

**tapirのようなものをScala3 x JDBCで作りたい**

# 自己紹介

名前: 富永 孝彦

趣味で色々なもの作ってます。

今回も趣味で作ったものの紹介です。

<https://github.com/takapi327>

<https://twitter.com/takapi327>

# tapirとは

エンドポイントの記述とビジネスロジックを分離して記述できるScalaのライブラリ

<https://github.com/softwaremill/tapir>

# tapirを使うメリット

- ルーティング定義からOpenAPIドキュメントを自動生成できるので、ドキュメントの手動管理していると起こりがちな仕様と実装のずれを減らせる
- 実行サーバーの情報を持たないので、サーバーの実装に依存しにくいルーティング定義が書ける
- Interpreter モジュールを差し替えることでサーバーの実装を差し替えることもできる
- 型でしっかりモデリングされているのでコンパイラで間違いを検出できる
- APIクライアントとしても使用することができる

# 社内の課題

- プロダクト仕様書の運用
- テストの書きにくさ
- バックエンド/フロントエンドでモデル定義が乱立
- etc...

課題を解決できるのでは？と思い社内のプロダクトをScala2からScala3へ書き換えた時に使ってみました

## 実際にプロダクトで使ってみて

ルーティング定義からOpenAPIドキュメントを自動生成できる点はかなり良かった。  
ルーティング定義にコメントを書くことができる点も良かった。確かに仕様と実装のずれを減らせると思った。

生成された仕様書からフロントで使用するAPIクライアントの自動生成をすることもできたので、APIクライアントやモデル定義を書く必要がなくなり記述量を減らせた。

サーバーの実装に依存しにくいルーティング定義が書けるおかげでコントローラーなどの実装もどのサーバーで実行されるのかという情報を隠蔽して実装することができた。

**こんな感じのものをJDBCでも作りたい**

## 目指すもの

- 型制御によってコンパイラで間違いを検出できる
- テーブル定義からドキュメントを自動生成できる
- 実行するライブラリを選べるようにする



# 型制御によってコンパイラで間違いを検出できる

モデルを使ってTable定義を作成するようになる

```
case class User(  
  id:      Long,  
  name:    String,  
  age:     Option[Int],  
  updatedAt: LocalDateTime,  
  createdAt: LocalDateTime  
)
```

このモデルから作成するテーブルを作成する時には以下の制限がつけられるようになる

1. カラムの数はモデルのプロパティの数と同じになる
2. プロパティの型とカラムのデータタイプは、指定したものと同じになる  
(Long => BIGINTは○ Long => VARCHARはXという感じ)

これら目指すものをScala3の機能を使って作ってみました。

Scala3で大きく変わったDataType generic programmingを使って行います。

参考

[DataType generic programming with scala3](#)

[Tuples and Mirrors in Scala3 and Higher-Kinded Data](#)

# カラム定義

カラムには型パラメーターを持たせる

データタイプや属性はこの型パラメーターと同じにものを受け取るようにしておく

```
trait Column[T]:  
  
  /** Column Field Name */  
  def label: String  
  
  /** Column type */  
  def dataType: DataType[T]  
  
  /** Extra attribute of column */  
  def attributes: Seq[Attribute[T]]  
  
  /** Column comment */  
  def comment: Option[String]
```

# データタイプ

データタイプはそれぞれ自身が受け取れる型に境界を設けておく  
(複数の型を受け取れるデータタイプに関してはUnionタイプを使って定義する)  
範囲などはinlineとcompiletimeを使用して制限を設けておく

```
inline def BIGINT[T <: Long](inline length: Int): Bigint[T] =  
  inline if length < 0 || length > 255 then error("The length of the BIGINT must be in the range 0 to 255.")  
  else Bigint(length)
```

