



232,54

Рейтинг

Издательский дом «Питер»

Компания



ph_piter 4 сентября 2015 в 11:06

Акка, акторы и реактивное программирование

<https://blog.codecentric.de/en/2015/08/introduction-to-akka-actors/>

Программирование, Параллельное программирование, Java, Блог компании Издательский дом «Питер»

[Перевод](#)

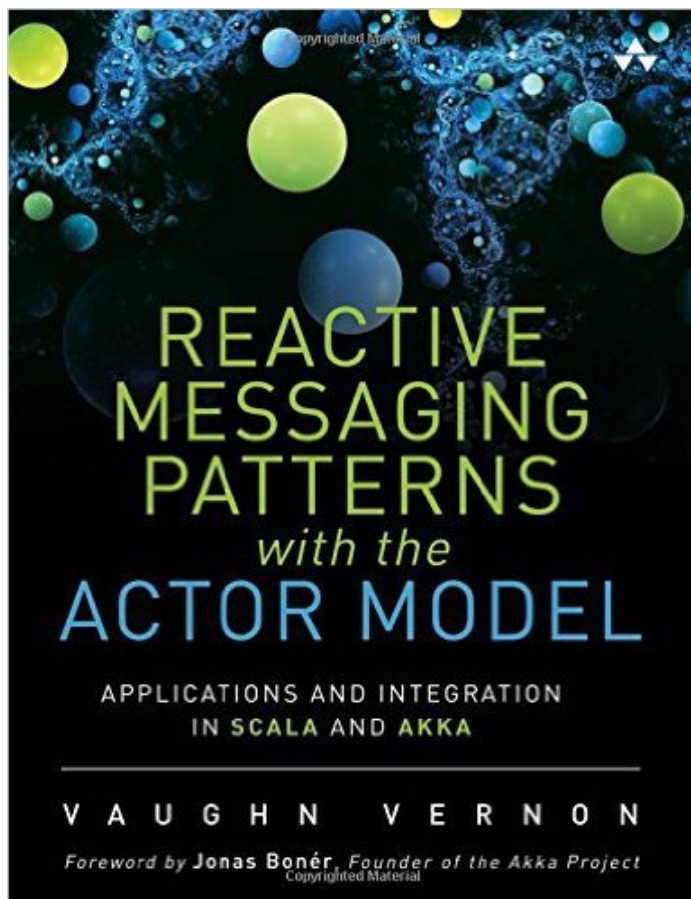
Здравствуйте, уважаемые читатели.

Сегодня мы хотели поговорить с вами на тему «все новое — это хорошо забытое старое» и вспомнить об акторах, описанных Карлом Хьюиттом еще в начале 70-х. А все дело в том, что недавно вышла вот такая [книга](#):

ИНФОРМАЦИЯ

Дата основания	05 сентября 1991
Локация	Санкт-Петербург Россия
Сайт	piter.com
Численность	201–500 человек
Дата регистрации	15 августа 2012

ВИДЖЕТ



Она довольно объемная — в переводе должна получиться более 500 страниц.

Несмотря на подчеркнутую элитарность книги (Akka и Scala), ее автор Вон Вернон (крупнейший специалист по DDD) уверен, что архитектурные паттерны, описанные в этой работе, вполне реализуемы на .NET и C#, о чем рассказывает в приложении. Мы же размещаем под катом перевод статьи, автор которой допускает перенос акторной парадигмы на язык Java. Поскольку рейтинг книги на Amazon стабильно высок, а тема универсальна, просим поделиться вашими мнениями как о ней, так и об акторной архитектуре в принципе.



Чтобы научиться хакингу нужно разбираться в программировании, машинной архитектуре, сетевых соединениях и знать конкретные приемы взлома.

БЛОГ НА ХАБРЕ

Вероятностное программирование и байесовский метод для хакеров

4,6k 1

Книга «Unity для разработчика. Мобильные мультиплатформенные игры»

В [первой статье](#) из этой серии был сделан общий обзор [Akka](#). Теперь мы как следует углубимся в сферу акторов Akka, вооружившись модулем akka-actor, который закладывает основы для всех остальных модулей Akka.

На наш взгляд, можно научиться программировать, даже не имея практики чтения/написания кода. Здесь мы пошагово разработаем маленькую акторную библиотеку: событийную шину PubSub, работающую по принципу «публикация-подписка». Разумеется, Akka поставляется с готовыми к работе локальным и глобальным решением такого рода, поэтому здесь мы просто повозимся с хорошо известным примером. Мы будем работать на языке [Scala](#), просто потому, что на нем гораздо удобнее писать Akka-образный код, но ровно таких же результатов можно достичь и на Java.

