実際に Go のランタイムではスライスはリスト 1.4 のような構造体で表現されています[*6]。

▼リスト 1.4　スライスを表す構造体

```
type slice struct {
  array unsafe.Pointer
  len int
  cap int
}
```

さて、問題のリスト 1.3 を見ると main関数内で nsというスライスを作り、関数 fに渡しています。変数@{ns}の値は関数 fの引数としてコピーされます。そのため、main関数の変数 nsと関数 fの引数 nsは別の領域に確保された変数となります。

[*5] https://play.golang.org/p/NM_KaC2XjiI
[*6] https://github.com/golang/go/blob/master/src/runtime/slice.go#L12-L16

しかし、スライスは内部に配列へのポインタを持つため、ns[1] = 200という式はポインタ経由で背後にある配列への代入が行われます。そのため、関数 f内の nsと main関数内の nsが持つ配列へのポインタの値は同じであるため、main関数内の nsにも代入の影響を受けます。

一方、関数 f内で行われる ns = append(ns, 40, 50について考えてみます。組込み関数の appendはスライスを第 1 引数に受け取り、第 2 引数以降で渡された値をそのスライスの後ろに追加していきます。

スライスの容量（cap）が十分に大きい場合、つまり、スライスのサイズ（len）と追加予定の要素の数を足した値が、capを超えない場合には、appendは背後にある配列の要素を追加予定の要素の値で上書きし、スライスの lenを更新します。

一方で、capの大きさを超えるような数の要素を追加する場合は、追加予定の要素がすべて入るように背後にある配列を再確保します。そのため、appendは新しい配列に、スライスが参照しているすべての要素をコピーし、追加予定の要素のコピーを行います。そして、スライスが保持する配列へのポインタと len、capの更新を行います。

このように、appendでは第 1 引数で渡したスライスが保持する配列のポインタや len、capなどの更新を行います。そのため、関数 fの中で引数 nsの値が変更されますが、その変更は呼び出し元の main関数に影響しません。

つまり、関数 f内で追加された 40と 50は、main関数内の nsで追加されたことになりません。そのため、[10 200 30]と表示されます。

1.5 スライスと defer

1.5.1 問題

▼リスト 1.5　スライスと defer

```
func main() {
  ns := []int{10, 20, 30}
  defer fmt.Println(ns)
  ns[1] = 200
  ns = append(ns, 40, 50)
}
```

リスト 1.5 を実行した結果は次のうちどれになるでしょうか？

1. コンパイルエラー
2. [10 20 30]と表示される
3. [10 200 30]と表示される

4. [10 200 30 40 50]と表示される

1.5.2 解答と解説

問題のコードを実行してみる[*7]と、[10 200 30]と表示されるため、答えは 3 となります。

deferで実行される関数の引数は deferが記述されている時点での値が用いられます。そのため、リスト 1.5 では deferのあとに変数 nsを書き換えても main関数終了時に fmt.Printlnを実行する際には影響を受けません。

しかし、前述の通り、スライスは内部に配列へのポインタを持つため、ns[1] = 200における代入はそのポインタ経由で行われるため出力される値は [10 200 30]となります。

1.6 ポインタとメソッドセット

1.6.1 問題

▼リスト 1.6 ポインタとメソッドセット

```
package main

import "fmt"

type Hoge interface {
  F() int
}

type Piyo struct {
  N int
}

func (p *Piyo) F() int {
  p.Inc()
  return p.N
}

func (p Piyo) Inc() {
  p.N++
}

func main() {
  var h Hoge = Piyo{N: 100}
  fmt.Println(h.F(), h.F())
}
```

[*7] https://play.golang.org/p/UNXGfcDL8Nn

リスト 1.6 を実行した結果は次のうちどれになるでしょうか？

1. コンパイルエラー
2. 101 102と表示される
3. 100 100と表示される
4. パニックが発生する

1.6.2 解答と解説

問題のコードを実行してみる[*8]と、次のようにコンパイルエラーが発生します。

```
./main.go:23:6: cannot use Piyo literal (type Piyo) as type Hoge in assignment:
  Piyo does not implement Hoge (F method has pointer receiver)
```

そのため、答えは 1 となります。

main関数で定義されている変数 hの方は Hogeです。Hogeはインタフェースであるため、hに代入できるのは Hogeインタフェースを実装した型の値となります。

ある型が Hogeインタフェースを実装するには、その型が F() intというメソッドを持っている必要があります。

リスト 1.6 では、構造体型の Piyo型が定義されています。そして、そのポインタ型をレシーバとしてメソッド Fが実装されています。

言語仕様[*9]を見ると、*T型のメソッドセットには Tのメソッドセットも含みますが、T型のメソッドセットには*T型のメソッドを含みません。そのため、main関数で変数 hに代入しようとしている値は Piyo型の値であるため、コンパイルエラーとなります。

リスト 1.7 のようなシンプルな例で見てみると、言語仕様で言わんとしていることが分かるでしょう。

▼リスト 1.7 T 型と*T 型のメソッドセット

```
package main

type T struct{}

func (_ T) F() {}
```

[*8] https://play.golang.org/p/xdfeX_IrMOq
[*9] https://golang.org/ref/spec#Method_sets

```
func (_ *T) G() {}

func main() {
  var _ interface{ F() } = &T{}
  var _ interface{ G() } = T{} // こちらはエラーになる
}
```

なお、リスト 1.7 を実行すると[*10]次のような結果になります。

```
main.go:11:6: cannot use T literal (type T)
as type interface { G() } in assignment:
  T does not implement interface { G() } (G method has pointer receiver)
```

また、リスト 1.6 の var h Hoge = Piyo{N: 100}を var h Hoge = &Piyo{N: 100 }とした場合には、Incメソッドのレシーバがポインタではないため、100 100と表示されます。

この問題はレシーバをポインタにしないとフィールドへの代入が意味がないことを確かめる問題に見せかけて、ポインタのメソッドセットの仕様を理解しているかをチェックする問題でした。

1.7 マップの初期化

1.7.1 問題

▼リスト 1.8 マップの初期化

```
package main

import "fmt"

type Key struct {
  V1 int
  V2 [2]int
}

func main() {
  var m map[Key]int
  key := Key{V1: 100, V2: [2]int{1, 2}}
  m[key] = 100
  fmt.Println(m[key])
}
```

[*10] https://play.golang.org/p/BdsE7GYUwig

リスト 1.8 を実行した結果は次のうちどれになるでしょうか？

1. コンパイルエラー
2. 100と表示される
3. 0と表示される
4. パニックが発生する

1.7.2 解答と解説

問題のコードを実行してみる[*11]と次のようにパニックが発生します。

```
panic: assignment to entry in nil map

goroutine 1 [running]:
main.main()
  /tmp/sandbox467872364/main.go:13 +0x80
```

変数 mはマップで初期化されていないため、値はゼロ値である nilになっています。そのため、m[key] = 100のように代入しようとするとパニックが発生してしまいます。

なお、変数 mを m := map[Key]int{}のように初期化しておいた場合には、100が表示されます。マップのキーにできる型は==で比較できる型であれば問題ありません。そのため、構造体や配列は==で比較可能であるため、マップのキーに用いることができます。しかし、構造体のフィールドや配列の要素に==で比較できない型が含まれている場合は、その構造体や配列も==で比較できなくなるため、マップのキーに用いることはできません。==で比較できない型とは、関数やスライスになります。

リスト 1.8 の Key構造体のフィールド V2は配列であるため、マップのキーとして用いることは問題ありません。しかし、V2がスライスであった場合は、Key型をマップのキーとして利用することはできなくなります。

この問題ではマップのキーに使える型はどういうものなのかという疑問を持ちながら解答することで、うっかりマップの初期化という初歩的なミスを見落としてしまうという点を突いたものです。

[*11] https://play.golang.org/p/5dXZGhxnILM

1.8 len と配列のポインタ

1.8.1 問題

▼リスト 1.9 len と配列のポインタ

```
func main() {
  ns := [...]int{10, 20, 30}
  ptr := &ns
  println(len(ptr))
}
```

リスト 1.9 を実行した結果は次のうちどれになるでしょうか？

1. コンパイルエラー
2. 3と表示される
3. 0と表示される
4. パニックが発生する

1.8.2 解答と解説

問題のコードを実行してみる[*12]と 3が表示されます。
組込み関数である lenは、次のような型の値を引数として受け付けます[*13]。

- スライス
- マップ
- チャネル
- 配列
- 配列のポインタ

そのため、ptrは配列のポインタであるため、そのポインタが指す配列の要素数となり 3が表示されます。

[*12] https://play.golang.org/p/0Xy_4aqOU_d
[*13] https://golang.org/ref/spec#Length_and_capacity

1.9 len とスライスのポインタ

1.9.1 問題

▼リスト 1.10 len とスライスのポインタ

```
func main() {
  ns := []int{10, 20, 30}
  ptr := &ns
  println(len(ptr))
}
```

リスト 1.10 を実行した結果は次のうちどれになるでしょうか？

1. コンパイルエラー
2. 3と表示される
3. 0と表示される
4. パニックが発生する

1.9.2 解答と解説

問題のコードを実行してみる[*14]と次のようなコンパイルエラーが発生します。

```
main.go:6:13: invalid argument ptr (type *[]int) for len
```

前述の通り、組み込み関数の lenはスライスのポインタを取ることができないため、コンパイルエラーが発生してしまいます。

1.10 len とチャネル

1.10.1 問題

▼リスト 1.11 len とチャネル

[*14] https://play.golang.org/p/ALWAWAtQSCJ

```
func main() {
  ch := make(chan int, 10)
  println(len(ch))
}
```

リスト 1.11 を実行した結果は次のうちどれになるでしょうか？

1. コンパイルエラー
2. 10と表示される
3. 0と表示される
4. パニックが発生する

1.10.2 解答と解説

問題のコードを実行してみる[*15]と 0が表示されます。

組込み関数の lenの引数にチャネルを渡した場合、チャネル内に保持されてる値の数が返されます。そのため、リスト 1.11 では 1 つも値を入れてないため、0が表示されます。

一方、組込み関数の capはチャネルの容量を返すため、リスト 1.11 のチャネルの場合は 10が返されます。

この問題は、チャネルに対して組込み関数の lenや capが使えることと、それぞれの挙動のち外を理解しているかを問う問題です。

1.11 close の複数回呼び出し

1.11.1 問題

▼リスト 1.12 close の複数回呼び出し

```
func main() {
  ch := make(chan int, 10)
  ch <- 100
  close(ch)
  close(ch)
  println(len(ch))
}
```

リスト 1.11 を実行した結果は次のうちどれになるでしょうか？

[*15] https://play.golang.org/p/AhQJj-pVNPd

1. コンパイルエラー
2. 10と表示される
3. 1と表示される
4. パニックが発生する

1.11.2 解答と解説

問題のコードを実行してみる[*16]と次のようにパニックが発生します。

```
panic: close of closed channel

goroutine 1 [running]:
main.main()
  /tmp/sandbox289574606/main.go:7 +0xa0
```

クローズされたチャネルに対して再度 closeを呼び出すとパニックが発生します。

また、closeの呼び出しが 1 度だけであった場合、closeされたチャネルであっても len関数を呼び指すとチャネル内に持つデータの数を返すため、1と表示されます。

1.12 close によるブロードキャスト

1.12.1 問題

▼リスト 1.13 close によるブロードキャスト

```
func main() {
  ch1 := make(chan int)
  ch2 := make(chan int)
  go func() { <-ch1; print("A"); close(ch2) }()
  close(ch1)
  <-ch2
  print("B")
}
```

リスト 1.13 を実行した結果は次のうちどれになるでしょうか？

1. コンパイルエラー
2. ABと表示される
3. BAと表示される

[*16] https://play.golang.org/p/IhlG2jK7vFA

4. デッドロックが発生する

1.12.2 解答と解説

問題のコードを実行してみる[*17]と ABと表示されます。

チャネルが closeされるとそのチャネルから受信を行っている箇所に直ちに closeされたことが第 1 戻り値のゼロ値と第 2 戻り値の falseで伝えられます。

そのため、ch1と ch2を初期化したあとにゴールーチンが生成されます。その後、mainゴールーチンが ch1を closeし、<-ch2で ch2からの受信を待ちます。

その間にもう一つのゴールーチンに処理が切り替わり、<-ch1で ch1からの受信を行います。しかし、ch1はすでに closeされているため、<-ch1は直ちに終了し、次の処理である print("A")が実行されます。

そして、ch2が closeされ、それが mainゴールーチンの<-ch2を行っている箇所に伝えられます。そこから次の処理に移り、print("B")が実行されます。最終的には ABが表示されていることになります。

