Actor Model

Курс «Параллельные вычисления»

Цесько Вадим Александрович
https://incubos.org

Санкт-Петербургский государственный политехнический университет

21 ноября 2014 г.

Содержание

1. Введение
2. Actor Model
3. Futures and Promises
4. Примеры систем
5. Заключение

Disclaimer

If I had asked people what they wanted, they would have said faster horses.

Henry Ford

Must Watch

Jonas Bonér. State: You're Doing It Wrong — Alternative Concurrency Paradigms For The JVM[a]. JavaOne 2009.

[a] http://www.slideshare.net/jboner/state-youre-doing-it-wrong-javaone-2009

Параллельное программирование

Damien Katz

- Beginner: "Threads are hard."
- Intermediate: "Don't fear multithreading."
- Expert: "Threads are hard."

Тем не менее

The free lunch is over[a].

[a] Herb Sutter. *The Free Lunch Is Over*. 2009

Проблема

Источник боли

Разделяемое состояние.

А если точнее, то

Изменяемое разделяемое состояние.

И

Часто оно **неизбежно**.

Классификация задач и подходов

- Shared State Concurrency (чугунная пуля)
  - Невероятно сложно
  - Даже для экспертов
- Координация независимых задач/процессов
  - Планирование, игры
  - Message-Passing Concurrency (Actor Model)
- Workflow-зависимые процессы
  - Банковская сфера, MapReduce, Spark, etc.
  - Dataflow Concurrency
- Консенсус и истинное разделяемое знание
  - Банковская сфера
  - Software Transactional Memory (STM)

Происхождение

- Carl Hewitt[1], Peter Bishop and Richard Steiger. A Universal Modular Actor Formalism for Artificial Intelligence. 1973.
- Gul Agha. Actors: A Model of Concurrent Computation in Distributed Systems. 1986.
- Изначально для описания параллельных вычислений
- Позднее в качестве основы для многочисленных реализаций

[1] http://letitcrash.com/post/20964174345/carl-hewitt-explains-the-essence-of-the-actor

Actor

- Всё это актор
- Функционируют параллельно
- Асинхронно обмениваются сообщениями
- При обработке сообщения актор может
  - отправить конечное число сообщений другим акторам
  - создать конечное число новых акторов
  - назначить поведение для обработки следующего сообщения
- Порядок доставки сообщений не специфицирован
- Акторы имеют «адреса»

Concurrency vs Parallelism

- Actor Model не о Parallelism, а о Concurrency[2]

Concurrency

Способность выполняться параллельно.

Parallelism

Действительно параллельное выполнение.

[2] Rob Pike. Concurrency is not Parallelism: `https://player.vimeo.com/video/49718712`

Реализации

Языки[3] с «родной» поддержкой:

- Erlang
- Scala
- ...

Библиотеки для языков:

- Scala
- Java
- F#
- ...

[3] http://en.wikipedia.org/wiki/Actor_model

Erlang

- Joe Armstrong, Ericsson, 1986
- Для разработки распределённых, отказоустойчивых, неостанавливающихся приложений мягкого реального времени
- Поддерживает «горячую» замену кода
- Пример — ПО коммутатора Ericsson AXD301
  - миллион строк кода
  - доступность **0.999999999** (31 ms/year downtime)

Parallelism vs Concurrency

Поддержка SMP в 2006.

Akka

Будем использовать

Akka (Scala API).