Акторная модель

В акторной модели — которая была изобретена в 1973 году Карлом Хьюиттом и др. — акторы представляют собой «фундаментальные единицы вычислений, реализующие обработку, хранение и коммуникацию». Хорошо, давайте разберемся по порядку.

Понятие «фундаментальная единица вычислений» означает, что когда мы пишем программу в соответствии с акторной моделью, наша работа по проектированию и реализации строится вокруг акторов. В [потрясающем интервью](#), данном Эрику Мейеру, Карл Хьюитт объясняет, что «акторы повсюду», а также что «одиночных акторов не бывает, они существуют в системах». Мы уже резюмировали эту мысль: при использовании акторной модели весь наш код будет состоять из акторов.

Как же выглядит актор? Что такое, наконец, «обработка», «хранение» и «коммуникация»? В сущности, коммуникация — это **асинхронный обмен сообщениями**, хранение означает, что акторы могут иметь **состояние**, а обработка заключается в том, что акторы могут иметь дело с сообщениями. Обработка также именуется **«поведением»**. Не слишком сложно звучит, правда? Итак, давайте сделаем следующий шаг и рассмотрим акторы Akka.

Устройство актора Akka

3,3k

7

Книга «Глубокое обучение на Python»

8,6k

8

Google Cloud: новая платформа и возможности машинного обучения

4,1k

3

Книга «Программирование для детей. Учимся создавать игры на Scratch»

3,6k

5

Книга «Head First. Программирование для Android. 2-е изд»

3,7k

3

Советы по оптимизации кода на Java: как не наступать на грабли

10k

15

Книга «Теоретический минимум по Computer Science. Все что нужно программисту и разработчику»

18,7k

22

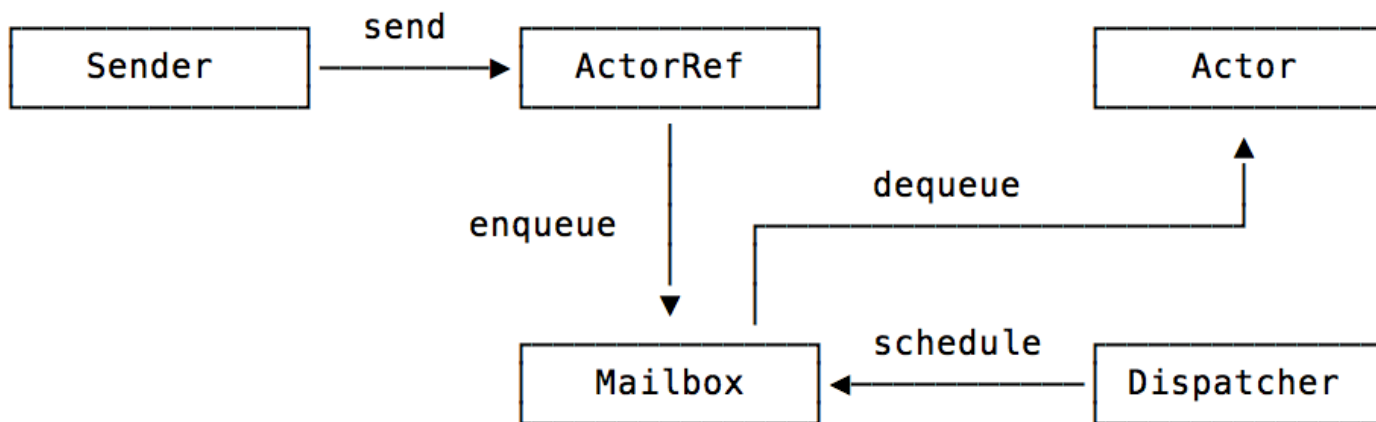
Промышленный IoT: изучение спроса

3k

1

Книга «Swift. Основы разработки приложений под iOS и macOS. 4-е изд. дополненное и переработанное»

Как понятно из следующей картинки, актор Akka состоит из нескольких взаимодействующих компонентов. *ActorRef* — это логический адрес актора, позволяющий асинхронно отправлять актору сообщения по принципу «послал и забыл». Диспетчер — в данном случае по умолчанию на каждую систему акторов приходится по одному диспетчеру — отвечает за постановку сообщений в очередь, ведущую в почтовый ящик актора, а также приказывает этому ящику изъять из очереди одно или несколько сообщений, но только по одному за раз — и передать их актору на обработку. Последнее, но немаловажное: актор — обычно это единственный API, который нам приходится реализовать — инкапсулирует состояние, и поведение.