この問題は特に引っ掛けみたいなものはなく、単純に closeの挙動を理解しているかを問う問題です。

1.13 nil チャネル

1.13.1 問題

▼リスト 1.14　nil チャネル

```
func main() {
  var ch1 chan chan int
  select {
  case <-ch1:
    println("A")
  default:
    println("B")
  }
}
```

リスト 1.14 を実行した結果は次のうちどれになるでしょうか？

1. コンパイルエラー
2. Aと表示される

[*17] https://play.golang.org/p/Mkza5ajAg8T

3. Bと表示される
4. パニックが発生する

1.13.2 解答と解説

問題のコードを実行してみる[*18]と Bと表示されます。

リスト 1.14 を見ると、チャネルのチャネルである var ch1 chan chan intに目が行きがちですが、この部分は問題を解く上で重要ではありません。チャネルはファーストクラスオブジェクトであり、引数や戻り値に取ることができます。そのため、チャネルのチャネルを定義することができることは、特に不自然ではありません。

一方、チャネル型のゼロ値は nilであり、リスト 1.14 の ch1も make関数で明示的に初期化されているわけではないため、値は nilです。

値が nilであるチャネルを selectの caseに用いた場合、その caseは絶対に実行されることはありません。これをうまく使い、nilを代入することで特定のチャネルからの受信を一時的に止めることに使うようなパターンを nilチャネルパターンと呼びます。

1.14 型エイリアス

1.14.1 問題

▼リスト 1.15　型エイリアス

```
func main() {
  type A = int
  var a A
  fmt.Printf("%T", a)
}
```

リスト 1.15 を実行した結果は次のうちどれになるでしょうか？

1. コンパイルエラー
2. Aと表示される
3. intと表示される
4. パニックが発生する

[*18] https://play.golang.org/p/dd2DZZ5N2cG

1.14.2 解答と解説

問題のコードを実行してみる[*19]と intと表示されます。

type A = intのように記述すると型 intのエイリアスとして、型 Aを定義するという意味になり、この機能を型エイリアスと呼びます。型エイリアスは、type A intのように新しい型として定義しているわけではなく、まったく同じ型としてエイリアスを作成しているだけです。

そのため、fmt.Printfの %T書式で型名を表示させると、エイリアス元である intが表示されます。

なお、type A = intではなく、type A intのように定義してあった場合は、Aが出力されます。

この問題は Go1.9 で導入された型エイリアスがどのようなものか理解していることが問われる問題です。

1.15 変数 true

1.15.1 問題

▼リスト 1.16 変数 true

```
func main() {
  true := false
  println(true == false)
}
```

リスト 1.16 を実行した結果は次のうちどれになるでしょうか？

1. コンパイルエラー
2. trueと表示される
3. falseと表示される
4. パニックが発生する

[*19] https://play.golang.org/p/NPL652NE-14

1.15.2 解答と解説

問題のコードを実行してみる[*20]と trueと表示されます。

trueや falseは予約語ではありません。そのため、trueや falseといった名前の変数を作ることが可能です。

リスト 1.16 では trueという名前の変数を作り、そこに値として falseを代入しています。そして、println(true == false)は、変数である trueと値の falseを比較しており、変数 trueには falseが入っているため、false == falseとなり、trueが表示されます。

この問題は、予約語とはなにか、予約語にはどのようなものがあるかなどを理解しているかを問う問題です。

1.16 大きな定数の演算

1.16.1 問題

▼リスト 1.17 大きな定数の演算

```
func main() {
  n := 1000000000000000000 / 1000000000000000000
  println(n)
}
```

リスト 1.17 を実行した結果は次のうちどれになるでしょうか？

1. コンパイルエラー
2. 1と表示される
3. NaNと表示される
4. パニックが発生する

1.16.2 解答と解説

問題のコードを実行してみる[*21]と 1と表示されます。

定数は明示的に型を指定しない場合、特定の型を持ちません[*22]。そのため、十分に大

[*20] https://play.golang.org/p/dd2DZZ5N2cG
[*21] https://play.golang.org/p/pHO_-1Mkfu2
[*22] https://blog.golang.org/constants

きいな定数であっても intの範囲に収まっていなくても、型なしの定数同士の演算であれば問題ありません。

演算の結果を変数に代入するときになってはじめて、その変数の型のサイズに収まるかが問題になってきます。リスト 1.17 で用いられている定数の 100000000000000000000は十分に int型の範囲を超えています。しかし、1000000000000000000 / 10000000000000000000の結果は 1となり、println に渡され t る段階では int型の範囲に収まっているため、特にコンパイルエラーは発生せずに実行ができます。

この問題は Go の定数を正しく理解しているかを問われる問題です。

1.17 nil なレシーバ

1.17.1 問題

▼リスト 1.18 nil なレシーバ

```
type Hoge struct{N int}

func (h *Hoge) F() {
  fmt.Println(h)
}

func main() {
  var h *Hoge
  h.F()
}
```

リスト 1.18 を実行した結果は次のうちどれになるでしょうか？

1. コンパイルエラー
2. {0}と表示される
3. <nil>と表示される
4. パニックが発生する

1.17.2 解答と解説

問題のコードを実行してみる[*23]と<nil>と表示されます。

メソッドのレシーバはほとんど引数と同じ扱いを受けます。そのため、レシーバはメソッド呼び出し毎にコピーされてメソッドに渡されます。また、レシーバの値が nilであっても呼び出すことができます。

この問題はレシーバがどういう扱いを受けるのか理解していることを問う問題です。

[*23] https://play.golang.org/p/1GfQZ3QWAXi

1.18 おわりに

この章では Go の文法や言語仕様、ランタイムの挙動の知識が問われる Go クイズについて扱いました。ここで扱ったコードはほとんどがプロダクトコードで書くようなコードでは無いうえに、コンパイラが指摘したりテストで気づけるものが多かったでしょう。

しかし、言語を深く理解することは悪いことではなく、効率よくコードを書く支えになってくれます。この章を通じて言語仕様やランタイムの挙動に興味を持ち、コンパイラやランタイムのコード読む方が増えてくれると嬉しいです。

第 2 章

HTTP パケットが飛んでいくまで

2.1　はじめに

ネクストカレンシー株式会社でバックエンドエンジニアをしている sonatard[*1]です。

本章では、Go の HTTP リクエストの送信処理の概要を紹介します。

以下のような関数は、Go で開発している方なら多くの方がお世話になっている関数だと思います。しかし最終的にどのようにパケットが送られるかご存知でしょうか？ 本章では、そのゴールを突き止めたいと思います。

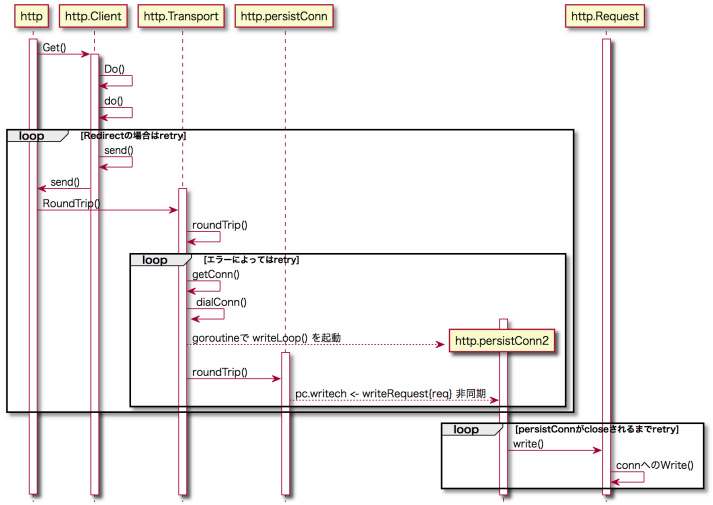
```
resp, err := http.Get(url)
resp, err := http.PostForm(url, values)
```

2.2　HTTP リクエストの送信シーケンス

2.2.1　送信処理の概要

以下が HTTP リクエストのシーケンス図になります。以降の章は、このシーケンスと比較しながら読んでみてください。

[*1] https://twitter.com/sonatard



▲図 2.1　HTTP リクエストシーケンス

2.2.2　func Get(url string) (resp *Response, err error)

ユーザが自身で `http.Clinet{}` を作成しなかった場合に利用する関数です。

http package 内で初期化されている `DefaultCliet` の `Celint.Get()`を実行します。

```
// DefaultClient is the default Client and is used by Get, Head, and Post.
var DefaultClient = &Client{}

func Get(url string) (resp *Response, err error) {
    return DefaultClient.Get(url)
}
```

2.2.3　func (c *Client) Get(url string) (resp *Response, err error)

`NewRequest` では、URL のパースや Host の設定などをして `http.Request` を作成します。作成した `http.Request` を引数に渡して `Client.Do()` を実行しています。

```
func (c *Client) Get(url string) (resp *Response, err error) {
    req, err := NewRequest("GET", url, nil)
    if err != nil {
        return nil, err
    }
    return c.Do(req)
}
```

2.2.4 func (c Client) Do(req *Request) (Response, error)

```
func (c *Client) Do(req *Request) (*Response, error) {
    return c.do(req)
}
```

2.2.5 func (c *Client) do(req *Request) (retres *Response, reterr error)

ここから具体的な HTTP リクエストの送信処理のための準備が始まります。今までは HTTP の GET メソッドに依存した処理でしたが、ここからは POST なども共通に利用されます。

次に実行される `Client.send` を実行しているだけですが、 ここでは Redirect について制御をしています。レスポンスのステータスコードから Redirect が必要だと判断した場合には、Location ヘッダを読み、次の Request の URL や Host として設定されます。

```
func (c *Client) Do(req *Request) (*Response, error) {
    // リダイレクトされる限りループする
    for {
        // 1 度でも Redirect された場合
        if len(reqs) > 0 {
            // Location ヘッダの読み込みや次の Request の作成
            // 省略
        }

        reqs = append(reqs, req)
        resp, didTimeout, err = c.send(req, deadline);
        // 省略
        redirectMethod, shouldRedirect, includeBody
                = redirectBehavior(req.Method, resp, reqs[0])
        if !shouldRedirect {
            return resp, nil
        }
    }
```



```
}
```

2.2.6 func (c *Client) send(req *Request, deadline time.Time) (resp *Response, didTimeout func() bool, err error)

Cookie の処理をして `http.sned` を実行します。

```
// didTimeout is non-nil only if err != nil.
func (c *Client) send(req *Request, deadline time.Time)
                (resp *Response, didTimeout func() bool, err error) {
    if c.Jar != nil {
        for _, cookie := range c.Jar.Cookies(req.URL) {
            req.AddCookie(cookie)
        }
    }
    resp, didTimeout, err = send(req, c.transport(), deadline)
    if err != nil {
        return nil, didTimeout, err
    }
    if c.Jar != nil {
        if rc := resp.Cookies(); len(rc) > 0 {
            c.Jar.SetCookies(req.URL, rc)
        }
    }
    return resp, nil, nil
}
```

2.2.7 func send(ireq *Request, rt RoundTripper, deadline time.Time) (resp *Response, didTimeout func() bool, err error)

ここでは、まず改めて http.Request の値を設定します。http.Request 構造体の nil の部分にメモリを確保と Basic 認証のために Authorization ヘッダの設定を行なっています。このとき元の request を更新してしまうと、Redirect 時の次のループで影響するため更新する場合はコピーを行なってから更新しています。http.Request の準備ができたら送信処理の `RoundTrip(req *Request) (*Response, error)` を実行します。これは `RoundTripper` Interface の `RoundTripper` メソッドで、あり `Client` の `Transport` に設定することで異なる処理を実行することもできます。何も挟まなければ、`net/http/transport.go` の `func (t *Transport) RoundTrip(req *Request) (*Response, error)` が実行されます。

```
func send(ireq *Request, rt RoundTripper, deadline time.Time)
           (resp *Response, didTimeout func() bool, err error)
    req := ireq // req is either the original request, or a modified fork
    //省略

    // 1 度だけコピーを行うための関数
    forkReq := func() {
        if ireq == req {
            req = new(Request)
            *req = *ireq // shallow clone
        }
    }

    if req.Header == nil {
        forkReq()
        req.Header = make(Header)
    }

    // Basic 認証のヘッダを作成
    if u := req.URL.User; u != nil && req.Header.Get("Authorization") == "" {
        username := u.Username()
        password, _ := u.Password()
        forkReq()
        req.Header = cloneHeader(ireq.Header)
        req.Header.Set("Authorization", "Basic "+basicAuth(username, password))
    }
    // 省略

    resp, err = rt.RoundTrip(req)
    // 省略
}
```

2.2.8 func (t Transport) RoundTrip(req *Request) (Response, error)

Transport.RoundTrip は RoundTrip Interface の実装です。以下の通り `Transport.roundTrip` を実行しています。

```
func (t *Transport) RoundTrip(req *Request) (*Response, error) {
    return t.roundTrip(req)
}
```

