

Scalaコードとともに考えるドメインモデリング

Scala福岡 2019

かとじゅん(@j5ik2o)



自己紹介

- Chatwork テックリード
- github/j5ik2o
 - [scala-ddd-base](#)
 - [scala-ddd-base akka-http.g8](#)
 - [reactive-redis](#)
 - [reactive-memcached](#)
- 翻訳レビュー
 - [エリックエヴァンスのドメイン駆動設計](#)
 - [Akka実践バイブル](#)



最近の発表ネタ

1. ドメインモデリングの始め方 - AWS Dev Day Tokyo 2018
 - ドメインオブジェクトの発見・実装・リファクタリングの方法論をカバー
2. Scalaでのドメインモデリングのやり方 - Scala関西Summit 201

アジェンダ

- 同じネタはやりません
 - 過去のネタについて議論したいなら、懇親会で捕まえてください！
1. ドメインイベントを使ったモデリングと実装
 2. 集約を跨がる整合性の問題

ドメインモデリングの前提

- 要件定義が済んでいること
 - RDRA(コンテキストモデル、要求モデル、業務フロー/利用シーン、ユースケースモデル、概念モデル, etc)
 - ユーザーストーリマッピング

対象のドメイン

ウォレットサービス

- ユーザが電子マネーをウォレットという概念で管理できるサービス
 - Kyash のようなサービスです
- APIサーバを開発する想定で考える

ウォレットサービスの概要

- 主な機能
 - ユーザがサインアップすると電子マネーを管理するウォレットが一つ作成される
 - ユーザがクレジットカードなどからウォレットにチャージできる
 - ユーザ間で請求や支払いができる（飲み会の割り勘のときに使える）
 - 料金プランはパーソナルプランとファミリープランがあり、プラン変更もできる

ドメインモデルの輪郭を捉える

想定するユースケース

- アクター
 - ユーザ
- ユースケース
 - ユーザが、ウォレットにチャージできる（クレジットカードなど）
 - ユーザが、他のウォレットに請求する
 - ユーザが、他のウォレットに支払う
 - ユーザが、他のウォレットからの請求を受け取る
 - ユーザが、他のウォレットからの支払を受け取る
 - ユーザが、ウォレットの残高を確認できる
 - ユーザが、支払履歴を確認する
 - ユーザが、請求履歴を確認する
 - ユーザが、プランをパーソナルプランもしくはファミリープランに切り替える
 - ユーザが、ウォレットを追加/削除/一覧確認できる

FYI: ユースケース記述

ユーザが、他のウォレットに支払う

- 晴れの日コース
 - ユーザは、支払ボタンをクリックする
 - システムは、支払画面を表示する
 - ユーザは、支払画面上に支払(支払元ウォレットID, 支払先ウォレットID、名目、金額)を入力し、支払ボタンをクリックする
 - システムは、受け取った支払から以下を行う
 - 支払元ウォレットIDをユーザが所有しているか確認する
 - 支払元ウォレットと支払先ウォレットをストレージから読み出す
 - 支払元から支払先への支払を、支払元ウォレットに履歴として残す(To側)

• 雨の日コース

- 上記がうまくいかないときはどうなる？
- 残高がマイナスになる請求はどうするのかなど

ユースケースの文言だけだと詳細がイメージできない場合は、
ユースケース記述を書いた方がよい。**雨の日の状況を考えると
ビジネスルールも見えてくる**

ビジネスルールに注目する

ユースケース記述を書き起こしながら、ビジネスルールについても議論する

- 支払・請求は、誰から誰へ、名目、いくらかが分かる必要がある
- 支払には請求があるものとないものがある。請求がなくても支払はできる
- 残高が0になる支払は行えるのか？行えないのか？
- プランでできること・できないこととは？パーソナルは1ウォレットのみ、ファミリは10ウォレットのみ
- 契約ってどう表現するの？そもそも契約とは？

Part-1

ドメインイベントを使ったモデリングと実装

ドメインモデルと集約を明らかにする

FYI: 関心事の分離

- ヒト
 - 個人、企業、担当者、など
- モノ
 - 商品、サービス、店舗、場所、権利、など
- コト(ドメインイベント)
 - 予約、注文、支払、出荷、キャンセル、など

(出典:現場で役立つシステム設計の原則)

コトからモノを整理する

コトに注目することで次の関係も明らかになる

- コトはヒトとモノとの関係として出現する(だれの何についての行動か)
- コトは時間軸に沿って明確な前後関係を持つ

(出典:現場で役立つシステム設計の原則)

