Scala School

Лекция 14: Клиенты базы данных в Scala

План лекции

- Slick
- Doobie
- Play anorm
- Scalikejdbc
- MongoDB
- Casbah
- Salat

Java DataBase Connectivity

- ```
  object ScalaJdbcConnectSelect {


    def main(args: Array[String]) {
      // connect to the database named "mysql" on the localhost
      val driver = "com.mysql.jdbc.Driver"
      val url = "jdbc:mysql://localhost/mysql"
      val username = "root"
      val password = "root"

      // there's probably a better way to do this
      var connection:Connection = null
  ```
  ```

Java DataBase Connectivity

```
try {
   // make the connection
   Class.forName(driver)
   connection = DriverManager.getConnection(url, username, password)

   // create the statement, and run the select query
   val statement = connection.createStatement()
   val resultSet = statement.executeQuery("SELECT host, user FROM user")
   while ( resultSet.next() ) {
     val host = resultSet.getString("host")
     val user = resultSet.getString("user")
     println("host, user = " + host + ", " + user)
   }
 } catch { case e => e.printStackTrace }
 connection.close()
} }
```

Slick

- https://habrahabr.ru/company/cit/blog/305682/
- http://eax.me/slick/
- http://slick.lightbend.com/doc/3.2.0/

Slick: Schema

```
// Definition of the SUPPLIERS table
class Suppliers(tag: Tag) extends Table[(Int, String, String, String, String, String)](tag, "SUPPLIERS") {
  def id = column[Int]("SUP_ID", O.PrimaryKey) // This is the primary key column
  def name = column[String]("SUP_NAME")
  def street = column[String]("STREET")
  def city = column[String]("CITY")
  def state = column[String]("STATE")
  def zip = column[String]("ZIP")
  // Every table needs a * projection with the same type as the table's type parameter
  def * = (id, name, street, city, state, zip)
}
val suppliers = TableQuery[Suppliers]

// Definition of the COFFEES table
class Coffees(tag: Tag) extends Table[(String, Int, Double, Int, Int)](tag, "COFFEES") {
  def name = column[String]("COF_NAME", O.PrimaryKey)
  def supID = column[Int]("SUP_ID")
  def price = column[Double]("PRICE")
  def sales = column[Int]("SALES")
  def total = column[Int]("TOTAL")
  def * = (name, supID, price, sales, total)
  // A reified foreign key relation that can be navigated to create a join
  def supplier = foreignKey("SUP_FK", supID, suppliers)(_.id)
}
val coffees = TableQuery[Coffees]
```

Slick: Populating the Database

```
val setup = DBIO.seq(
  // Create the tables, including primary and foreign keys
  (suppliers.schema ++ coffees.schema).create,

  // Insert some suppliers
  suppliers += (101, "Acme, Inc.", "99 Market Street", "Groundsville", "CA", "95199"),
  suppliers += ( 49, "Superior Coffee", "1 Party Place", "Mendocino", "CA", "95460"),
  suppliers += (150, "The High Ground", "100 Coffee Lane", "Meadows", "CA", "93966"),
  // Equivalent SQL code:
  //   insert into SUPPLIERS(SUP_ID, SUP_NAME, STREET, CITY, STATE, ZIP) values (?,?,?,?,?,?)

  // Insert some coffees (using JDBC's batch insert feature, if supported by the DB)
  coffees ++= Seq(
    ("Colombian",         101, 7.99, 0, 0),
    ("French_Roast",       49, 8.99, 0, 0),
    ("Espresso",          150, 9.99, 0, 0),
    ("Colombian_Decaf",   101, 8.99, 0, 0),
    ("French_Roast_Decaf", 49, 9.99, 0, 0)
  ),
  // Equivalent SQL code:
  //   insert into COFFEES(COF_NAME, SUP_ID, PRICE, SALES, TOTAL) values (?,?,?,?,?)
)

val setupFuture = db.run(setup)
```

Slick: Querying

```
// Read all coffees and print them to the console
println("Coffees:")
db.run(coffees.result).map(_.foreach {
  case (name, supID, price, sales, total) =>
    println("  " + name + "\t" + supID + "\t" + price + "\t" + sales + "\t" + total)
})
// Equivalent SQL code:
// select COF_NAME, SUP_ID, PRICE, SALES, TOTAL from COFFEES
```

Slick: Querying

