함수형 도메인 주도 설계 구현

박지수

CTO, 두물머리

liftIO 2021

오늘 얘기하지 않을 것

- 도메인 주도 개발(DDD) 자체에 대해서는 설명하지 않습니다.
- DDD의 전략적 패턴은 중요하지만 다루지 않습니다.
- 다행히 DDD 몰라도 됩니다.

목차

- 대수적 자료형을 이용한 도메인 모델링
- ZIO로 실제로 유용한 프로그램 만들기

목차

- **대수적 자료형을 이용한 도메인 모델링**
- ZIO로 실제로 유용한 프로그램 만들기

예제: 결제 시스템

두 가지 할인 유형

- 비율
  - 퍼센트 (정수)
- 정액
  - 액수 (실수)
  - 통화 (통화코드)

ANTIPATTERN: 테이블 주도 설계

```
enum DiscountType:
  case Percentage    // 퍼센트
  case FixedAmount   // 정액할인

class Discount(
  var discountType: DiscountType,
  var percentage: Option[Int],
  var fixedAmount: Option[BigDecimal],
  var fixedAmountCurrency: Option[Currency]
)
```

TABLE DRIVEN DESIGN의 문제

- 높은 복잡도
- 인지 부하가 늘어남

유지보수가 어렵다!

복잡도 증가

```
enum DiscountType:
  case Percentage
  case FixedAmount

class Discount(
  var discountType: DiscountType,
  var percentage: Option[Int],
  var fixedAmount: Option[BigDecimal],
  var fixedAmountCurrency: Option[Currency]
)
```

```
C(Discount) = C(discountType) * C(percentage) *
              C(fixedAmount) * C(fixedAmountCurrency)
            = C(DiscountType) C(Option[Int]) *
              C(Option[BigDecimal]) * C(Option[Currency])
            = 2 * 2 * 2 * 2
            = 16
```

Discount가 가질 수 있는 경우의 수는 16개

잘못된 상태를 표현 가능

```
val valid = new Discount(
  discountType = FixedAmount,
  percentage = None,
  fixedAmount = Some(10.0),
  fixedAmountCurrency = Some(Currency.USD)
)

val invalid = new Discount(
  discountType = FixedAmount,
  percentage = Some(10),   /* 값이 없어야 함 */
  fixedAmount = None,      /* 값이 있어야 함 */
  fixedAmountCurrency = Some(Currency.USD)
)

valid.discountType = Percentage // ???
```

잘못된 상태를 표현 가능

```
 1 val valid = new Discount(
 2   discountType = FixedAmount,
 3   percentage = None,
 4   fixedAmount = Some(10.0),
 5   fixedAmountCurrency = Some(Currency.USD)
 6 )
 7
 8 val invalid = new Discount(
 9   discountType = FixedAmount,
10   percentage = Some(10),   /* 값이 없어야 함 */
11   fixedAmount = None,      /* 값이 있어야 함 */
12   fixedAmountCurrency = Some(Currency.USD)
13 )
14
15 valid.discountType = Percentage // ???
```

잘못된 상태를 표현 가능

```
 1 val valid = new Discount(
 2   discountType = FixedAmount,
 3   percentage = None,
 4   fixedAmount = Some(10.0),
 5   fixedAmountCurrency = Some(Currency.USD)
 6 )
 7
 8 val invalid = new Discount(
 9   discountType = FixedAmount,
10   percentage = Some(10),   /* 값이 없어야 함 */
11   fixedAmount = None,      /* 값이 있어야 함 */
12   fixedAmountCurrency = Some(Currency.USD)
13 )
14
15 valid.discountType = Percentage // ???
```

잘못된 값에 접근 가능

```
1 def discount(subtotal: Money, d: Discount): Money =
2   if (d.discountType == Percentage) {
3     d.fixedAmount.get // FixedAmount?
4     // ...
5   } else if (d.discountType == FixedAmount) {
6     d.percentage.get // Percentage?
7     // ...
8   }
```