Greg Young氏考案のEvent Sourcingもモノではなくコトをモデリングの主役と位置づけている

コト=ドメインイベント

FYI: ドメインイベントの現れ方

「...するときに」や「...した場合」という用語法に現れる

- XXXするときに
- YYYだったら気にしないが、もしXXXだったら注意が必要
- XXXの場合は、把握しておく必要がある
- XXXが発生した場合に

過去形で表現される関心時がそのままドメインイベントとして表現される

ドメインイベントから集約を見つける手順

1. ユースケースからドメインイベントを洗い出す
2. そのドメインイベントを発生させる元となるコマンドを洗い出す
3. コマンドを発行するアクターを洗い出す(またそのときの入力となるリードモデルや条件も洗い出す)
4. そのコマンドを受け取って、副作用を起こし、ドメインイベント発行する集約の名前を付ける

ドメインイベントを見つける

- ユースケース(記述含む)からドメインイベントを見つける
 - ユーザが、ウォレットにチャージできる = MoneyDeposited
 - ユーザが、他のウォレットに請求する = MoneyRequested
 - ユーザが、他のウォレットに支払う = MoneyPaid
 - ユーザが、他のウォレットからの請求を受け取る = MoneyRequestReceived
 - ユーザが、他のウォレットからの支払を受け取る = MoneyPaymentReceived
- ドメインイベントの抜け漏れに注意
 - タイムラインにドメインイベントを並べて確認
 - 前後の依存関係がないかも確認("申請した"のあとに"承認した"があるかなど)
 - ドメインイベントの重複が起きないか(集約の不变条件が壊れる原因になる)

ドメインイベントの例

- チャージイベント(MoneyDeposited)
 - チャージ元口座(クレジットカードなど)
 - チャージ金額
 - チャージ日時
- 請求イベント(MoneyRequested)
 - 請求元のウォレットID
 - 請求先のウォレットID
 - 名目(Optional)
 - 金額
 - 請求日時
- 支払イベント(MoneyPaid)
 - 支払元のウォレットID
 - 支払先のウォレットID
 - 名目(Optional)
 - 金額
 - 支払日時
 - 請求ID(Optional)

コマンドを洗い出す

- ドメインイベントが分かればコマンドも分かる場合が多い
- チャージコマンド(DepositMoney)
 - チャージ元口座
 - 金額
 - 日時
- 請求コマンド(RequestMoney)
 - 請求元のウォレットID
 - 請求先のウォレットID
 - 名目(Optional)
 - 金額
 - 請求日時
- 支払コマンド(PayMoney)
 - 支払元のウォレットID
 - 支払先のウォレットID
 - 名目(Optional)
 - 金額
 - 支払日時
 - 請求ID(Optional)
- コマンドは、アクターではメッセージに、クラスではメソッドに対応する

アクターを洗い出す

- ユースケースを書いていたらすぐにわかる。今回の場合はすべてユーザ
- ワークフローがあるようなドメインでは、コマンドを発行する、複数のアクターを考慮する

コマンド発行条件とは？

アクターはどんなときに、そのコマンドを発行するのか？

- リードモデルが判断条件に一致したときに発行される
- アクターがシステムでも、人でも同じ

FYI: リードモデルの例

発生したイベントからDTOを作る

- 取引DTO
 - 取引ID
 - 発生日時
 - 取引種別(チャージ/請求/支払)
 - リソースID(Optional)
 - FromウォレットID(Optional)
 - Fromユーザーアカウント名(Optional)
 - Toウォレット元ID
 - Toユーザーアカウント名
 - 適用
 - 金額

集約の名前を探す(命名)

- コマンドを受け付けて、副作用を起こし、ドメインイベントを発生させる集約をイメージする
 - DepositMoney -> 副作用 -> MoneyDeposited
 - ReuestMoney -> 副作用 -> MoneyRequested
 - PayMoney -> 副作用 -> MoneyPaid
- コマンドはメソッド名になる
- 副作用を扱うオブジェクト(集約)の名前を見つける
 - Wallet集約とする
- Walletの責務も定義する
 - 電子マネーのチャージ・請求・支払の管理

