Инженерный вестник Дона, №3 (2025)
ivdon.ru/ru/magazine/archive/n3y2025/9934

Программирование с использованием модели акторов на платформе Akka: концепции, паттерны и примеры реализации

В.И. Шиян, Д.И. Вишняков, С.М. Власов, А.А. Костров, Е.Е. Тузовский, М.И. Щербаков

Кубанский государственный университет, Краснодар

Аннотация: В данной статье рассматриваются основные концепции и практические аспекты программирования с использованием модели акторов на платформе Akka. Акторная модель представляет собой мощный инструмент для создания параллельных и распределённых систем, обеспечивая высокую производительность, отказоустойчивость и масштабируемость. В статье подробно описываются основные принципы работы акторов, их жизненный цикл, механизмы обмена сообщениями, а также приводятся примеры реализации типичных паттернов, таких, как Master/Worker и Proxy. Особое внимание уделяется вопросам кластеризации и удалённого взаимодействия акторов, что делает статью полезной для разработчиков, работающих над распределёнными системами.
Ключевые слова: акторная модель, akka, параллельное программирование, распределённые системы, обмен сообщениями, кластеризация, отказоустойчивость, жизненный цикл актора, паттерны программирования, мастер-воркер, прокси-актор, синхронизация, асинхронность, масштабируемость, обработка ошибок.

Введение

Современные приложения всё чаще сталкиваются с необходимостью обработки больших объёмов данных в реальном времени, что требует использования параллельных и распределённых систем. Одним из наиболее эффективных подходов к реализации таких систем является акторная модель, которая была предложена Карлом Хьюиттом в 1973 году [1-3]. Акторная модель позволяет разработчикам создавать системы, которые легко масштабируются, обладают высокой отказоустойчивостью и способны эффективно использовать ресурсы многопроцессорных систем.

Akka – это популярная библиотека для JVM, которая реализует акторную модель и предоставляет инструменты для создания параллельных и распределённых приложений. В данной статье мы рассмотрим основные концепции Akka, а также приведём примеры кода, демонстрирующие использование акторов для решения типичных задач.

Основные концепции Akka

Акторная модель. Актор – это фундаментальная единица вычислений в акторной модели. Каждый актор инкапсулирует состояние и поведение, а также имеет собственную почтовую очередь, в которую поступают сообщения от других акторов. Акторы взаимодействуют друг с другом исключительно через обмен сообщениями, что обеспечивает высокую степень изоляции и предотвращает проблемы, связанные с параллельным программированием, такие как гонки данных, блокировки и взаимоблокировки.

Пример простого актора на языке Scala приведён на рис. 1. В этом примере создаётся актор SimpleActor, который реагирует на сообщение «hello», выводя приветствие на экран. Сообщения отправляются с помощью оператора !, который является синтаксическим сахаром для метода tell.

```
1.  import akka.actor._
2.
3.  class SimpleActor extends Actor {
4.    def receive = {
5.      case "hello" => println("Hello, world!")
6.      case _       => println("Unknown message")
7.    }
8.  }
9.
10. val system = ActorSystem("SimpleSystem")
11. val actor = system.actorOf(Props[SimpleActor], "simpleActor")
12.
13. actor ! "hello"
```

Рис. 1. – Пример реализации простого актора на языке Scala

Жизненный цикл актора. Акторы в Akka имеют чётко определённый жизненный цикл, который включает такие этапы, как создание, запуск, перезапуск и остановка. Каждый актор может быть перезапущен в случае возникновения ошибки, что обеспечивает высокую отказоустойчивость системы.