```
// Why not let the database do the string conversion and concatenation?
val q1 = for(c <- coffees)
  yield LiteralColumn("  ") ++ c.name ++ "\t" ++ c.supID.asColumnOf[String] ++
    "\t" ++ c.price.asColumnOf[String] ++ "\t" ++ c.sales.asColumnOf[String] ++
    "\t" ++ c.total.asColumnOf[String]
// The first string constant needs to be lifted manually to a LiteralColumn
// so that the proper ++ operator is found

// Equivalent SQL code:
// select '  ' || COF_NAME || '\t' || SUP_ID || '\t' || PRICE || '\t' SALES || '\t' TOTAL from COFFEES

db.stream(q1.result).foreach(println)
```

Slick: Querying

```
// Perform a join to retrieve coffee names and supplier names for
// all coffees costing less than $9.00
val q2 = for {
  c <- coffees if c.price < 9.0
  s <- suppliers if s.id === c.supID
} yield (c.name, s.name)
// Equivalent SQL code:
// select c.COF_NAME, s.SUP_NAME from COFFEES c, SUPPLIERS s where c.PRICE < 9.0 and s.SUP_ID = c.SUP_ID
```

Slick: Querying

```
val q3 = for {
  c <- coffees if c.price < 9.0
  s <- c.supplier
} yield (c.name, s.name)
// Equivalent SQL code:
// select c.COF_NAME, s.SUP_NAME from COFFEES c, SUPPLIERS s where c.PRICE < 9.0 and s.SUP_ID = c.SUP_ID
```

Doobie

- https://github.com/tpolecat/doobie
- http://tpolecat.github.io/doobie/book.html
- https://www.scala-exercises.org/doobie/connecting_to_database

Doobie: Fast start

```
import doobie._
import doobie.implicits._
import cats.effect.IO

val xa = Transactor.fromDriverManager[IO](
  "org.postgresql.Driver", "jdbc:postgresql:world", "postgres", ""
)

case class Country(code: String, name: String, population: Long)

def find(n: String): ConnectionIO[Option[Country]] =
  sql"select code, name, population from country where name = $n".query[Country].option

find("France").transact(xa).unsafeRunSync
res3: Option[Country] = Some(Country(FRA,France,59225700))
```

Doobie: Examples

```
scala> val program2 = sql"select 42".query[Int].unique
program2: doobie.ConnectionIO[Int] = Free(...)

scala> val io2 = program2.transact(xa)
io2: cats.effect.IO[Int] = IO$73654901

scala> io2.unsafeRunSync
res2: Int = 42
```

Doobie: Examples

```
scala> val program3a = {
  val a: ConnectionIO[Int] = sql"select 42".query[Int].unique
  val b: ConnectionIO[Double] = sql"select random()".query[Double].unique
  (a, b).tupled
}

scala> program3a.transact(xa).unsafeRunSync
res4: (Int, Double) = (42,0.799691379070282)
```

Doobie: Examples

```
scala> val valuesList = program3a.replicateA(5)
valuesList: doobie.ConnectionIO[List[(Int, Double)]] = Free(...)

scala> val result = valuesList.transact(xa)
result: cats.effect.IO[List[(Int, Double)]] = IO$1275612355

scala> result.unsafeRunSync.foreach(println)
(42,0.03204446332529187)
(42,0.2510692561045289)
(42,0.5603550099767745)
(42,0.2610341371037066)
(42,0.13996110623702407)
```

Doobie: Examples

```
scala> {
     |   sql"select name from country"
     |     .query[String]    // Query0[String]
     |     .to[List]         // ConnectionIO[List[String]]
     |     .transact(xa)     // IO[List[String]]
     |     .unsafeRunSync    // List[String]
     |     .take(5)          // List[String]
     |     .foreach(println) // Unit
     | }
Afghanistan
Netherlands
Netherlands Antilles
Albania
Algeria
```

Doobie: Examples

