Использование **Fetch**

Яблочкин Денис

NAUMEN

Решаем истинные задачи

Ссылки

- **Библиотека:**
  - **github.com/47degrees/fetch**
- **Примеры из презентации:**
  - **github.com/DenisNovac/Scala-Fetch-presentation**

Функционал

- *Fetch* **— это библиотека для упрощения и оптимизации доступа к данным из различных источников:**
  - Файловые системы;
  - Базы данных;
  - Веб-сервисы.

Функционал

- **Fetch требует абстрагирования над источником данных. После описания источников Fetch может:**
  - Запрашивать данные из нескольких источников одновременно;
  - Объединять несколько запросов к одному источнику в один (batching);
  - Дедуплицировать запросы;
  - Кэшировать запросы.

Структура

- **Для реализации доступа к источнику требуется реализовать для него:**
  - Описание источника Data[I, A];
  - Описание эффективных методов DataSource[F[_], I, A].

Термины: Data

```scala
trait Data[I, A] { self =>
  def name: String

  def identity: Data.Identity =
    H.fromUniversalHashCode.hash(self)
}

object Data {
  type Identity = Int
}
```

Термины: DataSource

```scala
trait DataSource[F[_], I, A] {
  def data: Data[I, A]

  implicit def CF: Concurrent[F]

  /** Fetch one identity, returning a None if it wasn't found.
   */
  def fetch(id: I): F[Option[A]]

  /** Fetch many identities, returning a mapping from identities to results. If an
   * identity wasn't found, it won't appear in the keys.
   */
  def batch(ids: NonEmptyList[I]): F[Map[I, A]] =
    FetchExecution
      .parallel(
        ids.map(id => fetch(id).tupleLeft(id))
      )
      .map(_.collect { case (id, Some(x)) => id -> x }.toMap)

  def maxBatchSize: Option[Int] = None

  def batchExecution: BatchExecution = InParallel
}

sealed trait BatchExecution extends Product with Serializable
case object Sequentially    extends BatchExecution
case object InParallel      extends BatchExecution
```

Реализация для листа

```
class ListSource(list: List[String])(implicit cf: ContextShift[IO]) extends Data[Int, String] with LazyLogging {
  override def name: String        = "My List of Data"
  private def instance: ListSource = this

  def source = new DataSource[IO, Int, String] {
    override def data: Data[Int, String] = instance

    override def CF: Concurrent[IO] = Concurrent[IO]

    override def fetch(id: Int): IO[Option[String]] =
      CF.delay {
        logger.info(s"Processing element from index $id")
        list.lift(id)
      }
  }

  def fetchElem(id: Int) = Fetch.optional(id, source)
}
```

Создание запроса к источнику

```
val list                                    = List("a", "b", "c")
val data: ListSource                        = new ListSource(list)
val source: DataSource[IO, Int, String] = data.source
val f: Fetch[IO, String]                    = Fetch(1, source)
val run: IO[String]                         = Fetch.run(f)
```

Создание запроса к источнику

```
object Example extends App {

  implicit val ec: ExecutionContext = global
  implicit val cs: ContextShift[IO] = IO.contextShift(ec)  // для Fetch.run и ListDataSource
  implicit val timer: Timer[IO]     = IO.timer(ec) // для Fetch.run

  val list = List("a", "b", "c")
  val data   = new ListSource(list)
  val source = data.source

  Fetch.run(Fetch(0, source)).unsafeRunSync
  // INFO ListDataSource - Processing element from index 0

  Fetch.run(Fetch(1, source)).unsafeRunSync
  // INFO ListDataSource - Processing element from index 1

  Fetch.run(Fetch(2, source)).unsafeRunSync
  // INFO ListDataSource - Processing element from index 2

  Fetch.run(Fetch(3, source)).unsafeRunSync
  // INFO ListDataSource - Processing element from index 3
  // Exception in thread "main" fetch.package$MissingIdentity
}
```

Возврат опционального значения

```
val f0: Fetch[IO, Option[String]] = Fetch.optional( id= 1, source)
val run0: IO[Option[String]]      = Fetch.run(f0)
```