# テーブル

- Tuple

Tuple.Mapなどでタプルの各メンバの型  $T$  を  $F[T]$  に変換する

- Mirror

モデルとタプルの相互変換を行う

- Dynamic

動的なメソッドを追加してカラム情報にアクセスできるようにする

# Dynamic

ScalaのDynamicは、実行時に存在しないメソッドやプロパティにアクセスするための機能です。

```
foo.method("blah")      ~~> foo.applyDynamic("method")("blah")
foo.method(x = "blah")  ~~> foo.applyDynamicNamed("method")(("x", "blah"))
foo.method(x = 1, 2)    ~~> foo.applyDynamicNamed("method")(("x", 1), ("", 2))
foo.field               ~~> foo.selectDynamic("field")
foo.varia = 10          ~~> foo.updateDynamic("varia")(10)
foo.arr(10) = 13        ~~> foo.selectDynamic("arr").update(10, 13)
foo.arr(10)             ~~> foo.applyDynamic("arr")(10)
```

```
import scala.language.dynamics

class MyDynamic extends Dynamic:
  def selectDynamic(name: String): String = s"Hello, $name!"

val myDynamic = new MyDynamic
println(myDynamic.world) // "Hello, world!"
```

この例では、MyDynamicクラスがDynamicトレイトをミックスインしています。  
selectDynamicメソッドを実装することで、クラスに動的なメソッドを追加することができます。

selectDynamicメソッドは、String型の引数を受け取り、String型の結果を返します。

# テーブル

テーブルはselectDynamicでフィールド名でカラム情報にアクセスできるようにしておきます

```
private[ldbc] trait Table[P <: Product] extends Dynamic:  
  ...  
  def selectDynamic[Tag <: Singleton](tag: Tag)(using  
    mirror: Mirror.ProductOf[P],  
    index: ValueOf[Tuples.IndexOf[mirror.MirroredElemLabels, Tag]])  
  ): Column[Tuple.Elem[mirror.MirroredElemTypes, Tuples.IndexOf[mirror.MirroredElemLabels, Tag]]]
```



フィールドへのアクセスはSingletonを使用します。

Singleton型を使用することで、特定の値を持つ唯一の型を表すことができるようになる

```
def single[Tag <: Singleton](x: Tag): Tag = x

val x = single("hello world")
// val x: String = hello world

val x = single[String]("hello world") // エラー
val x = single["hello world"]("hello world") // ok
val x = single[Singleton & String]("hello world") // ok
```

MirroredElemLabelsとTagが一致するIndexを生成し、ValueOfで値として扱えるようにする

```
override def selectDynamic[Tag <: Singleton](...)(
  ...
  index: ValueOf[Tuples.IndexOf[mirror.MirroredElemLabels, Tag]]
): Column[Tuple.Elem[mirror.MirroredElemTypes, Tuples.IndexOf[mirror.MirroredElemLabels, Tag]]]

...

import scala.compiletime.ops.int.S

object Tuples:

  type IndexOf[T <: Tuple, E] <: Int = T match
    case E *: _ => 0
    case _ *: es => S[IndexOf[es, E]]
```

Tableを継承したモデルを定義しておく

引数のcolumnsは、Tuple.Mapを使用して型パラメーターのTupleをColumn型で受け取るようにしている

(Long, String)というTupleの型が渡された場合に渡せる引数の型は、(Column[Long], Column[String])というTuple型になる

```
object Table extends Dynamic:  
  
  private case class Impl[P <: Product, T <: Tuple](  
    name:      String,  
    columns: Tuple.Map[T, Column],  
    ...  
  ) extends Table[P]:  
  
    ...
```

Tupleであるcolumnsから指定したIndexの値を取得する。

productElementの戻り値はAnyなため、Tuple.Elemを使用してTupleのIndexに対応した型に変更してあげる

Tuple.ElemはタプルXの位置Nにある要素の型を取得する型レベル関数

```
override def selectDynamic[Tag <: Singleton](tag: Tag)(using
  mirror: Mirror.ProductOf[P],
  index: ValueOf[Tuples.IndexOf[mirror.MirroredElemLabels, Tag]]
): Column[Tuple.Elem[mirror.MirroredElemTypes, Tuples.IndexOf[mirror.MirroredElemLabels, Tag]]] =
  columns
    .productElement(index.value)
    .asInstanceOf[Column[Tuple.Elem[mirror.MirroredElemTypes, Tuples.IndexOf[mirror.MirroredElemLabels, Tag]]]]
```