ドメインオブジェクトと集約を実装する

Walletはドメインイベントを履歴として扱う

FYI: ドメインイベントの利用例

- Event Message Pattern
 - 出来事をPub/Subを使って通知する仕組み

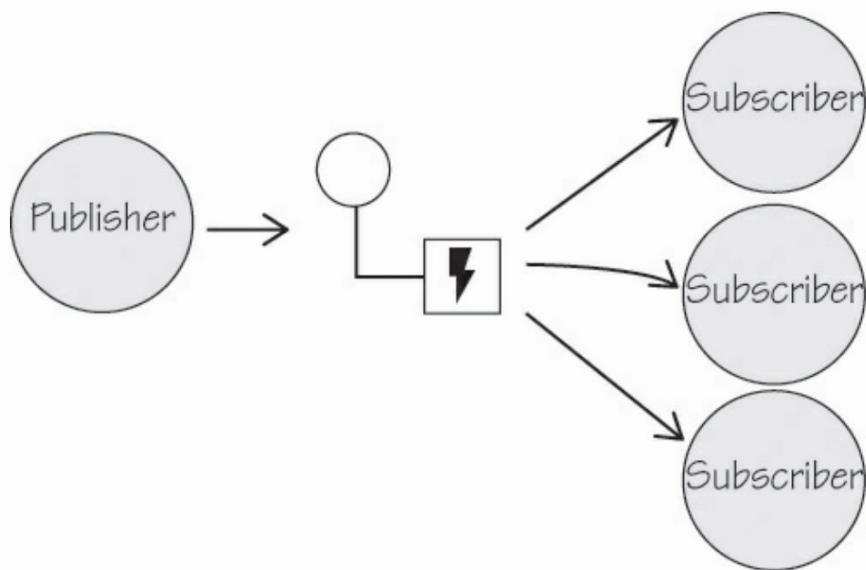
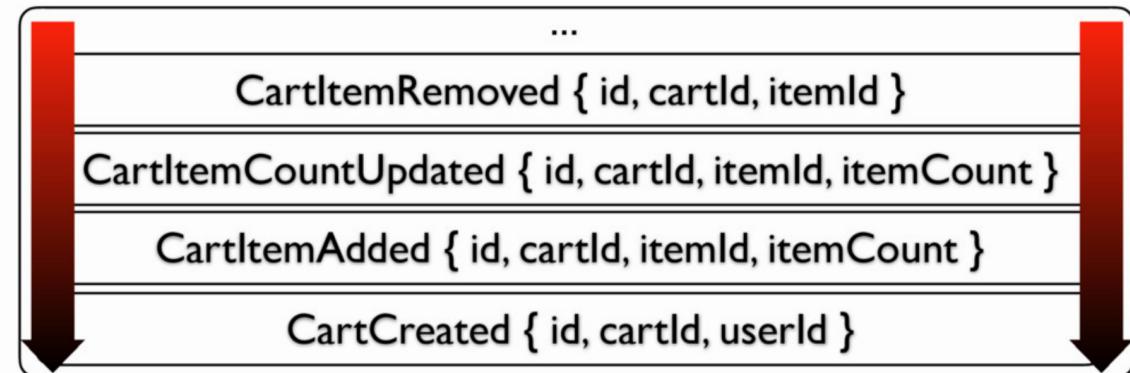