Пример обработки жизненного цикла актора приведён на рис. 2. В этом примере переопределены методы жизненного цикла актора, которые

вызываются при различных событиях, таких, как запуск, остановка и перезапуск.

```
class LifecycleActor extends Actor {
  override def preStart(): Unit = {
    println("Actor is starting")
  }

  override def postStop(): Unit = {
    println("Actor has stopped")
  }

  override def preRestart(reason: Throwable, message: Option[Any]): Unit = {
    println(s"Actor is restarting due to ${reason.getMessage}")
  }

  override def postRestart(reason: Throwable): Unit = {
    println("Actor has restarted")
  }

  def receive = {
    case _ => println("Received a message")
  }
}
```

Рис. 2. – Пример реализации актора с обработкой жизненного цикла

Механизмы обмена сообщениями

Гарантии доставки сообщений. Akka предоставляет различные гарантии доставки сообщений, такие как «at-most-once», «at-least-once» и «exactly-once» [4, 5]. Выбор гарантии зависит от требований приложения и может быть реализован с использованием различных паттернов, таких как подтверждение получения сообщения и повторная отправка. Например, гарантия «at-most-once» подходит для задач, где потеря сообщений допустима, а «at-least-once» используется в системах, где важно обеспечить доставку, даже если это приведёт к дублированию сообщений. Гарантия «exactly-once» требует более сложной реализации, включая дедупликацию и подтверждение получения, и используется в критически важных системах.

Пример реализации гарантии «at-least-once» приведён на рис. 3. В этом примере актор ReliableSender отправляет сообщение и планирует повторную отправку, если подтверждение не получено в течение заданного времени. Это позволяет обеспечить доставку сообщения даже в случае временных сбоев.

```scala
class ReliableSender extends Actor {
  var pendingMessages = Map[Int, (Any, ActorRef)]()

  def receive = {
    case (id: Int, message: Any) =>
      pendingMessages += (id -> (message, sender()))
      context.system.scheduler.scheduleOnce(1.second, self, id)
      recipient ! message

    case id: Int =>
      pendingMessages.get(id).foreach { case (message, originalSender) =>
        recipient ! message
        context.system.scheduler.scheduleOnce(1.second, self, id)
      }
  }
}
```

Рис. 3. – Пример реализации гарантии доставки сообщений «at-least-once»

Кластеризация и удалённое взаимодействие

Akka позволяет создавать кластеры акторов, которые могут работать на разных узлах сети [6, 7]. Это обеспечивает масштабируемость и отказоустойчивость системы. Кластеры Akka автоматически управляют членством узлов, отслеживают их состояние и обеспечивают прозрачное взаимодействие между акторами, даже если они находятся на разных узлах.

Пример создания кластера приведён на рис. 4. В этом примере актор подписывается на события кластера и получает уведомления о добавлении новых узлов.

```scala
import akka.actor._
import akka.cluster.Cluster

class ClusterActor extends Actor {
  val cluster = Cluster(context.system)

  override def preStart(): Unit = {
    cluster.subscribe(self, classOf[ClusterEvent.MemberUp])
  }

  def receive = {
    case ClusterEvent.MemberUp(member) =>
      println(s"Member is Up: ${member.address}")
  }
}

val system = ActorSystem("ClusterSystem")
val actor = system.actorOf(Props[ClusterActor], "clusterActor")
```

Рис. 4. – Пример реализации кластеризации в Akka

Паттерны программирования с использованием акторов

Паттерн Master/Worker. Паттерн Master/Worker используется для распределения задач между несколькими акторами [8, 9]. Мастер-актор разбивает задачу на подзадачи и распределяет их между воркерами, которые выполняют вычисления и возвращают результаты мастеру.

Пример реализации паттерна Master/Worker приведён на рис. 5. В этом примере мастер-актор создаёт несколько воркеров и распределяет между ними задачи. Воркеры выполняют задачи и возвращают результаты мастеру.

```
1.  class Master extends Actor {
2.    val workers = (1 to 4).map { i =>
3.      context.actorOf(Props[Worker], s"worker$i")
4.    }
5.
6.    def receive = {
7.      case task: Task =>
8.        workers.foreach(_ ! task)
9.      case result: Result =>
10.       println(s"Received result: $result")
11.   }
12. }
13.
14. class Worker extends Actor {
15.   def receive = {
16.     case task: Task =>
17.       val result = process(task)
18.       sender() ! result
19.   }
20.
21.   def process(task: Task): Result = {
22.     // Выполнение задачи
23.     Result(task.data)
24.   }
25. }
```

Рис. 5. – Пример реализации паттерна Master/Worker

Паттерн Proxy. Паттерн Proxy используется для создания промежуточного актора, который действует как посредник между отправителем и получателем сообщений [10]. Это может быть полезно для реализации таких функций, как кэширование, логирование или обеспечение надёжной доставки сообщений.