# インスタンス生成

Tableはコンパニオンオブジェクトで、Implを実装している。

```
val table: Table[User] = Table[User]("user")(
  column("id", BIGINT(64), AUTO_INCREMENT, PRIMARY_KEY),
  column("name", VARCHAR(255)),
  column("age", INT(255).DEFAULT_NULL),
  column("updated_at", TIMESTAMP.DEFAULT_CURRENT_TIMESTAMP()),
  column("created_at", TIMESTAMP.DEFAULT_CURRENT_TIMESTAMP(true))
)
```

# インスタンス生成

モデル <-> テーブルマッピングでパラメーターの数が違っていたり、型が違くとcompileでエラーになる

```
新規 *
case class User(id: Long, name: String, age: Int)

val table: Table[User] = Table[User]("user")(
  column(label = "id", BIGINT(length = 64)),
  column(label = "age",
Found: (ldbc.core.Column[Long], ldbc.core.Column[Int])
Required: ldbc.core.interpreter.ColumnTuples[(Long, String, Int), ldbc.core.Column]
```

```
case class User(id: Long, name: String, age: String)

val table: Table[User] = Table[User]("user")(
  column(label = "id", BIGINT(length = 64)),
  column(label = "name", VARCHAR(length = 255)),
  column(label = "age", INT(length = 255))
)
```

# インスタンス生成

Scalaの型とDBの型が一致していない場合もcompileでエラーとなる

```
val table: Table[User] = Table[User]("user")(
  column(label = "id", BIGINT(length = 64)),
  column(label = "name", BIGINT(length = 64)),
  column(label = "age", INT(length = 255))
)
```

データタイプの長さもDBの制限を超えるとcompileでエラーとなる

```
val table: Table[User] = Table[User]("user")(
  column(label = "id", BIGINT(length = 256)),
  column(label = "name", VARCHAR(length = 255)),
  column(label = "age", INT(length = 255))
)
```

The length of the BIGINT must be in the range 0 to 255.

モデル -> テーブル定義を作成することができた。

作成されたテーブル定義は何に使用されるかという情報を持っていない。

そのため、別の何かで使用する時に関係のない情報が邪魔をしない。

使う側がこのテーブル定義に対して、意味を与えてあげる。



今回はこのテーブル定義に以下2つの意味を与えてあげる。

- ドキュメント生成 => テーブル定義からドキュメントを自動生成できる
- DB接続 => 実行するライブラリを選ぶようにする

# テーブル定義からドキュメントを自動生成できる

ドキュメント生成はSchemaSpyを使用

# SchemaSpy

データベースの情報を元に、ER図やテーブル、カラム一覧などの情報をHTML形式のドキュメントとして出力するツール

maven対応していない。。。

しょうがないので、cloneしてゴニョゴニョして使ってたら

つい最近maven対応しました！

<https://github.com/schemaspy/schemaspy/issues/157>

<https://central.sonatype.com/artifact/org.schemaspy/schemaspy/6.2.2>

SchemaSpy自体は配布されているjarやDocker Imageを使用してコマンドベースでの処理を行い、  
内部でjdbcを使用してMeta情報など実際にDB接続を行ない取得を行なっている。

## SchemaSpyのカラムを情報を取得する処理

```
private void initColumns(Table table) throws SQLException {
    synchronized (Table.class) {
        try (ResultSet rs = sqlService.getDatabaseMetaData().getColumns(table.getCatalog(), table.getSchema(), table.getName(), "%")) {
            while (rs.next())
                addColumn(table, rs);
        } catch (SQLException exc) {
            if (!table.isLogical()) {
                throw new ColumnInitializationFailure(table, exc);
            }
        }
    }
}
```

