

Scalaの関数型ライブラリを活用した型安全な 業務アプリケーション開発

関数型まつり 2025

2025年6月15日

自己紹介



Tomoki Mizogami (ともちんと呼んでください)

株式会社ネクストビート
Webエンジニア (Scala歴6年)、 PdM

- 新規事業開発・共通ライブラリ開発・認証基盤等を担当
- 好きな言語 スペイン語 
- Scalaで始める圏論入門

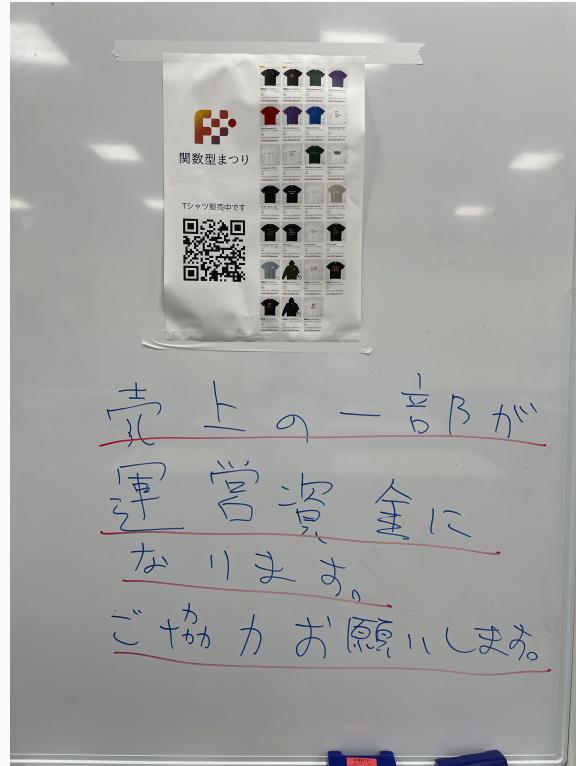
[@taretmch](#) 資料データ / デモコード



本日の資料

関数型まつりのTシャツ買いましたか？

QR チャンス！



会社紹介

株式会社ネクストビート (シルバースポンサー)

ミッション

「人口減少社会において必要とされるインターネット事業を創造し、ニッポンを元気にする。」

事業領域

- ライフィベント: 保育園/法人向けSaaS、ベビーシッター事業、子育てメディア、求人情報・転職支援 etc.
- 地方創生: 宿泊業界向け求人情報・転職支援 etc.
- グローバル: シンガポール事業、台湾事業 etc.

技術スタック

- Scala, TypeScript (Svelte, NestJS, Angular, ...), ...

Nextbeat Hub の紹介

- 毎週木曜日20:00～21:30
- ネクストビートの社員とカジュアルに話せる場
- 関数型の話、大歓迎です



何を話すのか

Web アプリケーションに関数型の考え方を取り入れ、型安全に開発する

そのための基本要素をご紹介します

なぜ話すのか

- 昨年新規事業開発において、Scala 3 の関数型スタイルでの開発をやってみた
- ドメインモデルの表現、ライブラリの組み合わせ方などの基本要素を抽出し、1つの事例として残したい
- 最小限のコードを GitHub に公開しているので、初めてでも始めやすい状態にしている（はず）

「型安全」



不正な値を作らせない

コンパイル時に不正な状態を排除



操作に副作用を含ませない

純粋関数による予測可能な処理

本日の流れ

① ドメインの表現

- ドメインモデルの表現
- ドメインのバリデーション

② ユースケースの表現

- ユースケースの表現
- 副作用の抽象化

③ 技術実装

- 永続化
- 外部サービス連携

④ 統合

- インターフェースと実装の繋ぎ込み
- Web API エンドポイントの型安全な定義

Part 1

ドメインの記述

型安全なドメインの定義

ドメインモデルの検討: パターン

モデルの構成要素は、主に以下の3つに分類できます。

パターン	説明
プリミティブな型	基本的な値の表現
直積型	複数の値を組み合わせ
直和型	いくつかの選択肢から一つ

ドメインモデルの検討: パターンの実装

Scalaでは、以下のような構造を用いてドメインモデルを表現できます。

パターン	目的	Scala での主な実装手法
プリミティブな型	値オブジェクトとして基本的な値を表現	Opaque Types
直積型	複数の値を組み合わせて一つのデータを表現	case class, Tuple
直和型	いくつかの選択肢から一つを表現	Enum, Union Types, Either

パターン: プリミティブな型

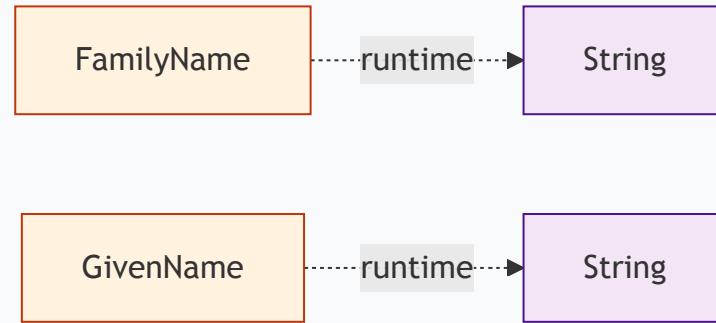
Opaque Types によってプリミティブな型を表現

```
opaque type GivenName = String
opaque type FamilyName = String

object GivenName:
    def apply(value: String): GivenName = value
object FamilyName:
    def apply(value: String): FamilyName = value

val givenName  = GivenName("太郎")
val familyName = FamilyName("山田")

// 型の混同を防ぐ - これはコンパイルエラー
val invalidAssign: GivenName = familyName // ✘
```