Метод fetchElem

- **То же, но с использованием метода fetchElem, создающего экземпляры Fetch:**

```
object SimpleExample extends App with ContextEntities {

  val list = List("a", "b", "c")
  val data = new ListSource(list)

  Fetch.run(data.fetchElem( id = 0)).unsafeRunSync          // INFO app.ListDataSource - Processing element from index 0
  Fetch.run(data.fetchElem( id = 1)).unsafeRunSync          // INFO app.ListDataSource - Processing element from index 1
  Fetch.run(data.fetchElem( id = 2)).unsafeRunSync          // INFO app.ListDataSource - Processing element from index 2
  println(Fetch.run(data.fetchElem( id = 2)).unsafeRunSync) // Some(c)
  println(Fetch.run(data.fetchElem( id = 3)).unsafeRunSync) // None

}
```

Кэширование

- **Fetch не кэширует «из коробки»:**

```
def fetch(id: Int): Option[String] = {
  val run   = Fetch.run(data.fetchElem(id))
  run.unsafeRunSync
}

fetch(1)
fetch(1)
fetch(1)

// INFO app.ListDataSource - Processing element from index 1
// INFO app.ListDataSource - Processing element from index 1
// INFO app.ListDataSource - Processing element from index 1
```

Кэширование

- **Трейт** *DataCache[F[_]]*
- **Встроенная имплементация** *InMemoryCache[F[_]: Monad]*
- **Можно задать начальное значение или создавать пустой:**

```
val cacheF: DataCache[IO] = InMemoryCache.from((data, 1) -> "b", (data, 2) -> "c")
val cache: DataCache[IO] = InMemoryCache.empty
```

InMemoryCache

```
val cacheF: DataCache[IO] = InMemoryCache.from((data, 1) -> "b", (data, 2) -> "c")

Fetch.run(data.fetchElem(1), cacheF).unsafeRunSync
Fetch.run(data.fetchElem(1), cacheF).unsafeRunSync
Fetch.run(data.fetchElem(1), cacheF).unsafeRunSync
Fetch.run(data.fetchElem(1), cacheF).unsafeRunSync
Fetch.run(data.fetchElem(0), cacheF).unsafeRunSync
Fetch.run(data.fetchElem(0), cacheF).unsafeRunSync

// INFO app.ListDataSource - Processing element from index 0
// INFO app.ListDataSource - Processing element from index 0
```

InMemoryCache

```scala
case class InMemoryCache[F[_]: Monad](state: Map[(Data[Any, Any], DataSourceId), DataSourceResult])
    extends DataCache[F] {
  def lookup[I, A](i: I, data: Data[I, A]): F[Option[A]] =
    Applicative[F].pure(
      state
        .get((data.asInstanceOf[Data[Any, Any]], new DataSourceId(i)))
        .map(_.result.asInstanceOf[A])
    )

  def insert[I, A](i: I, v: A, data: Data[I, A]): F[DataCache[F]] =
    Applicative[F].pure(
      copy(state =
        state.updated(
          (data.asInstanceOf[Data[Any, Any]], new DataSourceId(i)),
          new DataSourceResult(v)
        )
      )
    )
}
```

InMemoryCache

```scala
/** Пустой новый кэш */
var cache: DataCache[IO] = InMemoryCache.empty

def cachedRun(id: Int): Option[String] = {
  val (c, r) = Fetch.runCache(data.fetchElem(id), cache).unsafeRunSync
  cache = c // Пример ручного управления кэшем
  r
}

cachedRun( id = 1)
cachedRun( id = 1)
cachedRun( id = 2)
cachedRun( id = 2)
cachedRun( id = 4)
cachedRun( id = 4)

/**
Processing element from index 1
Processing element from index 2
Processing element from index 4
Processing element from index 4
 */
```

Имплементация произвольного кэша

```scala
/**
  * Обёртка над API Play для кэша, позволяющая использовать его в Fetch
  * @param asyncAkkaCache - API кэша Play
  * @param expiration - Длительность хранения
  * @param ec - Для Future
  * @param cs - Для IO
  */
case class CaffeineAkkaCache(asyncAkkaCache: AsyncCacheApi, expiration: FiniteDuration)(
    implicit val ec: ExecutionContext,
    implicit val cs: ContextShift[IO]
) extends DataCache[IO] with LazyLogging {

  override def lookup[I, A](i: I, data: Data[I, A]): IO[Option[A]] = {
    logger.debug(s"Searching in cache $i")
    val l: Future[Option[A]] = asyncAkkaCache.get(i.toString)
    IO.fromFuture(IO(l))
  }

  override def insert[I, A](i: I, v: A, data: Data[I, A]): IO[DataCache[IO]] = {
    logger.debug(s"Inserting to cache $i")
    val f: Future[Done] = asyncAkkaCache.set(i.toString, v, expiration) // Результат от апи Play вернуть не получится
    this.pure[IO]
  }
}
```