잘못된 값에 접근 가능

```
1 def discount(subtotal: Money, d: Discount): Money =
2   if (d.discountType == Percentage) {
3     d.fixedAmount.get // FixedAmount?
4     // ...
5   } else if (d.discountType == FixedAmount) {
6     d.percentage.get // Percentage?
7     // ...
8   }
```

유지보수할 때 생기는 일

- 특정 할인 유형에 새로운 필드 추가
  - 계산 과정에 반영하는걸 까먹었다!
  - 다른 할인 유형 계산식에 넣었다!
- 새로운 할인 유형 추가
  - `else if` 구문 추가를 잊었다!
    - 기존 코드에도 `else if` 대신 `else`가 있었다면!?

대수적 자료형 (ADT)

Algebraic Data Type, 합성 타입

PRODUCT TYPE

```
(3, "hello", true): (Int, String, Boolean)
```

SUM TYPE

```
enum Weekday:
  case Mon, Tue, Wed, Thu, Fri, Sat, Sun
```

SUM + PRODUCT 예: OPTION

```
enum Option[+A]:
  case Some(x: A)
  case None
```

ADT를 이용한 도메인 모델링

```
enum Discount:
  case Percentage(percent: Int)
  case Fixed(amount: BigDecimal, currency: Currency)
```

ADT를 이용한 도메인 모델링

```
enum Discount:
  case Percentage(percent: Int)
  case Fixed(amount: BigDecimal, currency: Currency)
```

```
할인
 - 비율: 퍼센트(정수)
 - 정액: 액수(실수), 통화(통화코드)
```

ADT를 이용한 도메인 모델링

```
enum Discount:
  case Percentage(percent: Int)
  case Fixed(amount: BigDecimal, currency: Currency)
```

```
할인
 - 비율: 퍼센트(정수)
 - 정액: 액수(실수), 통화(통화코드)
```

```
def discount(subtotal: Money, d: Discount): Money =
  d match {
    case Percentage(percent) => // amount 접근 불가
    case Fixed(amount, currency) => // percent 접근 불가
  }
```

ADT로 모델링 시 장점

- 유효한 상태만 표현 가능
- 복잡도 감소

ADT 케이스의 복잡도

```
enum Discount:
  case Percentage(percent: Int)
  case Fixed(amount: BigDecimal, currency: Currency)
```

```
C(Discount) = C(Percentage) + C(Fixed)
            = C(percent) + (C(amount) * C(currency))
            = C(Int) + (C(BigDecimal) * C(Currency))
            = 1 + (1 * 1)
            = 2
```

테이블 주도 vs. 함수형

테이블 주도

```
enum DiscountType:
  case Percentage
  case FixedAmount

class Discount(
  var discountType: DiscountType,
  var percentage: Option[Int],
  var fixedAmount: Option[BigDecimal],
  var fixedAmountCurrency: Option[Currency]
)
```

함수형

```
enum Discount:
  case Percentage(percent: Int)
  case Fixed(amount: BigDecimal, currency: Currency)
```

EITHER 타입을 이용한 도메인 예외 처리

Either ADT

```
enum Either[+A, +B]:
  case Left(value: A)
  case Right(value: B)
```

러스트의 Result도 같은 구조

```
pub enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

EITHER 타입은 우편향

```
1 case class LengthError()
2
3 def oddLength(str: String): Either[LengthError, Int] =
4   str.length match
5     case n if n % 2 == 0 => Right(n)
6     case n => Left(LengthError())
```

EITHER 타입은 우편향

```scala
case class LengthError()

def oddLength(str: String): Either[LengthError, Int] =
  str.length match
    case n if n % 2 == 0 => Right(n)
    case n => Left(LengthError())
```