```
scala> {
     |   sql"select name from country"
     |     .query[String]    // Query0[String]
     |     .stream           // Stream[ConnectionIO, String]
     |     .take(5)          // Stream[ConnectionIO, String]
     |     .compile.toList   // ConnectionIO[List[String]]
     |     .transact(xa)     // IO[List[String]]
     |     .unsafeRunSync    // List[String]
     |     .foreach(println) // Unit
     | }
Afghanistan
Netherlands
Netherlands Antilles
Albania
Algeria
```

Doobie: Examples

```
case class Country(code: String, name: String, pop: Int, gnp: Option[Double])
```

```
scala> {
     |   sql"select code, name, population, gnp from country"
     |     .query[Country]
     |     .stream
     |     .take(5)
     |     .quick
     |     .unsafeRunSync
     | }
  Country(AFG,Afghanistan,22720000,Some(5976.0))
  Country(NLD,Netherlands,15864000,Some(371362.0))
  Country(ANT,Netherlands Antilles,217000,Some(1941.0))
  Country(ALB,Albania,3401200,Some(3205.0))
  Country(DZA,Algeria,31471000,Some(49982.0))
```

Doobie: Examples

```
scala> def populationIn(range: Range) = sql"""
     |   select code, name, population, gnp
     |   from country
     |   where population > ${range.min}
     |   and   population < ${range.max}
     | """.query[Country]
populationIn: (range: Range)doobie.Query0[Country]

scala> populationIn(150000000 to 200000000).quick.unsafeRunSync
  Country(BRA,Brazil,170115000,Some(776739.0))
  Country(PAK,Pakistan,156483000,Some(61289.0))
```

Doobie: Examples

```
case class Country(code: Int, name: String, pop: Int, gnp: Double)

def biggerThan(minPop: Short) =
  sql"""
    select code, name, population, gnp, indepyear
    from country
    where population > $minPop
  """.query[Country]

scala> biggerThan(0).check.unsafeRunSync

      select code, name, population, gnp, indepyear
      from country
      where population > ?
```

Doobie: Examples

```
✓ SQL Compiles and Typechecks
  ✕ P01 Short  →  INTEGER (int4)
    - Short is not coercible to INTEGER (int4) according to the JDBC specification.
      Fix this by changing the schema type to SMALLINT, or the Scala type to Int or
      JdbcType.
  ✕ C01 code       CHAR     (bpchar)  NOT NULL  →  Int
    - CHAR (bpchar) is ostensibly coercible to Int according to the JDBC specification
      but is not a recommended target type. Fix this by changing the schema type to
      INTEGER; or the Scala type to Code or String.
  ✓ C02 name       VARCHAR  (varchar) NOT NULL  →  String
  ✓ C03 population INTEGER  (int4)    NOT NULL  →  Int
  ✕ C04 gnp        NUMERIC  (numeric) NULL      →  Double
    - NUMERIC (numeric) is ostensibly coercible to Double according to the JDBC
      specification but is not a recommended target type. Fix this by changing the
      schema type to FLOAT or DOUBLE; or the Scala type to BigDecimal or BigDecimal.

....
```

Doobie: Examples

```
case class Country(code: String, name: String, pop: Int, gnp: Option[BigDecimal])
scala> biggerThan(0).check.unsafeRunSync

      select code, name, population, gnp
      from country
      where population > ?