2.2.9 func (t Transport) roundTrip(req *Request) (Response, error)

引数の http.Request に基づいてヘッダのバリデーションやコネクションの管理をして、`persistConn.roundTrip` を実行します。


```
func (t *Transport) RoundTrip(req *Request) (*Response, error) {
    // 省略
        // HTTP ヘッダが正しいことを検証する
        for k, vv := range req.Header {
            if !httpguts.ValidHeaderFieldName(k) {
                return nil, fmt.Errorf("net/http: invalid header field name %q", k)
            }
            for _, v := range vv {
                if !httpguts.ValidHeaderFieldValue(v) {
                    return nil, fmt.Errorf("net/http: invalid header
                                           field value %q for key %v", v, k)
                }
            }
        }

    // 省略

    // err によってはリトライする
    for {
        // キャッシュされたコネクションの取得をするか、新たにコネクションを作成する。
        pconn, err := t.getConn(treq, cm)
        if err != nil {
            t.setReqCanceler(req, nil)
            req.closeBody()
            return nil, err
        }
        var resp *Response
        if pconn.alt != nil {
            // HTTP/2 path.
            t.setReqCanceler(req, nil) // not cancelable with CancelRequest
            resp, err = pconn.alt.RoundTrip(req)
        } else {
            resp, err = pconn.roundTrip(treq)
        }
        // 成功すれば response を返すが、成功しなければ for ループにより再実行する
        if err == nil {
            return resp, nil
        }
        if !pconn.shouldRetryRequest(req, err) {
            // Issue 16465: return underlying net.Conn.Read error from peek,
            // as we've historically done.
            if e, ok := err.(transportReadFromServerError); ok {
                err = e.err
            }
            return nil, err
        }
       // 省略
    }
```

2.2.10 func (t Transport) getConn(treq *transportRequest, cm connectMethod) (persistConn, error)

getConn ではキャッシュされた TCP コネクションを取得するか、キャッシュがない場合には新たに TCP コネクションを確立します。新規にコネクションを作成する際には

dial を実行しますが、既に確立されているコネクションの終了が先の場合にはそちらを使います。

```
func (t *Transport) getConn(treq *transportRequest,
                cm connectMethod) (*persistConn, error) {
    // 省略
    case pc := <-idleConnCh:
        // 省略
        return pc, nil

        // 省略
}
```

2.2.11 func (pc *persistConn) roundTrip()

persistConn では、gzip 圧縮の処理、送信、結果の待ち受けを行います。この後の送信処理は非同期で行われるため別の channel に渡すことになります。

こちらはユーザが実行したタスクで行われる送信処理の最深部になります。

```
func (pc *persistConn) roundTrip(req *transportRequest)
                                (resp *Response, err error) {
    if !pc.t.replaceReqCanceler(req.Request, pc.cancelRequest) {
        pc.t.putOrCloseIdleConn(pc)
        return nil, errRequestCanceled
    }

    requestedGzip := false
    if !pc.t.DisableCompression &&
        req.Header.Get("Accept-Encoding") == "" &&
        req.Header.Get("Range") == "" &&
        req.Method != "HEAD" {
        requestedGzip = true
        req.extraHeaders().Set("Accept-Encoding", "gzip")
    }

    // 送信処理
    startBytesWritten := pc.nwrite
    writeErrCh := make(chan error, 1)
    pc.writech <- writeRequest{req, writeErrCh, continueCh}

    // 結果の待ち受け
    for {
        select {
        // 省略
        case <-respHeaderTimer:
            pc.close(errTimeout)
            return nil, errTimeout
        case re := <-resc:
            if re.err != nil {
```



```
                return nil, pc.mapRoundTripError(req, startBytesWritten, re.err)
            }
            return re.res, nil
        // 省略
        }
    }
}
```

write する channle は以下の `chan writeRequest` になります。

```
type persistConn struct {
    // 省略
    writech   chan writeRequest   // written by roundTrip; read by writeLoop
    // 省略
    mutateHeaderFunc func(Header)
}
```

`chan writech` を受けているのは、`func (pc *persistConn) writeLoop()` になります。

2.2.12 func (pc *persistConn) writeLoop()

`writeLoop` は、ユーザ実行タスクから送信する `http.Request` を Channel 経由で受け取り送信処理を実行します。

```
func (pc *persistConn) writeLoop() {
    // 省略
    for {
        select {
        case wr := <-pc.writech:
            startBytesWritten := pc.nwrite
            err := wr.req.Request.write(pc.bw, pc.isProxy,
                        wr.req.extra, pc.waitForContinue(wr.continueCh))
    // 省略
}
```

このとき `write` の 第 1 引数に渡している `pc.pw` は以下になります。

```
type persistConn struct {
    // 省略
    bw          *bufio.Writer       // to conn
    // 省略
}
```

writeLoop の実行と bw *bufio.Writer がいつ設定されるかというと getConn() から呼ばれる dialConn() の内部になります。

```
func (t *Transport) dialConn(ctx context.Context, cm connectMethod)
                                                    (*persistConn, error) {
    // 省略
    pconn.bw = bufio.NewWriter(persistConnWriter{pconn})
    go pconn.writeLoop()
    return pconn, nil
}
```

persistConnWriter は、以下の通りです。persistConn の conn に Write していることがわかります。

conn は、 net.Conn になります。

```
type persistConnWriter struct {
    pc *persistConn
}

func (w persistConnWriter) Write(p []byte) (n int, err error) {
    n, err = w.pc.conn.Write(p)
    w.pc.nwrite += int64(n)
    return
}
```

```
type persistConn struct {
    conn      net.Conn
}
```

あとは func (req *Request) write(w io.Writer, usingProxy bool, extraHeaders Header, waitForContinue func() bool) (err error) に渡している w に対して実際に Write しているところを見つければゴールです。

2.2.13 func (req *Request) write(w io.Writer, usingProxy bool, extraHeaders Header, waitForContinue func() bool) (err error)

ここでいよいよ HTTP リクエストで実際に送信するテキストを組み立てていきます。

HTTP1 はバイナリフォーマットではなくテキストフォーマットの珍しいプロトコルであるため、文字列にして w に対して書き込んでいきます。ここにたどり着くまでに送るべきデータの構造体は完成しているので、それを HTTP の Header や Body データに変換していきます。

net.Conn の Write への書き込みは *bufio.Writer 型なので、バッファが行われるため、書き込んだタイミングで即座に送信されるというわけではありません。そのため必要なタイミングで flush を実行しています。

```
func (req *Request) write(w io.Writer, usingProxy bool, extraHeaders Header,
                                        waitForContinue func() bool) (err error) {
    // 省略

    // メソッドを送信
    _, err = fmt.Fprintf(w, "%s %s HTTP/1.1\r\n",
                         valueOrDefault(r.Method, "GET"), ruri)
    if err != nil {
        return err
    }

    // ヘッダを送信
    // Header lines
    _, err = fmt.Fprintf(w, "Host: %s\r\n", host)
    if err != nil {
        return err
    }

    // ユーザエージェントの送信
    userAgent := defaultUserAgent
    if _, ok := r.Header["User-Agent"]; ok {
        userAgent = r.Header.Get("User-Agent")
    }
    if userAgent != "" {
        _, err = fmt.Fprintf(w, "User-Agent: %s\r\n", userAgent)
        if err != nil {
            return err
        }
        if trace != nil && trace.WroteHeaderField != nil {
            trace.WroteHeaderField("User-Agent", []string{userAgent})
        }
    }

    _, err = io.WriteString(w, "\r\n")
    if err != nil {
        return err
    }
    // 省略

    // Body の書き込み
    // Write body and trailer
    err = tw.writeBody(w)
    if err != nil {
        if tw.bodyReadError == err {
            err = requestBodyReadError{err}
        }
        return err
    }
```

```
    if bw != nil {
        // バッファをフラッシュ、本当の送信
        return bw.Flush()
    }
}
```

2.2.14 HTTP パケット送信シーケンスまとめ

Go ではとても簡単に HTTP リクエストを送ることができますが、これは Go が標準ライブラリでこれだけの処理を実装してくれているためです。net.Conn への Write からシステムコールの socket の send の実装は？ Response はどのようにして受け取るのか? など、まだ疑問は残りますが続きはみなさんで調べてみてください。

2.3 学び

次に送信の一連の処理を読んでいて気が付いた学びを紹介します。

2.3.1 Channel の実例

Channel のサンプルでよく登場するタイマーによるリクエストのキャンセル処理が書かれています。

```
func setRequestCancel(req *Request, rt RoundTripper, deadline time.Time)
                              (stopTimer func(), didTimeout func() bool) {
    // 省略

    stopTimerCh := make(chan struct{})
    var once sync.Once
    stopTimer = func() { once.Do(func() { close(stopTimerCh) }) }

    timer := time.NewTimer(time.Until(deadline))
    var timedOut atomicBool

    go func() {
        select {
        case <-initialReqCancel:
            doCancel()
            timer.Stop()
        case <-timer.C:
            timedOut.setTrue()
            doCancel()
        case <-stopTimerCh:
            timer.Stop()
        }
    }()
```



```
    // 省略
}
```

2.3.2 utility package

単純な文字列処理をする httpguts パッケージを利用しています。httpguts は、内部ではなく外部の `golang_org/x/net/http/httpguts` ライブラリに分割しています。util パッケージのようなものは否定されることもありますが、標準ライブラリでも行われていることがわかります。

このような汎用的な処理を、無理にドメインに配置するのではなく切り出すのは正しいということです。ただしこのときに様々な機能を 1 つの utilily パッケージのようなものを作ることは避けるべきでしょう。

```
func (t *Transport) RoundTrip(req *Request) (*Response, error) {
    // 省略
        for k, vv := range req.Header {
            if !httpguts.ValidHeaderFieldName(k) {
                return nil, fmt.Errorf("net/http: invalid header field name %q", k)
            }
            for _, v := range vv {
                if !httpguts.ValidHeaderFieldValue(v) {
                    return nil, fmt.Errorf("net/http: invalid header
                                    field value %q for key %v", v, k)
                }
            }
        }
    // 省略
```

2.3.3 ループのリトライ処理

いくつかでてきたがリトライ処理が必要なものは無限ループで、成功した場合にだけ抜けるという書き方をしています。

```
    for {
        // 省略
        resp, err = pconn.alt.RoundTrip(req)
        if err == nil {
        // 成功すれば response を返すが、成功しなければ for ループにより再実行する
            return resp, nil
        }
```

2.3.4　1 つの関数内でのみ実行される関数の定義場所

handlePendingDial などその関数でしか使わない共通処理は、関数内部でクロージャとして定義されています。

```
func (t *Transport) getConn(treq *transportRequest, cm connectMethod)
                                                    (*persistConn, error) {
        handlePendingDial := func() {
        testHookPrePendingDial()
        go func() {
            if v := <-dialc; v.err == nil {
                t.putOrCloseIdleConn(v.pc)
            } else {
                t.decHostConnCount(cmKey)
            }
            testHookPostPendingDial()
        }()
    }

    // 省略
        case pc := <-idleConnCh:
        handlePendingDial()
        return pc, nil
    case <-req.Cancel:
        handlePendingDial()
        return nil, errRequestCanceledConn
    case <-req.Context().Done():
        handlePendingDial()
        return nil, req.Context().Err()
    case err := <-cancelc:
        handlePendingDial()
        if err == errRequestCanceled {
            err = errRequestCanceledConn
        }
}
```

2.3.5　1 度だけコピー

1 度だけコピーしたいというテクニックの紹介です。

```
func send(ireq *Request, rt RoundTripper, deadline time.Time)
                (resp *Response, didTimeout func() bool, err error)
    // 初期状態の ireq を保存
    req := ireq // req is either the original request, or a modified fork
    //省略

    // 1 度だけコピーする処理
    forkReq := func() {
        // もし初期状態と変わっていなければ新しい Request を作成してコピー
        if ireq == req {
            req = new(Request)
            *req = *ireq // shallow clone
        }
    }

    if req.Header == nil {
        // req を書き換える前にコピー
        forkReq()
        req.Header = make(Header)
    }
    // 省略
}
```


2.3.6　Hook 関数を用意したテスト

テスト用に Hook 関数が実行挿入している。普段は nop なので、何もしませんがテストのときだけ挿入するようになっています。

```
func nop() {}

// testHooks. Always non-nil.
var (
    testHookEnterRoundTrip   = nop
)

func (pc *persistConn) roundTrip(req *transportRequest)
                                (resp *Response, err error) {
    testHookEnterRoundTrip()
    if !pc.t.replaceReqCanceler(req.Request, pc.cancelRequest) {
        pc.t.putOrCloseIdleConn(pc)
        return nil, errRequestCanceled
    }
    pc.mu.Lock()
    pc.numExpectedResponses++
    headerFn := pc.mutateHeaderFunc
    pc.mu.Unlock()

    if headerFn != nil {
        headerFn(req.extraHeaders())
    }
```