```scala
val result =
  for
    s <- oddLength("scala")     // Right(5)
    h <- oddLength("haskell")   // Right(7)
  yield s + h

assert(result == Right(12))
```

EITHER 타입은 우편향

```scala
case class LengthError()

def oddLength(str: String): Either[LengthError, Int] =
  str.length match
    case n if n % 2 == 0 => Right(n)
    case n => Left(LengthError())
```

```scala
val result =
  for
    s <- oddLength("scala")     // Right(5)
    h <- oddLength("haskell")   // Right(7)
  yield s + h

assert(result == Right(12))
```

```scala
val result =
  for
    s <- oddLength("scala")     // Right(5)
    j <- oddLength("java")      // Left(LengthError())
    h <- oddLength("haskell")   // 평가하지 않음
  yield s + j + h

assert(result == Left(LengthError()))
```

EITHER 타입은 우편향

```scala
case class LengthError()

def oddLength(str: String): Either[LengthError, Int] =
  str.length match
    case n if n % 2 == 0 => Right(n)
    case n => Left(LengthError())
```

```scala
val result =
  for
    s <- oddLength("scala")     // Right(5)
    h <- oddLength("haskell")   // Right(7)
  yield s + h

assert(result == Right(12))
```

```scala
val result =
  for
    s <- oddLength("scala")     // Right(5)
    j <- oddLength("java")      // Left(LengthError())
    h <- oddLength("haskell")   // 평가하지 않음
  yield s + j + h

assert(result == Left(LengthError()))
```

EITHER 타입은 우편향

```scala
case class LengthError()

def oddLength(str: String): Either[LengthError, Int] =
  str.length match
    case n if n % 2 == 0 => Right(n)
    case n => Left(LengthError())
```

```scala
val result =
  for
    s <- oddLength("scala")     // Right(5)
    h <- oddLength("haskell")   // Right(7)
  yield s + h

assert(result == Right(12))
```

```scala
val result =
  for
    s <- oddLength("scala")     // Right(5)
    j <- oddLength("java")      // Left(LengthError())
    h <- oddLength("haskell")   // 평가하지 않음
  yield s + j + h

assert(result == Left(LengthError()))
```

그 밖의 함수형 도메인 모델링 도구

- Opaque Type (Newtype)
- Refinement Type
- Phantom Type
- Lens
- ...

다 좋은데 실무에서 쓸 수 있나?

- 오류 처리?
- DB 연결?
- 설정 관리?
- 의존성 관리?

실제로 유용한 프로그램은 대부분 외부 세계와 상호작용

EFFECT SYSTEM

순수 함수형 프로그램이 외부 세계와 상호작용하는 방법

- 하스켈: IO
- 스칼라: ScalaZ, Cats Effect, **ZIO**

목차

- 대수적 자료형을 이용한 도메인 모델링
- **ZIO로 실제로 유용한 프로그램 만들기**

순수 함수형 프로그램의 특징

절차적

- 부분적: 모든 입력을 처리하지 않음 (예외)
- 비결정적: 같은 입력, 다른 출력
- 비순수: 부수효과, 실행 중 값이 변함

함수형

- 전체적: 모든 입력에 대한 출력이 존재함
- 결정적: 입력값이 같으면 출력값도 같음
- 순수함: 부수효과가 없음

출처: https://zio.dev/overview/overview_background

FAQ: 순수 함수형 이펙트가 가능한가요?

외부 세계와 상호작용한다매?

가능: 함수형 이펙트 동작원리

- 프로그램의 명세와 실행을 분리합니다.
- 함수를 이용해 작은 프로그램을 큰 프로그램으로 조립합니다.
- 비슷한 사례
  - 파이썬 스크립트와 인터프리터
  - Abstract Syntax Tree (AST)
  - Continuation-Passing Style (CPS)

예: CONSOLE

ADT

```
enum Console[+A]:
  case Return(value: () => A)
  case PrintLine(line: String, rest: Console[A])
  case ReadLine(rest: String => Console[A}])
```

출처: https://zio.dev/overview/overview_background

예: CONSOLE (CONT. 1)

프로그램

