Глава 14: Параллельные вычисления с помощью акторов

После нескольких статей о продвинутых возможностях системы типов Scala и как они могут быть использованы для повышения гибкости кода и стабильности за счёт объявления более жёстких ограничений на этапе компиляции, мы вернёмся к вопросу о параллельных вычислениях.

В прошлых статьях мы узнали как организовать асинхронные вычисления с помощью `Future`.

Этот способ очень хорошо подходит для решения многих проблем. Но есть и другие альтернативы. Модель акторов — это другой основополагающий подход к распараллеливанию программ. В этом подходе одновременно выполняющиеся процессы передают друг другу сообщения.

Идея акторов не нова. Наиболее выдающаяся реализация была сделана в языке Erlang. В стандартной поставке Scala также есть своя библиотека для реализации акторов, но она устарела и с версии 2.11 будет окончательно заменена на реализацию акторов из библиотеки Akka. Эта библиотека по-сути уже давно является стандартом.

В этой статье мы познакомимся с моделью акторов и узнаем основы применения акторов в библиотеке Akka. Это лишь поверхностное знакомство, в этом настоящая статья отличается от предыдущих, мы сосредоточимся на том, чтобы научить думать в терминах Akka, изучим достаточно материала для того чтобы Вы могли заинтересоваться этой замечательной библиотекой.

Проблема общего изменяемого состояния

На сегодняшний день доминирующим подходом к созданию параллельных программ является подход, основанный на общем изменяемом состоянии (shared mutable state) — большое число объектов с состоянием выполняются в своих потоках, состояние каждого объекта может быть изменено сразу в нескольких частях. Обычно код такого приложения насыщен семафорами, контролирующими чтение и запись, для того чтобы состояние изменялось атомарно, для избежания одновременного изменения состояния несколькими потоками. В то же время мы стараемся не окружать семафорами слишком большие куски кода, поскольку это приводит к существенному снижению скорости приложения.

Очень часто такой код с самого начала пишется без учёта возможности распараллеливания. Параллелизация происходит по необходимости. Когда мы начинаем с последовательного кода всё идёт гладко, но когда мы пытаемся распараллелить его с учётом всех потребностей вычислений в многопоточной среде, мы получаем код, крайне трудный для чтения, который очень сложно понять.

Корень проблемы в том, что такие низкоуровневые средства синхронизации вычислений как семафоры и потоки — очень сложны для понимания. А следовательно, в их использовании очень легко ошибиться. Если вы не понимаете, что происходит в вашей системе, не сомневайтесь рано или поздно в ней появятся трудно уловимые ошибки, связанные с взаимной блокировкой процессов, или просто что-то может пойти не так. Такие ошибки

могут начать проявляться после месяцев нормальной работы серверов, когда вы давно установили его на боевые машины.

Также с этими низкоуровневыми конструкциями очень сложно добиться приемлемой эффективности.

Модель акторов

Модель акторов может избавить нас от всех упомянутых выше проблем, позволяя нам писать высокоэффективный но в то же время очень ясный код. Она принуждает нас к тому, чтобы код был параллельным с самого начала. Ведь добавить распараллеливание позже на практике не представляется возможным.

Основная идея в том, что приложение построено из многих легковесных процессов, называемых акторами. Каждый актор отвечает за одну очень маленькую задачу, поэтому нам легко понять, что он делает. Более сложная логика возникает из взаимодействия нескольких акторов, мы решаем задачи с помощью одних акторов, в это время посылаем сообщения совокупности других.

Система акторов

Акторы — несамостоятельные существа, они не могут жить сами по себе. Каждый актор в Akka создаётся *системой акторов* (actor system). Кроме создания акторов и поиска, система акторов `ActorSystem` позволяет нам выполнять множество других операций, о которых мы пока умолчим.

