Functional Spark

Взгляд скалиста на Spark-разработку

Дима Зуев

Дима Зуев

- ✔ Программирую за деньги с 2012
- ✔ Заболел скалой в 2014 году
- ✔ С 2017 года руковожу DE
- ✔ Мы с моей командой используем все возможности скалы в продакшене

Функциональное программирование

- Чистые функции
- Переменные неизменяемы
- Ссылочная прозрачность
- Ленивые вычисления

Чистые функции

```
def sum(a: Int, b: Int): Int = a + b

def map[U : Encoder](
           func: T ⇒ U): Dataset[U]
```

```
def prettyPrint(a: Int): Unit =
print(s"Int: $a")

def json(path: String): Unit = {
 format("json").save(path)
}
```

Ленивые вычисления

```
lazy val a = 10 + 10

def map[U : Encoder](func: T ⇒ U): Dataset[U]

def flatMap[U : Encoder](func: T ⇒ TraversableOnce[U]):
Dataset[U]
```

Ссылочная прозрачность

```
var i: Int = 10
def add(j: Int): Int = {
  i=i+j
  i
}
val k: Int = add(20) // 30
val l: Int = add(k) // 60
```

```
val i = 10
def add(i0: Int, j0: Int): Int = i0+j0
val k: Int = add(i, 20) // 30
val l = add(i, k) // 40

val l = add(i, add(i, 20))
val l = add(10, add(10, 20))
```

Эффекты

- Effectful code — код, которому необходимо взаимодействовать с “внешним миром”, отделенному от нашей программы. Например чтение/запись из БД
- IO эффекты предоставляют способ “встроить” грязный код в чистую программу и сохранить ссылочную прозрачность
- Эффекты можно композировать

Чистота и ленивость в Spark

Ленивые вычисления - **стратегия вычисления**, согласно которой вычисления следует откладывать до тех пор, пока не понадобится их результат.

Ленивое вычисление в Spark означает, что выполнение не начнется, пока **не будет запущен Action**. В Spark ленивые вычисления возникают, когда описываются преобразования.

Преобразования являются ленивыми по своей природе, что означает, что **когда мы вызываем некоторую операцию над DS, она не выполняется немедленно.**

Поскольку преобразования носят ленивый характер, мы можем выполнять операцию в любое время, вызывая action. Следовательно, при ленивых вычислениях данные не загружаются до тех пор, пока в этом нет необходимости.

Чистота и ленивость в Spark

Чистые и ленивые

```
spark.read.json
spark.read.csv
spark.read.table
spark.read.sql

ds.map
ds.join
ds.cache
ds.unpersist
```

Грязные и жадные

```
ds.write.csv
ds.write.json
ds.write.parquet
ds.write.saveAsTable
ds.show
ds.collect
```

Чистота и ленивость в Spark

Чистые и ленивые

```
ds.map
ds.join
ds.cache
ds.unpersist
```

Грязные и жадные

```
ds.write.csv
ds.write.json
ds.write.parquet
ds.write.saveAsTable
ds.show
ds.collect
spark.read.json
spark.read.csv
spark.read.table
spark.read.sql
```

spark.read

1. Добавляет в **план выполнения** чтение из источника.
2. Spark автоматически **выводит схему из источника** и представляет типы источника в типы спарка

Типичная джоба

```scala
val cfg = ConfigFactory.load()

val vaultConfig: VaultConfig = new VaultConfig()
  .address(cfg.getString("vault.address"))
  .token(cfg.getString("vault.token"))
  .build()

val partition: String   = args(0)
val tableSuffix: String = args(1)

val vault: Vault = new Vault(vaultConfig)

val verticaOpts =
vault.logical().read("etl/dbs/vertica").getData.asScala

val spark = SparkSession.builder.appName("Simple
Application").getOrCreate()
```

```scala
import spark.implicits._

val dimClientDs =
broadcast(spark.read.table("dim_client").as[data.Client].a
lias("clients").cache())

val factCallDs = spark.read
  .csv(s"fact_call/date=$partition/")
  .withColumn(
    "startAt",
    to_utc_timestamp($"startAt", "MSK")
  )
  .withColumn(
    "endAt",
    to_utc_timestamp($"endAt", "MSK")
  )
  .as[data.Call]
  .alias("calls")
```

