

Swift の struct • enum と代数学

part1 (#4 Algebraic Data Types)

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の一つをまとめます

- **#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

※ **自分の解釈も多分に含まれます**

代数学とこのテーマの概要

- 代数学（引用：物理のかぎしっぽ）
 - 代数式：有限個の係数や未知数を「+, -, ×, ÷, √」の五つの演算だけを組み合わせて作った式（今回はこちらのみ）
 - 未知数が代数式の形で表される方程式を代数方程式と呼ぶ

$$a_0x^n + a_1x^{n-1} + \dots + a_{n-1}x + a_n = 0$$

- テーマの概要
 - struct と enum を代数学を使って見ていこう（ざっくり）

早速 struct から見ていきます

```
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)
```

オリジナルの enum を作って適用してみる

全部で六つのパターンが出来上がる

```
// オリジナルの enum
```

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

```
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)
```

Void についても見ていきます

- Void は奇妙な型である
 - 一つ目の理由：型と値を同じように参照できる
 - Void は型で、() はVoid の値

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

Void が奇妙である二つ目の理由：値が一つ

- Void の値は一つ（「()」）しかない
 - () には 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が、明示的に指定されていなくてもこっそり 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もできる

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
```

型の名前を全て一掃する

フィールドにどのようなデータが格納されているかだけに焦点を当てる

```
// 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むというイメージ
- あくまで直感の助けになるので、このように読もうという話

値が有限でないものは？

String には無限大の数が存在するが、 $2 \times \infty$ と考えて良い


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

↓ も無限大の要素数同士を掛け合わせていると考えることができる

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

他の型も一掃して読んでみる

```
// 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
- Swift を代数的に捉えることが可能になった 

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

気をつけるべきこと

- Haskell, PureScript などの言語では Void の扱いが異なる
 - Void で無人型 (uninhabited type) を表現している
 - Swift では Never に当たる
 - 他の言語と混同しないように注意しましょう

一意な値を持つ型の名前として Unit を定義

```
struct Unit {} // Void の代わりとなるものを定義  
let unit = Unit()
```

- Unit を定義したことによる利点は ↓ のように拡張できること

```
extension Unit: Equatable {  
    static func == (lhs: Unit, rhs: Unit) -> Bool {  
        return true  
    }  
}
```

- これで等価な値だけを求める関数に値を渡すことができる

Void は拡張できない

- Void で extension しようとするすると↓のようなエラーが起きる

```
Non-nominal type 'Void' cannot be extended
```

- なぜなら Void は空のタプルとして定義されている

```
typealias Void = ()
```

- Void は Swift において non-nominal types ではなく、structural types であるため、extension できない

Unit と Never の定義を並べてみる

```
struct Unit {}  
enum Never {}
```

- 「フィールドを持たない struct」と「case を持たない enum」という対称性が明らかにある
- しかし、struct には値が一つあって、enum には値がないのはなぜなのか？
- Swift の型と代数の対応関係を持って、この謎を解くための質問をすることができる

空の struct と enum にはどんな値がある？

例えば `let xs = [1, 2, 3]` のような整数の配列があったとして、
↓ のような関数を定義するにはどうすれば良いか？

```
func sum(_ xs: [Int]) -> Int {  
    fatalError()  
}  
  
func product(_ xs: [Int]) -> Int {  
    fatalError()  
}  
  
sum(xs)  
product(xs)
```

例えばこのように実装できる

```
func sum(_ xs: [Int]) -> Int {  
    var result: Int // result が定義されていないのでコンパイルはできない  
    for x in xs {  
        result += x  
    }  
    return result  
}  
  
func product(_ xs: [Int]) -> Int {  
    var result: Int // こちらも同じ。しかし、result には何を入れるべきなのか？  
    for x in xs {  
        result *= x  
    }  
    return result  
}
```

result には何を入れるべき？

- この質問に答えるためには、和と積が満たすべき性質を理解する必要がある
 - （もちろんこの問題は簡単であるため、理解せずとも解くことが可能ではある）
- そのためには、配列の連結について `sum` と `product` がどのように振る舞うかを考えれば良い

sum と product にはどう振る舞って欲しい？

普通の自然数の場合

```
sum([1, 2]) + sum([3]) == sum([1, 2, 3])  
product([1, 2]) * product([3]) == product([1, 2, 3])
```

もし空の配列を考えたら ↓ のようになるはず

```
sum([1, 2]) + sum([]) == sum([1, 2])  
product([1, 2]) * product([]) == product([1, 2])
```

代数学を使って先ほどの問題が解ける

さっきの例

```
sum([1, 2]) + sum([]) == sum([1, 2])  
product([1, 2]) * product([]) == product([1, 2])
```

このことから ↓ は強制される

```
sum([]) == 0 // 空の和型 (enum) には値がない  
product([]) == 1 // 空の積型 (struct) には値が一つしかない
```

代数学を使って簡単に解くことができた 🙌

答えがわかったので関数に適用してみる

```
func sum(_ xs: [Int]) -> Int {  
    var result: Int = 0 // 空の和型なので 0 を初期値とする  
    for x in xs {  
        result += x  
    }  
    return result  
}
```

