JULIEN PONGE

# Golo

## 「おまじない」が少なく、すぐに覚えられる高速JVM言語

筆者が後にGoloと呼ばれることになるプログラミング言語を書き始めたのは、2012年夏のことでした。その当時、同僚とソフトウェアの動的な変更に関する研究に取り組んでおり、Javaプログラムの実行中に変更点やアスペクトをインジェクションするためのJooFluxというJVMエージェントを提案しました。この提案は、JSR 292（JVMでの動的言語のサポート）とinvokedynamicバイトコード命令に基づくものでした。最終的には、JVM上で動作する動的型付け言語の設計を容易にするため、このバイトコードについて数多くの実験を行うことになりました。

invokedynamicのない既存のJVM言語を研究していた際に、シンプルで理解しやすいコンパイラとランタイム・コードベースを備えた言語を作り、新しいバイトコードのような言語の進化を実験できるようにするとおもしろいのではないかと思い当たりました。それから早くも数年が経った今、GoloはEclipse Technologyプロジェクトのインキュベータ・プロジェクトとなっています。このプロジェクトでは、それまで言語の開発経験がなかった愛好家たちがGoloの開発に携わり、また、Goloから派生した研究も行われています。

本記事では、このGoloというプログラミング言語のいくつかの機能を紹介します。すべての機能をカバーしたものではありませんが、よい入門記事となるはずです。

### まずは冗長的に

Goloは動的型付け言語です。命令型、関数型のイディオムをサポートしており、Javaとの相性も抜群です。その親和性を示すために、Java標準APIのjava.util.ArrayListを作成するシンプルなプログラムを見てみます。そして、各要素について反復処理を行い、出力します。このプログラムの構造は意図的にJavaに近いものにしてありますが、後ほどこの構造を簡潔にする方法について説明します。

```
module javamag.Hello

import java.util

# コメント
function main = |args| {
  let elements = ArrayList()
  elements: add("Hello")
  elements: add("world")
  elements: add("!")
  let size = elements: size()
  for(var i = 0, i < size, i = i + 1) {
    print(elements: get(i))
    if (i < size - 1) {
      print(" ")
    }
  }
  println("")
}
```

このコードから、いくつかのことがわかります。

- コードはモジュールとなっており、これがGoloのコンパイル・ユニットになります。
- シンボルの解決は、import文によって行います。
- 関数のパラメータはパイプ記号（|）で挟んで渡します。
- Javaクラスのインスタンスを作成するnew演算子はありませんが、コンストラクタを関数として呼び出すことで同様の操作を行うことができます。
- letで定数の参照を定義し、varで変数の参照を定義します。
- インスタンスのメソッドは:演算子を使って呼び出します。たとえば、elements: size()はelementsオブジェクトのsizeメソッドを呼び出しています（コロンの後の空白は慣例であり、必須ではありませ

写真：MATT BOSTOCK/GETTY IMAGES

ん)。

- #記号で始まる行はコメント行です。
- Goloモジュールには、エントリ・ポイントとしてmain関数を含めることができます。この関数は、Javaのmainメソッドと同様に、厳密に1つの引数として、コマンドラインの引数が格納される配列を取ります。

import文は完全にシンボリックなものであり、コンパイル時にチェックされません。この点は重要であるため覚えておいてください。import java.utilではなくimport javaと書いた場合は、ArrayListのコンストラクタがutil.ArrayList()として呼び出されることとなります。

Goloには、独自のコマンドライン・ツールgoloがあります。さらに、compile、run、goloなどのいくつかのサブコマンドが提供されています。先ほどのプログラムをコンパイルして実行することも簡単です。次の例をご覧ください。

```
$ golo compile hello-1.golo
$ ls javamag/
Hello.class
$ golo run --classpath . --module javamag.Hello
Hello world !
$
```

goloサブコマンドは、ソース・コードをコンパイルしてファイルに出力するのではなく、メモリ内でコンパイルしてモジュールを実行します。デフォルトでは、直近の.goloファイルが実行されます。

```
$ golo golo --files hello.golo
Hello world !
$
```