```
val example1: Console[Unit] =
  PrintLine("안녕하세요, 이름이 무엇인가요?",
    ReadLine(name =>
      PrintLine(s"${name}님, 반가워요.",
        Return(() => ()))))
```

출처: https://zio.dev/overview/overview_background

예: CONSOLE (CONT. 2)

인터프리터

```
def interpret[A](program: Console[A]): A = program match {
  case Return(value) =>
    value()
  case PrintLine(line, next) =>
    println(line)
    interpret(next)
  case ReadLine(next) =>
    interpret(next(scala.io.StdIn.readLine()))
}
```

출처: https://zio.dev/overview/overview_background

ZIO

- 순수 함수형 이펙트
- 강력한 타입 안정성
- 동시성, 스트리밍
- 활발한 라이브러리 생태계와 커뮤니티
  - 스칼라의 Spring
- 스칼라의 객체지향적인 면 활용
- Cats 등 다른 함수형 라이브러리와 호환

https://zio.dev

ZIO (CONT.)

```
sealed trait ZIO[R, E, A]

type Task[A]  = ZIO[Any, Throwable, A]

type UIO[A]   = ZIO[Any, Nothing  , A]
type IO[E, A] = ZIO[Any, E        , A]
// ...
```

- R: Environment
- E: Error
- A: Value

```
R => Either[E, A]
```

ZIO: 오류 처리

```scala
def findUserByName(name: String): IO[DBError, Option[User]]
def pureValidation(user: User): Either[DomainError, Unit]
def flakyApiCall(x: Int): IO[NetworkError, Remote]

def prog: ZIO[Has[Clock], AppError, Remote] =
  for
    user <- findUserByName("guersam")
      .catchAll(e => IO.fail(AppError.Unexpected(e)))
      .someOrFail(AppError.NotFound("User not found"))

    _ <- IO.from(pureValidation(user))
      .mapError(e => AppError.FromDomainError(e.msg))

    result <- flakyApiCall(user.apiRef)
      .retry(Schedule.spaced(1.second) ++
```

ZIO: 오류 처리

```scala
def findUserByName(name: String): IO[DBError, Option[User]]
def pureValidation(user: User): Either[DomainError, Unit]
def flakyApiCall(x: Int): IO[NetworkError, Remote]

def prog: ZIO[Has[Clock], AppError, Remote] =
  for
    user <- findUserByName("guersam")
      .catchAll(e => IO.fail(AppError.Unexpected(e)))
      .someOrFail(AppError.NotFound("User not found"))

    _ <- IO.from(pureValidation(user))
      .mapError(e => AppError.FromDomainError(e.msg))

    result <- flakyApiCall(user.apiRef)
      .retry(Schedule.spaced(1.second) ++
```

ZIO: 오류 처리

```scala
def findUserByName(name: String): IO[DBError, Option[User]]
def pureValidation(user: User): Either[DomainError, Unit]
def flakyApiCall(x: Int): IO[NetworkError, Remote]

def prog: ZIO[Has[Clock], AppError, Remote] =
  for
    user <- findUserByName("guersam")
      .catchAll(e => IO.fail(AppError.Unexpected(e)))
      .someOrFail(AppError.NotFound("User not found"))

    _ <- IO.from(pureValidation(user))
      .mapError(e => AppError.FromDomainError(e.msg))

    result <- flakyApiCall(user.apiRef)
      .retry(Schedule.spaced(1.second) ++
```

ZIO: 오류 처리

```scala
def findUserByName(name: String): IO[DBError, Option[User]]
def pureValidation(user: User): Either[DomainError, Unit]
def flakyApiCall(x: Int): IO[NetworkError, Remote]

def prog: ZIO[Has[Clock], AppError, Remote] =
  for
    user <- findUserByName("guersam")
      .catchAll(e => IO.fail(AppError.Unexpected(e)))
      .someOrFail(AppError.NotFound("User not found"))