特徴・メリット

- 実行時はプリミティブな型と解釈される。ランタイムコストゼロ
- 定義したスコープ以外では異なる型として解釈される

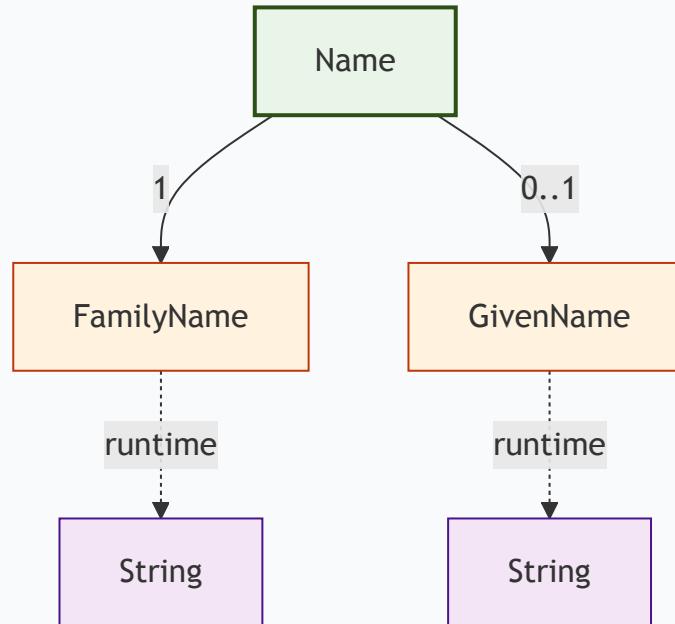
パターン: 直積型

case class を用いて値の組み合わせを表現

```
case class Name(
  familyName: FamilyName,
  givenName: Option[GivenName]
)

val name = Name(
  FamilyName("山田"),
  Some(GivenName("太郎"))
)

println(name)
// Name(山田, Some(太郎))
```



パターン: 直和型 (Enum)

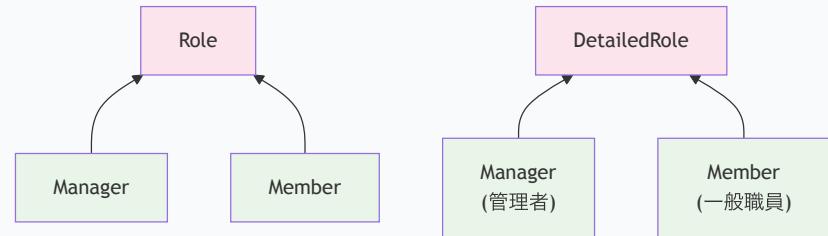
Enum を用いて選択肢を表現

```
enum Role:
    case Manager, Member

// パラメータ付きでより詳細に
enum DetailedRole(val displayName: String):
    case Manager extends DetailedRole("管理者")
    case Member   extends DetailedRole("一般職員")

val role: Role = Role.Manager
val detailedRole = DetailedRole.Manager

println(detailedRole.displayName)
// "管理者"
```



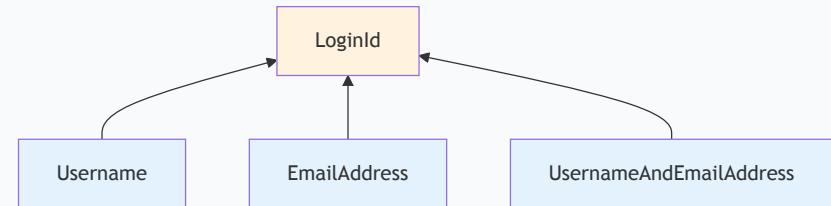
パターン: 直和型 (Union Types)

Union Types を用いて異なる型の選択肢を表現

```
// ログインIDの例
opaque type Username = String
opaque type EmailAddress = String
case class UsernameAndEmailAddress(
  username: Username,
  email:   EmailAddress
)

type LoginId = Username
  | EmailAddress
  | UsernameAndEmailAddress

val loginWithUsername: LoginId = username
val loginWithEmail: LoginId = email
val loginWithBoth: LoginId = UsernameAndEmailAddress(user
```



ドメインモデルの検討: 職員

以下のような職員データをモデル化することを考えます。

項目	説明
ID	職員は ID を持つ。ID の形式は UUIDv4 とする
職員名	姓名を持つ。姓名のうち、名は任意
メールアドレス	メールアドレスを持つ。未検証 or 検証済みの管理がされる
ロール	管理者 or 一般職員
ログインID	ユーザー名、またはメールアドレス。デフォルトはメールアドレス

👉 これらの要件を型安全に表現していきましょう

ドメインモデルの実装: 職員ID

要件 職員は ID を持つ。ID の形式は UUIDv4 とする

職員 ID を Opaque Types で定義します。UUID の生成は副作用を伴うため、生成はユースケース層で行い、ドメインモデルでは ID の型のみを定義します。

```
// UUID型からIDへの変換を提供する型クラス
trait FromUUID[K]:
    def fromUUID(uuid: UUID): K

object FromUUID:
    def apply[K](using instance: FromUUID[K]): FromUUID[K] = instance

// Opaque Type による職員 ID の定義
opaque type Id = UUID
object Id:
    def fromUUID(uuid: UUID): Id = uuid

    extension (id: Id)
        def value: UUID = id

given FromUUID[Id] = new FromUUID[Id]:
    def fromUUID(uuid: UUID): Id = Id.fromUUID(uuid)
```



ドメインモデルの実装: 職員名

要件 職員名を持つ。姓名のうち、名は任意

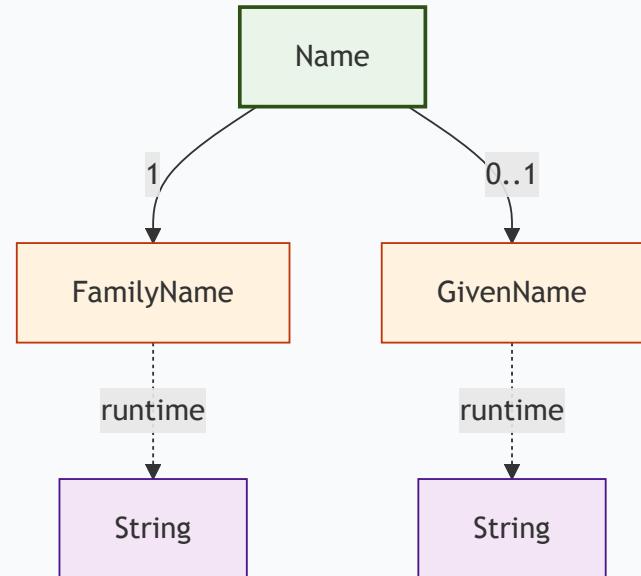
姓名をそれぞれ値オブジェクトで表現し、直積型で結合して表現しています。

```
// 各フィールドは値オブジェクトで表現
opaque type GivenName = String
opaque type FamilyName = String
object GivenName:
    def apply(value: String): GivenName = value
object FamilyName:
    def apply(value: String): FamilyName = value

// 姓名の組み合わせを直積型で表現。名は任意なので Option[_] 型で包む
case class Name(
    familyName: FamilyName,
    givenName: Option[GivenName]
)

val name = Name(FamilyName("山田"), Some(GivenName("太郎")))

println(name)
// Name(山田,Some(太郎))
```