ただ今回はDB接続を行うのではなく、アプリケーションで作成したテーブル定義やDB定義を使用してSchemaSpyのドキュメント生成を行う。

SchemaSpyはResultSetからTableやColumnのメタ情報など必要なデータ取得を行い、SchemaSpy内に定義されているモデルに格納をおこなっています。

そのモデルを使用してドキュメント生成をおこなっているので、ResultSetから取得を行うのではなく自作したテーブル、カラムからデータを取得しSchemaSpyのモデルに変換してあげることで割と簡単に実装は出来ました。

ただあまりScala3は関係ないので、実装は割愛



こんな感じのテーブル定義を使用してドキュメント生成してみる

```
val roleTable = Table[Role]("role")(
  column("id", BIGINT(64), AUTO_INCREMENT, PRIMARY_KEY),
  column("name", VARCHAR(255)),
  column("status", BIGINT(64))
)

val userTable = Table[User]("user")(
  column("id", BIGINT(64), "ユーザー識別子", AUTO_INCREMENT, PRIMARY_KEY),
  column("name", VARCHAR(255)),
  column("age", INT(255).DEFAULT_NULL),
  column("role_id", BIGINT(64)),
  column("updated_at", TIMESTAMP.DEFAULT_CURRENT_TIMESTAMP()),
  column("created_at", TIMESTAMP.DEFAULT_CURRENT_TIMESTAMP(true))
)
  .keySet(v =>
    CONSTRAINT("fk_id", FOREIGN_KEY(v.roleId, REFERENCE(roleTable)(roleTable.id)))
  )
```

データベースと生成先を指定する

```
val db = new Database:  
    override val databaseType: Database.Type = Database.Type.MySQL  
    override val name: String = "example"  
    override val schema: String = "example"  
    override val tables = Set(roleTable, userTable)  
    ...  
  
val file = java.io.File("./document")  
SchemaSpyGenerator(db).generateTo(file)
```

▼ document

> bower

> diagrams

> fonts

> images

> routines

summary

> tables

anomalies.html

anomalies.js

column.js

columns.html

constraint.js

constraints.html

★ favicon.png

index.html

info-html.txt

main.js

orphans.html

relationships.html

relationships.js

routines.html

routines.js

schemaSpy.css

schemaSpy.js

> src




# カラムの一覧

Columns

Columns

AllTablesViewsComments

Search:

| Table | Type  | Column  | Type         | Size | Nullable | Auto | Default | Comments |
|-------|-------|---|--------------|------|----------|------|---------|----------|
| user  | Table | age   | INT(255)     | 0    | √        |      | null    |          |
| user  | Table | created_at  | TIMESTAMP    | 0    |          |      | null    |          |
| user  | Table |  id        | BIGINT(64)   | 0    |          | √    | null    | ユーザー識別子  |
| role  | Table |  id        | BIGINT(64)   | 0    |          | √    | null    |          |
| user  | Table | name  | VARCHAR(255) | 0    |          |      | null    |          |
| role  | Table | name  | VARCHAR(255) | 0    |          |      | null    |          |
| user  | Table |  role_id | BIGINT(64)   | 0    |          |      | null    |          |
| role  | Table | status  | BIGINT(64)   | 0    |          |      | null    |          |
| user  | Table | updated_at  | TIMESTAMP    | 0    |          |      | null    |          |

