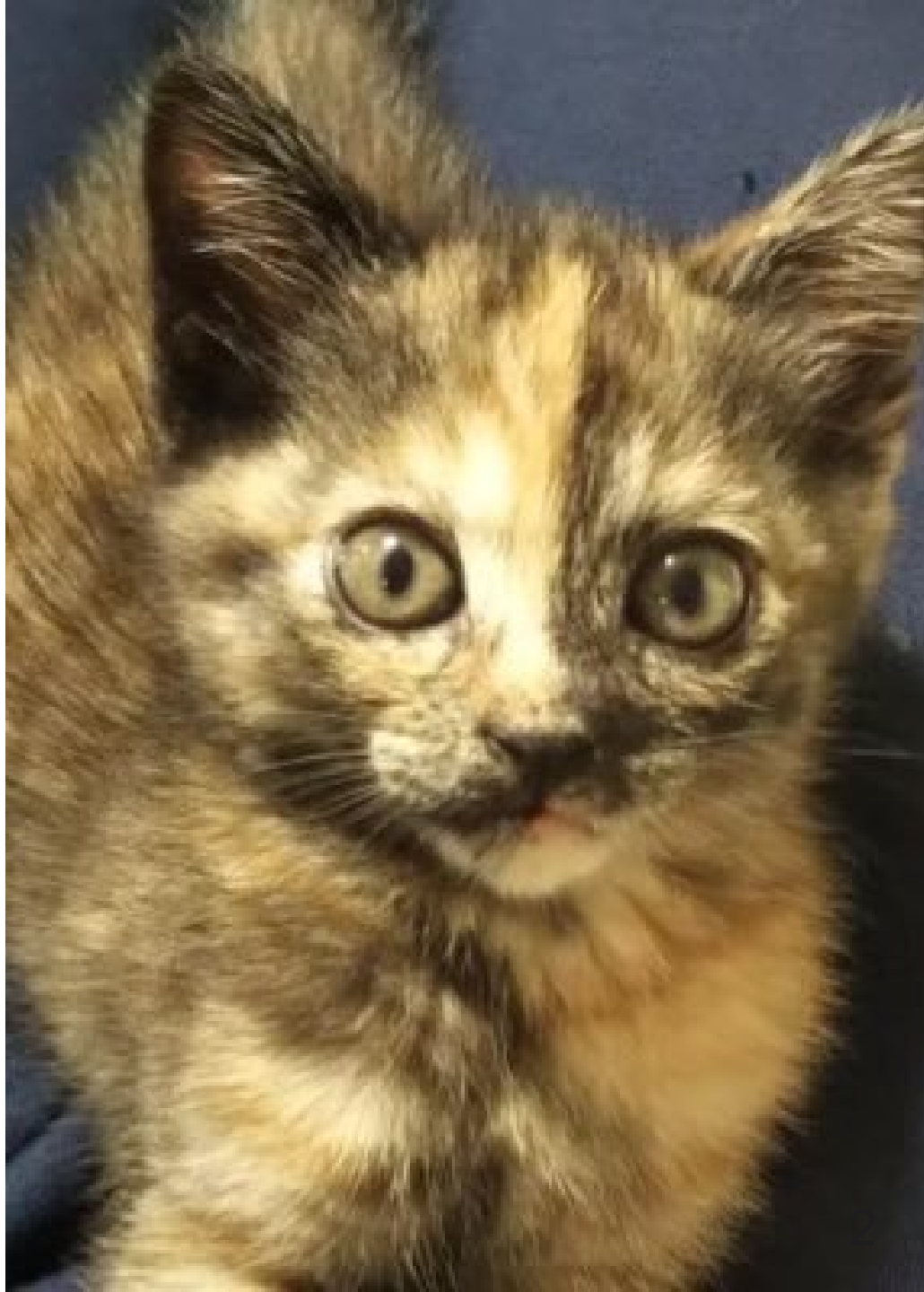
Eff（atnos-eff）による実践的なコーディング集

Kushiro Taichi at Alp,Inc.

自己紹介

- Kushiro Taichi (shiroichi315)
- アルプ株式会社 2021.06 ~
- キーワード
  - Scala
  - DDD
  - Agile
  - FP
  - Tortoiseshell



Scalebase»

Eff（Extensible Effects）とは

- 「Freer Monads, More Extensible Effects」の論文で紹介
  - Efficient Freer, Open Union
- 「Extensible Effects in Scala」- ねこはるさんの記事

Eff（Extensible Effects）の実装

- atnos-effライブラリを用いる
- 実装時の特徴
  - 複数のエフェクトをフラットに扱う
  - Interpreterによる実行の分離
  - 独自エフェクトの定義が可能

複数のエフェクトをフラットに扱う

- for式に含まれるエフェクトを型パラメータで受け取る
- スマートコンストラクタによって `Eff[R, A]` 型に変換

```
def execute[R: _option: _throwableEither: _future]: Eff[R, Int] =
  for {
    x <- fromOption[R, Int](Option(3))
    y <- fromEither[R, Throwable, Int](Right(4))
    z <- fromFuture[R, Int](Future.successful(5))
  } yield x * y - z
```

Interpreterによる実行の分離

- Open Unionによりエフェクトのスタックを定義
- Interpreterによる実行
- 実行順により型の結果が変わる - 値の結果は変わらない

```
type R = Fx.fx3[Option, ThrowableEither, TimedFuture]

val futureResult: Future[Either[Throwable, Option[Int]]] =
  Calculate
    .execute[R]                : Eff[R, Int]
    .runOption                 : Eff[Fx2[ThrowableEither, TimedFuture], Option[Int]]
    .runEither[Throwable]      : Eff[Fx1[TimedFuture], Either[Throwable, Option[Int]]]
    .runAsync                  : Future[Either[Throwable, Option[Int]]]
```