ドメインモデルの実装: メールアドレス

要件 メールアドレスを持つ。メールアドレスは未検証 or 検証済みの管理がされる

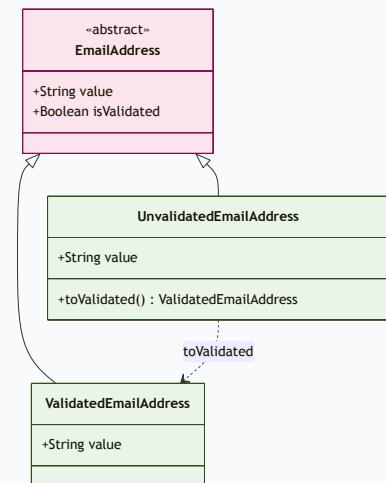
メールアドレス文字列、検証状態を持つ直和型を定義し、メールアドレスの状態を型で表現しています。

```
sealed abstract class EmailAddress(value: String, val isValidated: Boolean)
case class UnvalidatedEmailAddress(value: String) extends EmailAddress(value, false):
  def toValidated: ValidatedEmailAddress = ValidatedEmailAddress(value)
case class ValidatedEmailAddress(value: String)   extends EmailAddress(value, true)

val unvalidatedEmail = UnvalidatedEmailAddress("taro.yamada@example.com")
val validatedEmail  = unvalidatedEmail.toValidated

println(unvalidatedEmail)
// UnvalidatedEmailAddress(taro.yamada@example.com)
println(unvalidatedEmail.isValidated)
// false

println(validatedEmail)
// ValidatedEmailAddress(taro.yamada@example.com)
println(validatedEmail.isValidated)
// true
```



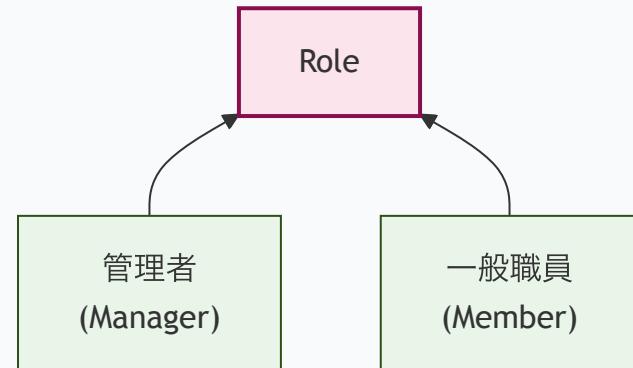
ドメインモデルの実装: ロール

要件 ロールは管理者 or 一般職員

ロールは階層構造を持った選択肢の構造なので、Enum として表現するのが適切でしょう。sealed abstract class + case object, Union Types でも表現できますが、今回は Enum で定義するのが最もシンプルです。

```
// パラメータを持たせることができる
enum Role(val value: String, val displayName: String):
    case Manager extends Role("manager", "管理者")
    case Member   extends Role("member", "一般職員")

// Enum で定義すると、values メソッドなど便利なメソッドが生えてくる
Role.values.foreach: role =>
    println(s"${role} : ${role.value}, ${role.displayName}")
// Manager, manager, 管理者
// Member, member, 一般職員
```



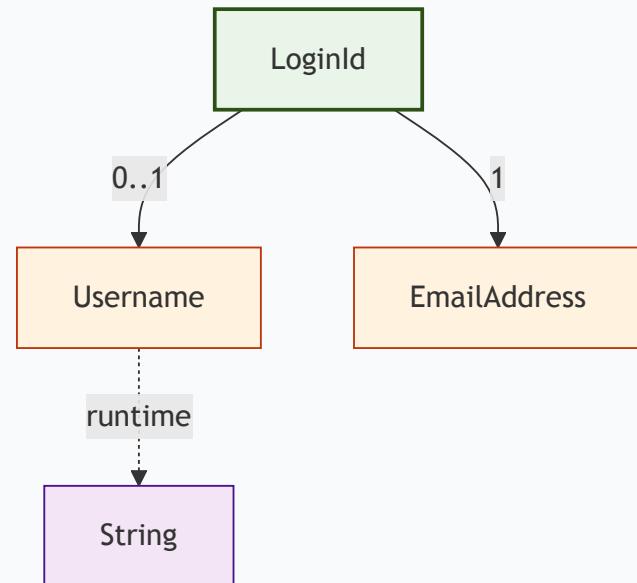
ドメインモデルの実装: ログインID

要件 ログインIDはユーザー名、またはメールアドレス。デフォルトはメールアドレス

ユーザー名は値オブジェクトとして定義し、ログインIDはユーザー名とメールアドレスの組み合わせで表現することができます。

```
opaque type Username = String
object Username:
    def apply(value: String): Username = value

case class LoginId(username: Option[Username], email: EmailAddress)
```



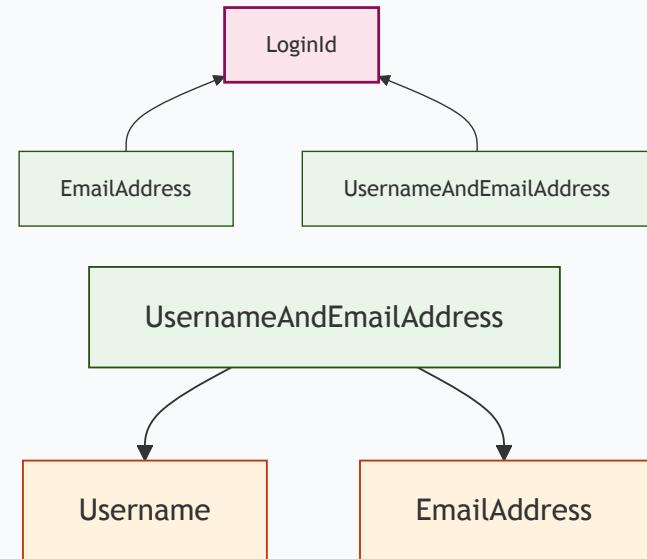
ドメインモデルの実装: ログインID (Union Types版)