以下では、RoundTrip 実行時に Cnacel の Request を送って正しくエラーが返ってくることを確認するテストしています。

```
    SetEnterRoundTripHook(func() {
        tr.CancelRequest(req)
```

```
    })
    defer SetEnterRoundTripHook(nil)
    res, err := tr.RoundTrip(req)
    if err != ExportErrRequestCanceled {
        t.Errorf("expected canceled request error; got %v", err)
        if err == nil {
            res.Body.Close()
        }
    }
```

2.3.7　デバッグ用コードやテスト用コードはベタがき

素直にベタがきをしています。さらに変数定義が関数内にあり、デバッグ時に true にして使うのだと思います。

正しさよりもシンプルな実装を心がけることについて、勇気を与えてくれるコードです。

```
func (pc *persistConn) roundTrip(req *transportRequest)
                          (resp *Response, err error) {

    // 省略
    const debugRoundTrip = false

    // 省略
    if debugRoundTrip {
            req.logf("writeErrCh resv: %T/%#v", err, err)
    }
}
```

2.4　おわりに

本章では、普段からよく利用する HTTP リクエストの送信処理の紹介とそこからの学びを紹介しました。中々この知識を業務で活かす機会は少ないかもしれませんが、何かのお役に立てば幸いです。

また学びについては、Go ではシンプルに書くことが正しいとされていますが、開発していると手の込んだ抽象化した設計を選定したくなりがちです。しかし抽象化のコードは本質的なロジックの見通しが悪くなることがあります。設計をする際には `if debugRoundTrip {` の愚直さをを思い出して、バランスを取った実装をしていきたいものです。

第 3 章

Go Modules 導入以前

3.1　はじめに

株式会社 Gunosy のバックエンド兼フロントエンドエンジニアの@timakin[*1]です。読者の皆様でも、もしかしたらご存知かもしれませんが、Go のバージョン 1.11 から、Modules という依存パッケージ管理機構が導入されました。`mod`というコマンドを通じてパッケージの管理ができます。もちろん、あくまで試験導入であり、正式に仕様が固まったという段階ではありません。

しかし、それを差し引いても、`glide`や `dep`などの既存パッケージ管理ツールを主に利用してきた人なら、コマンドそのもののインストールや、`go get`だけでは解決しない依存関係という、多少気になっても明確な解決策がなかった問題を、Modules を通じて解決することにメリットを感じることでしょう。本章では、そんな Modules について実戦投入を前提としつつ、いくつかの角度から掘り下げていきます。

3.2　Modules 以前

3.2.1　従来の依存パッケージ管理

Modules について書く前に、Modules の意義を知るためにも、従来の依存パッケージはどのように行われていたのかを振り返る必要があります。これまで、Go のプロジェクトで外部のパッケージへの依存関係を固定するためのツールはいくつも登場していました。

具体名を挙げるなら、非公式ではあるものの、有力な候補として活躍していたのは、`Godep`、`gb`、`glide`などでしょう。また、2016 年以降はほぼ公式のパッケージ管理ツールとして、`dep`が使われていました。`dep`が登場して以降は、`Godep`や `glide`は開発こそ継続

[*1] `https://twitter.com/__timakin__`

していたものの、depリリース直後から README で depへの移行を促す表明をしていました。depも、リリース当初は実験的な準公式ツールとしての色が強かったのですが、このように glideなどの過去の有力ツールたちが一斉に移行を促したことで、一気に depの導入が各所で行われました。

では、直近で準公式として活用されていた depについて見ていきましょう。depは直接の依存パッケージについてのマニフェストファイルである Gopkg.tomlと、直接の依存パッケージがさらに依存するものを深ぼって記録し、プロジェクトを完全に再現できるようにした Gopkg.lockというファイルの 2 つを生成します。設定ファイルに公式が toml 形式を使うあたりに好感が持てますね。それぞれリスト 3.1 とリスト 3.2 のような中身で、これを元にプロジェクトルートに vendor ディレクトリを生成します。

▼リスト 3.1　Gopkg.toml の例

```
[[constraint]]
  name = "github.com/fukata/golang-stats-api-handler"
  version = "1.0.0"

[[constraint]]
  name = "github.com/go-chi/chi"
  version = "3.3.2"

[[constraint]]
  name = "github.com/go-sql-driver/mysql"
  version = "1.3.0"
```

▼リスト 3.2　Gopkg.lock の例

```
[[projects]]
  digest = "1:289dd4d7abfb3ad2b5f728fbe9b1d5c1bf7d265a3eb9ef92869af1f7baba4c7a"
  name = "github.com/BurntSushi/toml"
  packages = ["."]
  pruneopts = ""
  revision = "b26d9c308763d68093482582cea63d69be07a0f0"
  version = "v0.3.0"

[[projects]]
  branch = "master"
  digest = "1:354e62d5acb9af138e13ec842f78a846d214a8d4a9f80e578698f1f1565e2ef8"
  name = "github.com/armon/go-metrics"
  packages = ["."]
  pruneopts = ""
  revision = "3c58d8115a78a6879e5df75ae900846768d36895"
```

この depは未だ現役で、今回紹介する modコマンドを使うのがためらわれる、という方はこちらを使えば現状ベストな形で対応できると思います。

3.2.2　従来のパッケージ管理で生じる課題

さて、従来のツールの名前をいくつか出してきましたが、これらのツールだといくつかの課題にぶつかる場面がありました。ツールが外出しされている点です。標準ツールではないため、Docker 利用時や CI 時にコマンドをインストールするための処理を記述しなければなりませんでした。go getコマンドをベースとして依存パッケージが管理されさえすればこの課題は乗り越えることができますが、現状では何らかの形で外部のツールとしてインストールし、CI のビルドを進める必要がありました。

さらに、dep以外のコマンドについても、特定パッケージのみのアップデートができなかったり、パッケージによってはそもそもダウンロードすることができないなどの不具合もありました。ダウンロード時にローカルのキャッシュが効きすぎてしまい、手元で工夫しないと意図したバージョンのパッケージがダウンロードできない、などのバグも発生しました。

3.3　Modules の登場

3.3.1　Go & Versioning

従来の各ツールがこのような問題を抱える中、2018 年初頭、Go & Versioning[*2]という一連の記事が公開されました。これは Russ Cox 氏による Go の標準的なバージョン管理方法に関する提案記事で、先日リリースされた Go1.11 の modの原型となるアイデアが記載されています。この記事群が公開された時点では、この Proposal の実装は vgoコマンドとして実装されており、go コマンドの配下ではなく別コマンドとして位置付けられていました。

主に modと共通している点としては、セマンティックバージョニング、git の tag による変更の適用や、GOPATH を意識しなくてもビルドが可能な点でしょうか。

特にセマンティックバージョニングについては、import を行うパッケージ利用者側だけではなく、パッケージ開発者も意識する必要があります。仮にとある開発者が v1.0.1 のパッケージを公開していたとして、後方互換性を伴うバグ修正をした場合は v1.0.2 にし、同じく後方互換性がありつつ、機能の修正を行う場合は v1.1.0 にします。さらに、後方互換性を伴わない、そのまま置き換えてしまうとバグを生じうる修正の場合は v2.0.0 にします。ここで、仮にパッケージがメジャーアップデートしていた場合、パッケージ開発者は git の tag をベースにバージョンを指定するとともに、後方互換性を伴わない機能

[*2] https://research.swtch.com/vgo

の変更が入ったパッケージに関しては、メジャーバージョンと同じ番号を持ったパス配下に置く必要があります。

例えば、v1 であれば/foo/barはそのままでいいですが、仮にメジャーアップデートが生じた場合、/v1/foo/barと/v2/foo/barを分けて開発する必要があります。Go のパッケージではなかなか大幅なメジャーアップデートを行うものは見かけませんが、例えば JavaScript のライブラリのように、メジャーバージョンをいきなり大幅に上げるような振る舞いは、あまりよろしくないものとなります。

特にメジャーバージョンを細かく刻んでアップデートするとさらに大変で、そのバージョンごとにパッケージを切って開発する必要が出てくるでしょう。このように、開発者は今までよりも明確にバージョンを意識しながら変更を加えていかなければなりません。

3.3.2 mod の誕生

vgoがそのまま Go1.11 に正式導入されると当初は考えられてましたが、正式リリースまで若干の変更が加えられ続け、modというコマンドで標準パッケージとして試験導入されました。vgoの最初の Proposal 時点からの差分としては、vendor ディレクトリを完全にスルーする機構ではなく、Go の API がトップレベルで vendor ディレクトリのサポートを行い、プロジェクトのローカルへのダウンロードコマンドを設けた点、go get -u=patchコマンドを通じて依存パッケージを最新に更新できるようになった点など、より自然な形で Go の標準パッケージとして組み込まれる流れになりました。

3.4 Modules 導入

3.4.1 Modules の利用方法

これまで Modules に至るまでの経緯や、Modules の概要を見てきましたが、本節では Modules を実際に利用してみた例を元に解説していこうと思います。まず、Modules の利用方法ですが、modコマンドとして実装されているのもあり、いくつかオプションがあります。Go1.11 をインストールし、ターミナル上で go mod helpと打つと、リスト 3.3 のような表示がされるかと思います。

▼リスト 3.3　go mod のオプション一覧

```
download    download modules to local cache
edit        edit go.mod from tools or scripts
graph       print module requirement graph
init        initialize new module in current directory
tidy        add missing and remove unused modules
vendor      make vendored copy of dependencies
verify      verify dependencies have expected content
why         explain why packages or modules are needed
```


中でも、go mod initや go mod tidyあたりは必ず使うでしょう。どうしても CI ビルドでキャッシュしたい場合などでは、go mod download を使ってローカルに vendor ディレクトリを作るというオプションの利用方法があります。

この modコマンド、試験導入なこともありまだ浸透していないかと思いきや、大きめの OSS プロジェクトでも利用していたりします。具体的には OSS になったソースコード探索サービス、Sourcegraph の中で使われていたりします。[*3]

こちらの事例は非常に参考になるとは思いますが、1 から進めていったときにどのような作業が必要なのか知るためにも、自分自身で Modules の機構に則りながら、API を書いてみました。当事例のパッケージ名を dratini と呼称します。こちらは実際に私の Github のプロジェクトとして開発しましたが、内容について詳しい言及はいたしません。

まず前提として、Modules は試験導入と再三書いた通り、GOPATH 配下だとデフォルトで有効ではありません。GO111MODULE=onという環境変数を設定する必要があります。GOPATH 配下ではない、全く関係のないディレクトリ配下だった場合は、このオプションはデフォルトで ON になり、常に buildや testの際に依存パッケージの存在確認を行って、なければ GO111MODULEオプションが有効になり、依存パッケージのインストールが行われます。

このオプションが有効になった状態で、まずは依存パッケージのマニフェストファイルを初期化するために、go mod initを実行します。すると、go.modファイルが生成されると思います。中身はシンプルに、リスト 3.4 のような状態になっているかと思います。

▼リスト 3.4　初期化時の go.mod

```
module github.com/timakin/dratini
```

開発を進めていき、このマニフェストの内容以外にも依存パッケージの設定を追加しなければならない場合は、go mod tidyを実行します。depに慣れしたんでいた方は dep ensureを思い浮かべるとわかりやすいでしょう。tidyは依存パッケージの追記や、使っていない依存関係を削除する役割を担います。

実行すると、go.modファイルへの依存パッケージ情報の追記と同時に、go.sumというファイルが新しく作成され、依存パッケージの細かい checksum 情報が記載され、再現性が取れる形でどの時点の該当パッケージをインストールすればいいか出力します。具体的

[*3] https://github.com/sourcegraph/sourcegraph

にはリスト 3.5 とリスト 3.6 のようなファイルになります。このファイルをコミットして、別の人がそのプロジェクトで依存パッケージをインストールしたい場合は、go mod downloadなどをしてローカルに vendor ディレクトリを作りつつパッケージを持ってくるか、go buildや go testを実行する際に GO111MODULE=onを指定して

▼リスト 3.5　tidy 実行後の go.mod

```
module github.com/timakin/dratini

require (
        github.com/BurntSushi/toml v0.3.0
        github.com/RobotsAndPencils/buford v0.12.0
        github.com/client9/reopen v1.0.0
        github.com/davecgh/go-spew v1.1.1 // indirect
        github.com/pkg/errors v0.8.0
        github.com/pmezard/go-difflib v1.0.0 // indirect
        github.com/stretchr/testify v1.2.2
        go.uber.org/atomic v1.3.2 // indirect
        go.uber.org/multierr v1.1.0 // indirect
        go.uber.org/zap v1.9.1
        golang.org/x/crypto v0.0.0-20180910181607-0e37d006457b // indirect
        golang.org/x/net v0.0.0-20180906233101-161cd47e91fd
        golang.org/x/text v0.3.0 // indirect
)
```