## 冗長性の削減

Goloでは、リスト、配列、ベクター、タプル、マップ、セット、範囲を作成するコレクション・リテラルがサポートされています。

```
[1, 2, 3]
list[1, 2, 3]
map[[1, "a"], [2, "b"]]
[1..10_000]
# ...
```

このようなオプションを使って先ほどの例を書き換えることができます。あわせて、foreachループを紹介します。

```
let elements = vector["Hello", "world", "!"]
foreach e in elements {
  print(e + " ")
}
println("")
```

foreachループには、when句によるガード条件を付けることができます。次の例では、範囲(ドット2つ)を使用して偶数のみを出力しています。

```
let ints = [1..100]
foreach i in ints {
  if i % 2 == 0 {
    println(i)
  }
}
```

このサンプルは、whenを使って次のように書き換えることができます。

```
let ints = [1..100]
foreach i in ints when i % 2 == 0 {
  println(i)
}
```

Goloでは、すべてのコレクション・リテラルでコレクション内包表記がサポートされています。この機能は、Pythonプログラミング言語のものに似ています。次に簡単な例を示します。

```
let odds = [i foreach i in [1..100] when i % 2 == 1]
println(odds)
```

このコードでは、1と100の間の奇数で構成されるタプルを定義しています。ご覧のとおり、whenガードを含めることもできるforeach句を埋め込むことによって、この定義を実現しています。

次の例のように、複数のforeach句を記述することもできます。この例では、タプルのタプルとして表現される整数のペアのコレクションを生成します。

```
let pairs = [ [i * 2, j * 3]
  foreach i in odds when (i >= 30) and (i <=
50)
  foreach j in [10..20] when j % 2 == 1 ]
println(pairs)
```

さらに、Goloではデータ型やコレクションの分割代入もサポートされています。

```
let l = list["1", "2", "3", "4"]
let a, b = l
let head, second, tail...= l

let m = map[["a", 123], ["b", 456]]
foreach key, value in m: entrySet() {
  # ...
}
```

最初の2つの例では、aの値は"1"になり、bの値は"2"になります。同様に、head、second、tailはそれぞれ"1"、"2"、list["3", "4"]となります。分割代入は、マップのエントリをキーと値のペアに分解する場合にも便利です（Goloのmap[...]リテラルは、java.util.HashMapを生成します）。

## 高階関数

ほとんどの最新プログラミング言語がそうであるように、Goloでも高階関数がサポートされています。高階関数とは、関数をパラメータとして受け取り、関数を返す関数です。

モジュール内で宣言された関数がある場合、^演算子によってその関数への参照を取得できます。関数値に参照が割り当てられると、次の例のようにしてその関数を呼び出すことができます。

```
module Foo

function hello = {
  println("Hello!")
}

function main = |args| {
  let f = ^hello
  f()       # "Hello!"を出力
}
```

もちろん、次の例のように関数を直接宣言することも可能です。

```
let f = |str| {
  println(">>> " + str)
}
f("Hello!")     # ">>> Hello!"を出力
```

また、単一の式から成る関数は、->表記法による省略形で表現することもできます。

```
let f = |str| -> println(">>> " + str)
f("Hello!")     # ">>> Hello!"を出力
```

関数の参照を操作できるメソッドもあります。この種のメソッドは、主に部分適用や合成といった操作を行います。提供されているメソッドは、java.lang.invokeパッケージのAPIのものにとてもよく似ています。次の例をご覧ください。

```
let add = |a, b, c| -> a + b + c
let times = |a, b| -> a * b

let add_1_and_2 = add: insertArguments(0, 1, 2)
let twice = times: bindTo(2)
let f = add_1_and_2:
  andThen(twice):
  andThen(|n| -> println(n))

# "12"を出力
f(3)
```