Union Types を使って表現すると起きうる状態を明確に表現できます。

```
case class UsernameAndEmailAddress(
    username: Username,
    email: EmailAddress
)
// 「メールアドレスのみ、または、メールアドレス&ユーザー名どちらも利用可能」を表現
// Username 単体の状態があると、Union Types の方が表現しやすくなる
type LoginId = EmailAddress | UsernameAndEmailAddress

val email = UnvalidatedEmailAddress("taro.yamada@example.com")
val username = Username("tyamada")

val loginId1: LoginId = UsernameAndEmailAddress(username, email)
val loginId2: LoginId = email
println(loginId1)
// UsernameAndEmailAddress(tyamada,UnvalidatedEmailAddress(taro.yamada@example.com))
println(loginId2)
// UnvalidatedEmailAddress(taro.yamada@example.com)
```

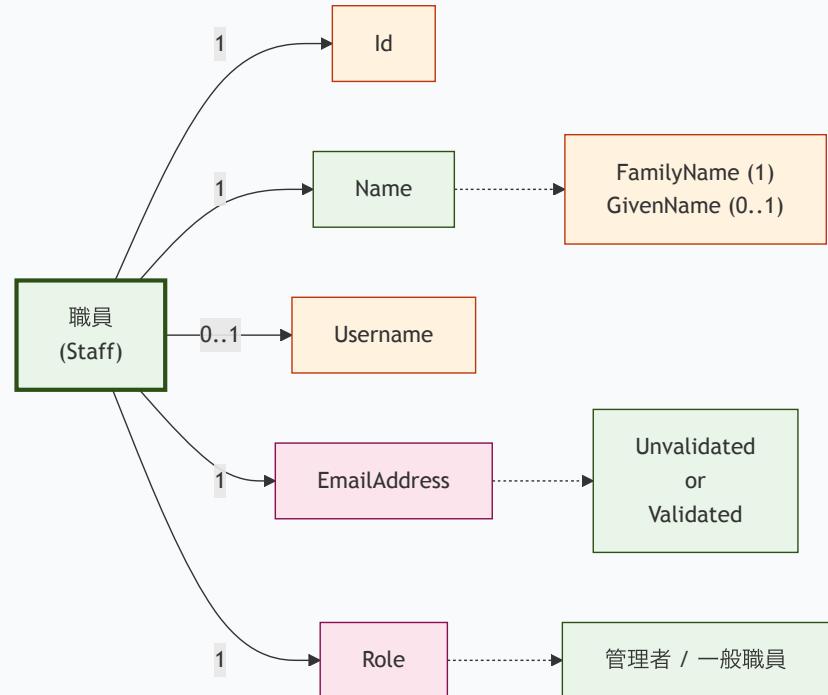


ドメインモデルの実装: 職員

これまでの値オブジェクトを組み合わせて、Staff モデルを作成しましょう。

```
case class Staff(
  id:      Staff.Id,
  name:    Staff.Name,
  username: Option[Staff.Username],
  email:   Staff.EmailAddress,
  role:    Staff.Role
):
  val loginId: Staff.LoginId = username match
    case Some(username) => Staff.UsernameAndEmailAddress(
      case None           => email

def create(id: Id, name: Name, email: EmailAddress, role:
  Staff(
    id      = id,
    name    = name,
    username = username,
    email   = email,
    role    = role
  )
)
```



ドメインモデル: バリデーション

職員モデルの要件に、以下の制約を追加します。

項目	制約
職員名	姓名それぞれ1文字以上32文字以下
メールアドレス	WHATWG で定義された正規表現
ユーザー名	4文字以上16文字以下

ドメインモデル: バリデーション実装

バリデーションの実装方法はいくつかありますが、ここでは以下の2つの方法を紹介します。

- スマートコンストラクタ
 - 値オブジェクトの生成時にバリデーションを行い、不正な値の生成を防ぐ
- Refinement Types
 - 値オブジェクトの制約を型で表現し、コンパイル時に不正な値を排除する

バリデーション実装: スマートコンストラクタ

スマートコンストラクタを用いて、値オブジェクトの生成時にバリデーションを行い、不正な値の生成を防ぎます。

実装

```
object GivenName {
  def apply(value: String): Either[String, GivenName] =
    if value.isEmpty then
      Left("GivenName must not be empty.")
    else if value.length > 32 then
      Left("GivenName must be less than or equal to 32.")
    else
      Right(value)
```

使用例と結果

```
val valid    = GivenName("太郎")
val invalid1 = GivenName("")
val invalid2 = GivenName("あ" * 33)
```

✓ valid → Right(太郎)

✗ invalid1 → Left("GivenName must not be empty.")

✗ invalid2 → Left("GivenName must be less than or equal to 32.")