О проекте

Jonas Bonér:

- Java Champion
- Terracotta JVM clustering, JRockit JVM, AspectWerkz AOP, Eclipse AspectJ

Ресурсы:

- http://akka.io/docs/
- http://letitcrash.com/

Код:

- https://github.com/akka/akka
- Apache V2 license

Производительность

Внимание

Синтетические тесты ☺

Erlang R14B04 vs **Akka** 2.0-SNAPSHOT[4]:

- 1M mps vs **2.1M mps**

Akka 2.0[5]:

- **50M mps**
- 48-core, 128 GB, `ForkJoinPool`

[4] http://letitcrash.com/post/14783691760/akka-vs-erlang

[5] http://letitcrash.com/post/20397701710/50-million-messages-per-second-on-a-single-machine

Особенности реализации

- **300 байт** на актор
- Актор: состояние, поведение, почтовый ящик, список детей, стратегия супервизора
- Множество акторов на множестве нитей
- Нет гарантированной доставки, семантика **at-most-once**, порядок сохраняется
- Сообщения обрабатываются **строго по порядку**
- Иерархия: создаваемые акторы — дети, родитель — супервизор
- Актор скрыт за **переносимой** `ActorRef`
- Подход «Let it crash»

Пути

Примеры

- `akka://system/user/a/b`
- `akka.tcp://system@server.yandex.ru:2552/user/a/b`

Конструирование ссылок

- Создание акторов: `ActorRefProvider.actorOf`
- Поиск акторов: `ActorRefProvider.actorSelection`
- Каждый актор знает себя, родителя и детей

Можно делать так:

```
1 context.actorSelection("../brother") ! msg
2 context.actorSelection("/user/service") ! msg
3 context.actorSelection("../*") ! msg
```

Определение актора

```
1  class Partitioner(partitionStorage: ActorRef) extends Actor {
2
3    def receive = {
4      case PartitionFeed(partner, offers) =>
5          partition(partner, offers)
6      case msg =>
7          log.error("Unsupported message received: {}", msg)
8    }
9
10   def partition(partner: Partner, offers: Traversable[Offer]) {
11     ...
12
13     partitionStorage ! UpdatePartitions(partner, partitioning)
14   }
15 }
```

Создание актора

```
1 val system = ActorSystem("sharder")
2
3 val partitionStorage = ...
4
5 val partitioner =
6   system.actorOf(
7     FromConfig.props(
8       Props[Partitioner](new Partitioner(partitionStorage))
9         .withDispatcher("dispatcher.cpu")),
10    "partitioner")
```

Конфигурация диспетчера

Конфигурация в HOCON[6]:

```
dispatchers.cpu {
  type = Dispatcher
  executor = "fork-join-executor"

  fork-join-executor {
    parallelism-min = 4
    parallelism-factor = 1.0
  }
}
```

[6] Human-Optimized Config Object Notation: https://github.com/typesafehub/config

HOCON

Независимая самоценная библиотека:

- Чистая Java без зависимостей
- Поддерживает Java properties и JSON superset
- Merge и `include` из файлов, URL, classpath
- Мощная поддержка вложенности
- Поддержка duration (`10 seconds`) и size (`512K`)
- Конвертация типов

Диспетчеры

- `Dispatcher`
- `PinnedDispatcher`
- `BalancingDispatcher`
- `CallingThreadDispatcher`

на

- `fork-join-executor`
- `thread-pool-executor`

Почтовые ящики

- UnboundedMailbox
- BoundedMailbox
- UnboundedPriorityMailbox
- BoundedPriorityMailbox

```
1 trait MessageQueue {
2   def enqueue(receiver: ActorRef, handle: Envelope): Unit
3   def dequeue(): Envelope
4   def numberOfMessages: Int
5   def hasMessages: Boolean
6   def cleanUp(owner: ActorRef, deadLetters: MessageQueue): Unit
7 }
```

Пулы акторов

```
1 akka.actor.deployment {
2   /partitioner {
3     router = smallest-mailbox-pool
4     nr-of-instances = 4
5   }
6 }
```

```
1 akka.actor.deployment {
2   /unifier {
3     router = round-robin-pool
4     resizer {
5       lower-bound = 2
6       upper-bound = 16
7     }
8   }
9 }
```

Типы роутеров

Реализации `Pool` и `Group`:

- `RoundRobin`
- `Random`
- `SmallestMailbox`
- `Broadcast`
- `ScatterGatherFirstCompleted`
- Самописные