このコードについて、順を追って説明します。add_1_and_2は、addの0から始まる場所にある引数に対して部分適用を行った結果です。これによって、aとbがそれぞれ1と2に固定されます。その結果、この関数は元のadd関数に引数cとして渡される1つの引数を取るようになります。bindToも同様の部分適用を行いますが、最初のパラメータの値1つだけに適用します。andThenは関数を合成します。fは、最初に1つの引数に1と2を足してその結果をtwiceに渡し、さらにその結果を、出力用の匿名関数に渡しています。

Goloの関数には、もう1つの興味深い特徴があります。名前付き引数です。

```
function startServer(address, port, options) {
  # (...)
}

function run = {
  # (...)
  startServer(
    port=8080,
    options=list["--watch", "--verbose"],
    address="0.0.0.0")
}
```

名前付き引数の便利な点は、特定のAPIを使用する際にパラメータの目的を明確化できることです。さらに、関数宣言とは違う順番でパラメータを渡すこともできます。匿名関数も含め、すべてのGolo関数は名前付き引数で呼び出すことができます。

注目すべき点は、GoloはJavaクラス・メソッドにも名前付き引数を使用できることです。ただし、この使用には1点だけ制限があります。クラスはjavac -parametersフラグを有効にしてコンパイルする必要がありますが、このフラグはデフォルトでは有効になっていません。

## Golo関数とJava関数インタフェース

ラムダ式が追加されたのはJava 8になってからですが、単一メソッド・インタフェースは長きにわたりJavaの言語やプラットフォームに利用されてきました。この典型的な例として、Threadコンストラクタにrunnableインスタンスを渡すことや、Swingコンポーネントに ActionListenerを渡すことが挙げられます。

具体例を見るために、Java 8で追加されたjava.util.concurrent.CompletableFutureクラスを使ってみます。

```
module SmiAndLambda

import java.util.concurrent.CompletableFuture

function main = |args| {
  supplyAsync(-> 1):
    thenApply(|n| -> n + 99):
    thenAcceptAsync(|n| -> println(">>> " +
n)):
    join()
}
```

このコードでは、CompletableFutureを作成するために、まず1を返すだけの関数を非同期的に実行しています。次に、最初の呼出しの結果に99を加える別の関数を適用し、さらに結果(>>> 100)を出力する別の関数を非同期的に実行しています。

CompletableFutureのsupplyAsyncメソッドは、単一メソッド・インタフェースであるjava.util.function.Supplierを受け取ります。thenApplyメソッドは、関数インタフェースである(つまり、defaultメソッドと合わせて単一の抽象メソッドを持つ)java.util.function.Functionを受け取ります。同様に、thenAcceptAsyncは、同じく関数インタフェースであるjava.util.function.Consumerを受け取ります。

Goloランタイムは、関数の参照をJavaの単一メソッド・インタフェースと関数インタフェースの両方に自動的に適応させます。

## 構造体

Goloでは、struct宣言を利用して次のように構造化データを定義できます。

```
struct Message = { id, date, payload }
```

構造体オブジェクトには、コンストラクタ、getterおよびsetterアクセッサ、実用的なequals()、hashCode()、toString()といったメソッドがあ

ります。上記のMessage構造体定義は、次の例のように使うことができます。

```
let m = Message(
  id=12345,
  date="2016-01-22 15:41 CEST",
  payload="""I hope this message finds you well.

Yours sincerely,

- Julien""")

println(m)
m: id(6789)
println(m: id())
```

このコードにより、次の結果が出力されます。

```
struct Message{id=12345, date=2016-01-22 15:41 CEST,
payload=I hope this message finds you well.

Yours sincerely,

- Julien}
6789
```

