静的型付け関数型言語Gleam

ndruger

Gleamとは？

- Erlang VM(BEAM)上で動作する静的型付け関数型プログラミング言語。
- 2024/3/4にバージョン1がリリースされた。
    - Gleam version 1 – Gleam

個人的に年一程度で新しい言語を試すようにしており、今回はGleamの文法を他の言語と比較しながら紹介する。

Erlang VM(BEAM)とは？

ElixirやErlangなどのプログラミング言語が動作する仮想マシンで、以下の特徴がある。

- 高い並行性
  - 軽量プロセスを用いた並行処理が可能で、数百万のプロセスを効率的に管理できる。
- スケーラビリティ
  - 多数のコアや分散システムでのスケーラビリティをサポートし、大規模な分散システムの構築に適している。
- フォールトトレランス
  - プロセスの隔離により、一部のプロセスがクラッシュしてもシステム全体には影響しない。

Elixirとは？

- Erlang VM(BEAM)上で動作する関数型プログラミング言語。
- 一見Rubyに似たシンタックスを持ち、Erlangよりも慣れ親しんだ文法で書くことができる。
- **静的型付け言語ではない。**
  - 静的コード分析ツールのDialyzerがあるが、言語仕様とは独立したツールであり、型推論が不十分だったり、時間がかかったり、結果がわかりづらい場合がある。
  - 現在、Elixirの言語仕様に静的型付けを追加するプロジェクトが進行中。
    - Type system updates: moving from research into development - The Elixir programming language

Gleamの特徴

- Erlang VM(BEAM)上で動作する**静的型付け**関数型プログラミング言語。
- 型推論が可能で、型エラーをコンパイル時に検出できる。
- Elixir同様に強力なパターンマッチングが可能。
- データ構造はImmutable。
- Resultが言語レベルで提供されている。
- ジェネリクスにより柔軟な型定義が可能。

・・・最強では？

Playground

スライドのコードは全てGleamのPlaygroundで実行可能。

- Welcome to the Gleam language tour! 💫 - The Gleam Language Tour

Hello, World!

```
import gleam/io

pub fn main() {
  io.println("Hello, World!")
  // io.println(1) // String引数にIntを渡しているのでコンパイルエラーになる
}
```

関数

```
import gleam/io

pub fn main() {
  let a = double(10) // 型推論によりInt型として扱われる
  io.debug(a)
}

fn double(a: Int) -> Int {
  a * 2
}
```

ラベル付き引数(1)

結構珍しい機能。

```
import gleam/io

pub fn main() {
  // ラベルなし呼び出し
  io.debug(calculate(1, 2, 3))

  // ラベル付き呼び出し
  io.debug(calculate(1, add: 2, multiply: 3))

  // ラベル付き呼び出し(順番を変えてもOK)
  io.debug(calculate(1, multiply: 3, add: 2))
}

// 引数名とは別にラベルをつけるとそのラベルで引数に値を渡せる
fn calculate(value: Int, add addend: Int, multiply multiplier: Int) {
  value * multiplier + addend
}
```

ラベル付き引数(2)

引数名を指定して外部から値を渡せない。
ラベルを付けることで明示的に外部からラベル名で引数を渡すことができるようになる。

```
import gleam/io

pub fn main() {
  // 引数名を指定すると、その名前のラベルが存在しないのでエラーになる
  io.debug(calculate(1, addend: 2, multiplier: 3))
}

// ラベルがない関数
fn calculate(value: Int, addend: Int, multiplier: Int) {
  value * multiplier + addend
}
```

ラベル付き引数(3)

Swiftでも関数の引数にラベルをつけることができるが、ラベルをつけない場合は引数名で呼び出すことができる。デフォルト動作が非公開・公開の違いがある。

- Functions | Documentation

```
func greet(person: String, from hometown: String) -> String {
    return "Hello \(person)!  Glad you could visit from \(hometown)."
}
print(greet(person: "Bill", from: "Cupertino"))
// Prints "Hello Bill!  Glad you could visit from Cupertino."
```

レコードの定義

```
import gleam/io

type Person {
  Person(name: String, age: Int)
}

pub fn main() {
  let person = Person("Joe", 20)
  io.debug(person.name)
}
```

カスタムタイプ(1)