# 外部キー制約

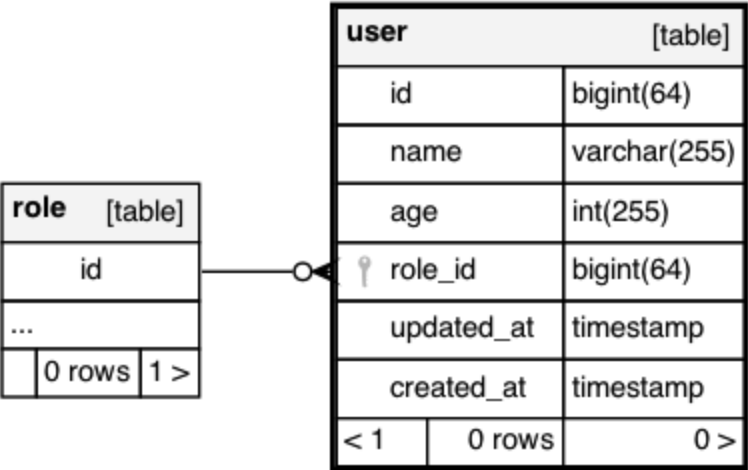
| Constraint Name | Child Column | Parent Column | Delete Rule     |
|-----------------|--------------|---------------|-----------------|
| FOREIGN KEY     | user.role_id | role.id       | Restrict delete |

# リレーションシップ

## 🔗 Relationships [🔗](#)

Close relationships within degrees of separation

One



# 実行するライブラリを選ぶようにする

- 自作
- Slick

# 自作

文字列補完などでSQL文を生成して、DataSource -> Connection -> Statement -> ResultSetというようなJDBCの標準的なアクセスを行うdoobieのような感じで実装しました。



ResultSet -> モデルだけではなく指定したカラムだけ取得とかも行いたいのので、テーブルのカラム全体ではなくカラム単体ごとにResultSetからデータを取得する処理を作成してあげる。

フィールド名でのアクセス時にResultSetからカラム名を指定してデータ取得を行う  
ResultSetReaderというものを暗黙的に渡してあげる

戻り値はEffect SystemでラップされたResultSetを使用するKleisli

```
def applyDynamic[Tag <: Singleton](
  tag: Tag
)()(using
  mirror: Mirror.ProductOf[P],
  index: ValueOf[Tuples.IndexOf[mirror.MirroredElemLabels, Tag]],
  reader: ResultSetReader[F, Tuple.Elem[mirror.MirroredElemTypes, Tuples.IndexOf[mirror.MirroredElemLabels, Tag]]]
): Kleisli[F, ResultSet[F], Tuple.Elem[mirror.MirroredElemTypes, Tuples.IndexOf[mirror.MirroredElemLabels, Tag]]] =
  Kleisli { resultSet =>
    val column = table.selectDynamic[Tag](tag)
    reader.read(resultSet, column.label)
  }
```

ResultSetReaderはこんなやつ。

ResultSetからカラム名を指定して取得を行う処理を記載しておく、このとき取得する型はパラメーターによって決められるようにしておく。

```
trait ResultSetReader[F[_], T]:  
  def read(resultSet: ResultSet[F], columnLabel: String): F[T]  
  
object ResultSetReader:  
  def apply[F[_], T](func: ResultSet[F] => String => F[T]): ResultSetReader[F, T] =  
    new ResultSetReader[F, T]:  
      override def read(resultSet: ResultSet[F], columnLabel: String): F[T] =  
        func(resultSet)(columnLabel)
```

あとはScala標準の型とかは予め定義しておき暗黙的に渡せるようにしておく。  
定義されていない型が必要になったら、同じように定義してあげれば良い。

```
given [F[_]]: ResultSetReader[F, String] = ResultSetReader(_.getString)
given [F[_]]: ResultSetReader[F, Boolean] = ResultSetReader(_.getBoolean)
...
```

これでカラム単体でResultSetから値を取得できるKleisliを構築できるようになった

```
given Kleisli[IO, ResultSet[IO], Long] = table.id()
```

あとはResultSet -> Userへの変換を行うKleisliを定義してあげる

```
given Kleisli[I0, ResultSet[I0], User] =  
  for  
    id      <- table.id()  
    name    <- table.name()  
    age     <- table.age()  
    status  <- table.status()  
    updatedAt <- table.updatedAt()  
    createdAt <- table.createdAt()  
  yield User(id, name, age, status, updatedAt, createdAt)
```