Goloでは、"""で複数行テキストを定義できる点に注意してください。

structオブジェクトのfrozenCopy()メソッドを呼び出すと、不変コピーを作成できます。すべての構造体は、不変オブジェクトを作成するコンストラクタを備えています。このコンストラクタ関数の名前には、Immutableという接頭辞がつきます。先ほどの例では、Messageの代わりにImmutableMessageを呼び出すことで、mを不変オブジェクトとして作成できます。その場合、mは不変であるため、m: id(6789)の呼出しは失敗します。

構造体のフィールドの可視性はデフォルトでpublicです。上記のMessage構造体が定義されたモジュールを別のモジュールにインポートすると、インポート先のモジュールでは、アクセッサ（たとえば、id()、id(newValue)）を通じてすべてのフィールドにアクセスできます。フィールド名の先頭にアンダースコアを付けると、可視性を制限できます。そのようなフィールドは、構造体が定義されているモジュールのコードからのみ参照でき、他のモジュールからはアクセスできなくなります。

また、構造体オブジェクトでは、分割代入や列挙も可能です。そのため、次のような記述も可能です。

```
let id, date, payload = m
println(id)
println(date)
println(payload)

foreach field, value in m {
  println(field + " -> " + value)
}
```

## タグ付き共用体

構造体型を補完するものとして、代数和データ型の直和型とも呼ばれるタグ付き共用体もサポートされています。Figure型の定義の例を次に示します。

```
union Figure = {
  Nothing
  Square = { sideLength }
  Rectangle = { firstSideLength, secondSideLength }
  Circle = { radius }
}
```

この定義では、FigureはNothing、Square、Rectangle、Circleという具象型のいずれかになります。それぞれの型に（不変）フィールドとコンストラクタを持たせることができます。いくつかのFigureインスタンスを持つタプルを作成する例を次に示します。

```
let figures = [
  Figure.Square(23),
  Figure.Rectangle(firstSideLength=10,
                   secondSideLength=20),
  Figure.Circle(30),
```

```
  Figure.Nothing()
]
```

Figureのそれぞれの具象型について、インスタンスがその型のものかどうかを確認するためのメソッドが提供されています。たとえば、f: isSquare()、f: isCircle()などです。次の例は、このようなメソッドの使い方を示しています。また、when/then/otherwise句によっていくつかの条件を反復的に評価して値を返すmatch演算子も導入されています。

```
println(figures: map(|f| -> match {
  when f: isRectangle()
    then
      "[ " +
      f: firstSideLength() +
      ", " +
      f: secondSideLength() +
      " ]"
  when f: isSquare()
    then "[ " +
      f: sideLength() + " ]"
  when f: isCircle()
    then "( " +
      f: radius() + " )"
  otherwise "."
}): join("\n"))
```

Goloのコレクションは、要素に関数を適用して新しいコレクションを作成するmapや、セパレータで要素を連結した文字列を生成するjoinのような関数イディオムを提供しています。上記のコードを実行すると、次のテキストが出力されます。

```
[ 23 ]
[ 10, 20 ]
( 30 )
.
```

さらに、インスタンスの型だけでなく、値をテストすることも可能です。

```
# false
println(Figure.Square(23): isSquare(20))

# true
println(Figure.Square(23): isSquare(23))
```

共用体型のフィールドの一部にしか興味がない場合もあるでしょう。そのような場合には、あるフィールドの値がマッチングに利用できないことを示す特別なUnknown.get()値を使うことができます。

```
let figs = [
  Figure.Rectangle(10, 10),
  Figure.Rectangle(10, 30),
  Figure.Rectangle(30, 30),
  Figure.Circle(20)
]
let _ = Unknown.get()
println(figs: filter(|f| -> match {
  when f: isRectangle(10, _)
    then true
  otherwise
    false
}): join("\n"))
```

このコードでは、figsコレクションから始めて、filterを使って最初の辺の値が10である長方形以外の図形を破棄し、その結果を表示しています。出力は次のようになります。

```
union Figure.Rectangle{firstSideLength=10,
    secondSideLength=10}
union Figure.Rectangle{firstSideLength=10,
    secondSideLength=30}
```