Для того чтобы код примеров запускался сначала добавьте следующие зависимости в SBT-проект с компилятором Scala 2.10:

```
resolvers += "Typesafe Releases" at "http://repo.typesafe.com/typesafe/releases"

libraryDependencies += "com.typesafe.akka" %% "akka-actor" % "2.2.3"
```

Теперь давайте создадим систему акторов `ActorSystem`. Она будет выступать в роли среды обитания для наших акторов:

```
import akka.actor.ActorSystem

object Barista extends App {
  val system = ActorSystem("Barista")
  system.shutdown()
}
```

Мы создали новое значение типа `ActorSystem` и дали ей имя `"Barista"` (или "бармен"" по англ.). Мы возвращаемся к кофейному примеру, с которым мы уже встречались в статье про `Future`.

Кроме того мы добропорядочно завершили работу системы акторов, по окончанию работы приложения.

Определение акторов

Наше приложение может состоять из нескольких десятков или нескольких миллионов акторов, Akka с этим прекрасно справляется. Но постойте-постойте, несколько миллионов? Вас может удивить это невероятно огромное число. Важно понимать, что между акторами и потоками нет соответствия один к одному. Если бы это было так мы бы очень скоро исчерпали бы всю память. Поскольку акторы не блокируют потоки вычисления, один поток может выполнять несколько акторов, переключаясь между ними, в зависимости от того к какому актору приходят сообщения.

Для иллюстрации давайте создадим очень простой актор. Бармен может принимать заказы, но пока он ничего не будет делать, только выводить сообщения на печать:

```
sealed trait CoffeeRequest
case object CappuccinoRequest extends CoffeeRequest
case object EspressoRequest extends CoffeeRequest

import akka.actor.Actor

class Barista extends Actor {
  def receive = {
    case CappuccinoRequest => println("I have to prepare a cappuccino!")
    case EspressoRequest => println("Let's prepare an espresso.")
  }
}
```

Мы определили несколько типов сообщений, которые принимает наш актор. Обычно для передачи сообщений, которые содержат значения, между акторами используются `case`-классы. Если сообщение ничего не содержит мы будем использовать `case`-объекты, как в данном примере.

В любом случае очень важно, чтобы наши сообщения были бы неизменяемыми значениями, иначе могут произойти очень плохие вещи.

Давайте присмотримся к нашему классу `Barista` для описания бармена. Он наследует от трэйта `Actor`. Этот трэйт определяет метод `receive`, который возвращает значения типа `Receive`. Этот тип это просто синоним для `PartialFunction[Any, Unit]`.

Обработка сообщений

Так в чём же суть метода `receive`? Тип `PartialFunction[Any, Unit]` может показаться странным сразу по нескольким причинам.

По смыслу частично определённая функция, возвращаемая методом `receive` отвечает за обработку сообщений. Если к нам приходит сообщение из любой другой части приложения, будь то актор или нет, Akka вызовет метод `receive`, передав сообщение в качестве аргумента.

Выполнение побочных эффектов

Во время обработки сообщения актор может делать всё что угодно, кроме возвращения значения.

> Что за!?

Как видно из типа метода `receive` наша частично определённая функция выполняет побочные эффекты. Это может отпугнуть поборников функционального программирования, ведь мы привыкли отдавать предпочтение чистым функциям. И это справедливо и для параллельных программ. Но наше состояние заключено внутри акторов и каждое сообщение обрабатывается одно за другим независимо от других акторов. Поэтому у нас не возникает необходимости в синхронизации процессов или семафорах. Это здорово, когда для побочных эффектов выделено отдельное место и они находятся под контролем.

Отсутствие типизации

Но... эта частично определённая функция не только выполняет побочные эффекты, она ещё и не типизирована настолько насколько это возможно в Scala. Мы можем передавать сообщения любого типа. Зачем нам это нужно, если у нас под рукой есть такая мощная система типов?

Это связано с некоторыми очень важными архитектурными решениями в Akka. Благодаря этому мы можем перенаправлять сообщения другим акторам, управлять вычислительной нагрузкой или создавать прокси акторы, в тайне от отправителя сообщений и так далее.

Иногда это и вправду приводит к весьма неприятным ошибкам, которые компилятор не способен распознать. Для того чтобы сделать передачу сообщений более безопасной, в ущерб некоторым возможностям нашей системы, в Akka мы можем воспользоваться экспериментальным типом `Channel`.