  ✓ SQL Compiles and Typechecks
  ✓ P01 Int  →  INTEGER (int4)
  ✓ C01 code        CHAR    (bpchar)  NOT NULL  →  String
  ✓ C02 name        VARCHAR (varchar) NOT NULL  →  String
  ✓ C03 population  INTEGER (int4)    NOT NULL  →  Int
  ✓ C04 gnp         NUMERIC (numeric) NULL      →  Option[BigDecimal]
```

Doobie: Examples

```
scala> val a = fr"select name from country"
a: doobie.util.fragment.Fragment = Fragment("select name from country ")

scala> val b = fr"where code = 'USA'"
b: doobie.util.fragment.Fragment = Fragment("where code = 'USA' ")

scala> val c = a ++ b // concatenation by ++
c: doobie.util.fragment.Fragment = Fragment("select name from country where code = 'USA' ")

scala> c.query[String].unique.quick.unsafeRunSync
  United States
```

Doobie: Examples

```
import doobie.postgres._


def safeInsert(s: String): ConnectionIO[Either[String, Person]] =
  insert(s).attemptSomeSqlState {
    case sqlstate.class23.UNIQUE_VIOLATION => "Oops!"
  }
```

Play anorm

- https://www.playframework.com/documentation/2.6.x/ScalaAnorm
- https://gist.github.com/davegurnell/4b432066b39949850b04

Play anorm: Examples

```
import anorm._

database.withConnection { implicit c =>
  val result: Boolean = SQL("Select 1").execute()
}
```

Play anorm: Examples

```
val id: Option[Long] =
  SQL("insert into City(name, country) values ({name}, {country})")
  .on('name -> "Cambridge", 'country -> "New Zealand").executeInsert()
```

Play anorm: Examples

```
import anorm.SqlParser.{ str, float }
// Parsing column by name or position
val parser =
  str("name") ~ float(3) /* third column as float */ map {
    case name ~ f => (name -> f)
  }

val product: (String, Float) = SQL("SELECT * FROM prod WHERE id = {id}").
  on('id -> "p").as(parser.single)
```

Play anorm: Examples

```
val lang = "French"
val population = 10000000
val margin = 500000

val code: String = SQL"""
  select * from Country c
    join CountryLanguage l on l.CountryCode = c.Code
    where l.Language = $lang and c.Population >= ${population - margin}
    order by c.Population desc limit 1"""
  .as(SqlParser.str("Country.code").single
```

Play anorm: Examples

```
import anorm.{ Macro, RowParser }

case class Info(name: String, year: Option[Int])

val parser: RowParser[Info] = Macro.namedParser[Info]
/* Generated as:
get[String]("name") ~ get[Option[Int]]("year") map {
  case name ~ year => Info(name, year)
}
*/

val result: List[Info] = SQL"SELECT * FROM list".as(parser.*)
```

Play anorm: Streaming Examples

```
val countryCount: Either[List[Throwable], Long] =
  SQL"Select count(*) as c from Country".fold(0L) { (c, _) => c + 1 }
```

Scalikejdbc

- http://scalikejdbc.org/
- https://habrahabr.ru/post/256545/
- http://blog.seratch.net/post/112407302678/why-is-scalikejdbc-efficient-when-working-with
- http://skinny-framework.org/documentation/orm.html
- http://skinny-framework.org/documentation/micro.html

Scalikejdbc: Examples

```
import scalikejdbc._

// initialize JDBC driver & connection pool
Class.forName("org.h2.Driver")
ConnectionPool.singleton("jdbc:h2:mem:hello", "user", "pass")

// ad-hoc session provider on the REPL
implicit val session = AutoSession

// table creation, you can run DDL by using #execute as same as JDBC
sql"""
create table members (
  id serial not null primary key,
  name varchar(64),
  created_at timestamp not null
)
""".execute.apply()
```

Scalikejdbc: Examples

```
// insert initial data
Seq("Alice", "Bob", "Chris") foreach { name =>
  sql"insert into members (name, created_at) values (${name}, current_timestamp)".update.apply()
}

// for now, retrieves all data as Map value
val entities: List[Map[String, Any]] = sql"select * from members".map(_.toMap).list.apply()

// defines entity object and extractor
import java.time._
case class Member(id: Long, name: Option[String], createdAt: ZonedDateTime)
object Member extends SQLSyntaxSupport[Member] {
  override val tableName = "members"
  def apply(rs: WrappedResultSet) = new Member(
    rs.long("id"), rs.stringOpt("name"), rs.zonedDateTime("created_at"))
}
```

Scalikejdbc: Examples