## オーグメンテーション

Goloではクラスを定義する構造は提供されませんが、Goloが操作できるあらゆる型にメソッドを追加する方法が提供されています。操作できる型には、Java APIで定義されているクラスと、Goloのstruct型およびunion型が含まれます。オーグメンテーションは「添加」という意味で、メ

ソッドとして呼び出すことができる一連の関数を定義することです。このような関数では、慣習的に第1パラメータにthisを取ります。このパラメータがレシーバ・オブジェクトを参照しているためですが、別の名前を使うこともできます。

オーグメンテーションは型を指定して定義できます。

```
augment java.lang.String {

  function wrap = |this, s1, s2| ->
    s1 + this + s2

  #（...他のメソッドも追加できます）
}
```

上記のコードでは、JavaのStringインスタンスにwrapメソッドを追加しています。たとえば、次のコードの結果は"{abc}"となります。

```
"abc": wrap("{", "}")
```

オーグメンテーションは型とそのすべてのサブタイプに適用されます。java.lang.Objectのオーグメンテーションはすべての型に適用されます。オーグメンテーションは、その定義が含まれるモジュールと、そのモジュールをインポートしているモジュールから参照できます。

オーグメンテーションを定義するもう1つの方法は、対象の型ではなく名前を使うことです。

```
augmentation Wrap = {
  function wrap = |this, s1, s2| ->
    s1 + this: pretty() + s2
}

augmentation PrettyContact = {
  function pretty = |this| ->
    this: name() + " <" +
    this: email() + ">"
}
```

この2つの名前付きオーグメンテーションの定義を使うと、次のstruct型を作成してオーグメンテーションを追加できます。

```
struct Contact = { name, email }

augment Contact with Wrap, PrettyContact
```

そうすると、オーグメンテーションを追加した型を次のように使うことができます。

```
let dan = Contact("Dan", "dan@tld")
println(dan: pretty())
println(dan: wrap("/* ", " */"))
```

このコードにより、次の結果が出力されます。

```
Dan <dan@tld>
/* Dan <dan@tld> */
```

名前付きオーグメンテーションが型へのオーグメンテーションよりも優れている点は、複数の型にオーグメンテーションを追加（まったく関係ない型でも構いません）する際にコードの重複が回避されることです。

## アダプタ・クラスの生成

GoloからのJava APIの呼出しで問題が生じることはほとんどありませんが、Java APIが、あるインタフェースやベース・クラスを拡張したオブジェクトを引数として要求することがあります。そのような場合のために、Goloにはアダプタ・クラスを生成するAPIがあります。アダプタ・クラスは、ベース・クラスの拡張、一連のインタフェースの実装、メソッドのオーバーライドや実装を行います。

Adapter APIはgololang.Adaptersインポート文によって有効になります。有効になると、次のようにしてオブジェクトを作成できるようになります。

```
let obj = Adapter():
  extends("java.lang.Object"):
  interfaces([
    "java.io.Serializable",
    "java.util.Enumeration"]):
```

```
  implements("hasMoreElements",
    |this| -> true):
  implements("nextElement",
    |this| -> 100):
  overrides("toString",
    |this, super| -> "Strange!"):
  newInstance()

println(obj: getClass())
println(obj: getClass():
             getInterfaces():
             toString())
println(obj: hasMoreElements())
println(obj: nextElement())
println(obj: toString())
```

このコードを実行すると、次の結果が出力されます。

```
class $Golo$Adapter$0
[interface gololang.GoloAdapter,
  interface java.io.Serializable,
  interface java.util.Enumeration]
true
100
Strange!
```

このAPIは、健全性の検証（たとえば、すべてのインタフェース・メソッドが実装されているかどうかの確認）を行い、アダプタ・クラスをJVMバイトコードとして動的に生成します。メソッド名の代わりに「*」を使用することもできます。これによって、1つの関数で複数のメソッドを実装することや、すべてのメソッドをオーバーライドすることができるようになっています。Goloのドキュメントには、アダプタを使って動的にプロキシを生成する例が掲載されています。