バリデーション実装: スマートコントラクタ

メールアドレス、ユーザー名なども同様にスマートコントラクタを用いて実装できます。

```
object EmailAddress:
  val EMAIL_REGEX = """^[\w\.-]+@[a-zA-Z0-9\.\-]{0,61}[a-zA-Z0-9]\.[a-zA-Z0-9]{1,}\.?[a-zA-Z0-9]{1,}"""
  def apply(value: String): Either[String, EmailAddress] =
    if EMAIL_REGEX.matches(value) then
      Right(UnvalidatedEmailAddress(value))
    else
      Left("Invalid email address.")
```

バリデーション実装: スマートコンストラクタ

ユーザー名のバリデーションは名前のバリデーションとほぼ同じです。

```
object Username:
  def apply(value: String): Either[String, Username] =
    if value.length < 4 then Left("Username must be at least 4 characters long.")
    else if value.length > 16 then Left("Username must be less than or equal to 16 characters long.")
    else Right(value)
```

バリデーション実装: Refinement Types

Refinement Types を用いると、型自体に制約を持たせることができます。

今回は [iron](#) ライブラリを使用しています。

```
type GivenName = GivenName.T
object GivenName extends RefinedType[String, MinLength[1] & MaxLength[32]]

val refinedGivenName = GivenName.either("太郎")
val refinedInvalidGivenName1 = GivenName.either("")
val refinedInvalidGivenName2 = GivenName.either("あ" * 33) // 32文字を超える

println(refinedGivenName)
// Right(太郎)
println(refinedInvalidGivenName1)
// Left(Should have a minimum length of 1 & Should have a maximum length of 32)
println(refinedInvalidGivenName2)
// Left(Should have a minimum length of 1 & Should have a maximum length of 32)

// ユーザー名も同様に定義
type Username = Username.T
object Username extends RefinedType[String, MinLength[4] & MaxLength[16]]
```

バリデーション実装: Refinement Types

制約はカスタマイズすることが可能です。

```
final class EmailConstraint
object EmailConstraint:
  val EMAIL_REGEX = """^([a-zA-Z0-9.!#$%&'*+\/=?_-`{|}~-]+@[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?(?:\.[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])*)$"""
  given Constraint[String, EmailConstraint] with
    override inline def test(inline value: String): Boolean = EMAIL_REGEX.matches(value)
    override inline def message: String = "Should be valid email address."

type Email = Email.T
object Email extends RefinedType[String, EmailConstraint]
```

バリデーション実装: Refinement Types

Email 型を使って EmailAddress の定義を更新してみましょう。

```
sealed abstract class EmailAddress(val value: Email, val isValidated: Boolean)
object EmailAddress:
  case class Unvalidated(override val value: Email) extends EmailAddress(value, false)
  case class Validated(override val value: Email)    extends EmailAddress(value, true)
  def either(email: String): Either[String, EmailAddress] = Email.either(email).map(Unvalidated(_))

val validEmailAddress = EmailAddress.either("taro.yamada@example.com")
val invalidEmailAddress1 = EmailAddress.either("taro.yamada@example.com")
val invalidEmailAddress2 = EmailAddress.either("taro.yamadaxample.com")

println(validEmailAddress)
// Right(Unvalidated(taro.yamada@example.com))
println(invalidEmailAddress1)
// Left(Should be valid email address.)
println(invalidEmailAddress2)
// Left(Should be valid email address.)
```

ドメインモデル: 職員モデルの型安全な生成

各値オブジェクト生成時にバリデーションを行うことで、職員モデルの生成時に不正な値が入ることを防ぐことができます。

```
def create(
    id:     Id,
    name:   Name,
    email:  EmailAddress,
    role:   Role
): Staff =
    Staff(
        id      = id,
        name    = name,
        username = None,
        email   = email,
        role    = role
    )
```

Part 2

ユースケースの記述

副作用を抽象化し、ユースケースの記述に集中する

題材: 職員を作成する

以下のような「職員を作成する」というユースケースを考えます。このユースケースはいくつかのステップに分解できます。

1 メールアドレスの重複チェック



2 職員の作成



3 職員の永続化



4 サービス連携 (作成イベント発火)

ユースケースの分解 (1/3)

1 メールアドレスの重複チェック

DB読み込み

- ・メールアドレス検索

```
def findByEmail: EmailAddress => F[Option[Staff]]
```

- ・存在チェック

```
def checkExists: Option[Staff] => Either[CreateStaffError, Unit]
```

ユースケースの分解 (2/3)

2 職員の作成

ID生成 (UUID)

- ・ID発行

```
def generateId: => F[Id]
```

- ・職員作成

```
def createStaff: CreateStaffRequest => Id => Staff
```

3 職員の永続化

DB書き込み

```
def save: Staff => F[Staff]
```

ユースケースの分解 (3/3)

4 サービス連携 (作成イベント発火)

タイムスタンプ生成

外部サービス連携

- ・イベント作成 (日付生成)

```
def now: => F[Instant]
```

- ・イベント作成

```
def createEvent: Staff => Instant => StaffEvent
```

- ・イベント発火

```
def publishEvent: StaffEvent => F[Unit]
```

ユースケースの副作用の抽象化

`F[_]` という形で抽象化することによって、ユースケース設計時に実装を意識する必要がなくなり、テストで実装を切り替えやすくなります。

```
// 職員情報のリポジトリ
trait StaffRepository[F[_]]:
  def findByEmail(email: Staff.EmailAddress): F[Option[Staff]]
  def save(staff: Staff): F[Staff]

// イベント発行
trait EventPublisher[F[_]]:
  def publish[E <: Event](event: E): F[Unit]

// ID 生成器
trait IdGenerator[F[_]]:
  def generate[K: FromUUID]: F[K]

// 時刻取得
trait TimeProvider[F[_]]:
  def now: F[Instant]

// 上記を組み合わせて実装するユースケース
trait CreateStaffService[F[_]]:
  def create(command: CreateStaffCommand): F[Either[CreateStaffError, CreateStaffEvent]]
```

CreateStaffService の実装 (1/2)

```
class CreateStaffServiceImpl[F[_]: Sync](
    staffRepository: StaffRepository[F],
    eventPublisher: EventPublisher[F],
    idGenerator: IdGenerator[F],
    timeProvider: TimeProvider[F]
) extends CreateStaffService[F]:  
  
  def create(command: CreateStaffCommand): F[Either[CreateStaffError, CreateStaffEvent]] =  
  
    val process = for
      // 1. メールアドレスの重複チェック
      existingStaff <- EitherT.liftF(
        staffRepository.findByEmail(command.email)
      )
      _ <- EitherT.cond[F](
        existingStaff.isEmpty,
        (),
        CreateStaffError.EmailAlreadyExists(command.email)
      )
```