Как будет показано ниже, Akka не позволяет получить непосредственный доступ к актору и поэтому гарантирует, что единственный способ взаимодействия с актором — это асинхронные сообщения. Невозможно вызвать метод в акторе.

Кроме того, необходимо отметить, что отправка сообщения актору и обработка этого сообщения актором — это две отдельные операции, которые, скорее всего, происходят в разных потоках. Разумеется, Akka обеспечивает необходимую синхронизацию, чтобы гарантировать, что любые изменения состояния будут видимы всем потокам.

Соответственно, Akka как бы разрешает нам запрограммировать иллюзию однопоточности, и мы можем не пользоваться в акторном коде никакими синхронизационными примитивами наподобие

volatile или *synchronized* — более того, не следует этого делать.

Реализация актора

Довольно слов, переходим к коду! В Akka актор — это класс, к которому подмешивается типаж `Actor`:

```
class MyActor extends Actor {  
  override def receive = ???  
}
```

Метод `receive` возвращает так называемое исходное поведение актора. Это просто частично вычисляемая функция, используемая Akka для обработки сообщений, отправляемых актору. Поскольку поведение равно `PartialFunction[Any, Unit]`, в настоящее время невозможно определять такие акторы, которые принимают сообщения лишь заданного типа. В Akka уже есть экспериментальный модуль *akka-typed*, обеспечивающий на этой платформе безопасность типов, но он еще дорабатывается. Кстати, поведение актора может изменяться, и именно поэтому в исходном поведении вызывается возвращаемое значение метода `receive`.

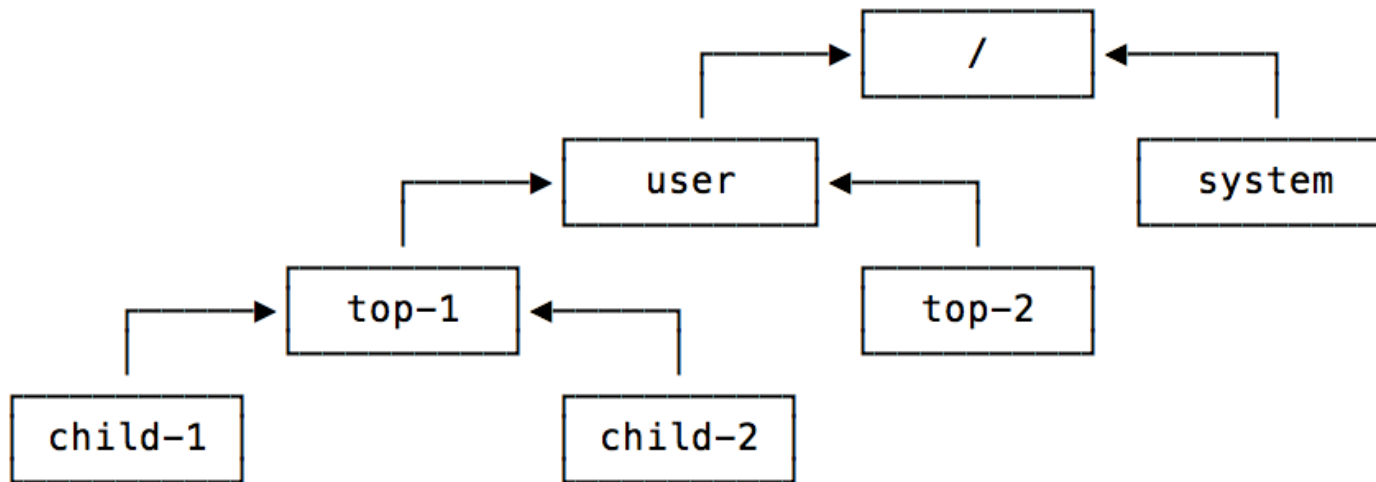
Хорошо, давайте реализуем базовый актор для нашей библиотеки PubSub:

```
class PubSubMediator extends Actor {  
  override def receive = Actor.emptyBehavior  
}
```

Пока нам не требуется, чтобы `PubSubMediator` обрабатывал какие-либо сообщения, поэтому мы используем обычную частично вычисляемую функцию `Actor.emptyBehavior`, для которой не определено какое-либо значение.

Акторные системы и создание акторов

Как было указано выше, «одиночных акторов не бывает, они существуют в системах». В Akka система акторов представляет собой взаимосвязанный ансамбль, члены которого организованы иерархически. Таким образом, у каждого актора есть свой родительский актор, как показано на следующей картинке.