独自エフェクトの定義が可能

- エフェクト定義
- スマートコンストラクタ
- Interpreter - 差し替え可能

```
trait MyError
type MyErrorEither[A] = MyError Either A
type _myErrorEither[R] = MyErrorEither /= R
type _myErrorEitherMember[R] = MyErrorEither <= R
```

コーディング集

実務に近いコードを紹介していきます

実務のコードでは...？

- 例として `Factory` や `Repository` 内でエフェクトを用いることも
- `Eff[R, A]` 型を複数のメソッドで引き回すことが多い

```
def execute[R: _dbio: _idgenm: _myErrorEither](
    args: CreateBillingUseCaseArgs
): Eff[R, CreateBillingUseCaseResult] =
  for {
    billing <- billingFactory.create[R](args.billingDate)
    storedBilling <- billingRepository.store[R](billing)
  } yield CreateBillingUseCaseResult(storedBilling)
```

map

- A型の値に関数を適用

```
for {
  providerIdAndBillings <- billingRepository
    .findByIds[R](args.billingIds)                    // Eff[R, Seq[Billing]]
    .map(billings => (args.providerId, billings)) // Eff[R, (ProviderId, Seq[Billing])]
} yield GetProviderIdAndBillingsUseCaseResult(providerIdAndBillings)
```

pureEff

- pureな `A` 型の値を `Eff[R, A]` 型に変換

```
for {
  maybeContract <- contractRepository.findById[R](args.contractId)
  billings <- maybeContract match {
    case Some(contract) =>
      billingRepository.findByContractId[R](contract.id)
    case None => Nil.pureEff[R] // Eff[R, Seq[Billing]]
  }
} yield GetBillingsByContractIdUseCaseResult(billings)
```

traverseA

- `Eff[R, A]`型を返す処理を走査する
- `sequenceA`も存在

```
for {
  dataModels <- {
    domainModels.toList // List[DataModel]
      .traverseA(convertToDataModel[R]) // Eff[R, List[DataModel]]
  }
} yield dataModels
```

runPure

- `Eff[R, A]` から `A` を `Option` で囲んで取り出す
  - 全て実行済みであれば `Some`、未実行があれば `None`
- 結果は式に依存するのでmockをスキップして `Some` を取得できる

```
type R = Fx.fx3[Option, ThrowableEither, TimedFuture]

val result: Option[Option[Int]] =
  Calculate
    .execute[R]
    .runOption
    .runPure // None
```

コーディングレベルでの悩みポイント

flatMapのコンテキスト指定

- for式の最初の処理の型にコンテキストが束縛される

```
for {
  billing <- billingFactory.create(args.billingDate)
  storedBilling <- billingRepository.store(billing)(...)
} yield CreateBillingUseCaseResult(storedBilling)
```

```
Type mismatch.
Required: Eff[Fx1[IdGenM],                B_                    ]
Found:    Eff[Fx1[ DBIO  ],CreateBillingUseCaseResult]
```

どのタイミングでEff[R, A]に変換するか

- テストが煩雑になることも
- ドメインロジックはピュアに書く？

```
def calculateBillingPrice[R: _myErrorEither]: Eff[R, Price] =
  Price(100).pureEff[R]

type R = Fx.fx1[MyErrorEither]

val result = calculateBillingPrice[R].runMyEither.runPure.value

result mustBe Right(Price(100))
```

option, list エフェクトを用いるか

- `runXxx` によって実行順を制御
- 処理に対する実行順の制御に気をつける

実行の配線問題

- 一つのエフェクト実行で別のエフェクトが展開されることも
- 展開される順序通りの実行が必要
- 例: `TransactionTask` -> `MyError` ・ `Task`

実装をカプセル化はするが実装知識も大事

- DBのコネクション、セッションどうなってる？

とはいえ個人・会社的にもEffはポジティブ

ドメインに集中できる

- 実装がカプセル化されユースケースの見通しが良くなる
- シグネチャに現れるエフェクトによって可読性が増す
  - ユースケースに対して発生するエフェクトが把握できる

学習コストが低い（という見方もできる）

- モナトラの型合わせの方が脳のメモリを使う印象
- 覚えるAPIの数は少ない
- チーム内に理論含め詳しい人は必要

アルプ独自の実装

独自エフェクトが多数用いられている

- 独自エラー型
- ID生成
- DBトランザクション（doobie公式） etc.

独自エラーエフェクトに対するスマートコンストラクタを多数定義

```
def fromPpError[R, A](ea: PpError Either A)(implicit member: _ppErrorEither[R]): Eff[R, A] =
  org.atnos.eff.all.fromEither[R, PpError, A](ea.fold(Left.apply, Right.apply))

def fromPpErrorRight[R: _ppErrorEither, A](a: A): Eff[R, A] =
  org.atnos.eff.all.right(a)

def fromPpErrorLeft[R: _ppErrorEither, A](e: PpError): Eff[R, A] =
  org.atnos.eff.all.left(e)

def fromPpErrorLeftIf[R: _ppErrorEither](condition: Boolean)(e: => PpError): Eff[R, Unit] =
  if (condition) fromPpErrorLeft(e) else fromPpErrorRight(())

def optionErrorEither[R: _ppErrorEither, A](option: Option[A], e: => PpError): Eff[R, A] =
  org.atnos.eff.either.optionEither(option, e)
```

To Be Continued...

AlpのEff独自Effect集

14:35 - 14:55 | Japanese

Tsubasa Matsukawa