## JavaからのGoloの呼出し

一般的に、JavaからGoloを呼び出すことも非常に簡単です。この呼出しは、直接呼出しとコード評価という2つの方法で実現できます。

**直接呼出し：**最初のオプションは、Goloのソース・ファイルをコンパイルし、生成されたバイトコード、GoloランタイムJARファイル、依存コンポーネントをJavaアプリケーション・クラスパスに追加する方法です。Javaクラスから見ると、Goloモジュールは静的メソッドを持ったクラスです。次のモジュールについて考えてみます。

```
module my.great.Module

struct Point = {x, y}

function a = |n| -> n + 1
function b = |a, b| -> a * b
```

javap逆コンパイラを使うと、次のpublicメソッドが含まれていることがわかります。

```
$ javap my.great.Module
Compiled from "compil.golo"
public class my.great.Module {
  public static java.lang.String[] $imports();
  public static java.lang.String[] $augmentations();
  public static java.lang.String[]
      $augmentationApplications();
  public static java.lang.String[]
      $augmentationApplications(int);
  public static java.lang.Object
      a(java.lang.Object);
  public static java.lang.Object
      b(java.lang.Object, java.lang.Object);
  public static java.lang.Object Point();
  public static java.lang.Object
      Point(java.lang.Object, java.lang.Object);
  public static java.lang.Object
      ImmutablePoint(java.lang.Object,
                     java.lang.Object);
}
```

[編集注：インデントされている行は、前の行から折り返されている行です]aやPointなどのメソッドは、Javaから「そのまま」呼び出すことができます。$で始まるメソッドは、Goloランタイムが使用するメソッドです。

**コード評価：**もう1つのオプションは、Goloランタイムの一部であるgololang.EvaluationEnvironmentクラスを使うことです。このクラスは、Goloコードを文字列として渡して評価または実行するさまざまな方法を提供します。

Javaからこのクラスを使用する例を示します。

```
package sample;

import gololang.EvaluationEnvironment;
import gololang.FunctionReference;

public class GoloFromJava {

  public static void main(String[] args)
    throws Throwable {
      EvaluationEnvironment env =
        new EvaluationEnvironment();

      FunctionReference f1 =
        (FunctionReference) env.def(
          "|a, b| -> println(a + \" ~ \" + b)");
      f1.invoke("hello", "world");

      FunctionReference f2 = (FunctionReference)
        env.asFunction("println(a + b)", "a", "b");
      f2.invoke(5, 10);
  }
}
```

このコードにより、次の結果が出力されます。

```
hello ~ world
15
```

EvaluationEnvironmentクラスは、テキストからモジュールのソース・コード全体を評価する、インポートを定義するなど、Goloコードを評価する追加メソッドを提供します。

## まとめ

本記事では、Goloプログラミング言語のいくつかの興味深い機能を紹介しました。もちろん、この言語はこれだけのものではありません。実験をしながらさらに踏み込んでみることをお勧めします。プログラミング言語はどのように実装できるのかをいつも疑問に思っている方であれば、ソース・コードをのぞいてみるのもよいでしょう。筆者はソース・コードが教育的価値の高いものとなるよう常に努力しています。

大事なことを書き忘れていましたが、Goloは趣味レベルで簡単に楽しむことができる言語です。皆様の貢献をお待ちしています。**</article>**

---

**Julien Ponge**（@jponge）：長年にわたってオープンソースに貢献しており、現在はフランス国立応用科学院リヨン校のコンピュータ・サイエンスおよびエンジニアリング准教授を務める。プログラミング言語、仮想マシン、ミドルウェアを中心に研究中。

筆者は、Goloに貢献している方々、とりわけ言語機能の設計や実装に関わったYannick Loiseau氏、Sylvain Desgrais氏、Daniel Petisme氏、Philippe Charrière氏に感謝の意を表します。