

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.

<http://simonmar.github.io/hib/papers/haxl-icfp14.pdf>

Структура

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

Термины: Data

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

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

object Data {
  type Identity = Int
}
```

Термины: DataSource

```
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 batchSize: 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
```

Библиотеки для кэширования и Fetch

- **Fetch не является библиотекой для кэширования сама по себе и поэтому предоставляет только механизмы для кэширования запросов, сделанных через Fetch.**
- **Она не включает в себя никакой собственной имплементации кэша (кроме самой простой карты InMemoryCache).**
- **Через реализацию интерфейса DataCache можно подключить к Fetch любую библиотеку для кэширования.**

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

```
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

```
/** Пустой новый кэш */  
var cache: DataCache[I0] = 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

```
/**
 * 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 batchSize: 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)
```

Дедупликация

```
val tupleD: Fetch[IO, (Option[String], Option[String])] = (data.fetchElem( id = 0), data.fetchElem( id = 0)).tupled
Fetch.run(tupleD).unsafeRunSync()
//INFO app.sources.ListSource - Processing element from index 0
//(Some(a),Some(a))

val l: Fetch[IO, (Option[String], Option[String])] = for {
  d1 <- data.fetchElem( id = 0)
  d2 <- data.fetchElem( id = 0)
} yield (d1, d2)
Fetch.run(l).unsafeRunSync()
//INFO app.sources.ListSource - Processing element from index 0
//(Some(a),Some(a))
```

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

```
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)
```

Обработка ошибок

- Из **Fetch** можно вернуть не только **Option**, но и **Either**. Для этого не нужно создавать специфичный объект, как это было с `Fetch.optional(...)`:

```
/**  
val i: String = Fetch.run(Fetch(5, data.source)).unsafeRunSync // Exception in thread "main" fetch.package$MissingIdentity  
*/  
// Более безопасный запуск завернёт исключение в Either  
val i1: Either[Throwable, String] = Fetch.run(Fetch(1, data.source)).attempt.unsafeRunSync // Right(b)  
val i2: Either[Throwable, String] = Fetch.run(Fetch(5, data.source)).attempt.unsafeRunSync // Left(fetch.package$MissingIdentity)
```

- Для обработки ошибок предусмотрены:
 - Трейт **FetchException**;
 - Специальное исключение **MissingIdentity**;
 - Общее исключение **UnhandledException** для всего остального.

Отладка

- **Fetch предоставляет метод `Fetch.runLog`, возвращающий трейс запуска;**
- **Fetch предоставляет метод `describe` для его обработки в отдельном пакете `fetch-debug`;**
- **Сырой трейс будет выглядеть примерно так:**

```
FetchLog(Queue(Round(List(Request(FetchOne(2,app.sources.ListSource@ea6147e),10767139,10767203))),  
Round(List(Request(FetchOne(10,app.sources.RandomSource@58b31054),10767211,10767213))), Round(List(Request(Batch(NonEmptyList(3,  
4),app.sources.ListSource@ea6147e),10767234,10767242))),  
Round(List(Request(FetchOne(150,app.sources.RandomSource@58b31054),10767252,10767252),  
Request(FetchOne(5,app.sources.ListSource@ea6147e),10767252,10767252))))))
```

Отладка

- **Метод `describe` подходит для обработки:**
 - Типов `Log`, полученных в результате запуска **`Fetch.runLog`**;
 - Типов `Throwable`, полученных в результате запуска **`Fetch.runLog`**.

```
// libraryDependencies += "com.47deg" %% "fetch-debug" % "1.3.0"
import fetch.debug.describe
val t: Either[Throwable, (Log, String)] = Fetch.runLog(Fetch(5, data.source)).attempt.unsafeRunSync
println(t.fold(describe, identity))
// [ERROR] Identity with id `5` for data source `My List of Data` not found, fetch interrupted after 1 rounds
//Fetch execution 0.00 seconds
//
// [Round 1] 0.00 seconds
// [Fetch one] From `My List of Data` with id 5 0.00 seconds
```

Отладка

```
val list = List("a", "b", "c", "d", "e", "f", "g", "h")
```

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

// нет среди раундов вообще
val cached = Fetch(1, listSource)

// только #1, больше не повторяется
val notCached = Fetch(2, listSource)

// #2
val random = Fetch(10, randomSource)

// #3
val batched: Fetch[IO, (String, String)] =
  (Fetch(3, listSource), Fetch(4, listSource)).tupled

// #4
val combined = (Fetch(5, listSource), Fetch(150, randomSource)).tupled

/** End of fetches */
val complicatedFetch: Fetch[IO, (String, Int)] =
  cached >> notCached >> notCached >> random >> batched >> combined

val result: IO[(Log, (String, Int))] = Fetch.runLog(complicatedFetch, cacheF)
val tuple: (Log, (String, Int)) = result.unsafeRunSync()
```

```
[Round 1] 0.06 seconds
  [Fetch one] From `My List of Data` with id 2
[Round 2] 0.00 seconds
  [Fetch one] From `Random numbers generator` with id 10
[Round 3] 0.01 seconds
  [Batch] From `My List of Data` with ids List(3, 4)
[Round 4] 0.00 seconds
  [Fetch one] From `Random numbers generator` with id 150
  [Fetch one] From `My List of Data` with id 5
```

Аналоги

- **ZQuery** - <https://github.com/ghostdogpr/caliban/tree/master/core/src/main/scala/zquery>
- **Clump** - <https://github.com/getclump/clump>
- **Haxl** - <https://github.com/facebook/haxl>

СПИСОК ИСТОЧНИКОВ

- <https://github.com/47degrees/fetch/tree/master/examples>
- <https://47degrees.github.io/fetch/docs>
- <https://www.scala-exercises.org/fetch/usage>
- <https://simonmar.github.io/bib/papers/haxl-icfp14.pdf>
- <https://github.com/DenisNovac/akka-play-integrations/tree/master/play-fetch-cache>

Спасибо за внимание!

NAUMEN

Решаем истинные задачи