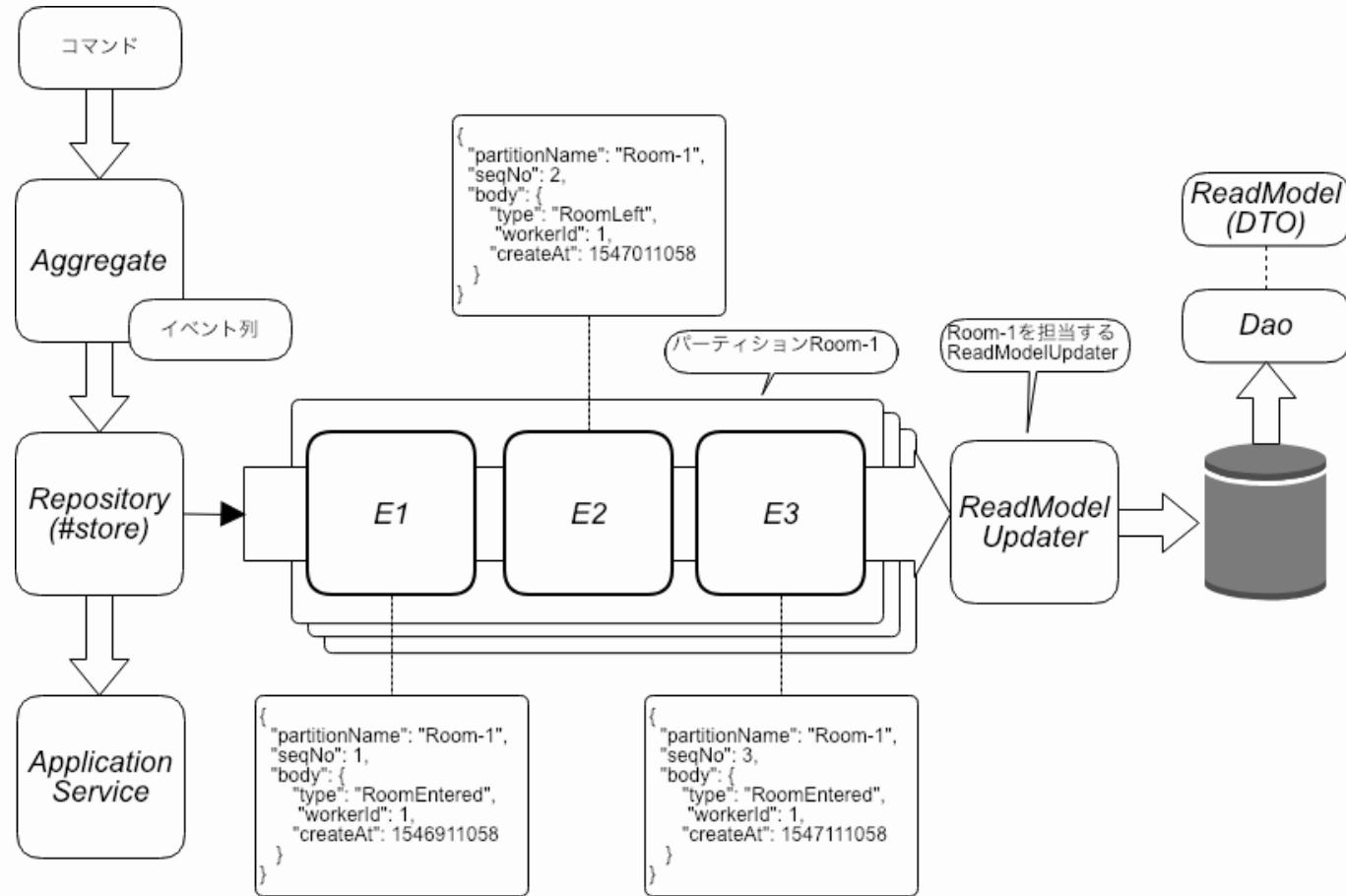
Figure 6.3 Using an Event Message, a Publisher may notify multiple Subscriber actors about something that happened in the domain model.

- Event Sourcing(ES)
 - 状態を永続化するのではなく、発生する出来事をすべて永続化(追記のみ)する



簡易的なEvent Sourcingを実装する

全体象



Wallet(1/2)

- WalletEventの集合はファーストクラスコレクション(WalletEvents)に委譲する
- 最新の状態(balanceなど)も必要。不变なイベントから導出する想定
- コマンドをメソッドとして展開する

```
trait Wallet {  
    def events: WalletEvents // ドメインイベントのコレクション  
  
    def id: WalletId  
    def userAccountId: UserAccountId  
    def balance: Money  
    def createdAt: Timestamp  
  
    // チャージする  
    def deposit(from: MoneyResource, money: Money, createdAt: Timestamp = ZonedDateTime.now()): Result[Wallet]  
    // 請求する  
    def request(toId: WalletId, money: Money, createdAt: Timestamp = ZonedDateTime.now()): Result[(Wallet, WalletEventId)]  
    // 支払う  
    def pay(toId: WalletId,  
           money: Money,  
           requestEventId: Option[WalletEventId] = None,  
           createdAt: Timestamp = ZonedDateTime.now()): Result[Wallet]  
    // ...  
}
```

Wallet(2/2)

- 請求された側と支払った側のメソッドも必要
- requestした後にreceiveRequestするのを忘れると問題...。

```
trait Wallet {  
    // ...  
    // 請求される  
    def receiveRequest(fromId: WalletId, money: Money, createdAt: Timestamp = ZonedDateTime.now()): Result[Wallet]  
    // 支払われる  
    def receivePayment(fromId: WalletId,  
                      money: Money,  
                      requestEventId: Option[WalletEventId],  
                      createdAt: Timestamp = ZonedDateTime.now()): Result[Wallet]  
}
```

WalletEvent(1/2)

- ケース漏れを防ぐためにsealedする

```
sealed trait WalletEvent extends Event {
    val id: WalletEventId
    val walletId: WalletId
    val userAccountId: UserAccountId
    val createdAt: Timestamp
}

// チャージした
case class MoneyDeposited(id: WalletEventId,
                           walletId: WalletId,
                           userAccountId: UserAccountId,
                           from: MoneyResource,
                           money: Money,
                           createdAt: Timestamp)
    extends WalletEvent

// 請求した
case class MoneyRequested(id: WalletEventId,
                          walletId: WalletId,
                          userAccountId: UserAccountId,
                          toId: WalletId,
                          money: Money,
                          createdAt: Timestamp)
    extends WalletEvent

// 支払った
case class MoneyPaid(id: WalletEventId,
                      walletId: WalletId,
                      userAccountId: UserAccountId,
                      toId: WalletId,
                      money: Money,
                      requestEventId: Option[WalletEventId],
                      createdAt: Timestamp)
    extends WalletEvent
```