```
// find all members
val members: List[Member] = sql"select * from members".map(rs => Member(rs)).list.apply()

// use paste mode (:paste) on the Scala REPL
val m = Member.syntax("m")
val name = "Alice"
val alice: Option[Member] = withSQL {
  select.from(Member as m).where.eq(m.name, name)
}.map(rs => Member(rs)).single.apply()
```

Scalikejdbc: Examples

```
val name = "Alice"
// implicit session represents java.sql.Connection
val memberId: Option[Long] = DB readOnly { implicit session =>
  sql"select id from members where name = ${name}" // don't worry, prevents SQL injection
    .map(rs => rs.long("id")) // extracts values from rich java.sql.ResultSet
    .single                   // single, list, traversable
    .apply()                  // Side effect!!! runs the SQL using Connection
}
```

Scalikejdbc: Examples

```
val (p, c) = (Programmer.syntax("p"), Company.syntax("c"))

val programmers: Seq[Programmer] = DB.readOnly { implicit session =>
  withSQL {
    select
      .from(Programmer as p)
      .leftJoin(Company as c).on(p.companyId, c.id)
      .where.eq(p.isDeleted, false)
      .orderBy(p.createdAt)
      .limit(10)
      .offset(0)
  }.map(Programmer(p, c)).list.apply()
}
```

Scalikejdbc: Examples

```
object Product {
  def create(name: String, price: Long)(implicit s: DBSession = AutoSession): Long = {
    sql"insert into products values (${name}, ${price})"
      .updateAndReturnGeneratedKey.apply() // returns auto-incremeneted id
  }
  def findById(id: Long)(implicit s: DBSession = AutoSession): Option[Product] = {
    sql"select id, name, price, created_at from products where id = ${id}"
      .map { rs => Product(rs) }.single.apply()
  }
}
Product.findById(123) // borrows connection from pool and gives it back after execution
DB localTx { implicit session => // transactional session
  val id = Product.create("ScalikeJDBC Cookbook", 200) // within transaction
  val product = Product.findById(id) // within transaction
}
```

Scalikejdbc: Examples

```
import scalikejdbc._

val id = 123

// simple example
val name: Option[String] = DB readOnly { implicit session =>
  sql"select name from emp where id = ${id}".map(rs => rs.string("name")).single.apply()
}

// defined mapper as a function
val nameOnly = (rs: WrappedResultSet) => rs.string("name")
val name: Option[String] = DB readOnly { implicit session =>
  sql"select name from emp where id = ${id}".map(nameOnly).single.apply()
}
```

Scalikejdbc: Examples

```
// define a class to map the result
case class Emp(id: String, name: String)
val emp: Option[Emp] = DB readOnly { implicit session =>
  sql"select id, name from emp where id = ${id}"
    .map(rs => Emp(rs.string("id"), rs.string("name"))).single.apply()
}

// QueryDSL
object Emp extends SQLSyntaxSupport[Emp] {
  def apply(e: ResultName[Emp])(rs: WrappedResultSet): Emp = new Emp(id = rs.get(e.id), name = rs.get(e.name))
}
val e = Emp.syntax("e")
val emp: Option[Emp] = DB readOnly { implicit session =>
  withSQL { select.from(Emp as e).where.eq(e.id, id) }.map(Emp(e.resultName)).single.apply()
}
```

Scalikejdbc: Examples

```
import scalikejdbc._
DB localTx { implicit session =>
  val batchParams: Seq[Seq[Any]] = (2001 to 3000).map(i => Seq(i, "name" + i))
  sql"insert into emp (id, name) values (?, ?)".batch(batchParams: _*).apply()
}
DB localTx { implicit session =>
  sql"insert into emp (id, name) values ({id}, {name})"
    .batchByName(Seq(Seq('id -> 1, 'name -> "Alice"), Seq('id -> 2, 'name -> "Bob")):_*)
    .apply()
}
val column = Emp.column
DB localTx { implicit session =>
  val batchParams: Seq[Seq[Any]] = (2001 to 3000).map(i => Seq(i, "name" + i))
  withSQL {
    insert.into(Emp).namedValues(column.id -> sqls.?, column.name -> sqls.?)
  }.batch(batchParams: _*).apply()
}
```

Scalikejdbc: Examples

https://github.com/scalikejdbc/scalikejdbc-async