▼リスト 3.6　go.sum の出力結果

```
github.com/BurntSushi/toml v0.3.0 h1:e1/Ivsx3ZOFVTV0NSOv/aVgbUWyQuzj7=
github.com/BurntSushi/toml v0.3.0/go.mod h1:xHWCNGjB5oqiDr8zfno3MHue2Ht5sIBk=
github.com/RobotsAndPencils/buford v0.12.0 h1:2nfOk+N/QVoQHwXIS0m5TFdvlUjEnqAj=
github.com/RobotsAndPencils/buford v0.12.0/go.mod h1:27KhJZ/wLQHRnsZF+mTWKvF5w8U=
github.com/client9/reopen v1.0.0 h1:8tpLVR74DLpLObrn2KvsyxJY++2iORGR=
github.com/client9/reopen v1.0.0/go.mod h1:caXVCEr+lUtoN1FlsRiOWdfQtdRHIYfc=
github.com/davecgh/go-spew v1.1.1 h1:vj9j/u1bqnvCEfJOwUhtlOARqs3+rkHY=
github.com/davecgh/go-spew v1.1.1/go.mod h1:J7Y8YcW2NihsgmVo/mv3lAwl/skON4iL=
github.com/pkg/errors v0.8.0 h1:WdK/asTD0HN+q6hsWO3/vpuAkAr+tw6a=
github.com/pkg/errors v0.8.0/go.mod h1:bwawxfHBFNV+L2hUp1rHADufV3IMtnDR=
github.com/pmezard/go-difflib v1.0.0 h1:4DBwDE0NGyQoBHbLQYPwSUPoCMWR5BEz=
github.com/pmezard/go-difflib v1.0.0/go.mod h1:iKH77koFhYxTK1pcRnkKkqfTogsbg7gZ=
github.com/stretchr/testify v1.2.2 h1:bSDNvY7ZPG5RlJ8otE/7V6gMiyenm9Rt=
github.com/stretchr/testify v1.2.2/go.mod h1:a8OnRcib4nhh0OaRAV+Yts87kKdq0PP7=
go.uber.org/atomic v1.3.2 h1:2Oa65PReHzfn29GpvgsYwloV9AVFHPDk=
```

3.4.2　Docker イメージの作成や CI 環境について

ここまでの流れを見ただけでは、Modules の機構のメリットはただコマンドが標準パッケージに統合されただけに感じるかもしれません。しかし、Docker コンテナ上でのバイナリビルドなど、依存パッケージを基にした buildコマンドの実行時には、Modules のメリットを強く実感できます。従来のバージョンでは、Go がデフォルトで入った Docker イメージであれば go getで、そうでなければ curlや apt-getでコマンドを入れる必要があり、ビルドステップとしては本来不要なものが混ざってきます。

Modules を使ったイメージ作成を実現する場合、リスト 3.7 のように、特に明示的なツールのインストールをすることなく、ビルドが可能になります。ここで特徴的なのが、WORKDIRを~/github.com/timakin/dratiniと、GOPATHを避けた形で設定している点です。こうすることで、デフォルトで GO111MODULEフラグが有効になるので、ビルドステップの中で自動で依存パッケージがインストールされます。

▼リスト 3.7　Modules 依存の Dockerfile の例

```
ROM golang:1.11.0 AS builder
ADD . ~/github.com/timakin/dratini
WORKDIR ~/github.com/timakin/dratini
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -o app

FROM alpine:latest as runner
COPY --from=builder ~/github.com/timakin/dratini/app /usr/local/bin/app
```

また、Docker ビルドを活用する場面で有力な候補としては、CircleCI 2.0 でのビルドでしょうか。その場合、ソースを Github からチェックアウトしてきてテストを実行するまでに必要な設定は、リスト 3.8 のようなものだけです。

▼リスト 3.8　Modules 依存の CircleCI のジョブ実行用 config ファイル

```
version: 2
jobs:
  build_and_test:
    working_directory: ~/github.com/timakin/dratini
    docker:
      - image: circleci/golang:1.11
    steps:
      - checkout
      - run: go test -race
```

非常に短くシンプルに書けていると思います。もちろんこれは GOPATH配下を避けてチェックアウトしてるから実現するステップですが、フラグを有効にしさえすればここまでシンプルに、Go 自体が勝手に依存パッケージをインストールしてくれるというのは、見てて気持ちの良いものです。さらに、go getでインストールできるソースであれば問題なく入るので、従来の他のツールで git tagやプライベートリポジトリ関係のバグが発生し、パッケージがインストールできないなどの不具合もありません。

3.5　おわりに

本章では、Go1.11 で新たに導入された Modules を取り巻く背景や、その利点、実際の使用例を見てきました。外部ツールではなく標準搭載のツールとして `mod`コマンドが誕生したことで、ビルドステップが非常に簡潔に記述できることがおわかりいただけたと思います。実際の OSS プロジェクトでも利用され始めているので、是非この機会に Modules にキャッチアップし、ご自身のプロジェクトに導入してみて欲しいと思います。

第 4 章

iota の使い方

4.1　はじめに

株式会社エウレカ[*1]で Pairs[*2]の開発や、エンジニア組織のマネジメントをしている kaneshin[*3]です。大学の専攻や前職では C 言語がメインだったため、C 言語での開発が一番得意なのと、数学についても大好きです。Go 言語では主に、CLI ツールや API サーバーを開発することが多いです。

本章では、Go 言語の定数宣言内の識別子である `iota`についての説明と、ビット演算と組み合わせて活用する解説をします。

4.2　iota とは

`iota`とは、ギリシャ文字の 9 番目のιにあたり、発音は『ァイオゥタ』です。日本人の多くは、イオタと発音する人が多いと思います。発音の『ァイオゥタ』の小文字で表記している発音が、発音リダクションによって脱落しているからです。

`iota`は**型なしの連続する整数定数**を暗黙的に展開する列挙子です。C 言語の `enum`のような列挙型ではなく、`const`の宣言グループ内で使用し、セットされた `iota`へ値が `0`からはじまる連続整数が展開されます。

▼リスト 4.1　iota の連続した整数の展開

[*1] `https://eure.jp/`
[*2] `https://www.pairs.lv/`
[*3] `https://twitter.com/kaneshin0120`

```
const (
  GolangTokyo  = iota // 0
  GoConference = iota // 1
  GopherCon    = iota // 2
)
```

iotaは新しく constのグループであれば、0から新しく展開されますし、同じ constグループ内であれば、iotaを省略した場合、コンパイル時に暗黙的にコードの文脈で補完されて展開されます。

▼リスト 4.2　const を分けると iota は 0 から展開

```
// 1st const group
const (
  GolangTokyo  = iota // 0
  GoConference = iota // 1
  GopherCon    = iota // 2
)

// 2nd const group
const (
  DroidKaigi = iota // 0
  TrySwift          // 1
  PyCon             // 2
)
```

4.2.1　ConstSpec 評価でインクリメント

iotaは 0からはじまる連続整数の値で展開されますが、インクリメントされるタイミングはコンパイル時に ConstSpecが評価されるタイミングでインクリメントを行っています。そのため、識別子と式のリストである IdentifierList, ExpressionListが同一の ConstSpecに存在する場合、同一の整数値が割り当てられます。

簡単に言えば、この ConstSpecは constをコンパイル時に一行ずつ解釈するものです。つまり、一行ずつ整数値が定義されることになります。

▼リスト 4.3　一行ずつ iota は展開

```
const (
  Foo1, Foo2 = iota, 2 * iota // 0, 0 (iota == 0)
  Bar1, Bar2                  // 1, 2 (iota == 1)
  Qux1, Qux2                  // 2, 4 (iota == 2)
)
```


4.2.2　ブランク識別子でスキップ

ブランク識別子とは、_（アンダースコア）を使用して表現します。Go の標準パッケージにて、database/sql/driverを満たした Driver をアンダースコアをつけてインポートした経験がある人もいると思います。

▼リスト 4.4　blank import の例

```
import (
  "database/sql"
  _ "github.com/go-sql-driver/mysql"
)
```

このブランク識別子を constグループ内で使用すると、ConstSpecとして解釈されますが、宣言において IdentiferListと ExpressionListへバインドはしません。そのため、コンパイル時に ConstSpecの解釈で iotaのインクリメントが実行されることになっています。

▼リスト 4.5　ブランク識別子でスキップ

```
const (
  Foo1, Foo2 = iota, 2 * iota // 0, 0 (iota == 0)
  _, _                        // _, _ (iota == 1)
  Qux1, Qux2                  // 2, 4 (iota == 2)
)
```

ConstSpec, IdentifierList, ExpressionList

本章の、constの話とは直接関係ありませんが、ConstSpec, IdentifierList, ExpressionListは、コンパイラが事前宣言における解釈をするためのリストです。コンパイラの方で typeや constの宣言を判断し、対応する TypeSpecや ConstSpecに適切にアサインしています。

▼リスト 4.6　ConstSpec, Identifier, Expression のリスト

```
ConstDecl      = "const" ( ConstSpec | "(" { ConstSpec ";" } ")" ) .
ConstSpec      = IdentifierList [ [ Type ] "=" ExpressionList ] .

IdentifierList = identifier { "," identifier } .
ExpressionList = Expression { "," Expression } .
```

IdentifierList, ExpressionListは、それぞれ「識別子」と「式」のリストになっており、カンマ区切りのものをバインドします。[*4]

4.3 iota の活用例

本家の iotaのページ[*5]に載っている例で解説します。

▼リスト 4.7　バイトサイズを iota で定義

```
type ByteSize float64

const (
  _           = iota // ignore first value by assigning to blank identifier
  KB ByteSize = 1 << (10 * iota)
  MB
  GB
  TB
  PB
  EB
  ZB
  YB
)
```

ブランク識別子をつかって最初の値はスキップするように実装しています。ブランク識別子を使わないでも実装することは可能です。

▼リスト 4.8　バイトサイズを iota で定義（ブランク識別子なし）

```
const (
  KB ByteSize = 1 << (10 * (iota + 1)) // 1 << 10
  MB                                   // 1 << 20
  GB                                   // 1 << 30
  // ...
)
```

リスト 4.8 では、1 << (10 * (iota + 1))と記述する必要がありますが、1 << (10 * iota + 1)と記述してしまったら期待動作通りにはなりません。演算子の評価の優先順位を踏まえれば、当たり前の挙動ですが、不具合の温床となってしまいます。このような観点で、ブランク識別子を有効活用すると不具合を発生させないコードになります。

[*4] https://golang.org/ref/spec#Constant_declarations:title
[*5] https://github.com/golang/go/wiki/Iota:title

4.3.1 数値に意味がある定数への使用

連続整数として定数に数値を展開することが iotaの機能であることが理解できたと思います。そのため、連続する数値に対して活用できると思いますが、数値として意味を持ってしまっている定数へは使用しないことが望ましいです。例えば、データーベースに入っている連続整数のマスターレコードの値を定数として定義する場合です。

▼リスト 4.9　データベースのレコード

```
mysql> SELECT * FROM conference_type;
+----+--------------------+
| id | conference_type    |
+----+--------------------+
|  1 | public_conference  |
|  2 | private_conference |
|  3 | other_conference   |
+----+--------------------+
```

これを Go のコードで表現するために iotaで設定するのは、あまり良い方法とは言えません。データーベースに数値が設定されているということは、既に、その数値は「データーベースに存在する値」として強い意味を持っていることになります。Go の定数の順番を間違えて変更してしまった場合、データーベースの値と違うものになってしまいます。

▼リスト 4.10　データーベースの値を定数で設定

```
type ConferenceType int

// 好ましい例
const (
  PublicConference  ConferenceType = 1
  PrivateConference ConferenceType = 2
  OtherConference   ConferenceType = 3
)

// 好ましくない例
const (
  PublicConference  ConferenceType = iota + 1 // 1
  PrivateConference                           // 2
  OtherConference                             // 3
)
```

連続整数だからと、すべてに iotaを使わない設計を考えることも重要です。

4.3.2 ビット演算との組み合わせ

さて、数値に意味がないものに対しては積極的に iotaを使っていきます。例えば、Pairs のユーザーの状態を管理をするために、ビット処理と組み合わせたとすれば、iota を使った定数は相性が良いものの一つとなります。