Асинхронные и неблокирующие

Ранее говорилось о том, что в Akka наши акторы рано или поздно обработают отправленные им сообщения. Важно понимать, что отправление сообщения и приём происходят асинхронно, без блокирования потока вычислений. Отправитель не будет заблокирован во время обработки сообщения. Вместо этого он продолжит выполнять свою работу. Возможно отправитель хочет получить ответ от нашего актора, но вполне возможно, что ответ ему совсем не важен.

На самом деле, когда мы отправляем сообщение, оно доставляется в почтовый ящик актора, который представляет собой очередь. Добавление сообщения в почтовый ящик не блокирует вычисления, то есть отправитель не ждёт пока сообщение будет добавлено в очередь адресата.

Обработчик событий заметит появление нового сообщения, опять же асинхронно. Если актор не занимается обработкой другого сообщения, для него выделяется отдельный поток, доступный в контексте вычислений. Как только актор закончит обработку предыдущего сообщения, обработчик отправит ему следующее сообщение из почтового ящика.

Актор блокирует выделенный ему поток вычислений до тех пор, пока не закончится обработка сообщения. Хотя это не скажется на отправителе, это всё же означает, что

медленные операции могут затормозить всё приложение, поскольку для остальных акторов останется меньше потоков.

Поэтому необходимо как можно быстрее проводить обработку сообщений и по возможности стараться избегать вызовов блокирующих операций.

Конечно мы не можем избежать возникновения таких ситуаций. Большинство драйверов для баз данных по-прежнему блокируют потоки вычисления, и нам скорее всего захочется пользоваться базами данных и в нашем приложении основанном на акторах. Эта проблема имеет решение, но мы пока не коснёмся его. Оно выходит за рамки ознакомительного материала.

Создание акторов

Пока мы научились определять акторы, но как они создаются? Как мы создадим актор для нашего бармена? Для этого нам нужно создать новое значение для актора `Barista`. Возможно Вам захочется сделать это так:

```
val barista = new Barista // приведёт к исключению
```

Но это не сработает! Akka ответит нам исключением `ActorInitializationException`. Суть в том, что для успешной работы акторов, они должны управляться системой `ActorSystem` и её сервисами. Поэтому нам нужно создать новый актор через систему акторов:

```
import akka.actor.{ActorRef, Props}

val barista: ActorRef = system.actorOf(Props[Barista], "Barista")
```

Метод `actorOf`, определённый в `ActorSystem`, ожидает значение типа `Props`, с помощью которого можно проводить настройку создаваемых акторов, также в методе `actorOf` мы задаём имя для актора. Мы воспользовались самой простой формой создания значений типа `Props`, вызвав метод `apply` из объекта-компаньона с указанием типа-параметра. Akka создаст новое значение для актора, вызовом конструктора по умолчанию.

Обратите внимание на то, что возвращаемое из `actorOf` значение имеет тип `ActorRef` а не `Barista`. Акторы никогда не взаимодействуют с другими акторами напрямую. Предполагается, что у нас нет прямых ссылок на значения акторов. Вместо этого акторы и другие компоненты приложения передают сообщения с помощью ссылок на акторы.

Так тип `ActorRef` выступает в роли прокси для настоящего актора. Это удобно, поскольку `ActorRef` может быть сериализован. Так мы можем передавать ссылки на акторы другим удалённым акторам, которые могут находиться на другом компьютере. При этом у нас есть один и тот же синтаксис для передачи сообщений вне зависимости от того находится ли актор на той же JVM или на другом компьютере. Мы называем это свойство прозрачностью по расположению.

Заметьте, что ссылка `ActorRef` не параметризована по типу. Все `ActorRef` равнозначны. Так мы можем передавать любые сообщения любому актору. Это намеренное решение. Благодаря

этому мы можем изменять топологию системы акторов, не затрагивая те акторы, что отправляют сообщения.

