Swift の struct・enum と代数学

part1 (#4 Algebraic Data Types)

iOSアプリ開発のためのFunctional Architecture情報共有会2

今回のテーマについて

- なぜこのテーマ？
  - TCA で使われている Case Paths について学びたかった
  - Case Paths を学ぼうとしたら #51 Structs 🤝 Enums を先に読むべきとお勧めされた
  - #51 Structs 🤝 Enums を読もうとしたら、以下を先に読むと良いとお勧めされた
    - #4 Algebraic Data Types
    - #9 Algebraic Data Types: Exponents
    - #19 Algebraic Data Types: Generics and Recursion

そこで今回は三つのうち一つををまとめます

- **#4 Algebraic Data Types**
  - **代数データ型**
- #9 Algebraic Data Types: Exponents
  - 代数データ型：指数
- #19 Algebraic Data Types: Generics and Recursion
  - 代数データ型： Generics と再帰

早速構造体を見ていきます

```
struct Pair<A, B> {
    let first: A
    let second: B
}
```

↓４つのパターンを作ることができる

```
Pair<Bool, Bool>(first: true, second: true)
Pair<Bool, Bool>(first: true, second: false)
Pair<Bool, Bool>(first: false, second: true)
Pair<Bool, Bool>(first: false, second: false)
```

↓の enum を使ってみる

```
enum Three {
    case one
    case two
    case three
}
```

Three enum を利用したパターン

```
Pair<Bool, Three>(first: true, second: .one)
Pair<Bool, Three>(first: true, second: .two)
Pair<Bool, Three>(first: true, second: .three)
Pair<Bool, Three>(first: false, second: .one)
Pair<Bool, Three>(first: false, second: .two)
Pair<Bool, Three>(first: false, second: .three)
```

全部で 6 つのパターンができます

Void についても見ていきます

- Void は奇妙な型である
  - 型と値を同じように参照できる

```
_: Void = Void()
_: Void = ()
_: () = ()
```

Void の値は一つ

- Void は値を一つしか持たない
  - () の実体は重要ではない
  - Void の中にあるものを表す値があるだけで、() は何もできない
  - 返り値を持たない関数宇賀、明示的に指定されていなくても Void を返すのはこのため

```
func foo(_ x: Int) /* -> Void */ {
    // return ()
}
```

Void を先ほどの Pair に適用してみる

- ↓は二つのパターンしか存在しない

```
Pair<Bool, Void>(first: true, second: ())
Pair<Bool, Void>(first: false, second: ())
```

- ↓は一つのパターンだけ！

```
Pair<Void, Void>(first: (), second: ())
```

もう一つの奇妙な型 Never

```
enum Never {}
```

- Never は case を持たない enum
- つまり値を持たない型

```
_: Never = ???
```

もちろん ↑ のようなことをしてコンパイルすることはできない

Never を Pair に適用すると？

```
Pair<Bool, Never>(first: true, second: ???)
```

- ↑ ??? に入れられるものは何もない
- Never もコンパイラによって特別な扱いを受けている
  - Never を返す関数は何も返さない関数として知られている
  - 例えば fatalError は Never を返す
  - コンパイラは fatalError を実行後のコードの全ての行と分岐は無意味になることを知っている
  - それを使ってコードの網羅性を証明することができる

Pair<A, B> の値の数の関係はどうなっている？

```
// Pair<Bool, Bool>  = 4
// Pair<Bool, Three> = 6
// Pair<Bool, Void>  = 2
// Pair<Void, Void>  = 1
// Pair<Bool, Never> = 0
```

↓ A の値の数と B の値の数の乗算で表されている！

```
// Pair<Bool, Bool>  = 4 = 2 * 2
// Pair<Bool, Three> = 6 = 2 * 3
// Pair<Bool, Void>  = 2 = 2 * 1
// Pair<Void, Void>  = 1 = 1 * 1
// Pair<Bool, Never> = 0 = 2 * 0
```

これは Pair 以外の構造体にも当てはまる

```
enum Theme {
  case light
  case dark
}

enum State {
  case highlighted
  case normal
  case selected
}

struct Component {
  let enabled: Bool // 2
  let state: State // 3
  let theme: Theme // 2
} // Bool * State * Theme = 2 * 3 * 2 = 12
```

簡単のために今後以下のような表現をします

```
// Pair<A, B>          = A * B
// Pair<Bool, Bool>    = Bool * Bool
// Pair<Bool, Three>   = Bool * Three
// Pair<Bool, Void>    = Bool * Void
// Pair<Bool, Never>   = Bool * Never
```

ざっくり、`Pair<A, B>` は A と B の要素数の乗算であるという風に直感的に読めれば OK です

値が有限でないものは？

```
// Pair<Bool, String> = Bool * String
```

String には無限大の数が存在するが、２x ∞ と考えて良い

```
// String * [Int]
// [String] * [[Int]]
```

これも無限大の数同士を掛け合わせていると考えることができる

型を代数的実体として捉える

```
// Never = 0
// Void = 1
// Bool = 2
```

- ↑ は Void, Never, Bool の名前を一掃して、型の中に含まれる値の数だけを表現している
- つまり今は特定の型について考えているのではなく、抽象的な代数的実体を考えているだけ

enum はどうか？

```
enum Either<A, B> {
    case left(A)
    case right(B)
}
```

↓ Bool, Bool なら（2 + 2）パターン

```
Either<Bool, Bool>.left(true)
Either<Bool, Bool>.left(false)
Either<Bool, Bool>.right(true)
Either<Bool, Bool>.right(false)
```

enum はどうか？

```
enum Either<A, B> {
    case left(A)
    case right(B)
}
```

↓ Bool, Three なら（2 + 3）パターン

```
Either<Bool, Three>.left(true)
Either<Bool, Three>.left(false)
Either<Bool, Three>.right(.one)
Either<Bool, Three>.right(.two)
Either<Bool, Three>.right(.three)
```

enum はどうか？

```
enum Either<A, B> {
    case left(A)
    case right(B)
}
```

↓ Bool, Void なら（2 + 1）パターン

```
Either<Bool, Void>.left(true)
Either<Bool, Void>.left(false)
Either<Bool, Void>.right(Void())
```

enum はどうか？

```
enum Either<A, B> {
    case left(A)
    case right(B)
}
```

↓ Never なら？

```
Either<Bool, Never>.left(true)
Either<Bool, Never>.left(false)
Either<Bool, Never>.right(???) // コードとしては有効ではない（説明のため）
```

Either を使うと、片方の case は無になる

まとめると

```
Either<Bool, Bool>  = 4 = 2 + 2
Either<Bool, Three> = 5 = 2 + 3
Either<Bool, Void>  = 3 = 2 + 1
Either<Bool, Never> = 2 = 2 + 0
```

- `Either<A, B>` の値は「A の値の数 + B の値の数」
- これが enum が 「sum types」と呼ばれる所以である
- Either は論理学の観点から解釈することもできる
  - 二つの型の「または」を取る意味をカプセル化している