При создании акторной системы, Akka — на внутреннем уровне использующая множество так называемых «системных акторов» — создает три актора: это «корневой страж» (root guardian), расположенный в корне акторной иерархии, а также системный и пользовательский стражи. Пользовательский страж — зачастую именуемый просто «страж» — является родительским элементом для всех создаваемых нами акторов верхнего уровня (в данном контексте имеется в виду «наивысший уровень, к которому мы имеем доступ»).

Допустим, но как создать систему акторов? Нужно просто вызвать фабричный метод, предоставляемый объектом-одиночкой *ActorSystem*:

```
val system = ActorSystem("pub-sub-mediator-spec-system")
```

А зачем мы вообще создаем *ActorSystem*? Почему бы просто не создавать акторы? Последнее невозможно, поскольку при непосредственном вызове конструктора актора система выбросит исключение. Вместо этого нам придется использовать фабричный метод, предоставляемый — вы угадали — *ActorSystem* для создания актора верхнего уровня:

```
system.actorOf(Props(new PubSubMediator), "pub-sub-mediator")
```

Разумеется, *actorOf* возвращает не экземпляр *Actor*, а *ActorRef*. Так Akka не позволяет нам получить доступ к экземпляру *Actor*, что, в свою очередь, гарантирует: обмен информацией с актором возможен только через асинхронные сообщения. Имя, указываемое нами, должно быть уникальным среди сиблингов данного актора, иначе будет выдано исключение. Если мы не укажем имени, Akka создаст его за нас, поскольку у каждого актора должно быть имя.

А что за такая штука *Props*? Это просто конфигурационный объект для актора. Он принимает конструктор как параметр, передаваемый по имени (то есть, лениво) и может содержать другую важную информацию — например, о маршрутизации или развертывании.

Когда заходит речь о дистанционной связи, важно учитывать, что *Props* можно сериализовать, поэтому уже сложилась практика добавлять *Props*-фабрику к сопутствующему объекту актора. Здесь также удобно ставить константу, соответствующую имени актора.

Зная все это, давайте допишем *PubSubMediator*, а также создадим для него тест при помощи [ScalaTest](#) и Akka Testkit — еще один модуль Akka, упрощающий тестирование акторов Akka:

```
object PubSubMediator {  
  
    final val Name = "pub-sub-mediator"
```

```

def props: Props = Props(new PubSubMediator)
}

class PubSubMediator extends Actor {
  override def receive = Actor.emptyBehavior
}

class PubSubMediatorSpec extends WordSpec with Matchers with BeforeAndAfterAll {

  implicit val system = ActorSystem("pub-sub-mediator-spec-system")

  "A PubSubMediator" should {
    "be suited for getting started" in {
      EventFilter.debug(occurrences = 1, pattern = s"started.*${classOf[PubSubMediator].getName}").intercept {
        system.actorOf(PubSubMediator.props)
      }
    }
  }

  override protected def afterAll() = {
    Await.ready(system.terminate(), Duration.Inf)
    super.afterAll()
  }
}

```

Как видите, мы создаем *ActorSystem* и актер *PubSubMediator* в *PubSubMediatorSpec*. Сам тест немного надуманный, поскольку наш *PubSubMediator* пока довольно сырой. В нем используется отладка жизненного цикла и ожидается логирование отладочного сообщения вида “started ... *PubSubMediator* ...”. Полный код его актуальной версии находится по адресу GitHub под меткой [step-01](#).

Коммуникация

Итак, научившись создавать акторы, давайте поговорим о коммуникации, которая — как было указано выше — основывается на асинхронных сообщениях и тесно связана с двумя другими свойствами актора: поведением (то есть, возможностью обрабатывать сообщения) и состоянием.

Чтобы отправить актору сообщение, нужно знать его адрес, то есть, *ActorRef*:

```
mediator ! GetSubscribers("topic")
```

Как видите, в *ActorRef* есть оператор *!* — так называемый “bang”, который отправляет заданное сообщение соответствующему актору. Как только сообщение доставлено, операция завершается, и код отправки продолжает работу. Подразумевается, что здесь нет возвращаемого значения (кроме *Unit*), следовательно, сообщения действительно уходят по принципу «послал и забыл».