Конфигурирование из кода

```
1 val shardActorPaths: immutable.Seq[String] = ...
2
3 val shard = system.actorOf(
4   RoundRobinGroup(shardActorPaths).props(),
5   "shard")
```

Удалённые акторы

```
1  akka {
2    actor {
3      provider = "akka.remote.RemoteActorRefProvider"
4    }
5
6    remote {
7      enabled-transports = ["akka.remote.netty.tcp"]
8
9      netty.tcp {
10       hostname = "server.yandex.ru"
11       port = 2553
12     }
13   }
14 }
```

Тестирование акторов

- Модульное тестирование с `TestActorRef`
- Интеграционное тестирование с `Probe`
- Проверки с сопоставлением по шаблону

Пример теста

```scala
val probe = TestProbe()
val burstScaler = TestActorRef(new BurstScaler(probe.ref))

before {
  burstScaler.underlyingActor.sent =
      burstScaler.underlyingActor.sent.empty
}

"A BurstScaler" should {
  "always forward the first message" in {
    probe.within(1 second) {
      burstScaler ! 1
      probe.expectMsg(1)
    }
  }
}
```

Супервизор

Решение при сбое:

1. `Resume`
2. `Restart`
3. `Stop`
4. `Escalate`

- Принятое решение (1-3) действует рекурсивно
- Функция `Exception` ⇒ `Directive`
- `Terminated`, `preStart`, `preRestart`, `postStop`, `postRestart`
- `OneForOneStrategy` и `AllForOneStrategy`
- Ограничение количества перезапусков

Супервизор по умолчанию

```
1 final val defaultDecider: Decider = {
2   case _: ActorInitializationException => Stop
3   case _: ActorKilledException      => Stop
4   case _: DeathPactException        => Stop
5   case _: Exception                 => Restart
6 }
7
8 final val defaultStrategy: SupervisorStrategy =
9   OneForOneStrategy()(defaultDecider)
```

Акторы

- Храните состояние вовне
- Стройте **всю систему** на акторах
- Пишите асинхронный код
- Избегайте косвенного взаимодействия акторов
- Баги есть, но быстро чинят

Память и ящики

- Неограниченные ящики $\Rightarrow$ неограниченная память при перегрузке
- Ограниченные ящики $\Rightarrow$ возможные deadlock'и при наличии циклов
- $\Rightarrow$ стройте системы без циклов ☺
- Существует диспетчер по умолчанию
- Нет доступа к размеру ящика[7]

[7] http://letitcrash.com/post/17707262394/why-no-mailboxsize-in-akka-2

Alan Kay on OOP, 1967

I thought of objects being like biological cells and/or individual computers on a network, only able to communicate with messages (so messaging came at the very beginning – it took a while to see how to do messaging in a programming language efficiently enough to be useful).
OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things. It can be done in Smalltalk and in LISP. There are possibly other systems in which this is possible, but I'm not aware of them.

Futures and Promises

David Ross

So a Promise can be broken, but you can't change the Future. Love the metaphors.

Происхождение

- Henry Baker and Carl Hewitt. The Incremental Garbage Collection of Processes[8]. 1977.

Future

Структура данных для **синхронного** (блокирующегося) или **асинхронного** (неблокирующегося) извлечения результата параллельной операции.

[8] http://en.wikipedia.org/wiki/Futures_and_promises

Реализации

- Java (`java.util.concurrent.Future`)
- Scala
- C#
- Python
- Haskell
- Clojure
- Oz
- R
- Scheme
- Node.js
- **Akka**

Futures and Actors

Общение с акторами вне Actor System:

```
1 implicit val timeout = Timeout(5 seconds)
2
3 val future = actor ? msg
4 val result =
5   Await.result(future, timeout.duration)
6     .asInstanceOf[String]
```