- transaction => DataSource -> Connection
- query => Connection -> Statement -> ResultSet

```
val user: IO[User] = sql"SELECT * FROM user".query.transaction.run(dataSource)
```

# Slick

強く型付けされ、高度に構成可能なAPIを持つ、Scalaのための高度で包括的なデータベースアクセスライブラリ

Scalaのコレクションを扱うようにデータベースを操作することができるのが特徴

※ 2023/04時点ではまだScala3対応版はリリースされていない。そのため対応進行中のものをクローンして使っています。

# Slick

SlickはモデルからTable定義を生成する

```
class UserTable(tag: Tag) extends Table[User](tag, "user"):
  def id = column[Long]("id", 0.AutoInc, 0.Primary)
  def name = column[String]("name")
  def age = column[Option[Long]]("age")
  def roleId = column[Long]("role_id")
  def updatedAt = column[Long]("updated_at")
  def createdAt = column[Long]("created_at")

  def * = (id, name, age, roleId, updatedAt, createdAt).mapTo[User]
```

DB接続はTableQueryとDatabaseを使用して行われる。

```
val tableQuery = TableQuery[UserTable]
val db = Database.forDataSource(...)

db.run(tableQuery.filter(_.name === "takapi").result)
```



Slickでは、RepとTypedType[T]というものが存在しています。

まず、RepはSlickがデータベーステーブルの列を表現するために使用する型です。Rep型は値を保持するためのものではなく、データベーステーブルの列に対応するSQL文を生成するために使用されます。

```
val id: Rep[Int] = column[Int]("id")  
val name: Rep[String] = column[String]("name")
```

これらのコードは、idとnameという2つの列を表し、それぞれの型はRep[Int]とRep[String]です。これらのRep型は、後に使用するクエリに対応するSQL文を生成するために使用されます。

次に、`TypedType[T]`は、Slickがカラムの型を表現するために使用する型です。Slickは、基本的な型（`Int`、`String`、`Boolean`など）に加えて、自動生成されたユーザー定義の型や、さまざまなデータベースシステムでサポートされる型をサポートしています。`TypedType[T]`は、Slickがデータベースからデータを読み取る際に使用する型を指定するために使用されます。

```
val id: Rep[Int] = column[Int]("id")(summon[TypedType[Int]])  
val name: Rep[String] = column[String]("name")(summon[TypedType[String]])
```

このコードでは、`summon[TypedType[Int]]`と`summon[TypedType[String]]`を使用して、それぞれの列の型を指定しています。これにより、Slickはデータベースからデータを読み取る際に、正しい型を使用することができます。

他にもSlickではScalaの型とデータベーステーブルの列の型をマッピングするためのShapeという機能が必要です。

SlickはRepをShapeに変換 -> ShapeをTupleに変換 -> その値を使用してTuple <-> モデルのマッピングを定義している。

Slickはこの変換処理を暗黙におこなっています

<https://github.com/slick/slick/blob/main/slick/src/main/scala/slick/lifted/ExtensionMethods.scala>

Slickのテーブル定義の `*` を分解すると暗黙に変換が行われていることがわかります。  
ShapedValueは、Scalaの型とデータベーステーブルの列の値のタプルを表すものです。

```
val shapedValue: ShapedValue[
  (Rep[Option[Long]], Rep[String], Rep[Option[Int]]),
  (Option[Long], String, Option[Int])
] = (id, name, age)

def * = shapedValue <> ((User.apply _).tupled, User.unapply)
```

つまりSlickで自作したテーブル定義を使用するためには、カラムをこのRepに持ち上げてあげる必要があります、その際にTypedTypeも合わせて持たせてあげる必要があります。