Типичная джоба

1

```
val cfg = ConfigFactory.load()

val vaultConfig: VaultConfig = new VaultConfig()
  .address(cfg.getString("vault.address"))
  .token(cfg.getString("vault.token"))
  .build()

val partition: String   = args(0)
val tableSuffix: String = args(1)

val vault: Vault = new Vault(vaultConfig)

val verticaOpts =
vault.logical().read("etl/dbs/vertica").getData.asScala

val spark = SparkSession.builder.appName("Simple
Application").getOrCreate()
```

```
import spark.implicits._

val dimClientDs =
broadcast(spark.read.table("dim_client").as[data.Client].a
lias("clients").cache())

val factCallDs = spark.read
  .csv(s"fact_call/date=$partition/")
  .withColumn(
    "startAt",
    to_utc_timestamp($"startAt", "MSK")
  )
  .withColumn(
    "endAt",
    to_utc_timestamp($"endAt", "MSK")
  )
  .as[data.Call]
  .alias("calls")
```

Типичная джоба

```scala
val cfg = ConfigFactory.load()

val vaultConfig: VaultConfig = new VaultConfig()
  .address(cfg.getString("vault.address"))
  .token(cfg.getString("vault.token"))
  .build()

val partition: String   = args(0)
val tableSuffix: String = args(1)

val vault: Vault = new Vault(vaultConfig)

val verticaOpts =
vault.logical().read("etl/dbs/vertica").getData.asScala

val spark = SparkSession.builder.appName("Simple
Application").getOrCreate()
```

2

```scala
import spark.implicits._

val dimClientDs =
broadcast(spark.read.table("dim_client").as[data.Client].a
lias("clients").cache())

val factCallDs = spark.read
  .csv(s"fact_call/date=$partition/")
  .withColumn(
    "startAt",
    to_utc_timestamp($"startAt", "MSK")
  )
  .withColumn(
    "endAt",
    to_utc_timestamp($"endAt", "MSK")
  )
  .as[data.Call]
  .alias("calls")
```

Типичная джоба

```scala
val cfg = ConfigFactory.load()

val vaultConfig: VaultConfig = new VaultConfig()
  .address(cfg.getString("vault.address"))
  .token(cfg.getString("vault.token"))
  .build()

val partition: String   = args(0)
val tableSuffix: String = args(1)

val vault: Vault = new Vault(vaultConfig)

val verticaOpts =
vault.logical().read("etl/dbs/vertica").getData.asScala

val spark = SparkSession.builder.appName("Simple
Application").getOrCreate()
```

3

```scala
import spark.implicits._

val dimClientDs =
broadcast(spark.read.table("dim_client").as[data.Client].a
lias("clients").cache())

val factCallDs = spark.read
  .csv(s"fact_call/date=$partition/")
  .withColumn(
    "startAt",
    to_utc_timestamp($"startAt", "MSK")
  )
  .withColumn(
    "endAt",
    to_utc_timestamp($"endAt", "MSK")
  )
  .as[data.Call]
  .alias("calls")
```

Типичная джоба

```scala
val cfg = ConfigFactory.load()

val vaultConfig: VaultConfig = new VaultConfig()
  .address(cfg.getString("vault.address"))
  .token(cfg.getString("vault.token"))
  .build()

val partition: String   = args(0)
val tableSuffix: String = args(1)

val vault: Vault = new Vault(vaultConfig)

val verticaOpts =
vault.logical().read("etl/dbs/vertica").getData.asScala
```

4

```scala
val spark = SparkSession.builder.appName("Simple
Application").getOrCreate()
```

```scala
import spark.implicits._

val dimClientDs =
broadcast(spark.read.table("dim_client").as[data.Client].a
lias("clients").cache())

val factCallDs = spark.read
  .csv(s"fact_call/date=$partition/")
  .withColumn(
    "startAt",
    to_utc_timestamp($"startAt", "MSK")
  )
  .withColumn(
    "endAt",
    to_utc_timestamp($"endAt", "MSK")
  )
  .as[data.Call]
  .alias("calls")
```