```
// set up connection pool (that's all you need to do)
AsyncConnectionPool.singleton("jdbc:postgresql://localhost:5432/scalikejdbc", "sa", "sa")

// create a new record within a transaction
val created: Future[Company] = AsyncDB.localTx { implicit tx =>
  for {
    company <- Company.create("ScalikeJDBC, Inc.", Some("http://scalikejdbc.org/"))
    seratch <- Programmer.create("seratch", Some(company.id))
    gakuzzzz <- Programmer.create("gakuzzzz", Some(company.id))
    xuwei_k <- Programmer.create("xuwei-k", Some(company.id))
  } yield company
}
```

MongoDB

Распределенная нереляционная документо-ориентированная база данных с открытым исходным кодом

https://www.mongodb.com/

https://university.mongodb.com/

https://docs.mongodb.com/ecosystem/drivers/scala/

MongoDB

- JSON - подобные документы
- Язык запросов на основе JavaScript
- Индексация
- Репликация, балансировка нагрузки
- Агрегация, MapReduce
- Использование в качестве файлового хранилища
- ...

MongoDB: пример документа

```
{
    "_id": "123",
    "params": {
        "x": 15,
        "y": [1, 2, 3]
    }
}
```

Документы содержатся в коллекциях - аналог таблиц в реляционных БД, коллекции объединены в базы

MongoDB: запросы

```
> db.test.find({_id: "123"})
{ "_id" : "123", "params" : { "x" : 15, "y" : [ 1, 2, 3 ] } }

> db.test.update({ _id: "123"}, { $set: {"params.x" : 24}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })

> db.test.insert({_id: "345", params: { x: 23, y: [5, 6]}})
WriteResult({ "nInserted" : 1 })

> db.test.remove({_id: "345"})
WriteResult({ "nRemoved" : 1 })
```

Casbah

Scala - обертка для Java-библиотеки к MongoDB

https://mongodb.github.io/casbah/

build.sbt

```
libraryDependencies += "org.mongodb" %% "casbah" % "3.1.1"
```

Casbah

```
val host = "localhost:27017"
val db = "test"
val collection = "test"

val mongoClient: MongoClient = MongoClient(host)
val dbClient: MongoDB = MongoClient(host)(db)
val collectionClient: MongoCollection = MongoClient(host)(db)(collection)
```

Casbah

```
import com.mongodb.casbah.Imports._
import com.mongodb.{BasicDBList, DBObject}
import scala.collection.JavaConverters._

case class Entity(_id: String, params: Param)
case class Param(x: Int, y: Seq[Int])
```

Casbah

```
val res: Option[Entity]= collectionClient.findOneByID( "333").map { o =>
  val id = o.get("_id").toString // original method
  val params = o.get("params").asInstanceOf[DBObject]
  val x = params.as[Int]("x") // implicit method
  val y =
params.get("y").asInstanceOf[BasicDBList].iterator().asScala.toList.map(_.to
String.toInt)

  Entity(id, Param(x, y))
}
```

Casbah: запросы

```
val cursor = collectionClient.find(MongoDBObject("params.x" -> 15))

val ids = cursor.map { o =>
  o.get("_id").toString
}.toList

cursor.close() // Важно не забыть закрыть курсор
```

Casbah: запросы

В Casbah есть implicit - методы, позволяющие писать запросы на DSL, близком к оригинальному языку запросов MongoDB