WalletEvent(2/2)

- 支払われた=MoneyPaymentReceivedはMoneyDepositedに似ているが意味が異なるので別の型とした

```
// 請求された
case class MoneyRequestReceived(id: WalletEventId,
                                 walletId: WalletId,
                                 userAccountId: UserAccountId,
                                 fromId: WalletId,
                                 money: Money,
                                 createdAt: Timestamp)
    extends WalletEvent

// 支払われた
case class MoneyPaymentReceived(id: WalletEventId,
                                 walletId: WalletId,
                                 userAccountId: UserAccountId,
                                 fromId: WalletId,
                                 money: Money,
                                 requestEventId: Option[WalletEventId],
                                 createdAt: Timestamp)
    extends WalletEvent
```

WalletEvents

- ファーストクラスコレクションの例
- 煩雑さを軽減する
- ユビキタス言語でメソッドを表現することで、実装を読まなくても設計が予測可能になる
- breachEncapsulationOfEventsは妥協して使えるが、乱用するとドメインロジックがドメイン層から流出するので注意

```
case class WalletEvents(breachEncapsulationOfEvents: Seq[WalletEvent]) {  
    // SeqじゃなくてStreamがよかもしれない  
    private val values = breachEncapsulationOfEvents  
    require(values.nonEmpty)  
    def walletId: WalletEventId      = values.head.id  
    def userAccountId: UserAccountId = values.head.userAccountId  
    def createdAt: Timestamp         = values.head.createdAt  
    def add(other: WalletEvents): WalletEvents = WalletEvents(values ++ other.values)  
    def add(other: WalletEvent): WalletEvents = add(WalletEvents(Seq(other)))  
}
```

イベントから状態を導出する

- 単純な実装としては、すべてのドメインイベントがコンストラクタに渡される。寿命が長い場合は最適化が費用。
- コマンドに反応する際、必ずしもすべてのイベント列が必要ではない。最新のスナップショット + 差分ドメインイベントの集合でよい場合がほとんど。この場合、最適化可能になる

```
case class WalletImpl(events: WalletEvents, snapshotBalance: Money = Money.zero(Money.JPY)) extends Wallet {  
  
    override val id: WalletEventId          = events.walletId  
    override val userAccountId: UserAccountId = events.userAccountId  
    override val createdAt: Timestamp        = events.createdAt  
  
    override lazy val balance: Money = {  
        // FIXME: ファーストクラスコレクションのリファクタ  
        events.breachEncapsulationOfEvents.foldLeft(snapshotBalance) {  
            case (m, MoneyDeposited(_, _, _, _, _, money, _)) =>  
                m.plus(money)  
            case (m, MoneyPaid(_, _, _, _, _, money, _, _)) =>  
                m.minus(money)  
            case (m, MoneyWasPaid(_, _, _, _, _, money, _, _)) =>  
                m.plus(money)  
        }  
    }  
    // ...  
}
```

振る舞いの実装

- ・ クライアントが契約を満たしているか確認する
- ・ イベントを生成して、新しいインスタンスのイベントの最後に追加する

```
override def pay(toId: WalletId,
                 money: Money,
                 requestEventId: Option[WalletEventId] = None,
                 createdAt: Timestamp): Result[Wallet] = money match {
    case m if m.currency != balance.currency =>
      Left(new InvalidCurrencyError("Invalid currency"))
    case m if balance.minus(money).isNegative =>
      Left(new InvalidBalanceError(s"fromId: $id, toId: $toId, money: $money"))
    case _ =>
      val event = MoneyPaid(ULID.random(), id, userAccountId, toId, money, requestEventId, createdAt)
      Right(
        copy(
          events = events.add(event)
        )
      )
}
```

ドメインサービスの例

- 支払は、支払う側と支払われる側の状態がある。どちらが欠けることは許されない(不变条件)
- 現状のWalletの実装はよくないが、二つのインスタンスに跨がる状態の制御は実装しにくいし、理解もしにくい
 - receivePaymentメソッドをprivate化し、paymentメソッド内部でコールする方法(読む側に負担が大きそう)
 - paymentメソッドとreceiveRequestメソッドをパッケージプライベートにして、ドメインサービスとして実装する方法

```
object PaymentService {  
  
  case class FromTo(from: Wallet, to: Wallet)  
  
  def execute(from: Wallet,  
             to: Wallet,  
             money: Money,  
             requestEventId: Option[WalletEventId] = None,  
             createdAt: ZonedDateTime = ZonedDateTime.now()): Result[FromTo] = {  
    for {  
      newFrom <- from.pay(to.id, money, requestEventId, createdAt)  
      newTo   <- to.wasPaid(from.id, money, requestEventId, createdAt)  
    } yield FromTo(newFrom, newTo)  
  }  
}
```