Пусть это и просто, нам часто требуется отклик. Благодаря тому, что оператор *!* неявно принимает отправителя как *ActorRef*, сделать это можно без труда:

```
override def receive = {  
  case Subscribe(topic) =>  
    // ИМЕННО ТУТ обрабатывается подписка  
    sender() ! Subscribed  
}
```

В данном примере поведение актора-получателя обрабатывает конкретное сообщение — команду *Subscribe* — и передает сообщение — событие *Subscribed* — обратно отправителю. Затем метод *sender* используется для доступа к отправителю того сообщения, которое сейчас обрабатывается.

Учитывая все это, давайте дополнительно усовершенствуем *PubSubMediator* и соответствующий тест. Для начала добавим протокол сообщения — множество всех сообщений, относящихся к *PubSubMediator*

– к сопутствующему объекту:

```
object PubSubMediator {  
  
  case class Publish(topic: String, message: Any)  
  case class Published(publish: Publish)  
  
  case class Subscribe(topic: String, subscriber: ActorRef)  
  case class Subscribed(subscribe: Subscribe)  
  case class AlreadySubscribed(subscribe: Subscribe)  
  
  case class Unsubscribe(topic: String, subscriber: ActorRef)  
  case class Unsubscribed(unsubscribe: Unsubscribe)  
  case class NotSubscribed(unsubscribe: Unsubscribe)  
  
  case class GetSubscribers(topic: String)  
  
  final val Name = "pub-sub-mediator"  
  
  def props: Props = Props(new PubSubMediator)  
}
```

Далее давайте реализуем поведение, которое до сих пор оставалось незаполненным:

```
class PubSubMediator extends Actor {  
  import PubSubMediator._  
  
  private var subscribers = Map.empty[String, Set[ActorRef]].withDefaultValue(Set.empty)  
  
  override def receive = {  
    case publish @ Publish(topic, message) =>
```

```

    subscribers(topic).foreach(_ ! message)
    sender() ! Published(publish)

    case subscribe @ Subscribe(topic, subscriber) if subscribers(topic).contains(subscr
iber) =>
        sender() ! AlreadySubscribed(subscribe)

    case subscribe @ Subscribe(topic, subscriber) =>
        subscribers += topic -> (subscribers(topic) + subscriber)
        sender() ! Subscribed(subscribe)

    case unsubscribe @ Unsubscribe(topic, subscriber) if !subscribers(topic).contains(s
ubscriber) =>
        sender() ! NotSubscribed(unsubscribe)

    case unsubscribe @ Unsubscribe(topic, subscriber) =>
        subscribers -= topic -> (subscribers(topic) - subscriber)
        sender() ! Unsubscribed(unsubscribe)

    case GetSubscribers(topic) =>
        sender() ! subscribers(topic)
}
}

```

Как видите, поведение обрабатывает все команды – например, *Publish* или *Subscribe* – и всегда посылает утвердительный или отрицательный отклик отправителю. Тот факт, валидна ли команда и выдает ли она положительный результат – напр., *Subscribed* – зависит как от команды, так и от состояния, представляемого в виде приватного изменяемого поля `subscribers`.

Как было указано выше, одновременно обрабатывается всего одно сообщение, и Akka гарантирует, что изменения состояния останутся видимы и при обработке следующего сообщения, поэтому не потребуется вручную синхронизировать весь доступ к подписчикам. Конкурентность без проблем!

Наконец, давайте рассмотрим фрагмент расширенного теста:

```
val subscribe01 = Subscribe(topic01, subscriber01.ref)
mediator ! subscribe01
sender.expectMsg(Subscribed(subscribe01))

val subscribe02 = Subscribe(topic01, subscriber02.ref)
mediator ! subscribe02
sender.expectMsg(Subscribed(subscribe02))

val subscribe03 = Subscribe(topic02, subscriber03.ref)
mediator ! subscribe03
sender.expectMsg(Subscribed(subscribe03))
```

Как видите, мы отправляем сообщения `Subscribe` посреднику при помощи оператора `!` и ожидаем получить соответствующие отклики. Как и выше, весь код проекта по состоянию на текущий момент находится по адресу GitHub под меткой [step-02](#).

Жизненный цикл

До сих пор мы пренебрегали одним важным аспектом акторов: их существование конечно — то есть, они завершаются, и завершить актор можно в любой момент.

Имея доступ к *ActorRef*, мы не знаем, «жив» ли соответствующий актор. В частности, мы не получим исключения, если будем посылать сообщения к завершенному актору. В таком случае *ActorRef* остается валиден, но Akka выполняет переадресацию, и для сообщений, направляемых в мертвые почтовые ящики, действует негарантированная доставка. Таким образом, эти сообщения логируются, что полезно при тестировании, но этот способ отнюдь не подходит, чтобы реализовать нечто вроде повторной передачи или даже гарантированной доставки.

