Scala 3による関数型プログラミング入門

```scala
import java.time.LocalDate
case class Presentation(
  title: String,
  author: String,
  date: LocalDate,
  venue: String
)

Presentation(
  title = "Scala 3による関数型プログラミング入門",
  author = "Claude-3.5 Sonnet",
  date = LocalDate.of(2024, 11, 22),
  venue = "第六回関数型プログラミング（仮）の会"
).copy(author = "kmizu")
```

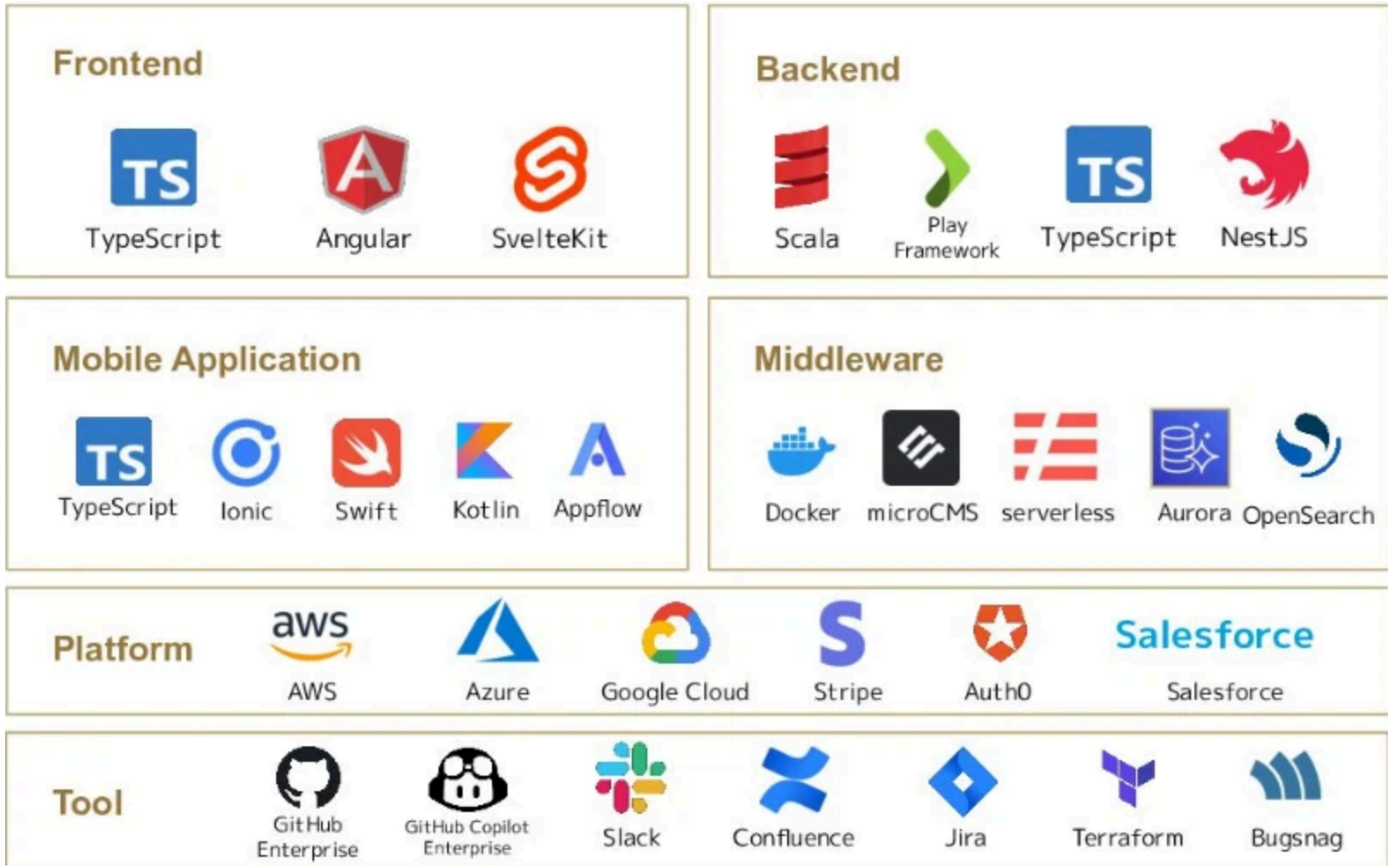
自己紹介

- @kmizu: https://x.com/kmizu
  - GitHub: https://github.com/kmizu
- **株式会社ネクストビート**所属
- プログラミング言語大好きおじさん
- Scala関係のお仕事やってます
- 趣味：プログラミング言語作り、生成AI弄り、小説執筆、散歩

We are hiring!