リポジトリの例

- ・ イベントの永続化はEventStoreに委譲
- ・ イベントの永続化にコールバックする

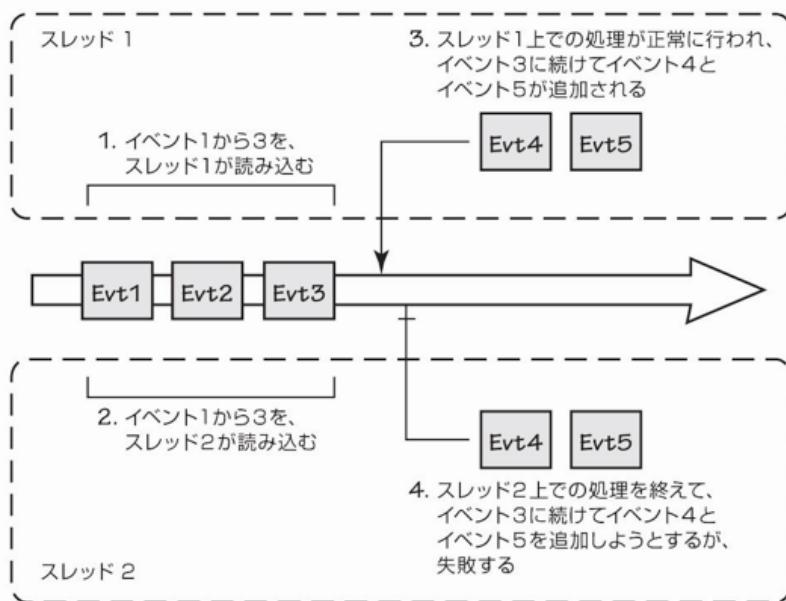
```
class WalletRepositoryOnMemory extends WalletRepository[Task] {  
    private val listeners: mutable.Seq[WalletEvent => Unit] = mutable.Seq.empty  
    private val eventStore = new EventStoreOnMemory[WalletEvent]()  
    override def addListeners(listeners: Seq[WalletEvent => Unit]): Unit = this.listeners ++ listeners  
  
    override def store(aggregate: Wallet): Task[Unit] =  
        eventStore.add(aggregate.id, aggregate.events.breachEncapsulationOfEvents).doOnFinish {  
            case None =>  
                fireEvents(aggregate)  
            // ...  
        }  
  
    override def resolveById(id: WalletId): Task[Wallet] = eventStore.iterator(id).map { events =>  
        Wallet(WalletEvents(events.toSeq)) // Streamのほうがよいかも  
    }  
    // ...  
}
```

EventStoreの例

- イベント列には順序を保証する概念が必要

```
class EventStoreOnMemory[E <: Event] extends EventStore[E] {  
    private val partitions: mutable.Map[String, mutable.Queue[E]] =  
        mutable.Map.empty  
  
    override def add(aggregateId: String, event: E): Task[Unit] = Task {  
        val queue = partitions.getOrElseUpdate(aggregateId, mutable.Queue.empty)  
        queue.enqueue(event)  
    }  
  
    override def add(aggregateId: String, events: Seq[E]): Task[Unit] =  
        Task.sequence(events.map(event => add(aggregateId, event))).map(_ => ())  
  
    override def fetch(aggregateId: String): Task[E] = Task {  
        val queue = partitions.getOrElseUpdate(aggregateId, mutable.Queue.empty)  
        queue.dequeue()  
    }  
  
    override def iterator(aggregateId: String): Task[Iterator[E]] = Task {  
        val queue = partitions.getOrElseUpdate(aggregateId, mutable.Queue.empty)  
        queue.iterator  
    }  
}
```