▼リスト 4.11　ビット演算と iota

```
package main

import (
  "fmt"
)

type RelationshipStatus int

const (
  LikeFromMe       RelationshipStatus = 1 << iota // 0001 (1 << 0)
  LikeFromPartner                                 // 0010 (1 << 1)
  BlockFromMe                                     // 0100 (1 << 2)
  BlockFromPartner                                // 1000 (1 << 3)
)

const (
  MutualLike = LikeFromMe | LikeFromPartner   // 0011
  Blocking   = BlockFromMe | BlockFromPartner // 1100
)

func (st RelationshipStatus) IsMatching() bool {
  // お互いのいいね！ 判定
  // MutualLike のビットが立っているかを AND 演算で判定
  if st&MutualLike != MutualLike {
    return false
  }

  // ブロックしている場合はマッチングにしない判定（案 1、2 どちらでも）
  // 案 1: Blocking のビットが立っているかを AND 演算で判定
  if st&Blocking != 0 {
    return false
  }
  // 案 2: お互いのいいね！ を AND NOT 演算でクリアして判定
  if st&^MutualLike != 0 {
    return false
  }

  return true
}
```

読みやすいようにビットを記載していますが、このコードを実行する場合、定数に入っている数値自体には意味がありません。昨今、ビット演算を使うことは少ないですが、覚えていて損は無いテクニックです。

4.4 おわりに

本章では、何気なく使う iotaについて、ConstSpecのコンパイル解釈を含めて解説しました。ConstSpecのように、コンパイル時の解釈を踏まえておくことによって、iota への深い理解へと繋がります。C 言語や Swift の enumのようなことはできませんが、Go 言語ならではの使い方として、使用していってもらえればと思います。

第 5 章

go test のマニアックなオプション大全

メルペイのバックエンドエンジニアの@knsh14[*1]です。

Go ではテストツールとして go testというコマンドが標準ツールとして提供されています。go testはテストを実行して動作を検証するだけでなく、ベンチマークを計測したり、ドキュメント用に Example を作成したりすることができます。

本章では go testで使うことのできるフラグの役割を解説し、実行にどのような影響を与えるか解説します。特に普段使う場面の少ないであろうフラグに絞って説明をしたいと思います。testing パッケージの使い方などの解説はしません。そちらについては公式のドキュメントや書籍などを参照してください。なお、対象にする Go のバージョンは 2018 年 10 月 1 日時点で最新の 1.11 とします。

5.1 go test のフラグ一覧

go testで使うことのできるフラグを見るためには go test -hを使います。ブラウザで見たい場合には GitHub に go test -hで出力されるものと同じもの[*2]があるので最新のものを見ることができます。

go testのフラグにはいくつか種類があります。

1. -jsonなどの go testそのものによって解釈されるもの
2. -runや-vなど go testによって実行されるテストが影響をうけるもの

[*1] https://twitter.com/knsh14

[*2] https://github.com/golang/go/blob/release-branch.go1.11/src/cmd/go/internal/test/test.go

これらは go test -hで見た場合には特に区別されていませんが、ソースコードを読んでみるとそれぞれ別の変数で定義されていることがわかります。[*3][*4] 本章では便宜上カテゴリ 1、カテゴリ 2 という呼び方で両者を区別します。

5.2 カテゴリ 1 go test そのものによって解釈されるもの

この種類に分類されるフラグは以下のものがあります。

- -args
- -c
- -exec xprog
- -i
- -json
- -o file

5.2.1 args

go test自体にフラグを渡すのではなく実行するテストコードにフラグを渡すことができます。-args以降のフラグは全て渡されます。

少し無理やりですがリスト 5.1 は argsフラグを使うことを前提としたコードです。flagパッケージで引数を処理し、IsMultiple関数の動作を変更できるようになっています。

▼リスト 5.1 コマンドライン引数を取るテストコード

```
package main

import (
  "flag"
  "testing"
)

var (
  num int
)

func init() {
  flag.IntVar(&num, "num", 1, "")
}
```

[*3] https://github.com/golang/go/blob/release-branch.go1.11/src/cmd/go/internal/test/test.go#L136

[*4] https://github.com/golang/go/blob/release-branch.go1.11/src/cmd/go/internal/test/test.go#L197

```
func IsMultiple(n, m int) bool {
  return n%m == 0
}

func TestSampleOdd(t *testing.T) {
  flag.Parse()
  cases := []struct {
    input int
  }{
    {input: 1},
    {input: 3},
    {input: 5},
    {input: 7},
  }
  for _, c := range cases {
    if IsMultiple(c.input, num) {
      t.Log(c.input)
    } else {
      t.Errorf("%d is not multiple of %d", c.input, num)
    }
  }
}
```

リスト 5.1 を実行する際に go test -v -args -num 3と実行するとリスト 5.2 のような出力になります。

▼リスト 5.2　実行結果

```
knsh14% go test -v -args -num 3
=== RUN   TestSampleOdd
--- PASS: TestSampleOdd (0.00s)
    sample_test.go:34: 1 is not multiple of 3
    sample_test.go:32: 3
    sample_test.go:34: 5 is not multiple of 3
    sample_test.go:34: 7 is not multiple of 3
PASS
ok      github.com/knsh14/sample        0.007s
```

実践での使い所としては例のように、実行時に挙動を変えたいなどがありそうです。DB の接続先や、タイムアウトの調整などで役に立ちそうです。

5.2.2　c

このフラグをつけて go testを実行すると、テスト関数を実行するのではなく、テストを実行するバイナリを生成します。バイナリ名は何も指定しないとパッケージ名.testになります。バイナリの名前は後述する-oオプションで指定することもできます。go buildのテスト版だと考えるとイメージが付きやすいです。

このフラグが有効な場合、複数パッケージを同時に扱うことはできません。生成されたバイナリはカテゴリ 2 のフラグを利用することができます。

ただし、バイナリにカテゴリ 2 のフラグを渡す際にはそのまま渡すことができません。-runオプションなら-test.runと変更する必要があります。

使い所としてはテストを作成して実行する前に使います。事前にコンパイルしておいて最低限 build failed で失敗しないように対策することで安心して実行することができます。

5.2.3　exec xprog

go testで実行されるバイナリ対して外部コマンドを実行します。例を挙げると go test -exec "go tool test2json"のように使います。

-cや-oフラグと同時に使うことはできません。

使い所がなんとも難しいフラグです。-cフラグと使えるとハッシュが取得できてありがたいなどがあるかもしれません。しかしそれもできないので、現状だと go test -exec "go tool test2json"くらいしか使いみちがなさそうに見えます。

5.2.4　i

ドキュメントを見ると「テストが依存しているパッケージをインストールします。テストは実行されません。」とあります。が実際には何がされているか全くわかりません。

5.2.5　json

標準出力に出される結果を JSON 形式にしてテストを実行します。この際に、カテゴリ 2 の-vオプションも同時に付与され、詳細なログが出力されます。JSON の出力形式についてより詳しく知りたい場合には go doc test2jsonからドキュメントを見ることができます。[*5]

詳細なログはいらないという場合には-execフラグを使って go test -exec "go tool test2json"とします。

CI などでテストを実行した結果を jqなどに食わせて処理をしたい場合に便利かもしれません。

[*5] https://golang.org/cmd/test2json/

5.2.6 o

テストバイナリを渡された名前で生成します。これだけ聞くと-cと大きな差がないように見えます。しかし、このオプションだけをつけて実行した場合はテストが実行された上でバイナリが生成されます。

このフラグだけで使うよりも、-cオプションと組み合わせて使う場面のほうが多そうに見えます。

5.3 カテゴリ 2 go test によって実行されるテストが影響をうけるもの

このカテゴリに分類されるフラグは以下のものがあります。

- -bench regexp
- -benchtime t
- -count n
- -cover
- -covermode set,count,atomic
- -coverpkg pattern1,pattern2,pattern3
- -cpu 1,2,4
- -failfast
- -list regexp
- -parallel n
- -run regexp
- -short
- -timeout d
- -v
- -vet list

こちらに分類されるフラグは使っている人も多いのではないでしょうか。
さらに、プロファイル用のフラグとして、以下のフラグが用意されています。

- -benchmem
- -blockprofile block.out
- -blockprofilerate n
- -coverprofile cover.out

- -cpuprofile cpu.out
- -memprofile mem.out
- -memprofilerate n
- -mutexprofile mutex.out
- -mutexprofilefraction n
- -outputdir directory
- -trace trace.out

この中から以下のフラグについて解説しようと思います。

- -failfast
- -timeout d

5.3.1 failfast

このフラグを付けて実行した場合、テストが最初に失敗した段階で残りの項目を全てスキップします。テスト関数内で testing.T.Run関数でサブテストを実行している場合も、サブテストの一つが失敗した段階で終了します。

PR を出す前などに最終チェックをする場合や、CI で実行時間を短くしたい場合などに非常に便利です。

5.3.2 timeout

go testの実行自体のタイムアウトを設定します。これを過ぎた場合、panic を起こしてテストを強制終了します。

フラグを付けた時点でタイムアウトが有効になります。デフォルトは 10 分で設定されています。フラグを付けた場合 0 に設定するとタイムアウトを無効にできます。

5.4 最後に

本章では普段なかなか使わない go testのフラグを解説しました。普段使わないだけあって使い所に困るものもありましたが、調べているうちに便利なものも発見できました。

テストツールに何ができるか知っておくと、「ここはテストツールの機能でカバーしよう」といったより柔軟な選択肢を取ることができます。これをきっかけに他のツールも調べてみようと思っていただければ幸いです。

第 6 章

go-cloud と Wire コマンドの概要と活用方法

6.1　はじめに

freee 株式会社[*1]でバックエンドエンジニアをしている budougumi0617[*2]です。普段は主に Go と gRPC によるマイクロサービスを開発しています。

本章では 7 月に発表された `go-cloud`[*3]とそれに付属する Wire コマンドについて次の情報をまとめています。

- go-cloud の基本的な概要
- Wire の基本的な概要
- Wire を使った Dependency Inejection（依存性注入）の利用方法
- Wire を使った Dependency Inejection（依存性注入）の提供方法

6.1.1　注意

本稿は 2018/09/23 時点の go-cloud リポジトリを参考に記述されています。go-cloud は現時点で Alpha リリース (v0.2.0) であるため、今後本稿の情報には差異が生じる可能性があります。また、本章で引用している go-cloud のコード、記載しているコードは全て Apache License 2.0 です。

[*1] `https://corp.freee.co.jp`
[*2] `https://twitter.com/budougumi0617`
[*3] `https://github.com/google/go-cloud`

6.2　go-cloud の概要

go-cloud は Google が公開したポータブルなクラウドアプリケーションを開発するための Go 用の API ライブラリです。2018/07/24 に Go blog で"Portable Cloud Programming with Go Cloud"[*4]というタイトルでリリースアナウンスされました。

2018/09/23 時点で GitHub に公開されている go-cloud のリポジトリには次の内容が含まれています。

- BLOB ストレージのジェネリックな API 定義
- GCP、AWS の BLOB ストレージサービス、MySQL サービスをラップした実装
- ロギングやヘルスチェックなどの基本的な Web サーバの構成に必要な実装
- Dependency Injection[*5]を行なうための Wire コマンド

ジェネリックな API 定義とは、誤解を恐れずにいうと MySQL, postgreSQL などに対する `database/sql`パッケージに相当するようなものです。また、より簡単にクラウドサービスへの依存関係をコードで表現できるように、Wire コマンドを使った Dependency Injection(DI、依存性の注入) の仕組みが提供されています。

6.2.1　go-cloud の対応クラウドプロバイダー

各クラウドプロバイダー用の実装は 2018/09/23 時点で Google Cloud Platform(GCP)、Amazon Web Services(AWS) のみです。クラウドプロバイダーではありませんが、ローカル環境へのアクセスを想定した実装[*6]は同梱されています。GCP、AWS 別ベンダーのサポート計画などはとくに公表されていません。Azure のサポートについての issue[*7]などは立ち上がっています。

6.3　go-cloud がなぜ必要なのか？

クラウドプロバイダーを利用した Web アプリケーション開発が一般的になった今日ですが、kubernetes(k8s) や Docker を利用したコンテナ技術も身近になってきました。コンテナ技術を活用することで Web アプリケーションのポータビリティ性は高まっていると言えます。しかし、実際に Web アプリケーションは本当にマルチクラウドで利用でき

[*4] `https://blog.golang.org/go-cloud`
[*5] `https://en.wikipedia.org/wiki/Dependency_injection`
[*6] `https://github.com/google/go-cloud/tree/master/samples/guestbook/localdb`
[*7] `https://github.com/google/go-cloud/issues/76`

るようになっているのでしょうか？ AWS 上で動いているコンテナのアプリケーションロジックはそのまま GCP 上で可動できるのでしょうか？ 多くの場合、アプリケーションはクラウドプロバイダーの SDK やマネージド・サービスの API に大きく依存しており、アプリを k8s に載せたとしてもベンダー依存からは脱却できていません。クラウドプロバイダーを乗り換える、あるいはマルチクラウド対応を行う際はベンダーサービスへの依存部分を再実装する必要があります。このような現状で、go-cloud は Go のアプリケーションにクラウドサービスへのジェネリックな API を定義し、クラウドプロバイダーに依存しない開発を行うために公開されました。