CreateStaffService の実装 (2/2)

```
// 2. 職員の作成
staffId <- EitherT.liftF(idGenerator.generate[Staff.Id])
newStaff = Staff.create(
  id      = staffId,
  name    = command.name,
  email   = command.email,
  role    = command.role,
)

// 3. 職員情報の永続化
savedStaff <- EitherT.liftF(staffRepository.save(newStaff))

// 4. イベントの作成と発行
now <- EitherT.liftF(timeProvider.now)
event = StaffCreatedEvent(data = savedStaff, occurredAt = now)
_ <- EitherT.liftF(eventPublisher.publish(event))
yield event

process.value.handleError { throwable =>
  CreateStaffError.Unexpected(throwable).asLeft
}
```

異なる実装の提供

テストでは DBIO を走らせたくない・モックを行いたくない場合、異なる実装 (例: インメモリ、固定値) を提供することができます。

テスト用の実装クラス

```
// インメモリでの実装...
class InMemoryStaffRepository[F[_]: Sync] extends StaffRepos
class InMemoryEventPublisher[F[_]: Sync] extends EventPublis

object IdGenerator:
  def uuid[F[_]: Sync]: IdGenerator[F] = new IdGenerator[F]:
    def generate[K: FromUUID]: F[K] = // UUID生成...

  def fixed[F[_]: Sync](id: String): IdGenerator[F] = new Id
  def generate[K: FromUUID]: F[K] = // 固定値を返す...

object TimeProvider:
  def system[F[_]: Sync]: TimeProvider[F] = new TimeProvider
  def now: F[Instant] = // 現在時刻を取得...

  def fixed[F[_]: Sync](instant: Instant): TimeProvider[F] =
  def now: F[Instant] = Sync[F].pure(instant)
```

テスト環境での組み立て

```
val testTime = Instant.parse("2025-01-01T00:00:00Z")

val staffRepository = new InMemoryStaffRepository[IO]
val eventPublisher = new InMemoryEventPublisher[IO]
val idGenerator = IdGenerator.uuid[IO]
val timeProvider = TimeProvider.fixed[IO](testTime)

val createStaffService = new CreateStaffServiceImpl[IO](
  staffRepository,
  eventPublisher,
  idGenerator,
  timeProvider
)
```

ユースケースの副作用の抽象化

👍 メリット

- ✓ ユースケースの実装が型安全に記述できる
- ✓ 副作用を抽象化することで、テストが容易になる
- ✓ 実装の切り替えが容易 (IO, Test, InMemory など)

👎 デメリット

- ✗ 実装の複雑化 (特に高階関数の使用)
- ✗ インスタンスの依存関係が増える

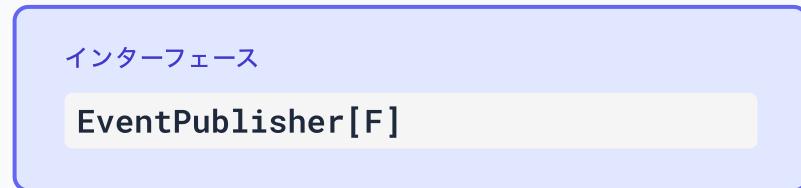
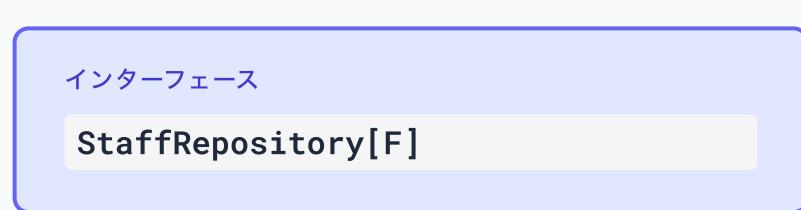
Part 3

インフラ層の記述

実装の提供

インフラ層：DBIO, 外部連携に依存する部分を実装

ユースケース層で定義したインターフェースを具体的な技術で実装



StaffRepository: インターフェースと実装

ユースケース層で定義したインターフェースをインフラ層で実装

⌚ StaffRepository

```
trait StaffRepository[F[_]]:
    def findById(id: Staff.Id): F[Option[Staff]]
    def findByEmail(email: Staff.EmailAddress): F[Option[Staff]]
    def save(staff: Staff): F[Staff]
    def delete(id: Staff.Id): F[Unit]
```

🔧 StaffRepositoryImpl

```
class StaffRepositoryImpl(transactor: Transactor[IO])
  extends StaffRepository[IO]:
    def findByEmail(email: Staff.EmailAddress): IO[Option[Staff]] =
        sql"""
            SELECT BIN_TO_UUID(id), family_name, given_name,
                   email, email_validated, role, username
            FROM staffs
            WHERE email = ${email.value}
        """.query[(Staff.Id, Staff.FamilyName,
                  Option[Staff.GivenName], Staff.Email,
                  Boolean, Staff.Role, Option[Staff.Username])
        .map { case (id, familyName, givenName, emailValue,
                    isValidated, role, username) =>
            Staff.create(id, Staff.Name(familyName, givenName),
                        Staff.EmailAddress(emailValue, isValidated),
                        role, username)
        }.option.transact(transactor)
```

StaffRepository: ドメインモデルの永続化

ドメインモデルを永続化するには、値オブジェクトの変換が必要です

⌚ ドメインモデル

```
case class Staff(  
    id:       Staff.Id,  
    name:     Staff.Name,  
    username: Option[Staff.Username],  
    email:    Staff.EmailAddress,  
    role:     Staff.Role  
)
```