FYI: 集約の更新が競合するケース



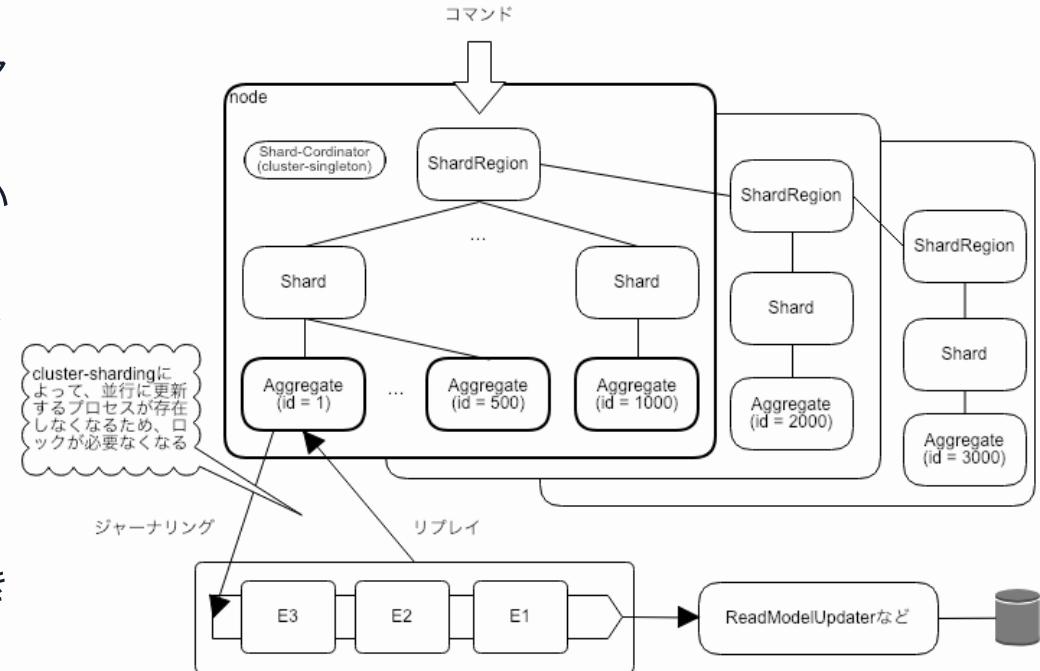
図A-10 : A+ES を使うように作られた集約のインスタンスに対して、二つのスレッドからのアクセスが競合する例

集約のイベントストリームに対して、複数のスレッドから同時にアクセスがあって、同時に読まれることもありえる。そうなると、並行処理の衝突が発生する可能性が出てくる。これを無視して放置していると、集約が不正な状態になってしまい問題が発生する。二つのスレッドが、イベントストリームを同時に変更しようとした場合を考えてみよう。そのようすを図A-10に示す。(実践ドメイン駆動設計より)

対策は、Evt4の追加時に同じキーがあれば、Evt4,Evt5の追加を失敗する(しかもこれがアトミックにできないと意味がない)というもののだが、ストレージの機能によって実装が難しい場合がある。多分、Kafkaは無理...。 つまるところ、状態を一次的にしか持たないアプリケーションでは、EventStore#addにロックが必要...。

FYI: なぜAkkaではESがうまくいくか

- 集約をイベント列から再生して、クラスター上で唯一のアクター(=集約アクター)として常時起動
 - イベントストリームに書き込むスレッドが他にいないので競合問題が回避できる
- メッセージはShardRegionをプロキシーとしてルーティングされる
- 集約がアクターが不变条件を維持する(並行制御からのロックなども含む)
- コマンドに反応した集約アクターは、イベントを追加書き込み
- 「Akka実践バイブル」の「第15章 アクターの永続化」あたりを読むとよい



Part-2

集約を跨がる整合性問題について

集約を跨がる整合性の問題

- ユースケース
 - ユーザが、プラン(Plan)をパーソナルプランもしくはファミリープランに切り替える
 - ユーザが、ウォレットを追加/削除/一覧確認できる
- 契約(Contract)に応じて、Walletのインスタンス数を制限しなければならない(**Wallet上限ルール**)
- ネタ元
 - 2018-12-03 集約の境界と整合性の維持の仕方に悩んで2ヶ月ぐらい結論を出せていない話
 - 「集約の境界と整合性(略」に対して頂いたアイデアの分類と現状での僕の回答らしきもの

PlanとContract

```
// 料金プラン
sealed trait Plan extends EnumEntry {
    def maxWallets: Int
    def minWallets: Int
}

object Plan extends Enum[Plan] {
    override def values: immutable.IndexedSeq[Plan] = findValues

    case object Personal extends Plan {
        override val minWallets: Int = 1
        override val maxWallets: Int = 1
    }
    case object Family extends Plan {
        override val minWallets: Int = 1
        override val maxWallets: Int = 10
    }
}

// 契約を表す集約
case class Contract(id: ContractId, ownerId: UserAccountId, plan: Plan, createdAt: Timestamp, updatedAt: Timestamp) {
    // ...
}
```

Planを超えてWalletが登録できる問題

- 以下のロジックを複数スレッドなどで同時に実行すると、整合性が破綻します
- 異なる集約では異なるトランザクション境界を持つため、アトミックなI/Oがそもそもできない(結果整合)

```
object WalletApplicationService {  
  
  def addWallet(wallet: Wallet): Result[Unit] = {  
    val contract = contractRepository.resolveById(contractId)  
    val walletCount = walletRepository.countEnabledByContractId(contractId)  
    if (contract.plan.maxWallets > walletCount)  
      Right(walletRepository.store(wallet))  
    else  
      Left(new WalletLimitOverError(s"contract = $contract, wallet = $wallet"))  
  }  
}
```

解決策とそのつらさ...