例えば、go-cloud に含まれている `samples/tutorial/main.go`(リスト 6.1) を見てみましょう。このサンプルは指定されたファイル名（`file`）のファイルをコマンドオプションで指定された AWS/GCP いずれかの BLOB ストレージにアップロードするコードです。

▼リスト 6.1　go-cloud/samples/tutorial/main.go から抜粋したファイルアップロード処理

```
ctx := context.Background()
// Open a connection to the bucket.
b, err := setupBucket(context.Background(), *cloud, *bucketName)

// Prepare the file for upload.
data, _ := ioutil.ReadFile(file)

w, _ := b.NewWriter(ctx, file, nil)
w.Write(data)
w.Close()
```

`go-cloud/blob`パッケージに含まれる `blob.Bucket`の API 定義に基づいてファイルアップロードのロジックが書かれています。`setupBucket()`メソッド内で `go-cloud/blob/s3blob`と `go-cloud/blob/gcsblob`のどちらかを生成する処理がありますが、`main.go`の実装にクラウドプロバイダーに依存した処理はありません。`blob.Bucket`オブジェクトの初期化処理を隠蔽すれば、コーディングを行う開発者はクラウドプロバイダーの SDK を考慮せずに開発することができます。それでは、次節から go-cloud リポジトリの中を確認していきます。

6.4　go-cloud リポジトリの内容

go-cloud リポジトリには大きく分けて表 6.1 の内容が含まれています。

まず go-cloud リポジトリの主目的であるクラウドサービスに対するジェネリックな API 定義が含まれています。また、API 定義に対する各クラウドプロバイダー用の実装には「6.8 Wire の概要」から説明する `go-cloud/wire`パッケージと Wire コマンドを利

▼表 6.1　go-cloud リポジトリの内容

分類	内容
メインコード	クラウドサービスのジェネリックな API 定義と AWS/GCP に対応した実装
`wire`パッケージ	Wire コマンドを利用した DI の仕組み
`samples`パッケージ	サンプルコード

用した DI を行うための設計が施されています。サンプルは 3 種類あり、それぞれの概要は表 6.2 のとおりです。

▼表 6.2　go-cloud/samples に用意されているコンテンツ内容

サンプル名	内容
tutorial	`go-cloud/blob`パッケージを使った CLI ツール
wire	Wire コマンドを使った DI のサンプル
guestbook	AWS/GCP 上で実行できる Wire を駆使した Web サーバのサンプル

では go-cloud で定義されたジェネリックな API を確認していきましょう。といっても、2018/09/23 現在、go-cloud で定義されたクラウドサービスのジェネリックな API 定義は BLOB ストレージ用しかありません[*8]。

6.5　go-cloud/blob パッケージ

`go-cloud/blob`パッケージはジェネリックな API として `blob.Bucket`を提供しています。`go-cloud/blob`パッケージのサブパッケージにある `OpenBucket`関数を用いると、各クラウドプロバイダー用の設定がされた `blob.Bucket`オブジェクトを生成することが出来ます。

▼リスト 6.2　go-cloud/blob/s3blob/s3blob.go

```
func OpenBucket(ctx context.Context,
                sess client.ConfigProvider,
                bucketName string
                ) (*blob.Bucket, error)
```

▼リスト 6.3　go-cloud/blob/gcsblob/gcsblob.go

[*8] AWS の RDS, GCP Cloud SQL のラッパーもありますが、go-cloud 独自の API ではなく `database/sql.DB`オブジェクトを生成するビルド関数です。

```
func OpenBucket(ctx context.Context,
                bucketName string,
                client *gcp.HTTPClient
               ) (*blob.Bucket, error)
```

6.6 go-cloud/mysql/{cloudmysql, rdsmysql}パッケージ

go-cloud v0.2.0 時点では BLOB ストレージへの API 定義以外に独自定義された API 定義はありません。ただし、「6.8 Wire の概要」で説明する Wire コマンドによる DI を考慮した設計の SQL サービスの実装は用意されています。

▼リスト 6.4　go-cloud/mysql/cloudmysql.Open 関数

```
// Open opens a Cloud SQL database.
func Open(ctx context.Context, certSource proxy.CertSource,
          params *Params) (*sql.DB, error)
```

▼リスト 6.5　mysql/rdsmysql/rdsmysql.Open 関数

```
// Open opens an encrypted connection to an RDS MySQL database.
func Open(ctx context.Context, provider CertPoolProvider,
          params *Params) (*sql.DB, func(), error)
```

その他に、Stackdriver へのコネクタなどが用意されています。こちらも Wire コマンドを使った DI の仕組みの上に実装されています。

6.7 go-cloud の何がよいのか？

と、ここまで go-cloud の概要をざっくり紹介してきましたが、「別に外部サービスに依存している部分のパッケージ外出しや抽象化はすでにしてあるし」という方も多いと思います。先に断っておくと、現時点の go-cloud を利用するだけでバイナリからベンダー依存のコードを完全に削除できるわけではありません[*9]。まだ全ての API 定義が揃っているわけでもありません。

では、今 go-cloud を利用するメリットはなんでしょうか？ デファクトになりえそうな Google によって定義されたクラウドサービスの API を使う、という意味以外にも

[*9] go-cloud パッケージが内部で各クラウドプロバイダーの SDK パッケージを参照している

go-cloud を使う意味はあります。それは go-cloud リポジトリにある Wire コマンドの存在と、Wire コマンドによって実現する依存関係の自動解決です。

6.8 Wire の概要

Wire コマンドは go-cloud リポジトリに付属されているコマンドラインツールです。
https://github.com/google/go-cloud/tree/master/wire

Wire コマンドは `go-cloud/wire`パッケージをつかって定義された Dependency Injection(DI、依存性の注入) を解決するコードを自動生成します。go-cloud 配下のパッケージは `go-cloud/wire`パッケージを使って依存関係を定義しています。

Wire を使った実装を行うと、私たちは以下の恩恵を受けることができます。

- 依存関係を疎にするコードを強制できる
- 複雑な依存関係も Wire が自動的に判別し DI コードを自動生成してくれる
- Wire を使って定義された依存関係を組み合わせて、新しい依存関係も定義できる

各クラウドプロバイダーは BLOB ストレージや VM インスタンスという類似サービスを提供していますが、各社の SDK でそれらを利用する際に必要になる情報や設定名は各々で異なります。それとは別に各サービス用のオブジェクト群の初期化時にクラウドサービスへの接続設定やロギング設定を引き回したいというニーズもあります。大抵はオブジェクトごとに初期化を行うか、グローバル変数に置くことが多いのではないでしょうか。これらの問題を効率的に解決するのが Wire コマンド (パッケージ)[*10]です。

Wire を利用した実装ではコンポーネント間の依存関係が関数のパラメータとして現れるので、グローバル変数を使った場合などとは異なり明示的な初期化を表現できます。また、Wire は実行時の状態やリフレクションなしで動作するため、Wire 利用を想定して実装されたコードは Wire 未使用時でも利用しやすいです。具体的には、後述する Provider と Injector という概念に対応した実装を行なうことで、依存関係を解決するコードを Wire コマンドで自動生成することが可能になります。ちなみに `go-cloud/wire`パッケージ・Wire コマンドは go-cloud の他パッケージと独立しているので、go-cloud を使わずに Wire による DI の仕組みだけを別のパッケージで活用することも可能です。

6.8.1 Provider と Injector

`go-cloud/wire`パッケージが提供する DI の仕組みには **Provider** と **Injector** という概念があります。依存関係を定義する Provider と依存関係を注入する Injector です。

[*10] `https://github.com/google/go-cloud/blob/master/wire/README.md`

Provider は依存関係の解決が必要なコンポーネントの集合 (wire.ProviderSet) です。wire.Set関数を使って宣言します。リスト 6.6 は GCP の依存解決を行う Provider です。

▼リスト 6.6　go-cloud/gcp/gcpcloud/gcpcloud.go に定義されている Provider

```
var GCP = wire.NewSet(Services, gcp.DefaultIdentity)

var Services = wire.NewSet(
  cloudmysql.CertSourceSet,
  cloudmysql.Open,
  gcp.DefaultTransport,
  gcp.NewHTTPClient,
  runtimeconfigurator.Set,
  sdserver.Set)
```

Servicesは Cloud SQL 用の認証情報（CertSourceSet）、Cloud SQL への接続（Open）、Stackdriver に向けた Logger（sdserver.Set）の依存関係が定義されてる Provider です。GCPは Servicesの依存関係に加え、デフォルトの Project ID などを取得する gcp.DefaultIdentityも含めた依存関係の Provider です。

Injector は Provider を組み合わせて依存関係を注入するためのコードです。wire.Build関数を使って宣言します。リスト 6.7 は Provider で紹介した依存関係 GCPなどを組み合わせて依存関係を注入した*applicationオブジェクトを返す Injector の宣言です。Wire コマンドはこの Injector mの宣言から DI を行うコードを自動生成します。Injector の宣言自体は自動生成を行うためだけのものなので、ビルドタグ[*11]をつけてビルドには含めないようにします。

▼リスト 6.7　go-cloud/samples/guestbook/inject_gcp.go にある Injector

```
//+build wireinject

// setupGCP is a Wire injector function that sets up the application using GCP.
func setupGCP(ctx context.Context,
              flags *cliFlags) (*application, func(), error) {
  // This will be filled in by Wire with providers from the provider sets in
  // wire.Build.
  wire.Build(
    gcpcloud.GCP,
    applicationSet,
    gcpBucket,
    gcpMOTDVar,
    gcpSQLParams,
  )
  return nil, nil, nil
}
```

[*11] https://golang.org/pkg/go/build/#hdr-Build_Constraints

上記の例だと自動生成後のコードが膨大になるので、簡単な Inejctor を見ます。リスト 6.8 は３つのビルド関数と外部から与えられる文字列（phrase）を使って依存関係を解決した Eventオブジェクトを返す Injector[*12]です。

▼リスト 6.8　go-cloud/samples/wire/wire.go にある Injector

```
//+build wireinject

func InitializeEvent(phrase string) (Event, error) {
  wire.Build(NewEvent, NewGreeter, NewMessage)
  return Event{}, nil
}
```

実際に依存性を解決するコードは wire genコマンドで自動生成します。リスト 6.8 から Wire コマンドが自動生成したコードがリスト 6.9 です。

▼リスト 6.9　go-cloud/samples/wire/wire_gen.go

```
//go:generate wire
//+build !wireinject

func InitializeEvent(phrase string) (Event, error) {
  message := NewMessage(phrase)
  greeter := NewGreeter(message)
  event, err := NewEvent(greeter)
  if err != nil {
    return Event{}, err
  }
  return event, nil
}
```

どの関数をどの順番で呼び、どの関数の引数に使えば戻り値のオブジェクトの初期化が完了するか、などを自動で判別してコードを生成してくれます。Wire を使うと、開発者は依存性の解決に必要な情報を書くだけで依存性が注入された初期化済みオブジェクトを手に入れることができます。では、次節より Wire コマンドの使い方を確認し、Provider や Injector の設計方法を確認していきます。

6.9　Wire コマンドの使い方

まず go-cloud リポジトリのコードを使って Wire コマンドの使い方を確認してみましょう。Wire コマンドは go getコマンドで取得できます。

[*12] NewEventなどの宣言は go-cloud/samples/wire/main.goにあります。

```
$ go get github.com/google/go-cloud/wire/cmd/wire
```

Wire コマンドには 3 個のコマンドラインオプションがあります。

```
$ wire -h
usage: wire [gen] [PKG] | wire show [...] | wire check [...]
```

6.9.1 wire gen

wire_gen.goファイルを生成します。引数なしで Wire コマンドを実行した際も wire genコマンドが実行されます。

6.9.2 wire show

定義済みの Provider の使い方を調べる手段として、wire showコマンドが用意されています。wire showコマンドは引数に指定されたパッケージ内から以下の情報を表示します。

- パッケージ内の Provider の情報を出力する
  - Provider が依存する Provider を出力
  - 必要な Input とその Input を利用して得られる Output を知ることができる。
  - 取得できる Output を生成しているコードの場所も知ることができる
- Injector の情報を出力する

実際に go-cloud リポジトリの samples/guestbook で weire showコマンドを実行してみましょう。samples/guestbookパッケージにはリスト 6.10 のような applicationSetが Provider として用意されています。また、inject_{aws, gcp, local.go}ファイルには複数の Injector が宣言されています。

▼リスト 6.10　go-cloud/samples/guestbook/main.go

```
// applicationSet is the Wire provider set for the Guestbook application that
// does not depend on the underlying platform.
var applicationSet = wire.NewSet(
    newApplication,
    appHealthChecks,
    trace.AlwaysSample,
)
```