また指定したモデルへのマッピングを行うためにShapeも用意してあげる必要があります。

まずはShapdeValueを作成するために、RepのShapeのTupleとRepのTupleをカラムから生成して渡してあげる必要があります。

TupleShapeは、複数のShapeを結合して、タプルのShapeを表現するためのShapeです。TupleShapeでまずはカラムのリストからShapeのタプルを生成します。

```
val tupleShape = new TupleShape[
  FlatShapeLevel,
  Tuple.Map[mirror.MirroredElemTypes, RepColumnType],
  mirror.MirroredElemTypes,
  P
](
  columns.productIterator
    .map(v => {
      RepShape[FlatShapeLevel, Rep[Extract[v.type]], Extract[v.type]]
    })
    .toList: _*
)
```

ここで型をColumn[T]からRep[T]にしてあげる必要があるので、Scala3で追加された型マッチを使用してColumnが持つTの型を抽出してあげます。

```
type Extract[T] = T match  
  case Column[t] => t
```

次にカラムのTupleからRepのTupleを生成する  
単純に詰め替えを行う

```
val repColumns: Tuple.Map[mirror.MirroredElemTypes, RepColumnType] = Tuple
  .fromArray(
    columns.productIterator
      .map(v => {
        val column = v.asInstanceOf[TypedColumn[?]]
        new TypedColumn[Extract[column.type]] with Rep[Extract[column.type]]:

          ...

          override def encodeRef(path: Node): Rep[Extract[column.type]] =
            Rep.forNode(path)(using column.typedType)

          override def toNode =
            Select(
              (tag match
                case r: RefTag => r.path
                case _        => tableNode
              ),
              FieldSymbol(label)(Seq.empty, typedType)
            ) :@ typedType
      })
    .toArray
  )
.asInstanceOf[Tuple.Map[mirror.MirroredElemTypes, RepColumnType]]
```



生成されたShapedValueとMirror、Tupleを使用してモデル <-> タプルの変換を定義してあげる。

```
val shapedValue = new ShapedValue[Tuple.Map[mirror.MirroredElemTypes, RepColumnType], mirror.MirroredElemTypes](
  repColumns,
  tupleShape
)

shapedValue <> (
  v => mirror.fromTuple(v.asInstanceOf[mirror.MirroredElemTypes]),
  Tuple.fromProductTyped
)
```

## 自作したテーブル用のTableQueryを作成

```
class TableQuery[T <: Table[?]](table: T) extends Query[T, TableQuery.Extract[T], Seq]:  
  override lazy val shaped: ShapedValue[T, TableQuery.Extract[T]] =  
    ShapedValue(table, RepShape[FlatShapeLevel, T, TableQuery.Extract[T]])  
  
  override lazy val toNode = shaped.toNode  
  
object TableQuery:  
  type Extract[T] = T match  
    case Table[t] => t  
  
  def apply[T <: Table[?]](table: T): TableQuery[T] =  
    new TableQuery[T](table)
```

Slickで実行する準備ができたので、テーブル定義を自作したものに置き換える

```
val table: Table[User] = Table[User]("user")(
  column("id", BIGINT(64), AUTO_INCREMENT, PRIMARY_KEY),
  column("name", VARCHAR(255)),
  column("age", INT(255).DEFAULT_NULL),
  column("updated_at", TIMESTAMP.DEFAULT_CURRENT_TIMESTAMP()),
  column("created_at", TIMESTAMP.DEFAULT_CURRENT_TIMESTAMP(true))
)

val tableQuery = TableQuery(table)
val db = Database.forDataSource(...)

db.run(tableQuery.filter(_.name === "takapi").result)
```

# 目指すもの

- 型制御によってコンパイラで間違いを検出できる  
=> モデル -> テーブルの型制御, データタイプの型制御
- テーブル定義からドキュメントを自動生成できる  
=> SchemaSpyのドキュメント生成
- 実行するライブラリを選べるようにする  
=> 自作/Slick両方で動作可能

Scala3のDataType generic programmingのおかげで型の受け渡しや変換を楽に行うことができました。

他にもこんな使い方があるよ！などあれば教えていただけると嬉しいです。

おわり