Но иногда нам действительно требуется знать, «жив» ли актор до сих пор, либо нет. В описываемом

случае нам нужна возможность избавиться от завершенных подписчиков, так как в противном случае *PubSubMediator* отправляет ненужные сообщения и даже может рано или поздно израсходовать всю память.

По всем этим причинам Akka предоставляет возможность отслеживать жизненный цикл акторов. Поскольку мы можем наблюдать только завершение актора, этот механизм называется «**мертвая вахта**» (death watch). Для отслеживания актора мы просто вызываем метод `watch`, предоставляемый *ActorContext*, доступный в Actor посредством *context*:

```
context.watch(subscriber)
```

Затем Akka отправит наблюдающему актору сообщение `Terminated` после того, как наблюдающий актор завершится. Это сообщение гарантированно будет последним, полученным от актора, даже при удаленной связи.

Хорошо, доделываем *PubSubMediator*:

```
class PubSubMediator extends Actor {  
  import PubSubMediator._  
  
  ...  
  
  override def receive = {  
    ...  
  
    case subscribe @ Subscribe(topic, subscriber) =>  
      subscribers += topic -> (subscribers(topic) + subscriber)  
      context.watch(subscriber)  
      sender() ! Subscribed(subscribe)  
  
    ...  
  }
```

```
case Terminated(subscriber) =>
  subscribers = subscribers.map { case (topic, ss) => topic -> (ss - subscriber) }
}
```

Как видите, мы отслеживаем всех подписчиков, обрабатывая валидную команду *Subscribe* и удаляя каждый завершенный подписчик при работе с соответствующим ему сообщением *Terminated*. Опять же, полный актуальный код этого примера находится на GitHub под меткой [step-03](#).

Заключение

На этом заканчивается предварительное знакомство с акторами Akka. Итак, мы рассмотрели самые важные аспекты акторной модели — коммуникацию, поведение и состояние, а также поговорили о системах акторов. Также мы обсудили реализацию этих концепций при помощи Akka и поговорили о мертвой вахте.

Разумеется, пришлось опустить массу интересного и важного материала: создание дочерних акторов, слежение (supervision) и т.д. Отсылаем вас к интересным дополнительным ресурсам, например, к отличной [документации Akka](#).

Только зарегистрированные пользователи могут участвовать в опросе. [Войдите](#), пожалуйста.

Мнение о книге

☐ Перспективная серьезная тема, на такую книгу денег не жалко

☐ Актеры не прижились за сорок лет, не приживутся и далее, не издавайте

☐ Хотелось бы более универсальную книгу по Scala

Проголосовали 162 пользователя. Воздержались 53 пользователя.

Метки: akka, актеры, scala, java, enterprise, архитектура, программирование, книги

↑ +12 ↓ 161 38,6k 23



Издательский дом «Питер» 232,54

Компания



99,0

Карма

338,9

Рейтинг

262

Подписчики

@ph_piter

Пользователь

Сайт

Поделиться публикацией



ПОХОЖИЕ ПУБЛИКАЦИИ

23 марта 2016 в 16:01

Увлекательное программирование: изучаем Minecraft

↑ +14 👁 35,1k 📖 127 💬 17

18 сентября 2015 в 17:10

Rust, дисциплинирующий язык программирования

↑ +35 👁 37k 📖 115 💬 47

20 декабря 2013 в 14:02

Программирование под Android. Для профессионалов

↑ +3 👁 25,4k 📖 61 💬 25

Комментарии 23



lair 04.09.15 в 11:15 🗑 📖

↑ +2 ↓

Жалко, книги выходят иногда не в том порядке, в котором было бы осмысленно.

У мэннинга скоро выйдет более общая [Reactive Design Patterns](#) (и в этой серии еще как минимум две книги), мне кажется, ее осмысленнее читать первой.



ph_piter 04.09.15 в 11:18 🗑 📖 🔄 ⬆

↑ 0 ↓

Спасибо, рассмотрим




HotWaterMusic 04.09.15 в 12:39 🗑 📖 🔄 ⬆

↑ 0 ↓

Для тех, кто как и я заинтересовался комментарием: если запросить бесплатный отрывок с www.typesafe.com, то на почту прилетит купон на скидку на саму книгу в 45% для покупки на manning.com.



 lair 04.09.15 в 12:41 # 📖 ⬆


↑ +2 ▼

Печаль в том, что эта книга, на самом деле, еще не вышла, а читать ее MEAP оказалось не очень удобно.