これだけ見てもこの Provider でどんなことができるかわかりませんね。samples/guestbookパッケージに対して wire showコマンドを実行した結果が以下になります。

```
$ pwd
.../github.com/google/go-cloud/samples/guestbook
$ wire show
"github.com/google/go-cloud/samples/guestbook".applicationSet
Outputs given no inputs:
  go.opencensus.io/trace.Sampler
    at .../go.opencensus.io/trace/sampling.go:64:6
Outputs given *database/sql.DB:
  []github.com/google/go-cloud/health.Checker
    at .../github.com/google/go-cloud/samples/guestbook/main.go:313:6
Outputs given *database/sql.DB, *github.com/google/go-cloud/blob.Bucket,
*github.com/google/go-cloud/runtimevar.Variable,
*github.com/google/go-cloud/server.Server:
  *github.com/google/go-cloud/samples/guestbook.application
    at .../github.com/google/go-cloud/samples/guestbook/main.go:135:6
Injectors:
  "github.com/google/go-cloud/samples/guestbook".setupAWS
  "github.com/google/go-cloud/samples/guestbook".setupGCP
  "github.com/google/go-cloud/samples/guestbook".setupLocal
```

wire showコマンドの出力は以下のフォーマットになっています。

▼リスト 6.11　wire show の出力フォーマット

```
{{ 定義されている Provider 1}}
  {{ 依存している Provider の一覧 }}
Outputs given {{ 必要な入力パターン 1 }}
  {{ 必要な入力パターン 1 があったとき ProvideSet から得られるオブジェクト一覧 1 }}
Outputs given {{ 必要な入力パターン 2 }}
  {{ 必要な入力パターン 2 があったとき ProvideSet から得られるオブジェクト一覧 2 }}
...
{{ 定義されている Provider 2}}
  {{ 依存している Provider の一覧 }}
...
Injectors:
  {{ 定義されている Injector の一覧 }}
```

まず、リスト 6.11 では表示されていませんが、wire showコマンドで表示する Providerが他の Provider に依存した定義になっている場合、依存している Provider の一覧も表示されます。次のコマンドライン出力の結果は go-cloud/gcp/gcpcloudパッケージに対して wire showコマンドを実行した結果です。GCP という Provider から得られる Output を出力する前に、GCP が依存する Provider の一覧が表示されています。

```
$ wire show
"github.com/google/go-cloud/gcp/gcpcloud".GCP
  "github.com/google/go-cloud/gcp".DefaultIdentity
  "github.com/google/go-cloud/gcp/gcpcloud".Services
  "github.com/google/go-cloud/mysql/cloudmysql".CertSourceSet
  "github.com/google/go-cloud/runtimevar/runtimeconfigurator".Set
  "github.com/google/go-cloud/server".Set
  "github.com/google/go-cloud/server/sdserver".Set
Outputs given no inputs:
  ...
```

次に何をインジェクトすればどんなオブジェクトが取得できるのか、が確認できます。またその提供元の実装コードの場所の情報も出力されます。例えば、以下の情報は`*sql.DB`をインジェクトできれば、`health.Checker`の配列が得られることを示しています。

```
Outputs given *database/sql.DB:
  []github.com/google/go-cloud/health.Checker
    at ../go-cloud/samples/guestbook/main.go:313:6
```

複数の Provider があった場合は全ての `ProbiderSet`の情報が表示されます。最後に、`samples/guestbook`内に定義されている Injector 関数の一覧が表示されます。これらの情報を確認すれば、ある Provider を使ってどんなオブジェクトが取得できるのか、また Provider を使ってオブジェクトを取得したいときは Builder でどんなオブジェクトを用意すればよいのかわかります。

6.9.3 wire check

`wire check`コマンドは与えられたパッケージ内の Provider, Injector の定義を検査します。エラーがあった場合はそのエラー情報を出力しますし、エラーがなかった場合は何も表示せずに終了します。正常実行時の出力がでないだけで、内部処理は `wire show`コマンドと同じです。

6.10 Provider の実装を読み解く

Provider の概要は次の箇条書きのとおりです。

- `wire.NewSet`でコンポーネントをまとめた Provider を定義できる。
- `wire.NewSet`には他の Provider を含めることもできる
- 関数の戻り値をインターフェースとして扱いたい場合は `wire.Bind`関数を使う
- 後処理が必要なオブジェクトを提供する場合は、戻り値に `func`を含める

まず Provider の定義済みのコードを確認してみましょう。リスト 6.12 は「6.8.1 Provider と Injector」の冒頭でも引用した GCP の一連のサービスのコンポーネント群を生成できる Provider です。

▼リスト 6.12　go-cloud/gcp/gcpcloud/gcpcloud.go に定義された Provider

```
var Services = wire.NewSet(
  cloudmysql.CertSourceSet,
  cloudmysql.Open,
  gcp.DefaultTransport,
  gcp.NewHTTPClient,
  runtimeconfigurator.Set,
  sdserver.Set)
```

Provider を定義するときはグローバル変数で `wire.Set`関数を使って `wire.ProviderSet`オブジェクトの宣言をします。`wire.Set`にはその Provider で提供したいコンポーネントを返すビルド関数あるいは別の `wire.ProviderSet`を引数に与えます。リスト 6.12 の ProviderSet の場合、Cloud SQL へ接続する`*sql.DB`オブジェクトや`*requestlog.StackdriverLogger`オブジェクトを取得できます。`wire.Set`の引数にしたビルド関数や別の `wire.ProviderSet`が必要とする入力を全てここで列挙する必要はありません。例えば、リスト 6.12 で `wire.Set`の第 2 引数となっている `cloudmysql.Open`関数はリスト 6.13 のような宣言で`*sql.DB`オブジェクトを提供するビルド関数です。

▼リスト 6.13　go-cloud/mysql/cloudmysql/cloudmysql.go に定義された cloudmysql.Open 関数

```
// Open opens a Cloud SQL database.
func Open(ctx context.Context,
          certSource proxy.CertSource,
          params *Params) (*sql.DB, error) {
  // 省略
}
```

第 1 引数の `context.Context`は `Services`内では生成されないオブジェクトなので、Injector 作成時か `Services`を含んだ別の新しい `Provider`を定義するときに含めます[*13]。第 2 引数の `proxy.CertSource`インターフェースはリスト 6.12 の `wire.Set`の第 1 引数となっている `cloudmysql.CertSourceSet`が提供しているので、リスト 6.12 の Provider 内で完結しています。

`cloudmysql.CertSourceSet`の宣言を見ると `cloudmysql_certsource`のようになっています。

[*13] 実際に `context.Context`を Provider で用意することはないでしょう

▼リスト 6.14　go-cloud/mysql/cloudmysql/cloudmysql.go に定義された CertSourceSet

```
// CertSourceSet is a Wire provider set that binds a Cloud SQL proxy
// certificate source from an GCP-authenticated HTTP client.
var CertSourceSet = wire.NewSet(
    NewCertSource,
    wire.Bind((*proxy.CertSource)(nil), (*certs.RemoteCertSource)(nil)))
```

NewCertSourceは*certs.RemoteCertSourceオブジェクトを返すビルド関数ですが、cloudmysql.CertSourceSetでは proxy.CertSourceインターフェースを提供します。そのため、 wire.Bind関数を用いて*certs.RemoteCertSourceを proxy.CertSource インターフェースとして返す定義をしています。

6.11　Injector を自作する

「6.9.2 wire show」で wire showコマンドを使えば既成の Provider の提供情報を一覧できることは確認しました。wire showコマンドの情報をを参考に Injector を簡単に自作してみます。Injector の概要は次の箇条書きのとおりです。

- wire.Build を内部でよぶ。
- Injector メソッドのメソッド名に特別な制限はない
- Injector メソッドの引数は wire.Build で呼ぶ Provider で用意できない情報
- Injector メソッドの戻り値は以下の 4 種類しか定義できない
  - Injector で生成したい Output
  - Injector で生成したい Output、func()
  - Injector で生成したい Output、error
  - Injector で生成したい Output、func(), error
- あるオブジェクト変数をインターフェースとして扱いたい場合は wire.InterfaceValue関数を使う

func()は Provider が Close()などの関数を一緒に返す時に強制されます。たとえば、RDS MySQL の Provider は db.Close関数を呼んで cleanup するための関数を返します。

▼リスト 6.15　mysql/rdsmysql/rdsmysql.Open 関数

```
func Open(ctx context.Context, provider CertPoolProvider,
          params *Params) (*sql.DB, func(), error)
  // ...
  return db, func() { db.Close() }, nil
}
```


ある structオブジェクトを特定のインターフェース値として Provider に注入する場合は wire.InterfaceValue関数を使います。os.Stdinを io.Readerとして注入する場合は wire.Build関数内で以下のように宣言します。

▼リスト 6.16　wire.InterfaceValue の例

```
wire.Build(wire.InterfaceValue(new(io.Reader), os.Stdin))
```

では実際に Injector を一つ実装してみます。「6.9 Wire コマンドの使い方」で確認した samples/guestbookパッケージの applicationSet の情報はリスト 6.17 です。

▼リスト 6.17　s

```
$ wire show
"github.com/google/go-cloud/samples/guestbook".applicationSet
...
Outputs given *database/sql.DB:
  []github.com/google/go-cloud/health.Checker
    at .../go-cloud/samples/guestbook/main.go:313:6
...
```

applicationSetから []health.Checkerを生成したい場合は*sql.DBが必要でした。Injector はリスト 6.18 のような宣言になります。

▼リスト 6.18　[]health.Checker を取得するための Injector

```
func setupChecker(db *sql.DB) ([]health.Checker, func()) {
    // This will be filled in by Wire with providers from the provider sets in
    // wire.Build.
    wire.Build(
        applicationSet,
        // Injector 引数で*sql.DB を用意しない場合は*sql.DB を生成する関数をここにセット
する
    )
    return nil, nil
}
```

Provider が要求する*sql.DBは Injector の引数として渡すか、wire.Build関数の引数の別の関数で生成できるようにします。戻り値は自動生成されるので特に指定する必要はありません。

func()を第二戻り値にしているのは []health.Checkerの Provider が cleanup()関数を一緒に返すためです。が、これを知らなくても wire genコマンド実行時にエラー

メッセージで指摘してくれます。

```
$ wire
.../github.com/google/go-cloud/samples/guestbook/inject_gcp.go:53:1: \
inject setupChecker: provider for []github.com/google/go-cloud/health.Checker \
returns cleanup but injection does not return cleanup function

wire: generate failed
```

この Injector 定義から wire_gen.go にはリスト 6.19 のようなコードが生成されます。

▼リスト 6.19　wire gen コマンドで生成された Injector

```
func setupChecker(db *sql.DB) ([]health.Checker, func()) {
    v, cleanup := appHealthChecks(db)
    return v, func() {
        cleanup()
    }
}
```

このような小規模な DI では何もメリットを感じられません。が、「6.8.1 Provider と Injector」でも引用した`*application`オブジェクトを取得するための`setupGCP`関数の定義から自動生成される`*application`の初期化コードがリスト 6.20 です。省略していますが、50 行強のコードになります。注入するオブジェクトが BLOB クライアント、SQL クライアント (と、ロガーなどの設定) のみでもかなりのコード量が自動生成されることを考えると必要なコンポーネントさえ Injector に並べる[*14]だけでコーディングが済むのはかなり魅力的です。

▼リスト 6.20　GCP に依存した*application オブジェクトを取得する Injector

```
func setupGCP(ctx context.Context,
              flags *cliFlags) (*application, func(), error) {
  stackdriverLogger := sdserver.NewRequestLogger()
  roundTripper := gcp.DefaultTransport()
  credentials, err := gcp.DefaultCredentials(ctx)
  if err != nil {
    return nil, nil, err
  }
  tokenSource := gcp.CredentialsTokenSource(credentials)
  httpClient, err := gcp.NewHTTPClient(roundTripper, tokenSource)
  if err != nil {
    return nil, nil, err
  }
  remoteCertSource := cloudmysql.NewCertSource(httpClient)
```

[*14] `wire.Build`関数に Provider を並べるときもとくに依存関係順にしなくてはいけないなどの条件はありません。


```
  projectID, err := gcp.DefaultProjectID(credentials)
  ...
  // このあと blob.Bucket などの初期化が始まる
  ....
```

6.12　おわりに

今回は go-cloud リポジトリと付属する Wire コマンドの概要を確認しました。また、サンプルコードから`go-cloud/wire`パッケージを使って DI を行う流れを解説しました。Wire コマンドは go-cloud のパッケージに依存しない DI ツールでそれ単独でも利用することができます。go-cloud を使わなずとも Go で DI を行う時に Wire を使うのはいかがでしょうか。

ゴーファーの書

2018 年 10 月 8 日　技術書典 5 版 v1.0.0
著　者　golang.tokyo
編　集　tenntenn