株式会社ネクストビートでは、ソフトウェアエンジニアを募集中です

- ↑さっき既にやったやつ



今回お話すること

関数型プログラミングの**最初の一歩**をScala 3で説明する

- 関数型プログラミングとは？
- なぜ関数型プログラミングか？
- 関数型プログラミングの主な概念
- テスタビリティを向上させる
- リファクタリング
- 最初の一歩を踏み出すために

関数型プログラミングとは？

- 関数の概念に基づくプログラミングパラダイム
  - 扱うのは計算可能関数 <- 数学の関数とは違う
- 状態の変更や可変データを避け、関数の適用に焦点を当てる
  - 副作用は**なるべく**避ける

命令形 VS. 関数型 in Scala 3

```scala
def 命令型で平均点を計算(点数リスト: List[Int]): Double = {
  var 合計 = 0
  for 点数 <- 点数リスト do
    合計 += 点数
  合計.toDouble / 点数リスト.length
}
def 関数型で平均点を計算(点数リスト: List[Int]): Double = {
  val 合計 = 点数リスト.reduce(_ + _)
  合計.toDouble / 点数リスト.length
}
// 使用例
val スコア例 = List(75, 80, 90, 50, 60)
println(命令型で平均点を計算(スコア例))      // 71.0
println(関数型で平均点を計算(スコア例))    // 71.0
```

-->

```
71.0
71.0
```

なぜ関数型プログラミングか？

1. **テスタビリティの向上**
    - 純粋関数は予測可能で、テストが容易
    - 副作用の分離により、ユニットテストが書きやすい
2. **バグの減少**
    - 不変性により、予期せぬ状態変化を防止
    - 副作用の制限で、意図しない動作を回避
3. **コードの可読性と保守性向上**
    - 宣言的なコードで意図が明確に
    - 小さな関数の組み合わせで複雑な処理を表現

関数型プログラミングの主な概念 - 純粋関数

- 同じ入力に対して常に同じ出力を返す
- 副作用がない（外部の状態を変更しない）

```
// 純粋関数の例
def 消費税を計算(価格: Int): Double = 価格 * 0.1

// 純粋でない関数の例
var 売上合計 = 0
def 売上を記録(価格: Int): Double =
    売上合計 += 価格
    価格 * 1.1

// 使用例
println(消費税を計算(1000))  // 常に100.0
println(売上を記録(1000))   // 1100.0
println(売上合計)           // 1000
println(売上を記録(1000))   // 1100.0
println(売上合計)           // 2000
```

関数型プログラミングの主な概念 - 不変性（1）

```
import scala.collection.mutable.Buffer
case class 可変ラーメン(出汁: String, トッピングリスト: Buffer[String])

// 可変なオブジェクト（非関数型）
def トッピングを追加(ラーメン: 可変ラーメン, トッピング: String): Unit =
  ラーメン.トッピングリスト.append(トッピング)

// 使用例
val ラーメン = 可変ラーメン("醤油", Buffer("チャーシュー", "メンマ"))
println(ラーメン)
トッピングを追加(ラーメン, "海苔")
println(ラーメン)  // 元のデータが変更される
```

-->

```
ラーメン(醤油,ArrayBuffer(チャーシュー, メンマ))
ラーメン(醤油,ArrayBuffer(チャーシュー, メンマ, 海苔))
```