Пример реализации паттерна Proxy приведён на рис. 6. В этом примере ProxyActor пересылает сообщения целевому актору, добавляя при необходимости логирование или кэширование.

```scala
class ProxyActor(target: ActorRef) extends Actor {
  def receive = {
    case message =>
      // Логирование или кэширование
      target ! message
  }
}

val targetActor = system.actorOf(Props[TargetActor], "targetActor")
val proxyActor = system.actorOf(Props(new ProxyActor(targetActor)), "proxyActor")

proxyActor ! "Hello"
```

Рис. 6. – Пример реализации паттерна Proxy

Заключение

Акторная модель, реализованная в библиотеке Akka, предоставляет мощные инструменты для создания параллельных и распределённых систем. В статье были рассмотрены основные концепции Akka, такие как жизненный цикл акторов, механизмы обмена сообщениями, кластеризация и удалённое взаимодействие. Также были приведены примеры реализации типичных паттернов, таких как Master/Worker и Proxy. Использование Akka позволяет разработчикам создавать высокопроизводительные, отказоустойчивые и масштабируемые системы, что делает эту библиотеку незаменимой для современных распределённых приложений.

Литература

1. Hewitt C., Bishop P., Steiger R. A Universal Modular Actor Formalism for Artificial Intelligence. IJCAI. 1973. Pp. 235–245.

2. Clinger W. Foundations of Actor Semantics. Cambridge: MIT, 1981. 120 p.

3. Greif I. Semantics of Communicating Parallel Processes. Cambridge: MIT, 1975. 150 p.

4. Agha G., Mason I., Smith S., Talcott C. Foundations of Actor Computations. Journal of Functional Programming. 1993. Vol. 3, №1. Pp. 1–72.

5. Baker H., Hewitt C. Laws for Communicating Parallel Processes. IFIP. 1977. Pp. 1–12.

6. Hewitt C. What is Commitment? Physical, Organizational, and Social. IEEE Internet Computing. 2008. Vol. 12, №5. Pp. 293–307.

7. Agha G. Actors: A Model of Concurrent Computation in Distributed Systems. Cambridge: MIT Press, 1986. 200 p.

8. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. СПб.: БХВ-Петербург, 2002. 608 с.

9. Оленев Н.Н. Основы параллельного программирования в системе MPI. М.: ВЦ РАН, 2005. 80 с.

10. Федотов И. Модели параллельного программирования. М.: Солон-Пресс, 2012. 384 с.

References

1. Hewitt C., Bishop P., Steiger R. IJCAI. 1973. Pp. 235–245.

2. Clinger W. Foundations of Actor Semantics. Cambridge: MIT, 1981. 120 p.

3. Greif I. Semantics of Communicating Parallel Processes. Cambridge: MIT, 1975. 150 p.

4. Agha G., Mason I., Smith S., Talcott C. Journal of Functional Programming. 1993. Vol. 3, №1. Pp. 1–72.

5. Baker H., Hewitt C. IFIP. 1977. Pp. 1–12.

6. Hewitt C. IEEE Internet Computing. 2008. Vol. 12, №5. Pp. 293–307.

7. Agha G. Actors: A Model of Concurrent Computation in Distributed Systems. Cambridge: MIT Press, 1986. 200 p.

8. Voevodin V.V., Voevodin Vl.V. Parallel'nye vychisleniya [Parallel Computing]. St. Peterburg: BHV-Petersburg, 2002. 608 p.

9. Olenev N.N. Osnovy parallel'nogo programmirovaniya v sisteme MPI [Basics of Parallel Programming in MPI]. Moskva: VTs RAN, 2005. 80 p.

10. Fedotov I. Modeli parallel'nogo programmirovaniya [Models of Parallel Programming]. Moskva: Solon-Press, 2012. 384 p.

Дата поступления: 1.02.2025

Дата публикации: 3.02.2025