Типичная джоба

```scala
val cfg = ConfigFactory.load()

val vaultConfig: VaultConfig = new VaultConfig()
  .address(cfg.getString("vault.address"))
  .token(cfg.getString("vault.token"))
  .build()

val partition: String   = args(0)
val tableSuffix: String = args(1)

val vault: Vault = new Vault(vaultConfig)

val verticaOpts =
vault.logical().read("etl/dbs/vertica").getData.asScala

val spark = SparkSession.builder.appName("Simple
Application").getOrCreate()
```

```scala
import spark.implicits._

val dimClientDs =
broadcast(spark.read.table("dim_client").as[data.Client].a
lias("clients").cache())

val factCallDs = spark.read
  .csv(s"fact_call/date=$partition/")
  .withColumn(
    "startAt",
    to_utc_timestamp($"startAt", "MSK")
  )
  .withColumn(
    "endAt",
    to_utc_timestamp($"endAt", "MSK")
  )
  .as[data.Call]
  .alias("calls")
```

5

Типичная джоба

```
val factChatDs = spark.read
  .json(s"fact_chat/date=$partition/")
  .withColumn("messages", $"messages".withField("sendAt", to_timestamp($"messages.sendAt", "MM/dd/yyyy")))
  .as[data.ChatMessage]
  .alias("chats")


val calls = dimClientDs.join(factCallDs, $"clients.id" === $"calls.clientId")
val chats = dimClientDs.join(factChatDs, $"clients.id" === $"chats.clientId")
calls.write
  .format("com.vertica.spark.datasource.DefaultSource")
  .options(verticaOpts + ("table" → s"calls$tableSuffix"))
  .mode(SaveMode.Overwrite)
  .save()
chats.write
  .format("com.vertica.spark.datasource.DefaultSource")
  .options(verticaOpts + ("table" → s"chats$tableSuffix"))
  .mode(SaveMode.Overwrite)
  .save()


dimClientDs.unpersist()


spark.stop()
```

3, 6

Типичная джоба

```
val factChatDs = spark.read
  .json(s"fact_chat/date=$partition/")
  .withColumn("messages", $"messages".withField("sendAt", to_timestamp($"messages.sendAt", "MM/dd/yyyy")))
  .as[data.ChatMessage]
  .alias("chats")


val calls = dimClientDs.join(factCallDs, $"clients.id" === $"calls.clientId")
val chats = dimClientDs.join(factChatDs, $"clients.id" === $"chats.clientId")
calls.write
  .format("com.vertica.spark.datasource.DefaultSource")
  .options(verticaOpts + ("table" → s"calls$tableSuffix"))
  .mode(SaveMode.Overwrite)
  .save()
chats.write
  .format("com.vertica.spark.datasource.DefaultSource")
  .options(verticaOpts + ("table" → s"chats$tableSuffix"))
  .mode(SaveMode.Overwrite)
  .save()

5 dimClientDs.unpersist()

spark.stop()
```

Типичная джоба

```
val factChatDs = spark.read
  .json(s"fact_chat/date=$partition/")
  .withColumn("messages", $"messages".withField("sendAt", to_timestamp($"messages.sendAt", "MM/dd/yyyy")))
  .as[data.ChatMessage]
  .alias("chats")


val calls = dimClientDs.join(factCallDs, $"clients.id" === $"calls.clientId")
val chats = dimClientDs.join(factChatDs, $"clients.id" === $"chats.clientId")
calls.write
  .format("com.vertica.spark.datasource.DefaultSource")
  .options(verticaOpts + ("table" → s"calls$tableSuffix"))
  .mode(SaveMode.Overwrite)
  .save()
chats.write
  .format("com.vertica.spark.datasource.DefaultSource")
  .options(verticaOpts + ("table" → s"chats$tableSuffix"))
  .mode(SaveMode.Overwrite)
  .save()


dimClientDs.unpersist()

spark.stop()
```

4

Проблемы

1. Ошибка при парсинг конфига + Обращение к несуществующей части конфига
2. Ошибка при отсутствии аргументов
3. Обращение к несуществующим ключам из Vault
4. Непрозрачные границы использования Spark Session
5. Непрозрачные границы использования cached ds
6. Нет выраженного end-of-the-world

Что будем использовать?

```
"org.typelevel"          %% "cats-core"          // FP абстракции
"org.typelevel"          %% "cats-effect"        // Абстракции для эффектов
"com.monovore"           %% "decline"            // парсер аргументов
"com.github.pureconfig" %% "pureconfig"         // парсер конфига
"org.manatki"            %% "derevo-pureconfig" // аннотации для вывода парсера
"dev.zio"                %% "zio"                // Реализация эффектов
"io.7mind.izumi"         %% "distage-core"       // DI
```