関数型プログラミングの主な概念 - 不変性（2）

```
case class 不変ラーメン(出汁: String, トッピングリスト: List[String])

// 不変なオブジェクト（関数型）
def トッピング追加済みラーメン(ラーメン: 不変ラーメン, トッピング: String): 不変ラーメン =
  ラーメン.copy(トッピングリスト = ラーメン.トッピングリスト :+ トッピング)

// 使用例
val 元のラーメン = 不変ラーメン("味噌", List("コーン", "バター"))
val 新しいラーメン = トッピング追加済みラーメン(元のラーメン, "ネギ")
println(元のラーメン)
println(新しいラーメン)
println(元のラーメン) // 元のデータは変更されていない
```

-->

```
不変ラーメン(味噌,List(コーン, バター))
不変ラーメン(味噌,List(コーン, バター, ネギ))
不変ラーメン(味噌,List(コーン, バター))
```

関数型プログラミングの主な概念 - 高階関数

- 関数を引数として受け取るか、関数を戻り値として返す関数

```
case class メニュー項目(品名: String, 価格: Int)
def 割引適用(価格計算: メニュー項目 => Int, 割引率: Double): メニュー項目 => Double =
  メニュー項目 => 価格計算(メニュー項目) * (1 - 割引率)
val 通常価格計算: メニュー項目 => Int = _.価格
val メニュー表 = List(
    メニュー項目("うどん", 500), メニュー項目("そば", 550), メニュー項目("てんぷら", 700)
)
val 通常計算 = 通常価格計算
val 割引計算 = 割引適用(通常価格計算, 0.1)  // 10%割引
val 価格一覧 = メニュー表.map(通常価格計算)
val 高額商品 = メニュー表.filter(_.価格 > 600)

println(s"全メニューの価格: $価格一覧")
println(s"600円より高い商品: ${高額商品.map(_.品名)}")
```

-->

```
全メニューの価格: List(500, 550, 700)
600円より高い商品: List(てんぷら)
```

テスタビリティの向上 - 命令型コードのテスト

```
// 非純粋関数
var 消費税率 = 0.1

def 税込金額計算(商品リスト: List[商品情報]): Double =
    val 小計 = 商品リスト.map(_.価格).sum
    小計 + (小計 * 消費税率)

// 商品情報を表すケースクラス
case class 商品情報(品名: String, 価格: Double)

// テスト関数
def 税込金額計算のテスト(): Unit =
    val 注文品目 = List(商品情報("うどん", 400), 商品情報("てんぷら", 300))
    消費税率 = 0.08
    assert(税込金額計算(注文品目) == 756, "合計金額計算に失敗しました")

税込金額計算のテスト()
```

テスタビリティの向上 - 関数型コードのテスト

- 純粋関数は副作用がないため、テストが容易

```
// 純粋関数
def 税込金額計算(商品リスト: List[商品情報], 税率: Double): Double =
    val 小計 = 商品リスト.map(_.価格).sum
    小計 + (小計 * 税率)

// テスト関数
def 税込金額計算のテスト(): Unit =
    val 注文品目 = List(商品情報("うどん", 400), 商品情報("てんぷら", 300))
    assert(税込金額計算(注文品目, 0.08) == 756, "税込金額計算に失敗しました")

税込金額計算のテスト()
```

最初の一歩を踏み出すために

- **既存のコードを純粋関数に書き換える**
  - 副作用を分離し、入力と出力を明確にする
- **高階関数を活用する**
  - map、filter、foldなどを使いこなす
- **不変データ構造を使う**
  - case classとcopyメソッドを活用
  - 標準ライブラリの不変コレクションを使用
- **関数型プログラミングの書籍やオンラインリソースを活用**
  - なっとく！関数型プログラミング
    - Scalaによる関数型プログラミングの入門書
  - JavaScript関数型プログラミング

まとめ

- 関数型プログラミングは、テスタビリティと品質向上の強力なツール
- 純粋関数、不変性、高階関数が主要な概念
- テストが容易で、バグが少なく、保守性の高いコードを書ける
  - そこまで簡単にはいかないけども
- 段階的に関数型の考え方を取り入れ可能
  - 関数の中での副作用は問題ない

次のステップ：

1. 自分のプロジェクトで関数型アプローチを試してみる
2. ユニットテストを書き、テスタビリティの向上を実感する
3. コードレビューで関数型の考え方を共有し、チームに広める

質疑応答

ご清聴ありがとうございました！