☰ テーブルスキーマ

```
CREATE TABLE IF NOT EXISTS `staffs` (  
    `id` BINARY(16) PRIMARY KEY,  
    `family_name` VARCHAR(32) NOT NULL,  
    `given_name` VARCHAR(32),  
    `email` VARCHAR(255) NOT NULL UNIQUE,  
    `email_validated` BOOLEAN NOT NULL DEFAULT FALSE,  
    `username` VARCHAR(16) UNIQUE,  
    `role` VARCHAR(20) NOT NULL,  
    `created_at` TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  
    `updated_at` TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP  
    INDEX `idx_email` (`email`),  
    INDEX `idx_username` (`username`)  
);
```

doobie の型クラス: Put/Get/Meta

doobie では、ドメインモデルとデータベース間の変換を型クラスで抽象化しています

● Put 型クラス

ドメインモデル → 永続化用のデータ型

例: Staff.FamilyName → String

ドメインの型をDBに保存できる形に変換

● Get 型クラス

永続化用のデータ型 → ドメインモデル

例: String → Staff.FamilyName

DBから取得した値をドメインの型に変換

● Meta 型クラス

Get と Put をまとめたもの

双方向の変換を一つにまとめた型クラス

ドメインの型 ⇌ DB型の相互変換を定義

これらの Meta インスタンスを Opaque Types、直積型、直和型ごとに定義していきます

Meta インスタンスの定義

ドメインの型に応じた Meta インスタンスの定義方法

Opaque Types

```
// String型の制約付き型
given Meta[Staff.FamilyName] =
  Meta[String].timap(
    Staff.FamilyName.applyUnsafe
  )(_.value)

// UUID型
given Meta[Staff.Id] =
  Meta[String].timap(str =>
  Staff.Id.fromUUID(
    UUID.fromString(str)
  ))
```

基本型との相互変換を定義

Case Class (直積型)

```
// 各フィールドのMetaが定義済みなら
// doobieが自動的に導出
case class Staff.Name(
  familyName: FamilyName,
  givenName: Option[GivenName]
)

// SQLで直接利用可能
sql"""
  SELECT family_name, given_name
  FROM staffs
"""
query[Staff.Name]
```

Enum / Sealed Trait (直和型)

```
// Enum型の変換
given Meta[Staff.Role] =
  Meta[String].timap(str =>
  Staff.Role.find(str)
    .toRight(s"Unknown: $str")
  )(_.value)

// Union型の場合は個別に
// Metaインスタンスを定義
```

EventPublisher: インターフェースと実装

イベント発行の抽象化と環境別の実装

⌚ EventPublisher

```
trait EventPublisher[F[_]]:  
  def publish[E <: Event](event: E): F[Unit]
```

🔧 環境別実装

```
// 本番環境用  
class AwsSnsEventPublisher[F[_]: Async](  
  client: SnsClient,  
  topicArn: String  
) extends EventPublisher[F]:  
  def publish[E <: Event](event: E): F[Unit] =  
    // AWS SNSにイベントを送信  
  
// テスト環境用  
class InMemoryEventPublisher[F[_]: Sync]  
  extends EventPublisher[F]:  
  private val events = mutable.ArrayBuffer.empty[Event]  
  
  def publish[E <: Event](event: E): F[Unit] =  
    Sync[F].delay(events += event)
```

Part 4

統合

- インターフェースと実装の繋ぎ込み
- Web API エンドポイントの型安全な定義

統合: エントリーポイント

アプリケーションの起動と各層の統合

Web API の入出力定義

- ドメインモデルの型安全性を活用したバリデーション
- tapir によるエンドポイント定義

HTTPサーバー起動

- http4s によるサーバー実装
- tapirエンドポイントのルーティング組み込み

依存性注入 (DI)

- ユースケース層とインフラ層の結合
- DIコンテナを使わず手動で組み立て

Web API の入出力の定義

職員作成エンドポイントのリクエスト/レスポンス仕様

➡ リクエスト

```
{  
  "name": {  
    "familyName": "山田",  
    "givenName": "太郎"  
  },  
  "email": "taro.yamada@example.com",  
  "role": "manager"  
}
```

➡ レスポンス

```
{  
  "id": "12345678-1234-5678-123456789012",  
  "name": {  
    "familyName": "山田",  
    "givenName": "太郎"  
  },  
  "email": {  
    "value": "taro.yamada@example.com",  
    "isValidated": false  
  },  
  "role": "manager",  
  "username": null  
}
```

Web API の入出力の定義：リクエスト

JSONリクエストをドメイン型に変換 ([circe](#) を使用)

JSON



Decoder



CreateStaffRequest

リクエスト型定義

```
case class CreateStaffRequest(  
    name: Staff.Name, // ドメイン型を直接使用  
    email: Staff.Email, // バリデーション済み  
    role: Staff.Role // 型安全なEnum  
)
```

Decoderインスタンス

```
// Refinement Types / Enum  
given Decoder[Staff.Email] = Decoder[String].emap(Staff.Email.either)  
given Decoder[Staff.Role] = Decoder[String].emap { value =>  
    Staff.Role.find(value).toRight(s"Invalid role: $value")  
}  
  
// 直積型の自動導出  
given Decoder[Staff.Name] = deriveDecoder[Staff.Name]  
  
// 全体のDecoder  
given Decoder[CreateStaffRequest] = deriveDecoder[CreateStaffRequest]
```

Web API の入出力の定義：レスポンス

ドメイン型をJSONレスポンスに変換

StaffResponse



Encoder



JSON

📝 レスポンス型定義

```
case class StaffResponse(  
    id:      Staff.Id,           // UUID型  
    name:    Staff.Name,         // 直積型  
    email:   Staff.EmailAddress, // 直和型  
    role:    Staff.Role,         // Enum型  
    username: Option[Staff.Username] // Option型  
)
```

⌚ Encoderインスタンス

```
// Opaque Types / Refinement Types / Enum  
given Encoder[Staff.Id] = Encoder[UUID].contramap(_.value)  
given Encoder[Staff.Email] = Encoder[String].contramap(_.value)  
given Encoder[Staff.Role] = Encoder[String].contramap(_.value)  
  
// 直積型の自動導出  
given Encoder[Staff.Name] = deriveEncoder[Staff.Name]  
  
// 全体のEncoder  
given Encoder[StaffResponse] = deriveEncoder[StaffResponse]
```

Web API の入出力の定義：エンドポイント定義

tapirのSchema定義でWeb APIドキュメントとバリデーションを自動生成

フィールド単位のSchema定義

```
// Refinement Types
given Schema[Staff.FamilyName] = Schema
  .string[Staff.FamilyName]
  .description("姓")
  .validate(Validator.minLength(1))
  .validate(Validator.maxLength(32))
```

```
// Enum
given Schema[Staff.Role] = Schema
  .string[Staff.Role]
  .description("職員ロール")
  .validate(Validator.enumeration(
    List("manager", "member"),
    v => Some(v.value)
  ))
```