 grossws 04.09.15 в 14:32 # 📖 ⬆

↑ +2 ↓

В рамках курса reactive programming на coursera на неё давали скидку 50%.

 solver 04.09.15 в 12:54 # 📖


↑ 0 ↓

Однозначно нужна книга.
Очень мало структурированной информации на эту тему.
Когда перевод планируется к выходу?

 ph_piter 04.09.15 в 12:58 # 📖 ⬆

↑ 0 ↓

Нам хотя бы права получить). Обязательно сориентируем по срокам выхода, если решим издавать.

 wheercool 04.09.15 в 13:20 # 📖

↑ 0 ↓

Из названия статьи

| Akka, акторы и реактивное программирование

ожидал увидеть что-нибудь о реактивном программировании. Может быть я не внимательно читал. Можете пояснить как оно реализуется на акторах?

 lair 04.09.15 в 13:22 # 📖 ⬆

↑ 0 ↓

А что вас смущает? Акторы (в той терминологии, на которой построена книга) — изначально событийно-ориентированы, событие на входе, события на выходе. По-моему, это квинтэссенция реактивного программирования.

 wheercool 04.09.15 в 13:26 # 📖 ⬆

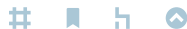
↑ 0 ↓

Книгу я не читал, поэтому и задал вопрос.

Событийная архитектура еще не означает реактивность, в том числе как реактивность не обязательно должна быть реализована при помощи событий.



lair 04.09.15 в 14:07



↑ +1 ↓

В (еще не вышедшей) [книге](#), на которую я ссылаюсь выше, есть набор определений, согласно которым модель акторов в акке — реактивна.

А какими определениями реактивности пользуетесь вы?



wheercool 04.09.15 в 15:11



↑ +1 ↓

Для меня реактивное программирование это программирование построенное на инвариантах (если конечно правомерно так выразиться).

Переменные образуют граф из этих инвариантов (зависимостей). Любое изменение значения переменной из этого графа вызывает его пересчет. Таким образом значение переменных всегда находится в консистентном (не противоречивом по отношению к инвариантам) состоянии.



lair 04.09.15 в 15:14



↑ 0 ↓

Это понимание не имеет никакого отношения ни к акторам, ни к той концепции реактивного программирования, которая использована в акке (или других известных мне современных реактивных моделях), поэтому я никак не могу прокомментировать его применимость к чему-либо.



wheercool 04.09.15 в 15:33



↑ 0 ↓

Так мой первоначальный вопрос как раз и был о том как реактивное программирование реализовано в акторах :)

Я если честно, вообще не знаком с понятием «реактивная модель».

Как мне кажется, я как раз и описал общую концепцию реактивного программирования своими словами.

На практике же существует несколько подходов в ее реализации: FRP например.

Если не сложно опишите что такое «реактивная модель» и с какими вы знакомы

 lair 04.09.15 в 15:41 # 📌 ↻ ⬆

↑ 0 ↓

Правильно спрашивать не «как реактивное программирование реализовано в акторах», а «что в акторах понимается под реактивным программированием».

А дальше просто идем и читаем [Reactive Manifesto](#). Мой вольный сильно сжатый перевод:

Реактивные системы:

- Отзывчивы, что означает, что система, если ответ вообще возможен, отвечает за разумное время
- Легко восстанавливаются: система сохраняет отзывчивость даже после отказа.
- Эластичны: система сохраняет отзывчивость при изменении нагрузки
- Основаны на сообщениях


 wheercool 04.09.15 в 15:50 # 📌 ↻ ⬆

↑ 0 ↓

Видимо мы о разном говорим)

Я так понимаю реактивные системы, к-ые вы упоминаете здесь — это «быстрореагирующие» системы.

Я же имел в виду вот это [Реактивное программирование](#)

 lair 04.09.15 в 16:01 # 📌 ↻ ⬆

↑ 0 ↓

Не я упоминаю, а авторы манифеста (аналогично с ними — авторы Акки, вышеозначенных книг, курса по FRP и так далее... зачастую это, правда, одни и те же люди).

 solver 04.09.15 в 22:56 # 📌 ↻ ⬆

↑ +3 ↓

У вас очень упрощенный взгляд на этот вопрос.

То реактивное программирование, которое упоминаете вы, если так можно

выразится «входит в состав» того реактивного программирования, которое сейчас принято понимать. И сокращенный манифест которого привели выше.