Начинаем: чтение аргументов

```
val partition: String   = args(0)
val tableSuffix: String = args(1)
// --
java.lang.ArrayIndexOutOfBoundsException: Index 0 out of
bounds for length 0
```

```
object arguments {
  val partitionOpt  = Opts.option[String]("partition",
short = "p", metavar = "2021-09-21", help = "Partition
date")
  val chatsTableOpt = Opts.option[String]("chatsTable",
metavar = "chats_stg_table", help = "Chats target table")
  val callsTableOpt = Opts.option[String]("callsTable",
metavar = "calls_stg_table", help = "Calls  target table")

  val command = Command(
    name = "job",
    header = "Call center job",
    helpFlag = true
  )(
    (partitionOpt, chatsTableOpt, callsTableOpt).tupled
  )
}
arguments.command.parse(args, sys.env) match {
  case Left(help) ⇒
    System.err.println(help)
    sys.exit(1)
  case Right((partition, chatsTable, callsTable)) ⇒ {???}
}
///
Usage: job --partition <2021-09-21> --chatsTable
<chats_stg_table> --callsTable <calls_stg_table>
```

Продолжаем: чтение конфига

```
val cfg = ConfigFactory.load()


val vaultConfig: VaultConfig = new VaultConfig()
    .address(cfg.getString("vault.address"))
    .token(cfg.getString("vault.token"))
    .build()
// -
com.typesafe.config.ConfigException$Missing: system
properties: No configuration setting found for key 'vault'
```

```
ConfigSource
  .fromConfig(ConfigFactory.load())
  .load[config.AppConfig]
  .leftMap(errors.ApplicationError.ConfigError)

val vaultConfig: VaultConfig = new VaultConfig()
  .address(cfg.vault.address)
  .token(cfg.vault.token)
  .build()
// ---
at 'vault':
  - (application.conf @
file:/home/dmzuev/fp-spark/common/target/scala-2.12/classe
s/application.conf: 1) Key not found: 'address'.
  - (application.conf @
file:/home/dmzuev/fp-spark/common/target/scala-2.12/classe
s/application.conf: 1) Key not found: 'token'.
```

Продолжаем: чтение из Vault

```
val verticaOpts =
vault.logical().read("etl/dbs/vertica").getData


verticaOpts("host")
// -
java.util.NoSuchElementException: key not found: host
```

```
(
  Validated.fromOption(cfgMap.get("host"),
   "host is not set"),
  Validated.fromOption(cfgMap.get("user"),
   "user is not set"),
  Validated.fromOption(cfgMap.get("db"),
   "db is not set"),
  Validated.fromOption(cfgMap.get("password"),
   "password is not set"),
  Validated.fromOption(cfgMap.get("stagingFsUrl"),
   "stagingFsUrl is not set"),
).tupled
  .toValidatedNel

.leftMap(errors.ApplicationError.VerticaConfigurationInitiali
zationError)
//

Vault cfg failures:
host is not set
user is not set
db is not set
password is not set
stagingFsUrl is not set
```

Ресурсы

- Распространенный паттерн - получить ресурс (например, файл или сокет), выполнить с ним какое-либо действие, а затем запустить финализатор (например, закрыть дескриптор файла), независимо от результата действия.
- Resource это абстракция, которая инкапсулирует логику для получения и завершения ресурса.
- Resource предоставляет возможности для композиции - можно создавать составные ресурсы

FP style try … catch .. finally

```
var ss: SparkSession = null
try {
 ss = SparkSession
   .builder
   .appName("Application")
   .getOrCreate()
} catch {
 case e ⇒ throw e
}
finally {
 ss.stop()
}
```

```
Resource
  .make(
    Task(
      SparkSession
        .builder
        .appName("Application")
        .getOrCreate()
    )
  )(
    ss ⇒ Task(ss.stop())
  )
```

ZIO

`Type-safe, composable asynchronous and concurrent programming for Scala`
ZIO — A ZIO is a value that models an **effectful program**, which might fail or succeed.