自動導出

```
// case classは自動導出可能
given Schema[CreateStaffRequest] =
  Schema.derived[CreateStaffRequest]

given Schema[StaffResponse] =
  Schema.derived[StaffResponse]
```

CreateStaffRequest ^ Collapse all object

```
name* ^ Collapse all object
familyName* > Expand all string [1, 32] characters
givenName > Expand all string [1, 32] characters
email* > Expand all string email
role* ^ Collapse all string
職員ロール
Enum ^ Collapse all array
#0="manager"
#1="member"
```

Web API の入出力の定義：エンドポイント定義

tapirによる型安全なエンドポイント定義

```
val createStaffEndpoint: PublicEndpoint[  
    CreateStaffRequest, // 入力  
    APIError,          // エラー  
    StaffResponse,     // 出力  
    Any  
] = endpoint.post  
    .in("api" / "staffs")  
    .in(jsonBody[CreateStaffRequest])  
    .out(jsonBody[StaffResponse])  
    .errorOut(  
        oneOf[APIError](  
            oneOfVariant(statusCode(BadRequest).and(jsonBody[APIError])),  
            oneOfVariant(statusCode(InternalServerError).and(jsonBody[APIError]))  
        )  
    )  
    .summary("新規職員を作成")  
    .description("メールアドレスは一意である必要があります")
```

Web API の入出力の定義：処理実装

エンドポイントにビジネスロジックを接続

```
def createStaffServerLogic[F[_]: Async](
  service: CreateStaffService[F]
): CreateStaffRequest => F[Either[APIError, StaffResponse]] = dto => {
  val command = CreateStaffRequest.toCommand(dto)

  service.create(command).map {
    case Right(event) =>
      Right(StaffResponse.fromEvent(event))

    case Left(CreateStaffError.EmailAlreadyExists(email)) =>
      Left(APIError(
        error = "Email already exists",
        details = Some(s"${email.value} is already registered")
      ))

    case Left(CreateStaffError.Unexpected(ex)) =>
      Left(APIError(
        error = "Internal server error",
        details = Some(ex.getMessage)
      ))
  }
}
```

HTTP サーバーの起動、ルーティング設定

```
for
...
// すべてのエンドポイント
allEndpoints = staffsEndpoints ++ swaggerEndpoints

// HTTPルート
routes = Http4sServerInterpreter[IO]().toRoutes(allEndpoints)

httpApp = ServerLogger.httpApp(logHeaders = true, logBody = false)(
    routes.orNotFound
)

server <- EmberServerBuilder
    .default[IO]
    .withHost(config.serverHost)
    .withPort(config.serverPort)
    .withHttpApp(httpApp)
    .build

yield ()
```

依存関係の組み立て

関数型スタイルの DI ライブリ、依存の部分適用など、より良い方法が他にもあります。

```
// インフラストラクチャ層
val staffRepository: StaffRepository[IO] = new StaffRepositoryImpl(xa)
val eventPublisher: EventPublisher[IO] = new AwsSnsEventPublisher(snsClient, config.snsTopicArn)
val idGenerator: IdGenerator[IO] = IdGenerator.uuid[IO]
val timeProvider: TimeProvider[IO] = TimeProvider.system[IO]

// アプリケーション層
val createStaffService: CreateStaffService[IO] = new CreateStaffServiceImpl[IO](
    staffRepository,
    eventPublisher,
    idGenerator,
    timeProvider
)

// エンドポイント
val staffsEndpoints = List(
    StaffsAPI.createStaffEndpoint.serverLogic(
        StaffsAPI.createStaffServerLogic(services.createStaffService)
    )
)
```

動作確認

✓ 正常なリクエスト

```
curl -X POST http://localhost:8080/api/staffs \
-H 'Content-Type: application/json' \
-d '{
  "name": {
    "familyName": "山田",
    "givenName": "太郎"
  },
  "email": "yamada@example.com",
  "role": "member"
}'
```

```
{
  "id": "123e4567-e89b-12d3-a456-426614174000",
  "name": {"familyName": "山田", "givenName": "太郎"},
  "email": "yamada@example.com",
  "role": "member"
}
```

✗ バリデーションエラー

```
curl -X POST http://localhost:8080/api/staffs \
-H 'Content-Type: application/json' \
-d '{
  "name": {"familyName": ""},
  "email": "invalid-email",
  "role": "admin"
}'
```

```
{
  "error": "Validation failed",
  "details": "Invalid email format"
}
```

まとめ



ドメイン型の設計

- Refinement Types (iron)
- Opaque Types
- case class
- Enum / sealed trait / Union Types



レイヤー構成

- ドメイン層（純粋な型・インターフェース）
- アプリケーション層（ユースケース）
- インフラ層（副作用の実装）
- エントリーポイント（Web API）



型安全な実装

- tapirによるエンドポイント定義
- OpenAPIスキーマ自動生成
- doobieによる型安全なDB操作

使用したライブラリ

型定義

iron, Scala 3標準

Web API

tapir, http4s

JSON変換

circe

永続化/エフェクト

doobie, Cats Effect (IO)

Part A

Appendix

Play Framework

Scala / Java 向けのデファクトスタンダードな Web フレームワーク。

- オールインワン: 必要な機能が最初から揃っている
- 高速な開発開始: プロジェクトテンプレートで即座にスタート
- 豊富なドキュメント: 成熟したエコシステム
- 設定の自動化: サーバー起動、ルーティングなど

ライブラリの組み合わせ

個別のライブラリを組み合わせて構築。

- 軽量性: 必要な機能のみを選択
- 学習機会: 各ライブラリの深い理解
- ベンダーロックイン回避: 特定フレームワークに依存しない
- 初期設定の手間: 各ライブラリの統合作業
- 学習コスト: 複数ライブラリの習得が必要
- 設計の責任: アーキテクチャ選定の自由度が高い

非同期ランタイムの選択

Future

- Scala 標準ライブラリに組み込み
- ✗ 即座に実行される（参照透過性がない）
- ✗ エラーハンドリングが限定的

IO

- 純粹関数型のエフェクトシステム
- 豊富なライブラリエコシステム
- 型クラスベースで拡張性が高い

ZIO

- エラー型を明示的に扱える（ZIOR, E, A）
- 統合されたエコシステム