Или так:

```
1 val future: Future[String] = ask(actor, msg).mapTo[String]
```

Прямое использование

```scala
val future = Future {
  "Hello" + "World"
}
val result = Await.result(future, 1 second)
```

```scala
val future = Promise.successful("Yay!")
```

```scala
val otherFuture =
  Promise.failed[String](
      new IllegalArgumentException("Bang!"))
```

Future — это монада: map (1)

```
val f1 = Future {
  "Hello" + "World"
}

val f2 = f1 map { x =>
  x.length
}

val result = Await.result(f2, 1 second)

result must be(10)

f1.value must be(Some(Right("HelloWorld")))
```

Future — это монада: map (2)

```
1 val f1 = Future {
2   "Hello" + "World"
3 }
4 val f2 = Promise.successful(3)
5 val f3 = f1 map { x =>
6   f2 map { y =>
7     x.length * y
8   }
9 }
```

Тип f3?

- Future[Future[Int]]
- А хотели: Future[Int]

Future — это монада: flatMap

```
1  val f1 = Future {
2    "Hello" + "World"
3  }
4  val f2 = Promise.successful(3)
5  val f3 = f1 flatMap { x =>
6    f2 map { y =>
7      x.length * y
8    }
9  }
10
11 val result = Await.result(f3, 1 second)
12
13 result must be(30)
```

Future — это монада: filter

```
1  val future1 = Promise.successful(4)
2  val future2 = future1.filter(_ % 2 == 0)
3  val result = Await.result(future2, 1 second)
4  result must be(4)
5
6  val failedFilter = future1.filter(_ % 2 == 1).recover {
7    case m: MatchError => 0
8  }
9
10 val result2 = Await.result(failedFilter, 1 second)
11 result2 must be(0)
```

Futures: композиция (1)

```
1 val f1 = ask(actor1, msg1)
2 val f2 = ask(actor2, msg2)
3
4 val a = Await.result(f1, 1 second).asInstanceOf[Int]
5 val b = Await.result(f2, 1 second).asInstanceOf[Int]
6
7 val f3 = ask(actor3, (a + b))
8
9 val result = Await.result(f3, 1 second).asInstanceOf[Int]
```

Блокировки

Каждый вызов `Await.result()` — блокировка.

Futures: композиция (2)

```
1  val f1 = ask(actor1, msg1)
2  val f2 = ask(actor2, msg2)
3
4  val f3 = for {
5    a <= f1.mapTo[Int]
6    b <= f2.mapTo[Int]
7    c <= ask(actor3, (a + b)).mapTo[Int]
8  } yield c
9
10 val result = Await.result(f3, 1 second).asInstanceOf[Int]
```

Блокировки

Уже лучше, но усложняется с ростом числа акторов.

Futures: композиция (3)

```
1 Future.sequence:
2   M[Future[A]] => Future[M[A]]
```

```
1 Future.traverse:
2   M[A] => (A => Future[B]) => Future[M[B]]
```

```
1 Future.fold:
2   Traversable[Future[T]] => R => ((R, T) => R) => Future[R]
```

```
1 Future.reduce[T, R >: T]:
2   Traversable[Future[T]] => ((R, T) => R) => Future[R]
```

Futures: Callbacks

```
1 future onSuccess {
2   case "bar"     => println("Got my bar alright!")
3   case x: String => println("Got some random string: " + x)
4 }
```

```
1 future onFailure {
2   case ise: IllegalStateException
3     if ise.getMessage == "OHNOES" =>
4       // OHNOES! We are in deep trouble, do something!
5   case e: Exception =>
6     // Do something else
7 }
```

```
1 future onComplete {
2   case Success(result) => doSomethingOnSuccess(result)
3   case Failure(error) => doSomethingOnFailure(error)
4 }
```

Futures: Порядок Callback'ов

Исполнение Callback'ов

Обработчики исполняются в **неопределенном порядке** и/или **параллельно**.

```
1 val result = Future { loadPage(url) } andThen {
2   case Left(exception) => log(exception)
3 } andThen {
4   case _ => watchSomeTV
5 }
```

Futures: Полезности

`Future.fallbackTo()`:

```
val future4 = future1 fallbackTo future2 fallbackTo future3
```

`Future.zip()`:

```
val future3 =
  future1 zip future2 map { case (a, b) => a + " " + b }
```

Futures and Exceptions

Исключения доставляются **вместо** результата `Future`.