Отправка сообщений

Теперь когда мы научились создавать акторы и получили ссылку, которая указывает на актор бармена, мы можем отправить ему сообщение. Это делается вызовом метода `!` на ссылке:

```
barista ! CappuccinoRequest
barista ! EspressoRequest
println("I ordered a cappuccino and an espresso")
```

Вызов `!` работает по принципу: отправь и забудь. Мы говорим актору, что мы хотим капучино, но мы не ждём ответа. Это наиболее распространённый сценарий взаимодействия с актором в Akka. Вызов метода приводит к тому, что Akka добавляет сообщение в очередь почтового ящика адресата. Как было сказано ранее этот вызов не блокирует поток вычисления и со временем актор-адресат обработает наше сообщение.

В силу асинхронности вызовов, результат этого примера непредсказуем. Он может выглядеть так:

```
I have to prepare a cappuccino!
I ordered a cappuccino and an espresso
Let's prepare an espresso.
```

не смотря на то, что мы сначала послали два запроса, а затем вывели текст на экран. Наш текст может оказаться между сообщениями от актора.

Отвечаем на сообщения

Иногда нам не достаточно просто сообщить другим о том, что делать. Нам важно уметь отправлять ответ именно тому кто послал нам сообщение. Конечно нам бы хотелось делать это асинхронно.

Как раз для этого в акторах предусмотрен метод `sender`. Он возвращает `ActorRef` отправителя последнего сообщения, то есть того сообщение от которого мы в данный момент обрабатываем.

Но как он узнает о том кто отправил сообщение? Ответ кроется во втором неявно передаваемом параметре метода `!`:

```
def !(message: Any)(implicit sender: ActorRef = Actor.noSender): Unit
```

Когда этот метод вызывается внутри актора его ссылка передаётся в качестве неявного параметра.

Давайте изменим нашего бармена. Теперь он будет мгновенно отправлять посетителю счёт перед тем как напечатать привычный ответ:

```
case class Bill(cents: Int)
case object ClosingTime

class Barista extends Actor {
  def receive = {
    case CappuccinoRequest =>
      sender ! Bill(250)
      println("I have to prepare a cappuccino!")
    case EspressoRequest =>
      sender ! Bill(200)
      println("Let's prepare an espresso.")
    case ClosingTime => context.system.shutdown()
  }
}
```

Между тем мы объявили новое сообщение `ClosingTime`. Бармен реагирует на него завершением работы системы акторов. Он может обратиться к ней как и любой другой актор через значение `ActorContext` (получен вызовом метода `context`).

Теперь давайте определим второй актор представляющий покупателя:

```
case object CaffeineWithdrawalWarning

class Customer(caffeineSource: ActorRef) extends Actor {
  def receive = {
    case CaffeineWithdrawalWarning => caffeineSource ! EspressoRequest
    case Bill(cents) => println(s"I have to pay $cents cents, or else!")
  }
}
```

Этот актор — настоящий кофейный наркоман, ему необходимо заказать новую порцию кофе. Мы передаём ему ссылку `ActorRef` в конструкторе. Ссылка указывает на источник кофеина. Актор не знает указывает ли она на бармена или на что-то ещё. Единственное что ему нужно знать — что по этой ссылке он может отправлять запросы `CoffeeRequest`.

Наконец, нам нужно создать эти два актора и отправить посетителю `CaffeineWithdrawalWarning`, чтобы запустить процесс:

```
val barista = system.actorOf(Props[Barista], "Barista")
val customer = system.actorOf(Props(classOf[Customer], barista), "Customer")
customer ! CaffeineWithdrawalWarning
barista ! ClosingTime
```

При создании актора для посетителя мы воспользовались другой формой вызова `Props`. Мы передаём тип актора и аргументы, которые принимает на вход конструктор актора. Так мы передаём ссылку на бармена в актор посетителя.

Отправление `CaffeineWithdrawalWarning` посетителю, принуждает его к отправлению `EspressoRequest` бармену, который в свою очередь среагирует отправкой счёта

посетителю. Вывод программы может выглядеть так:

```
Let's prepare an espresso.
I have to pay 200 cents, or else!
```

Сначала происходит обработка сообщения `EspressoRequest`, затем бармен отправляет сообщение отправителю сообщения, то есть актору посетителя. Но эта операция не блокирует вычисления, мы не ждём пока посетитель среагирует на сообщение. Актор бармена может сразу обработать `EspressoRequest` и вывести на консоль своё сообщение. Вскоре `Customer` начинает обработку сообщения `Bill` и выводит сообщение на консоль.