ZManaged = Resource

Ух бл… (Контроль ресурсов)

```scala
val cfg = ConfigFactory.load()

val partition: String   = args(0)
val tableSuffix: String = args(1)

val spark = SparkSession
  .builder
  .appName("Application")
  .getOrCreate()
```

```scala
(for {
  cfg       ← ZManaged.fromEither(
    ConfigSource
      .fromConfig(ConfigFactory.load())
      .load[config.AppConfig]
      .leftMap(errors.ApplicationError.ConfigError)
  )
  arguments ← ZManaged.fromEither(
      arguments.command.parse(args, sys.env).leftMap(
      errors.ApplicationError.CommandlineArgumentsError)
    )
  spark     ← ZManaged
    .make(Task(SparkSession
      .builder
      .appName("Application")
    .getOrCreate()))(ss ⇒ UIO(ss.stop()))
    .mapError(errors.ApplicationError.ExecutionError)
} yield runtime.ContextV1(cfg,
runtime.Args.tupled(arguments), spark))
  .use(ctx ⇒ {??? })
```

Ух бл… (Контроль ресурсов)

```scala
val cfg = ConfigFactory.load()

val partition: String   = args(0)
val tableSuffix: String = args(1)

val spark = SparkSession
  .builder
  .appName("Application")
  .getOrCreate()
```

```scala
(for {
  cfg       ← ZManaged.fromEither(
    ConfigSource
      .fromConfig(ConfigFactory.load())
      .load[config.AppConfig]
      .leftMap(errors.ApplicationError.ConfigError)
  )
  arguments ← ZManaged.fromEither(
      arguments.command.parse(args, sys.env).leftMap(
      errors.ApplicationError.CommandlineArgumentsError)
    )
  spark     ← ZManaged
    .make(Task(SparkSession
      .builder
      .appName("Application")
    .getOrCreate()))(ss ⇒ UIO(ss.stop()))
    .mapError(errors.ApplicationError.ExecutionError)
} yield runtime.ContextV1(cfg,
runtime.Args.tupled(arguments), spark))
  .use(ctx ⇒ {??? })
```

Ух бл… (Контроль ресурсов)

```scala
val cfg = ConfigFactory.load()

val partition: String   = args(0)
val tableSuffix: String = args(1)

val spark = SparkSession
  .builder
  .appName("Application")
  .getOrCreate()
```

```scala
(for {
  cfg       ← ZManaged.fromEither(
    ConfigSource
      .fromConfig(ConfigFactory.load())
      .load[config.AppConfig]
      .leftMap(errors.ApplicationError.ConfigError)
  )
  arguments ← ZManaged.fromEither(
      arguments.command.parse(args, sys.env).leftMap(
      errors.ApplicationError.CommandlineArgumentsError)
    )
  spark     ← ZManaged
    .make(Task(SparkSession
      .builder
      .appName("Application")
    .getOrCreate()))(ss ⇒ UIO(ss.stop()))
    .mapError(errors.ApplicationError.ExecutionError)
} yield runtime.ContextV1(cfg,
runtime.Args.tupled(arguments), spark))
  .use(ctx ⇒ {??? })
```

Ух бл… (Контроль ресурсов)

```
val cfg = ConfigFactory.load()

val partition: String   = args(0)
val tableSuffix: String = args(1)

val spark = SparkSession
  .builder
  .appName("Application")
  .getOrCreate()
```

```
(for {
  cfg       ← ZManaged.fromEither(
    ConfigSource
      .fromConfig(ConfigFactory.load())
      .load[config.AppConfig]
      .leftMap(errors.ApplicationError.ConfigError)
  )
  arguments ← ZManaged.fromEither(
      arguments.command.parse(args, sys.env).leftMap(
      errors.ApplicationError.CommandlineArgumentsError)
    )
  spark     ← ZManaged
    .make(Task(SparkSession
      .builder
      .appName("Application")
    .getOrCreate()))(ss ⇒ UIO(ss.stop()))
    .mapError(errors.ApplicationError.ExecutionError)
} yield runtime.ContextV1(cfg,
runtime.Args.tupled(arguments), spark))
  .use(ctx ⇒ {??? })
```

Cached DS – это ресурс

```
object functions {

  def cache[T](ds: Dataset[T]): ZManaged[Any, Throwable, Dataset[T]]     =
    ZManaged.make(Task(ds.cache()))(ds0 ⇒ UIO(ds0.unpersist()))

  def broadcast[T](ds: Dataset[T]): ZManaged[Any, Throwable, Dataset[T]] = cache(ds).map(sbroadcast)

}
…

(ZManaged.fromEffect(Task(ctx.spark.read.table("tbl"))) >>= func.broadcast).use(cachedDS ⇒ {})
```