いわゆるユニオンタイプを作りたい場合、明示的にタグを指定する必要がある。

```
import gleam/io
import gleam/string
import gleam/int

type IntOrFloat { // IntとFloatのどちらかを持つユニオンタイプ
  AnInt(Int)  // タグとしてAnIntが必要
  AFloat(Float) // タグとしてAFloatが必要
}

fn int_or_float(x: IntOrFloat) {  // `x: Int | Float`とは書けない。引数の型は1つのみ。
  case x {
    AnInt(n) -> string.append("It's an integer: ", int.to_string(n))
    AFloat(_) -> "It's a float"
  }
}

pub fn main() {
  io.debug(int_or_float(AnInt(1)))  // int_or_float(1)とは書けない
  io.debug(int_or_float(AFloat(1.0)))  // int_or_float(1.0)とは書けない
}
```

カスタムタイプ(2)

- TypeScriptのようにタグを指定しないユニオンタイプではなく、タグを指定するのでこのユニオンタイプは直和型である。
  - ユニオンタイプと直和型の違いは下記が参考になる。
    - 「ADT, 直和・直積, State Machine」 #TypeScript - Qiita
  - タグがあるので、同じ型でも異なるタグを持つ値を区別することができる。
    先ほどの例が両方Int型でも、タグが異なるので区別できる。
- ElmやHaskellと同じ。
  - Custom Types · An Introduction to Elm

Result

- Gleamはエラーハンドリングに例外を使わない。
- RustやElmのようにResultが言語レベルで提供されているので、TypeScript/Elixir/PythonでResultを使う時のように標準でないライブラリを探す必要がない。
    - https://hexdocs.pm/gleam_stdlib/gleam/result.html

パイプライン

- Elixirのパイプライン演算子`|>`のような機能がある。

```
import gleam/io
import gleam/string

pub fn main() {
  // 読みづらい
  io.debug(string.append(string.append("a", "b"), "c")) // abc

  // 下記のように書ける
  "a"
  |> string.append("b")
  |> string.append("c")
  |> io.debug // abc

  // デフォルトは先頭に渡されるが、_で位置を変更できる。
  "a"
  |> string.append("b")
  |> string.append("c", _)
  |> io.debug // cab
}
```

ジェネリック関数

```
import gleam/io

pub fn main() {
  let add_one = fn(x) { x + 1 }

  io.debug(twice(1, add_one))
  // io.debug(twice("1", add_one)) // エラー
}

fn twice(argument: value, my_function: fn(value) -> value) -> value {
  my_function(my_function(argument))
}
```

ジェネリックタイプ

```
pub type MyOption(inner) {
  Some(inner)
  None
}

pub const name: MyOption(String) = Some("test")
```

インターフェイスの実現

- ElixirのBehavioursに相当する、特定の関数セットの実装を強制するための「インターフェース」を実現する専用の機能はない。
- トレイトや継承もない。
- ただし、レコードのフィールドに関数を持たせることで、インターフェースのような機能を実現することができる。

インターフェイスの実装を強制するコード

```
import gleam/io
import gleam/string

// インターフェースとしてのレコード
// explain()関数を実装させる。
pub type Reporter(x) {
  Reporter(
    explain: fn(x) -> String,
  )
}

// 特定のインターフェースを強制する関数
// 引数が、 target というジェネリックな型を受け取り、それを処理できるReporterインターフェースを実装することを強制する。
pub fn show_report(d: Reporter(target), value: target) {
  let Reporter(explain) = d
  io.debug(explain(value))
}
```

利用側のコード

```
pub type Person {
  Person(name: String, age: Int)
}

// Person用のReporterインスタンスを返す関数
pub fn report_person() -> Reporter(Person) {
  Reporter(
    fn(person) {  // explain()の実装
      case person {
        Person(name, _age) -> string.append("こちらの方の名前は", name)
      }
    }
  )
}

pub fn main() {
  let person = Person(name: "Alice", age: 30)
  let person_reporter = report_person()

  show_report(person_reporter, person)
}
```

他

- マクロは現在はない。FAQでメタプログラミングについては検討していると記載がある。
  - Frequently asked questions – Gleam
- ErlangのASTにコンパイルされるので、Elixirから利用しやすい。
- OTPに相当するフレームワークとして下記が開発中。
  - [GitHub - gleam-lang/otp: 📫 Fault tolerant multicore programs with actors

まとめ

- Gleamは型による安全性を重視した関数型プログラミング言語。
- 構文がシンプル。