Задаём вопросы

Иногда нам нужно отправить сообщение и дождаться ответа, обычно это происходит когда некоторым сервисам приходится взаимодействовать с акторами, но сами они акторами не являются. Поскольку они живут за пределами мира акторов, они не могут обрабатывать сообщения.

Специально для таких случаев предусмотрен метод `?`. Это мост между параллельными вычислениями, основанными на акторах и на `Future`. Код пользователя будет выглядеть так:

```
import akka.pattern.ask
import akka.util.Timeout
import scala.concurrent.duration._

implicit val timeout = Timeout(2.second)
implicit val ec = system.dispatcher

val f: Future[Any] = barista2 ? CappuccinoRequest
f.onSuccess {
  case Bill(cents) => println(s"Will pay $cents cents for a cappuccino")
}
```

Для начала нам нужно импортировать поддержку специального синтаксиса из модуля `ask` и указать неявное ограничение по времени `timeout` для ожидания ответа при вызове `?`. Также `Future` нуждается в контексте вычисления `ExecutionContext`. В этом примере мы пользуемся контекстом из `ActorSystem`, этот тип наследует от `ExecutionContext`.

Как видно из примера результат не типизирован, значение имеет тип `Future[Any]`. Не удивительно, ведь мы получили сообщение от актора, и они также не типизированы.

Актор, у которого мы спрашиваем, просто ответит отправителю через метод `sender`. Вот почему, когда мы задаём вопрос к актору бармена, нам не нужно ничего менять.

Как только актор, которому задали вопрос, отвечает сообщением, значение типа `Promise` принадлежащее `Future`, будет завершено.

Получается, что говорить гораздо лучше чем спрашивать. Akka — не для вежливых людей, но иногда возникают ситуации, когда ответ нам очень нужен, и в Akka это предусмотрено.

Акторы с состоянием

Каждый актор может содержать внутреннее состояние, хотя это и не обязательно. Иногда большая часть состояния приложения состоит из информации передаваемой изменяемыми сообщениями, которые передаются от одного актора к другому.

Каждый актор в данный момент может обрабатывать лишь одно сообщение. При этом он может изменять внутреннее состояние. Это означает, что наш актор содержит некоторое изменяемое состояние, но поскольку каждое сообщение обрабатывается отдельно от других, целостность состояния не может быть нарушена. Мы свободны от проблем, которые присущи модели связанной с общим изменяемым состоянием.

Для примера, давайте введём состояние в наш актор для бармена. Мы будем считать число заказов:

```
class Barista extends Actor {
  var cappuccinoCount = 0
  var espressoCount = 0

  def receive = {
    case CappuccinoRequest =>
      sender ! Bill(250)
      cappuccinoCount += 1
      println(s"I have to prepare cappuccino #$cappuccinoCount")
    case EspressoRequest =>
      sender ! Bill(200)
      espressoCount += 1
      println(s"Let's prepare espresso #$espressoCount.")
    case ClosingTime => context.system.shutdown()
  }
}
```

У нас появилось две изменяемые переменные: `cappuccinoCount` и `espressoCount`. В этих заметках нам впервые встретились `var`-переменные! Несмотря на то, что мы стараемся избегать их в функциональном программировании, это единственный способ определения состояния для акторов. Поскольку каждое сообщение обрабатывается отдельно от остальных, пример выше эквивалентен использованию `AtomicInteger` в модели не связанной с акторами.

Заключение

На этом заканчивается наше введение в то как устроена модель акторов в Akka. Хотя мы затронули лишь малую часть материала, я надеюсь, что мне удалось дать достаточно информации для того, чтобы вы поняли основные понятия и смогли самостоятельно изучить остальное.

В следующей статье мы разовьём наш пример и рассмотрим несколько новых понятий из Akka, среди них вопрос обработки ошибок в системе акторов.

- <= Глава 13: Зависимые от пути типы
- => Глава 15: Обработка ошибок в системе акторов
- Содержание