Effectful операции

```
class read(spark: SparkSession) {
  def table[T: Encoder](table: String): Task[Dataset[T]] =
Task(spark.read.table(table).as[T])
  def csv[T: Encoder](path: String): Task[Dataset[T]]    =
Task(spark.read.csv(path).as[T])
  def json[T: Encoder](path: String): Task[Dataset[T]]   =
Task(spark.read.json(path).as[T])
  def text(path: String): Task[Dataset[String]]          =
Task(spark.read.textFile(path))
}


object functions {
  def cache[T](ds: Dataset[T]): ZManaged[Any, Throwable,
Dataset[T]]     =
    ZManaged.make(Task(ds.cache()))(ds0 ⇒
UIO(ds0.unpersist()))
  def broadcast[T](ds: Dataset[T]): ZManaged[Any,
Throwable, Dataset[T]] = cache(ds).map(sbroadcast)
}
```

```
object write {
  def vertica[T](
      host: String,
      user: String,
      db: String,
      password: String,
      stagingFsUrl: String,
  )(table: String, mode: SaveMode)(ds: Dataset[T]):
Task[Unit] = Task(
    ds.write

.format("com.vertica.spark.datasource.DefaultSource")
      .options(
        Map(
          "host"           → host,
          "user"           → user,
          "db"             → db,
          "password"       → password,
          "staging_fs_url" → stagingFsUrl,
          "table"          → table,
        )
      )
      .mode(mode)
      .save()
  )
}
```

Джоба – это один большой эффект

```
val eff: zio.IO[errors.ApplicationError, Unit] = (ZManaged.fromEffect(
  read.table[data.Call]("dim_client")
) >>= func.broadcast).use(dimClientDs => {
  (
    for {
      factCallRawDs <- read.csv[data.Call](s"fact_call/date=${ctx.args.partition}/")
      factCallDs     = factCallRawDs.map(call =>
                         call.copy(
                           startAt = call.startAt.minus(3, ChronoUnit.HOURS),
                           endAt = call.endAt.minus(3, ChronoUnit.HOURS),
                         )
                       )
      factChatDs    <- read.json[data.ChatMessage](s"fact_chat/date=${ctx.args.partition}/")


      calls = dimClientDs
                .join(factCallDs, $"clients.id" === $"calls.clientId")
                .as[data.ClientCall]
      chats = dimClientDs
                .join(factChatDs, $"clients.id" === $"chats.clientId")
                .as[data.ChatMessage]
      _    <- verticaWriter(ctx.args.callsTable, SaveMode.Overwrite)(calls)
      _    <- verticaWriter(ctx.args.chatsTable, SaveMode.Overwrite)(chats)
    } yield ()
  ).mapError(errors.ApplicationError.ExecutionError)
})
```

Запуск эффекта и обработка ошибок

```
zio.Runtime.default.unsafeRunSync(eff) match {
  case Exit.Success(value) ⇒ sys.exit(0)
  case Exit.Failure(cause) ⇒
    cause match {
      case Fail(value) ⇒
        value match {
          case ApplicationError.CommandlineArgumentsError(help)                  ⇒ println(help)
          case ApplicationError.ConfigError(failures)                            ⇒ failures.prettyPrint()
          case ApplicationError.SparkSessionError(t)                             ⇒ t.printStackTrace()
          case ApplicationError.VaultInitializationError(t)                      ⇒ t.printStackTrace()
          case ApplicationError.VerticaConfigurationInitializationError(failures) ⇒ println(failures.mkString_("\n"))
          case ApplicationError.ExecutionError(t)                                ⇒ t.printStackTrace()
        }
      case Die(value)  ⇒ value.printStackTrace()
      case _           ⇒ ()
    }
    sys.exit(1)
}
```

Запуск эффекта и обработка ошибок

```
zio.Runtime.default.unsafeRunSync(eff) match {
  case Exit.Success(value) ⇒ sys.exit(0)
  case Exit.Failure(cause) ⇒
    cause match {
      case Fail(value) ⇒
        value match {
          case ApplicationError.CommandlineArgumentsError(help)                ⇒ println(help)
          case ApplicationError.ConfigError(failures)                          ⇒ failures.prettyPrint()
          case ApplicationError.SparkSessionError(t)                           ⇒ t.printStackTrace()
          case ApplicationError.VaultInitializationError(t)                    ⇒ t.printStackTrace()
          case ApplicationError.VerticaConfigurationInitializationError(failures) ⇒ println(failures.mkString_("\n"))
          case ApplicationError.ExecutionError(t)                              ⇒ t.printStackTrace()
        }
      case Die(value)  ⇒ value.printStackTrace()
      case _           ⇒ ()
    }
    sys.exit(1)
}
```