```
import com.mongodb.casbah.Imports._

val rangeCursor = collectionClient.find("params.x" $gt 14 $lt 16)
val rangeIds = rangeCursor.map { o =>
  o.get("_id").toString
}.toList

rangeCursor.close()
```

Casbah: insert

```
val writeResult: WriteResult = collectionClient.insert(
  MongoDBObject("_id" -> "001", "params" ->
    MongoDBObject("x" -> 0, "y" -> List(0, 0, 1))
  )
)

writeResult.getN
writeResult.isUpdateOfExisting
```

Casbah: remove

```
val removedDoc: Option[DBObject] =
collectionClient.findAndRemove( MongoDBObject ("_id" -> "001"))
// удаляет первый документ, удовлетворяющий запросу и возвращает его (если
он был)

val removeResult: WriteResult = collectionClient.remove( MongoDBObject ("_id"
-> "0011", "params" -> MongoDBObject ("x" -> 1, "y" -> List(0, 0, 1))))
// удаляет документ, полностью соответствующий переданному
```

Casbah: update / upsert

```
  val updateResult: WriteResult = collectionClient.update(
    MongoDBObject("_id" -> "0011"),
    $set("params.x" -> 1),
    upsert = false,
    multi = true) // Если multi == false, будет проапдейчен только первый
подходящий документ
```

Casbah: save

```
val saveResult: WriteResult = collectionClient.save(
  MongoDBObject("_id" ->  "9", "params" ->
    MongoDBObject("x" -> 1, "y" -> List(0, 0, 1))
  )
)

// Если объект не содержит поле _id, будет создан новый объект, если
содержит, будет выполнен upsert
```

Casbah: и многое другое

- Авторизация
- WriteConcern
- Приоритет master / slave нод
- Aggregations

Salat

Позволяет работать с MongoDB объектами через case-class, внутри использует Casbah

https://github.com/salat/salat

build.sbt

```
libraryDependencies += "com.novus.salat" %% "salat" % "1.11.2"
```

Salat: DAO

```
import com.mongodb.casbah.Imports._
import com.novus.salat.dao.SalatDAO
import com.novus.salat.global._


class EntityDAO extends SalatDAO[Entity, String]
    (collection = MongoConnection()("test")("test"))
// Первый type-param - тип DTO, второй - тип поля _id в DTO
```

Salat

В `SalatDAO` имеются все те же методы для работы с MongoDB, что и в Casbah, но они возвращают не `DBObject`, а объект case-class DTO

```
val dao = new EntityDAO
val entity: Option[Entity] = dao.findOneById("333")
```

Salat: запросы

Для запросов используется Casbah - синтаксис, но возвращается `SalatMongoCursor[Entity]`

```
val cur: SalatMongoCursor[Entity] = dao.find(MongoDBObject("params.x" ->
15))

val res: List[Entity] = cur.toList
cur.close()
```

Salat: insert

```
val maybeId: Option[String] = dao.insert(Entity("4", Param(4, List(4, 5,
6))))

val ids: List[Option[String]] = dao.insert(List(Entity("5",
Param(5,List(1))),
    Entity("6", Param(6, List.empty))))

// Some(id), если запись успешна, None - не успешна
```

Salat: remove

```
val removeByIdResult = dao.removeById("4")

val removeByQueryResult = dao.remove(MongoDBObject("_id" -> "5"))

val removeEntityResult = dao.remove(Entity("6", Param(6, List.empty)))
```

Salat: update / upsert

Update / upsert реализован так же, как в Casbah

```
val updateResult: WriteResult = dao.update(

   MongoDBObject("_id" -> "4"), $set("params.x" -> 18), upsert = false)
```

Salat: save

Метод `.save` работает аналогично Casbah: если объект не существует, он будет создан, если существует - выполнится update

```
val saveResult: WriteResult = dao.save(Entity("5", Param(1, List(1))))
```

Salat: тонкие моменты

- https://github.com/novus/salat#what-doesnt-salat-support
- Быть аккуратным с численными типами (Int, Double, ..), особенно, если запись в коллекцию идет не только через SalatDAO
- Если в одном проекте и Casbah, и Salat, могут быть конфликты из-за transitive-dependency

https://sysgears.com/articles/how-to-build-a-simple-mongodb-dao-in-scala-using-salatdao/

Спасибо