    _ <- IO.from(pureValidation(user))
      .mapError(e => AppError.FromDomainError(e.msg))

    result <- flakyApiCall(user.apiRef)
      .retry(Schedule.spaced(1.second) ++
```

ZIO: 오류 처리

```scala
def findUserByName(name: String): IO[DBError, Option[User]]
def pureValidation(user: User): Either[DomainError, Unit]
def flakyApiCall(x: Int): IO[NetworkError, Remote]

def prog: ZIO[Has[Clock], AppError, Remote] =
  for
    user <- findUserByName("guersam")
      .catchAll(e => IO.fail(AppError.Unexpected(e)))
      .someOrFail(AppError.NotFound("User not found"))

    _ <- IO.from(pureValidation(user))
      .mapError(e => AppError.FromDomainError(e.msg))

    result <- flakyApiCall(user.apiRef)
      .retry(Schedule.spaced(1.second) ++
```

ZIO: 오류 처리

```scala
def findUserByName(name: String): IO[DBError, Option[User]]
def pureValidation(user: User): Either[DomainError, Unit]
def flakyApiCall(x: Int): IO[NetworkError, Remote]

def prog: ZIO[Has[Clock], AppError, Remote] =
  for
    user <- findUserByName("guersam")
      .catchAll(e => IO.fail(AppError.Unexpected(e)))
      .someOrFail(AppError.NotFound("User not found"))

    _ <- IO.from(pureValidation(user))
      .mapError(e => AppError.FromDomainError(e.msg))

    result <- flakyApiCall(user.apiRef)
      .retry(Schedule.spaced(1.second) ++
```

ZIO: 오류 처리

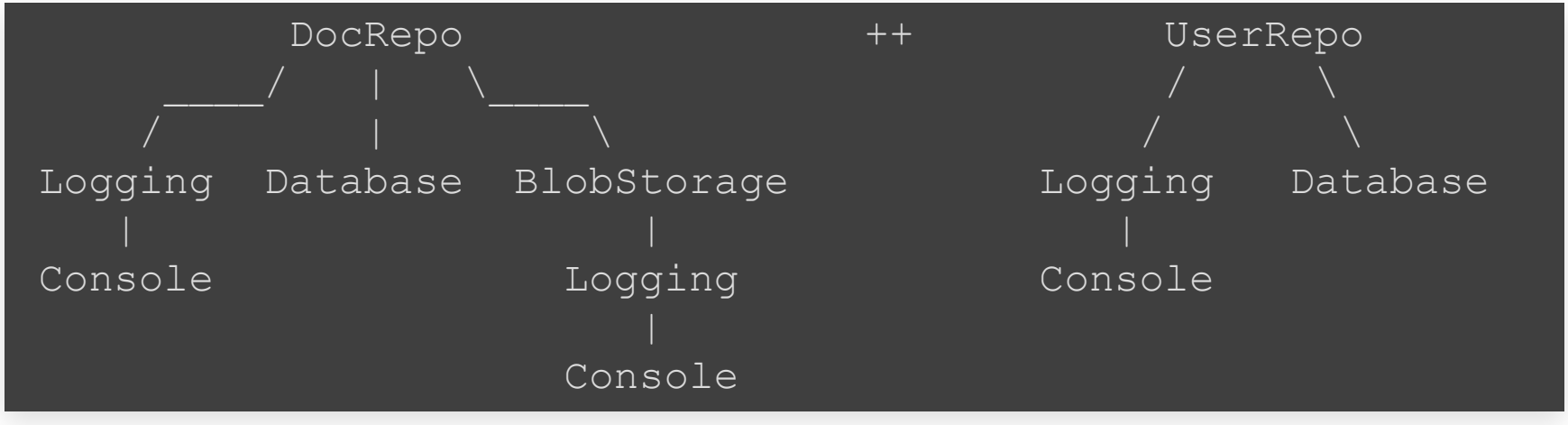
```
1  def findUserByName(name: String): IO[DBError, Option[User]]
2  def pureValidation(user: User): Either[DomainError, Unit]
3  def flakyApiCall(x: Int): IO[NetworkError, Remote]
4
5  def prog: ZIO[Has[Clock], AppError, Remote] =
6    for
7      user <- findUserByName("guersam")
8        .catchAll(e => IO.fail(AppError.Unexpected(e)))
9        .someOrFail(AppError.NotFound("User not found"))
10
11     _ <- IO.from(pureValidation(user))
12       .mapError(e => AppError.FromDomainError(e.msg))
13
14     result <- flakyApiCall(user.apiRef)
15       .retry(Schedule.spaced(1.second) ++
16              Schedule.recurs(5))
```

ZIO: 오류 처리