```
func product(_ xs: [Int]) -> Int {  
    var result: Int = 1 // 空の積型なので 1 を初期値とする  
    for x in xs {  
        result *= x  
    }  
    return result  
}
```

高いレベルでの型の構文についても考えていく

- Swift の型と代数の対応関係を理解するための概念が構築できた
- より高いレベルでもその直感を活かすことができるかを見ていく
- その前に、簡単に始めていきましょう

Void について見てみる

- Void は 1 に対応し、代数の世界では 1 を掛けても何も起きない

```
// Void = 1  
// A * Void = A = Void * A
```

- 型の世界で考えると？
 - struct のフィールドで Void を使用しても基本的には型を変更せずに済むということ

Never についても見てみる

- Never は 0 に対応し、代数の世界では 0 を掛けると 0 になる
- 型の世界では ↓ のようになる

```
// Never = 0  
// A * Never = Never = Never * A
```

- つまり、型の世界において struct のフィールドに Never を入れると
その構造体自体が Never 型になってしまうという結果になる
- これは構造体を完全に消滅させることを意味する

和の場合はこのようになる

- Never の場合
 - 0 を追加する、つまり値を変更せずに残すという結果になる

```
//  $A + \text{Never} = A = \text{Never} + A$ 
```

- 1 と追加するということは、Void を追加するという意味になる

```
//  $1 + A = \text{Void} + A$ 
```

1 + A = Void + A

この式は Either を使えば ↓ のように表すことができる

```
// Either<Void, A> {  
//   case left()  
//   case right(A)  
//}
```

つまり、これは右辺に A の値が全て存在して、そこに一つの特殊な値である left(Void()) が隣接している型であると捉えられる

Swift にはこのような型が存在している

```
// Either<Void, A> {  
//   case left()  
//   case right(A)  
//}
```

これと似ている Swift の型 -> **Optional**

```
enum Optional<A> {  
    case none  
    case some(A)  
}
```

この考えを使えば？

先ほど見た↓の式は

```
// 1 + A = Void + A
```

このように表すことができる

```
// Void + A = A?
```

代数を用いることで型が簡潔になる例

```
// Either<Pair<A, B>, Pair<A, C>>
```

↓

```
// A * B + A * C
```

これは因数分解すると

$$A(B + C)$$

と表すことができる。つまり Swift では↓のように直せる

```
// Pair<A, Either<B, C>>
```

他にも見てみる

```
// Pair<Either<A, B>, Either<A, C>>
```

数学の世界では

$$(A + B)(A + C)$$

これ以上因数分解はできないため、Swift でさらに簡潔に表すことはできない。もちろん展開して考えることはできる

このように代数学はデータ構造を考えるための一つの手段となる

今日やったことは結局何の役に立つのか？

- 今日は有効な Swift でさえない疑似コードの束を並べていただけ
- 直感のためには役立つことがわかったが、エンジニアにとってメリットはあるのか？

URLSession にそのメリットは隠れている

```
URLSession.shared
.dataTask(with: url,
          completionHandler: (data: Data?,
                              response: URLResponse?,
                              error: Error?) -> Void)
```

- completionHandler は全て Optional の値を返す
- また、Swift のタプルは単なる積であるため、以下のように考える

```
// (Data + 1) * (URLResponse + 1) * (Error + 1)
```

これを展開してみる

```
// (Data + 1) * (URLResponse + 1) * (Error + 1)
//   = Data * URLResponse * Error // これは絶対起きてはいけない
//   + Data * URLResponse
//   + URLResponse * Error // これも同時に存在してはいけない (議論あり)
//   + Data * Error // これも同時に存在してはいけない
//   + Data
//   + URLResponse
//   + Error
//   + 1 // これはただの Void であり、この場合は全て nil である
```

これを考慮するとすれば、予想されるケースを越える場合、必然的に fatalError が必要となってしまう

代数学の直感を使う

直感を使って本当に欲しいものを表現してみると↓のようになる

```
// Data * URLResponse + Error
```

さっきまで使っていた型を利用すれば↓のような感じ

```
// Either<Pair<Data, URLResponse>, Error>
```

Swift の Result

Swift では、先ほどのような状態を扱えるものがある

```
// Result<(Data, Response), Error>
```

- このように callback で適切な型を使用すれば、コンパイル時に許可される無効な状態の数を大幅に減らすことができる
- callback で必要とされるロジックを単純化することができる

Result 型についてさらに考えてみる

失敗することのない特定の操作を行う場合は？

```
// Result<A, Never>
```

- ↑ でエラーケースは存在しないことが証明できた
- キャンセルをサポートする非同期 API を使っている場合は？

```
// Result<A, Error>?
```

Optional にするだけ 🤖

今日のまとめ

- 代数学を使えば、複雑さをどうにかして処理し、自分のニーズに合った型に自然に誘導できることがわかった
 - 代数的な直感が日常のコードを改善できる可能性が見えた
- 代数学はまだまだ Swift に応用して考えることができる
 - TCA の後々の章では指数・再帰などについても見ていくらしい
- 代数学と Swift の関係性が見えた気分になって、Swift の見方が広がった気がします