Джоба – это функция

```
def job(
          spark: SparkSession,
          verticaCfg: runtime.Vertica,
          partition: String,
          callsTable: String,
          chatsTable: String): zio.IO[errors.ApplicationError, Unit] = {}
```

Distage

1. Инициализация только необходимых компонентов
2. Валидация на compile-time
3. Контроль жизненного цикла
4. Экспрессивный DSL, сокращает boilerplate
5. Неинвазивный

```
def spark = new ModuleDef {
  make[SparkSession].fromResource(
    Lifecycle.fromZIO(
      ZManaged
        .make(Task(SparkSession.builder.appName("Simple
Application").getOrCreate()))(ss ⇒ UIO(ss.stop()))
        .mapError(errors.ApplicationError.ExecutionError)
    )
  )
}
```

Distage

```
def spark = new ModuleDef {
  make[SparkSession].fromResource(
    Lifecycle.fromZIO(
      ZManaged
        .make(Task(SparkSession.builder.appName("Simple
Application").getOrCreate()))(ss ⇒ UIO(ss.stop()))
        .mapError(errors.ApplicationError.ExecutionError)
    )
  )
}


def appConfig = new ConfigModuleDef {
  makeConfig[config.AppConfig]("app")
}


def vault = new ModuleDef {
  make[Vault].fromEffect((cfg: config.AppConfig) ⇒
    Task(
      new Vault(
        new VaultConfig()
          .address(cfg.vault.address)
          .token(cfg.vault.token)
          .build()
      )
    ).mapError(errors.ApplicationError.VaultInitializationError)
  )
}
```

```
def runtimeConfig = new ModuleDef {
  make[runtime.Vertica].fromEffect((v: Vault) ⇒

Task(v.logical().read("etl/dbs/vertica").getData.asScala).flat
Map(cfgMap ⇒
      Task
        .fromEither(
          (
            Validated.fromOption(cfgMap.get("host"), "host is
not set"),
            Validated.fromOption(cfgMap.get("user"), "user is
not set"),
            Validated.fromOption(cfgMap.get("db"), "db is not
set"),
            Validated.fromOption(cfgMap.get("password"),
"password is not set"),
            Validated.fromOption(cfgMap.get("stagingFsUrl"),
"stagingFsUrl is not set"),

).tupled.toValidatedNel.toEither.leftMap(errors.ApplicationErr
or.VerticaConfigurationInitializationError)
        )
        .map(runtime.Vertica.tupled)
    )
  )
}


def job = new ModuleDef {
  make[SparkJob]
}
```

Рисуем Сову

```
object DistageZioSparkJob extends zio.App {

  final def run(args: List[String]) =
    myAppLogic(args).exitCode

  def myAppLogic(args: List[String]) =
    (zio.IO.fromEither(
      arguments.command.parse(args, sys.env).leftMap(errors.ApplicationError.CommandlineArgumentsError)
    ) >>= (Injector().produceGet[SparkJob](modules).unsafeGet().run _).tupled).tapError(
      {
        case ApplicationError.CommandlineArgumentsError(help)                  => putStr(help.toString())
        case ApplicationError.ConfigError(failures)                            => putStr(failures.prettyPrint())
        case ApplicationError.SparkSessionError(t)                             => putStr(t.toString)
        case ApplicationError.VaultInitializationError(t)                      => putStr(t.toString)
        case ApplicationError.VerticaConfigurationInitializationError(failures) => putStr(failures.mkString_("\n")
        case ApplicationError.ExecutionError(t)                                => putStr(t.toString)
      }
    )

}
```

Выводы

- В данном случае функциональный код больше
- Декларативный стиль позволяет описывать “ожидания”, а не реализацию
- С помощью функциональных паттернов, мы решили real world проблемы: обработка ошибок, высвобождение ресурсов, контроль запуска и внедрение зависимостей
- Напугали коллег

Ссылка на код

https://github.com/zuynew/smartdata-fp-spark