```
def pureValidation(user: User): Either[DomainError, Unit]
def flakyApiCall(x: Int): IO[NetworkError, Remote]

def prog: ZIO[Has[Clock], AppError, Remote] =
  for
    user <- findUserByName("guersam")
      .catchAll(e => IO.fail(AppError.Unexpected(e)))
      .someOrFail(AppError.NotFound("User not found"))

    _ <- IO.from(pureValidation(user))
      .mapError(e => AppError.FromDomainError(e.msg))

    result <- flakyApiCall(user.apiRef)
      .retry(Schedule.spaced(1.second) ++
             Schedule.recurs(5))
      .mapError(e => AppError.Unexpected(e))
```

ZIO: 모듈간 의존성 관리

예: 의존성 그래프



출처: https://zio.dev/howto/migrate/zio-2.x-migration-guide/

객체지향 설계 기법 재사용

- FP in small, OO in large
- 인터페이스와 구현 분리

객체지향 설계 기법 재사용

- FP in small, OO in large
- 인터페이스와 구현 분리

```
trait UserRepo:
  def getById(id: UserId): IO[DBError, User]
```

```
case class UserRepoLive(log: Logging, db: Database)
  extends UserRepo:
  // ...
```

```
case class TestUserLive(ref: Ref[Map[UserId, User]])
  extends UserRepo:
```

ZIO 모듈 타입

Has[A]

- 타입 수준 Map
- 컴파일 시점 합성
  - Has[A] & Has[B]

ZLayer[-RIn, +E, +ROut]

- 비동기 생성 지원
- 안전한 리소스 할당 & 해제
- 종적, 횡적 합성

ZLAYER: 의존성 그래프 정의

```
object UserRepo:
  val live: URLayer[Has[Logging] & Has[Database],
                    Has[UserRepo]] =
    (for
      log <- ZIO.service[Logging]
      db  <- ZIo.service[Database]
    yield UserRepoLive(log, db)).toLayer
```

ZLAYER: 컴파일 시점 조립

- a >>> b: 종적 조립 (Has[A] => Has[B])
- a ++ b: 횡적 조립 (Has[A] & Has[B])

```
val userRepo: URLayer[Has[UserRepo]] =
  ((Console.live >>> Logging.live) ++ Database.live) >>>
    UserRepo.live
```

출처: https://zio.dev/howto/migrate/zio-2.x-migration-guide/

ZLAYER: 매크로로 더 쉽게

ZIO 1.0

```
val appLayer: URLayer[Any, Has[DocRepo] with Has[UserRepo]] =
  (((Console.live >>> Logging.live) ++ Database.live ++ (Conso
    (((Console.live >>> Logging.live) ++ Database.live) >>> Us

val res: ZIO[Any, Nothing, Unit] = myApp.provideLayer(appLayer
```

ZIO 2.0

```
val res: ZIO[Any, Nothing, Unit] =
  myApp.inject(
    Console.live,
    Logging.live,
    Database.live,
    BlobStorage.live,
    DocRepo.live,
    UserRepo.live
  )
```

출처: https://zio.dev/howto/migrate/zio-2.x-migration-guide/

ZLAYER: 매크로로 더 쉽게 - 타입 안전