Фишка в том, что Akka это не табличный процессор. Это надо просто понимать. Akka это фреймворк общего назначения, на котором в том числе можно построить и табличный процессор. Это кстати одно из задач на курсах Одерского по Scala, Akka и реактивному программированию). В общем случае, программы которые строят на Akka гораздо сложнее чем просто пересчет формул в экселе. Поэтому сравнивать их очень странно. И сам вопрос «как реактивное программирование реализовано в акторах» звучит очень странно. Это как спросить «как реактивное программирование реализовано в C# или на Python» или на любом другом языке или фреймворке. Ибо, учитывая что это объекты общего назначения, ответ будет простым — так как вы сами реализуете.

Сама же реактивность в акторных системах реализуется просто, вся система реагирует на входящие сообщения и реагируя на них изменяет свое состояние на выходе. Без промежуточного сохранения. Сделано это через передачу сообщений, что дает много профита, особенно для многопоточной разработки. Сам актор — «формула», которая делает одно вычисление. Без состояния. Таким образом запуская много таких формул получаем тот самый поток данных который проходя через множество акторов на выходе дает результат.

Но надо понимать, что конкретное применение зависит от задач проекта. Сам фреймворк позволяет множество решений.



meIn1k 22.09.15 в 08:48



Сам актор — «формула», которая делает одно вычисление. Без состояния.

Это не самый подходящий способ использования акторов. Акторы больше предназначены для безопасного хранения состояния в многопоточной среде, а если состояние отсутствует, то фьюча часто бывает проще и удобнее.

 **solver** 22.09.15 в 10:36 # 📌 🔄 ⬆

Ну вообще-то это всего лишь один из вариантов, в контексте конкретного примера.

Никто и не говорил, что это самый лучший вариант использования.

 **Astashov_Anton** 04.09.15 в 19:43 # 📌 🔄 ⬆

Плюсую, во фронт-енде довольно много хороших примеров использования FRP (на сайте Elm, например), было бы круто найти хорошую книгу как это можно это еще применять на бекенде, с базами данных, application серверами и вот этим всем.

 **lair** 04.09.15 в 20:38 # 📌 🔄 ⬆

А чем вам уже упомянутая [Reactive Design Patterns](#) не нравится?

 **Nashev** 12.05.17 в 11:49 # 📌

Если книжка написана тем же языком, что и эта статья — то наверно лучше не надо было её выпускать. Чрезвычайно дырявое и рваное повествование. Постоянно появляющиеся ниоткуда новые понятия и ссылки в никуда. Не надо так.

Только [полноправные пользователи](#) могут оставлять комментарии. [Войдите](#), пожалуйста.

САМОЕ ЧИТАЕМОЕ

Сутки

Неделя

Месяц

Вирус VPNFilter, заразивший 500 тыс. роутеров оказался еще опаснее, чем считалось

↑ +35 👁 27,5k 📖 49 💬 38

Регистратор REG.RU лишил партнёра доступа к 70 тысячам доменов и забрал их обслуживание себе

↑ +148 👁 53k 📖 41 💬 412

Как я осилил английский

↑ +98 👁 38k 📖 346 💬 119

Деградация веба или как сделать веб человекочитаемым

↑ +35 👁 11,5k 📖 30 💬 44

NewSQL: SQL никуда не уходит

↑ +57 👁 15,6k 📖 93 💬 14

ИНТЕРЕСНЫЕ ПУБЛИКАЦИИ

Как не надо писать код

↑ +9 👁 964 📖 14 💬 0

Handmade: Программируемая клавиатура для онлайн-трейдинга своими руками

↑ +9 👁 1,6k 📖 9 💬 3

Начало

↑ +5

👁 3,8k

🔖 17

💬 27

Статус криптовалют в РФ не определен, но налоги с продажи платить придется

↑ +10

👁 4,2k

🔖 6

💬 41

К вопросу производительности старых и новых версий ноды

↑ +12

👁 2,7k

🔖 9

💬 12

Аккаунт

[Войти](#)

[Регистрация](#)

Разделы

[Публикации](#)

[Хабы](#)

[Компании](#)

[Пользователи](#)

[Песочница](#)

Информация

[Правила](#)

[Помощь](#)

[Документация](#)

[Соглашение](#)

[Конфиденциальность](#)

Услуги

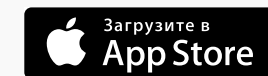
[Реклама](#)

[Тарифы](#)

[Контент](#)

[Семинары](#)

Приложения



© 2006 – 2018 «ТМ»

[О сайте](#)

[Служба поддержки](#)

[Мобильная версия](#)