Но можно делать так:

```
1 val future = akka.pattern.ask(actor, msg1) recover {
2   case e: ArithmeticException => 0
3 }
```

Или так:

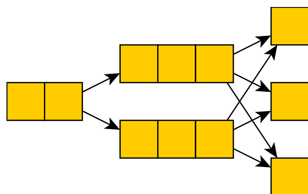
```
1 val future = akka.pattern.ask(actor, msg1) recoverWith {
2   case e: ArithmeticException =>
3     Promise.successful(0)
4   case foo: IllegalArgumentException =>
5     Promise.failed[Int](
6         new IllegalStateException("All br0ken!"))
7 }
```

Классификация

- Конвейерная индексация
  - Яндекс.{Авто, Недвижимость, Работа}
  - etc.
- Иерархическая обработка пользовательских запросов
  - Spray[9]
  - Маршрутизация и обработка кликов и показов
  - Матчинг объявлений на подписки
  - etc.

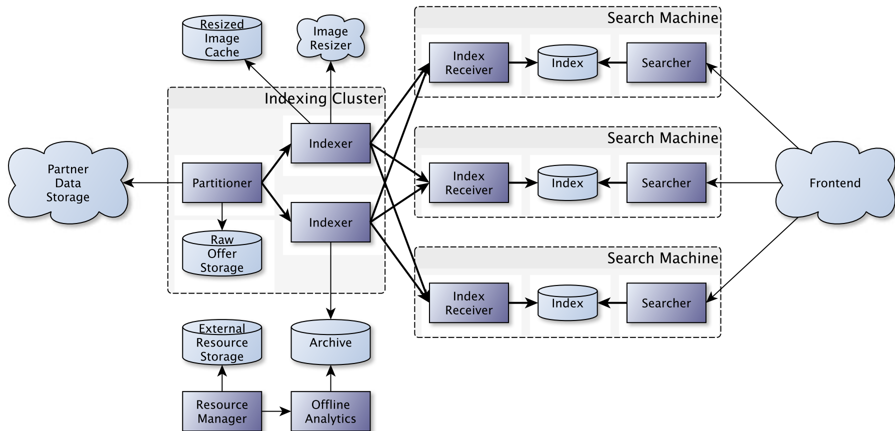
[9] http://spray.io

Конвейер

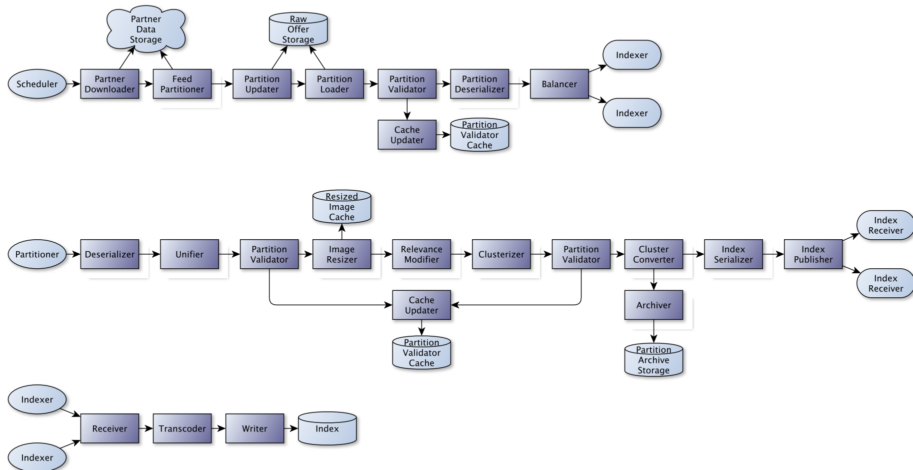


- Масштабируемость
- Устойчивость к сбоям
- Ограниченные очереди
- Back pressure
- Выделенные диспетчеры
- Детерминированное потребление памяти
- Графики размеров очередей
- Декомпозиция стадий
- Масштабирование пулами
- Загрузка CPU 100%