```
val app: ZIO[Any, Nothing, Unit] =
  myApp.inject(
    DocRepo.live,
    BlobStorage.live,
//    Logging.live,
    Database.live,
    UserRepo.live,
    Console.live
  )
```

출처: https://zio.dev/howto/migrate/zio-2.x-migration-guide/

ZLAYER: 매크로로 더 쉽게 - 타입 안전

```
val app: ZIO[Any, Nothing, Unit] =
  myApp.inject(
    DocRepo.live,
    BlobStorage.live,
//    Logging.live,
    Database.live,
    UserRepo.live,
    Console.live
  )
```

```
  ZLayer Wiring Error

❯ missing Logging
❯     for DocRepo.live

❯ missing Logging
❯     for UserRepo.live
```

출처: https://zio.dev/howto/migrate/zio-2.x-migration-guide/

ZLAYER: 디버깅

```
1 val layer = ZLayer.wire[Has[DocRepo] with Has[UserRepo]](
2   Console.live,
3   Logging.live,
4   DocRepo.live,
5   Database.live,
6   BlobStorage.live,
7   UserRepo.live,
8   ZLayer.Debug.mermaid
9 )
```

출처: https://zio.dev/howto/migrate/zio-2.x-migration-guide/

ZLAYER: 디버깅

```
1 val layer = ZLayer.wire[Has[DocRepo] with Has[UserRepo]](
2   Console.live,
3   Logging.live,
4   DocRepo.live,
5   Database.live,
6   BlobStorage.live,
7   UserRepo.live,
8   ZLayer.Debug.mermaid
9 )
```

출처: https://zio.dev/howto/migrate/zio-2.x-migration-guide/

디버깅 (CONT.)

```
 1 [info]    ZLayer Wiring Graph
 2 [info]
 3 [info] ◉ DocRepo.live
 4 [info] ├─◑ Logging.live
 5 [info] │ └─◑ Console.live
 6 [info] ├─◑ Database.live
 7 [info] └─◑ BlobStorage.live
 8 [info]   └─◑ Logging.live
 9 [info]     └─◑ Console.live
10 [info]
11 [info] ◉ UserRepo.live
12 [info] ├─◑ Logging.live
13 [info] │ └─◑ Console.live
14 [info] └─◑ Database.live
15 [info]
```

출처: https://zio.dev/howto/migrate/zio-2.x-migration-guide/

디버깅 (CONT.)

```
 3 [info] ◉ DocRepo.live
 4 [info] ├─◑ Logging.live
 5 [info] │ └─◑ Console.live
 6 [info] ├─◑ Database.live
 7 [info] └─◑ BlobStorage.live
 8 [info]   └─◑ Logging.live
 9 [info]     └─◑ Console.live
10 [info]
11 [info] ◉ UserRepo.live
12 [info] ├─◑ Logging.live
13 [info] │ └─◑ Console.live
14 [info] └─◑ Database.live
15 [info]
16 [info] Mermaid Live Editor Link
17 [info] https://mermaid-js.github.io/mermaid-live-editor/edi
```

출처: https://zio.dev/howto/migrate/zio-2.x-migration-guide/

남은 주제들

- Property Based Testing
- Concurrency
- Event Sourcing/CQRS
- GraphQL, gRPC
- Saga Pattern
- ...

정리

- 대수적 자료형을 이용한 도메인 모델링
  - 테이블 주도 설계와 함수형 설계
  - 통제 가능한 모델 복잡도 관리
- ZIO로 실제로 유용한 프로그램 만들기
  - 함수형 이펙트 시스템
    - 명세와 실행을 분리
  - ZIO
    - 오류 처리
    - 의존성 관리

감사합니다.

Q & A

오세요 두물머리

기술을 통해 다수의 사람들에게 뛰어난 투자기회를 제공합니다.

Direct Indexing: 당신만의 주가지수를 당신의 계좌에서

- recruit@doomoolmori.com
- https://doomoolmori.com