1. 集約をマージして1つのトランザクション境界にする
 - たとえば、ContractとWalletをマージする。Contract has Wallets ...
 - Walletsの参照を得るのに、いちいちContractを特定しなければならない
 - ロックが競合する頻度があがる
 - DB I/Oのパフォーマンス劣化
2. 集約をマージせずに、独立した集約間は結果整合性を用いる
 - 今回の場合はこちら。どうにもならない場合もある...。
3. アンチパターン: 集約間を跨がったトランザクション制御を用いる

```
def addWallet(wallet: Wallet): Result[Unit] = DB.localTx { ... }
```

- 1集約1トランザクション境界のルールが破綻し、ユースケースによって集約の整合性境界がバラバラになる。デッドロックの温床にもなりやすい
4. ユースケースやビジネスルールの見直し
 - 利害関係者の合意が難しい場合も...。

どうするといいのか

1. 万能ではないが結果整合性でカバーしつつ、ある程度の不整合を許すようにする
2. 一次的な破綻を受け入れて、結果整合を利用する
 - 無効なWalletを追加して、別プロセスで有効化する

今回は前者の例を示します

結果整合性をうまくつかう方法の一つ

Contractが有効なWalletIdsを持つ

```
case class Contract(id: ContractId, ownerId: UserAccountId,
  plan: Plan, walletIds: WalletIds, createdAt: Timestamp, updatedAt: Timestamp) {

  def addWalletIds(walletIds: WalletIds): Result[Contract] = {
    if (!AddWalletSpec(this, walletIds)) // 仕様パターン
      Left(new WalletLimitOverError(s"walletIds = $walletIds"))
    else
      Right(
        copy(
          walletIds = this.walletIds.add(walletIds)
        )
      )
  }

  def removeWalletIds(walletIds: WalletIds): Result[Contract] = ???
}
```

FYI: 仕様パターン:WalletSpec

- 暗黙的な概念を明示的にするためのパターン
- 複雑なルールはor, and, not演算子で合成して作れるようにするよい

```
// ウォレット追加仕様
object AddWalletSpec extends ((Contract, WalletIds) => Boolean) {
  override def apply(contact: Contract, walletIds: WalletIds): Boolean =
    contact.plan.maxWallets > (contact.walletIds.size + walletIds.size)
}

// ウォレット削除仕様
object RemoveWalletSpec extends ((Contract, WalletIds) => Boolean) {
  override def apply(contact: Contract, walletIds: WalletIds): Boolean =
    contact.plan.minWallets < (contact.walletIds.size - walletIds.size)
}
```

AddWalletUseCase

- Contractを更新してから、Walletを追加するようにすれば、リカバリ処理はいらない
- 逆の書き込み操作(Walletを追加してからContractを更新)は、リカバリ(Walletの削除)が必要になる

```
case class AddWalletDto(userAccountId: UserAccountId)

class AddWalletUseCase[M[_]](file:///Users/j5ik2o/Sources/slides/scala-fukuoka-2019/contractRepository: ContractRepository[M], walletR
    implicit ME: MonadError[M, Error]
) {

  def execute(dto: AddWalletDto): M[Unit] = {
    for {
      contract <- contractRepository.resolveByUserAccountId(dto.userAccountId)
      newWallet   <- ME.pure(Wallet(ULID.randomUUID(), contract.id, contract.ownerId))
      newContract <- contract.addWalletIds(newWallet.id) match { // ビジネスルールを強制
        case Right(v) => ME.pure(v)
        case Left(e)   => ME.raiseError(e)
      }
      _ <- contractRepository.store(newContract) // 楽観ロック
      _ <- walletRepository.store(newWallet)
    } yield ()
  }
}
```

このようなトランザクションを跨ぐ整合性の問題を調停する役割を**Process Manager**や**Saga**と読んだりする

まとめ

- ドメインイベントは、ドメインの分析と実装の両方で使えるツール
- 集約を跨ぐ整合性の問題は難しいが、解決方法がないわけではない

一緒に働くエンジニアを募集しています！

<http://corp.chatwork.com/ja/recruit/>