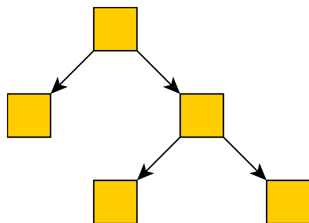
Яндекс.Авто: Архитектура



Яндекс.Авто: Гиперконвейер



Иерархическая обработка



- Масштабируемость
- Высокая доступность
- Естественные подсистемы
- Часто динамическая топология
- Actor per request
- Request-reply

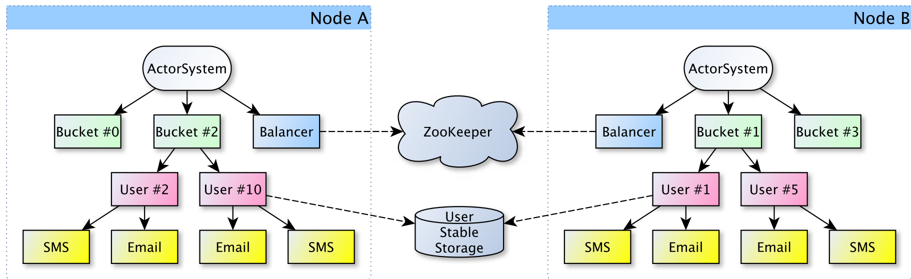
Иерархические системы

- Внутренние сервисы
- Иерархия акторов для каждого активного пользователя/подписки/визита/запроса
- Динамическая выгрузка/подгрузка сущностей
- Распределение пользователей на бакеты (и ноды) через ZooKeeper + Curator[10] (Consistent Hashing[11])

[10] http://curator.apache.org

[11] http://en.wikipedia.org/wiki/Consistent_hashing

Компонент сервиса уведомлений



О чём нужно думать

- Наблюдаемость
  - Логгирование
  - Агрегатные графики
  - Мониторинги
- Поведение при перегрузке
  - Детектирование перегрузки
  - Ограничение ресурсов
  - Превентивный reject
  - Восстановление после перегрузки
  - Выравнивание нагрузки

Заключение

- Параллельные программы — это неизбежно
- Нужны более простые способы создания **корректных параллельных** программных систем
- Низкоуровневый параллелизм — это чрезвычайно сложно
- Но есть альтернативы[12]:
  - Message-Passing Concurrency (Actor Model)
  - Dataflow Concurrency
  - Software Transactional Memory (STM)
- No Silver Bullet

[12] ScalaDays 2013. Concurrency — The good, the bad, the ugly: http://www.parleys.com/play/51c0bc58e4b0ed877035680a/

Куда двигаться дальше

- Typesafe Activator[13]
- Reactive Manifesto[14]
- Principles of Reactive Programming[15]
- Книги:
  - Jamie Allen. *Effective Akka*. 2013
  - Derek Wyatt. *Akka Concurrency*. 2013
  - Series: The Neophyte's Guide to Scala[16]
  - EAI Patterns Series[17]
  - Chris Okasaki. *Purely Functional Data Structures*. 1999
  - JCIP 2nd edition + JMM

[13] http://typesafe.com/activator

[14] http://www.reactivemanifesto.org

[15] https://class.coursera.org/reactive-001/class

[16] http://letitcrash.com/post/64667109914/series-the-neophytes-guide-to-scala

Вопросы?

- http://incubos.org/
- Общие вопросы — в Twitter: @incubos
- Вопросы по лекциям — в комментариях: http://incubos.org/blog/
- Частные вопросы — в почту incubos@yandex.ru