Объединение запросов

- **Fetch умеет объединять запросы к одному источнику в один запрос. Для этого их нужно связать аппликативным оператором.**

```
import fetch.fetchM  // инстансы Fetch для синтаксиса Cats

val tuple: Fetch[IO, (Option[String], Option[String])] = (data.fetchElem(0), data.fetchElem(1)).tupled

Fetch.run(tuple).unsafeRunSync()  // (Some(a),Some(b))
```

Объединение запросов

- **По умолчанию метод batch использует имплементацию метода fetch в параллели, но его можно переопределить под свои нужды:**

```
override def batch(ids: NonEmptyList[Int]): IO[Map[Int, String]] = {
  logger.info(s"Ids fetching: $ids")
  super.batch(ids)
}
```

Объединение запросов

- **Batch поддерживает специальные указания:**

```
override def maxBatchSize: Option[Int] = 2.some  // defaults to None
override def batchExecution: BatchExecution = Sequentially // defaults to `InParallel`
```

```
import fetch.fetchM

def findMany: Fetch[IO, List[Option[String]]] =
  List(0, 1, 2, 3, 4, 5).traverse(data.fetchElem)

Fetch.run(findMany).unsafeRunSync

// INFO app.ListSource - IDs fetching in batch: NonEmptyList(0, 5)
// INFO app.ListSource - IDs fetching in batch: NonEmptyList(1, 2)
// INFO app.ListSource - IDs fetching in batch: NonEmptyList(3, 4)
```

Комбинирование данных из разных источников

- **Внешне выглядит и работает как объединение запросов, но методы batch/fetch будут вызываться для конкретных источников.**

```
val listSource = new ListSource(List("a", "b", "c"))
val randomSource = new RandomSource()

def fetchMulti: Fetch[IO, (Int, String)] =
  for {
    rnd <- Fetch(3, randomSource.source)  // Fetch[IO, Int]
    char <- Fetch(rnd, listSource.source)  // Fetch[IO, String]
  } yield (rnd, char)

println(Fetch.run(fetchMulti).unsafeRunSync)  // например, (0,a)
```

Комбинирование данных из разных источников

```scala
class RandomSource(implicit cf: ContextShift[IO]) extends Data[Int, Int] with LazyLogging {

  override def name: String           = "Random numbers generator"
  private def instance: RandomSource = this

  def source: DataSource[IO, Int, Int] = new DataSource[IO, Int, Int] {
    override def data: Data[Int, Int] = instance

    override def CF: Concurrent[IO] = Concurrent[IO]

    override def fetch(max: Int): IO[Option[Int]] =
      CF.delay {
        logger.info(s"Getting next random by max $max")
        scala.util.Random.nextInt(max).some
      }
  }
}
```

Комбинаторы

- **Суть комбинирования запросов к одному или разным источникам в том, чтобы сделать из типа** Seq[Fetch[_]] **тип** Fetch[Seq[_]]**. Для этого подходят как** for **из стандартной библиотеки Scala, так и некоторые методы из Cats.**

Комбинаторы

```
def fetchRandomInt(max: Int) = Fetch(max, randomSource.source)

val listFetch: List[Fetch[IO, Int]] = List(
  Fetch(10, randomSource.source),
  Fetch(10, randomSource.source),
  Fetch(10, randomSource.source),
  Fetch(10, randomSource.source),
  Fetch(10, randomSource.source)
)

val fetchTuple: Fetch[IO, (Option[String], Option[String])] =
  (data.fetchElem(0), data.fetchElem(1)).tupled

val fetchTrv: Fetch[IO, List[Int]]
  = List(10, 10, 10, 10, 10).traverse(fetchRandomInt)

val fetchSeq: Fetch[IO, List[Int]] = listFetch.sequence

println(Fetch.run(fetchSeq).unsafeRunSync)  // List(8, 8, 8, 8, 8)
```

Список источников

- https://github.com/47degrees/fetch/tree/master/examples
- https://47degrees.github.io/fetch/docs
- https://www.scala-exercises.org/fetch/usage
- https://github.com/DenisNovac/akka-play-integrations/tree/master/play-fetch-cache

Спасибо за внимание!

NAUMEN

Решаем истинные задачи