Actor Model

Курс «Параллельные вычисления»

Цесько Вадим Александрович
http://kspt.ftk.spbstu.ru/people/tsesko/
@incubos

Санкт-Петербургский государственный политехнический университет

29 ноября 2013 г.

Содержание

Обо мне

- 5 лет в Digitek Labs
  - Параллельные системы цифровой обработки сигналов
  - Статический анализ кода программных систем
- 2.5 года в Яндекс
  - Распределённые высоконагруженные системы
- Преподавание
  - Базы данных
  - Формальные методы обеспечения качества ПО

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
  - 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).

Класс систем

- **Яндекс.Авто**
- Яндекс.Недвижимость

ETL-процесс:

1. **Extract** — загрузка внешних данных
2. **Transform** — унификация, кластеризация, ...
3. **Load** — построение и раскладка индекса

Статистика

Данные:

- До 10^4 источников
- До 10^7 сущностей
- 20% обновляется ежедневно
- 2x рост за год

Пользователи[4]:

- Яндекс.Авто — 4.5 млн. чел.
- Яндекс.Недвижимость — 2 млн. чел.

[4] `http://stat.yandex.ru`

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[5]:

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

Akka 2.0[6]:

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

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

[6] `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»

Ссылки

- Чисто локальные
- Локальные ссылки
- Ссылки для маршрутизации (`Router`)
- Ссылки на удалённых акторов
- Особые: `PromiseActorRef`, `DeadLetterActorRef`, `EmptyLocalActorRef` (`DeadLetterActorRef`)

Примеры путей

- `akka://system/user/service/worker`
- `akka://system@server.yandex.ru:2552/user/service`

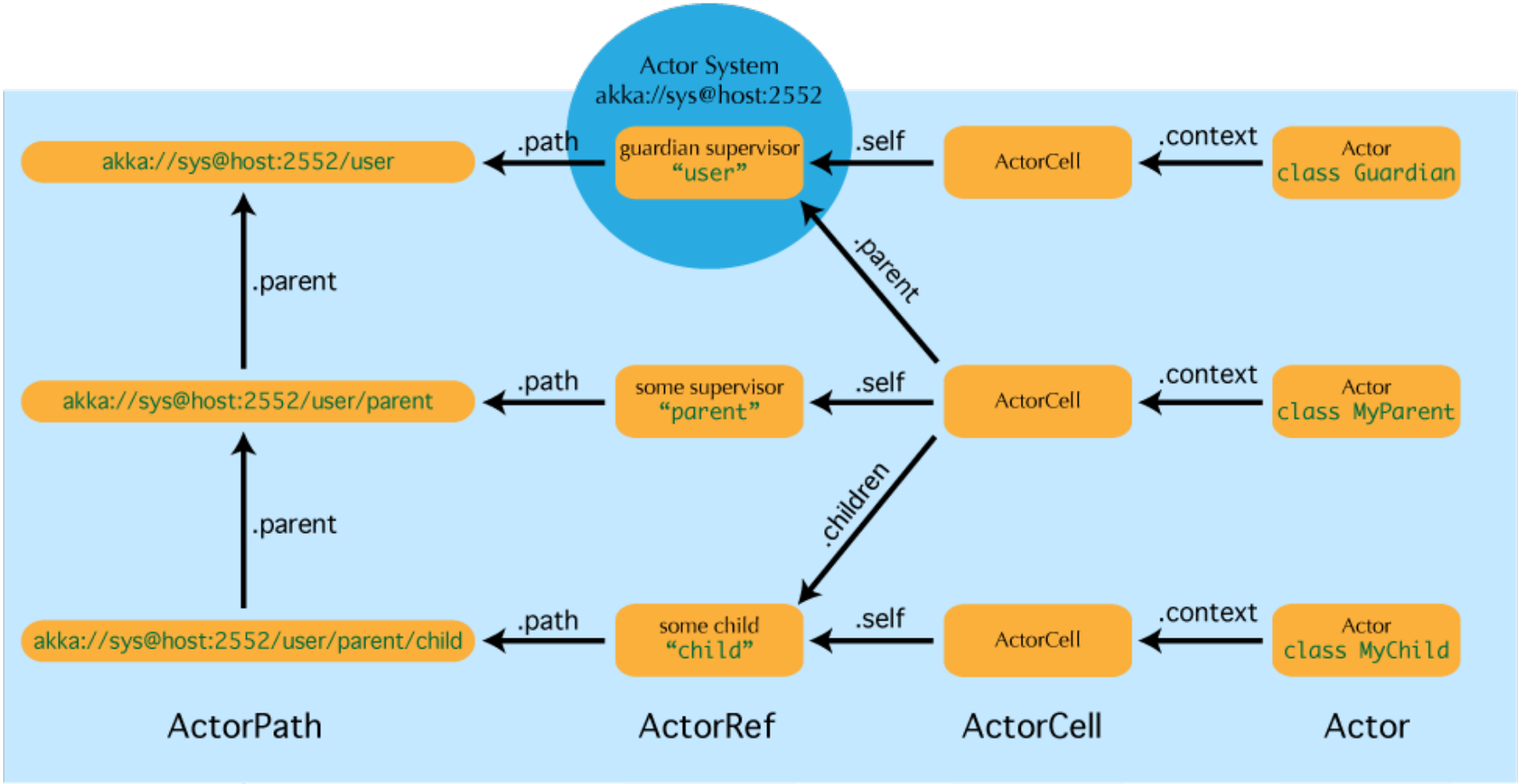
Конструирование ссылок

- Создание акторов: `ActorSystem.actorOf` или `ActorContext.actorOf`
- Поиск акторов: `ActorSystem.actorFor` или `ActorContext.actorFor`
- Каждый актор знает себя, родителя и детей

Можно делать так:

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

Именование в локальной системе



Пути

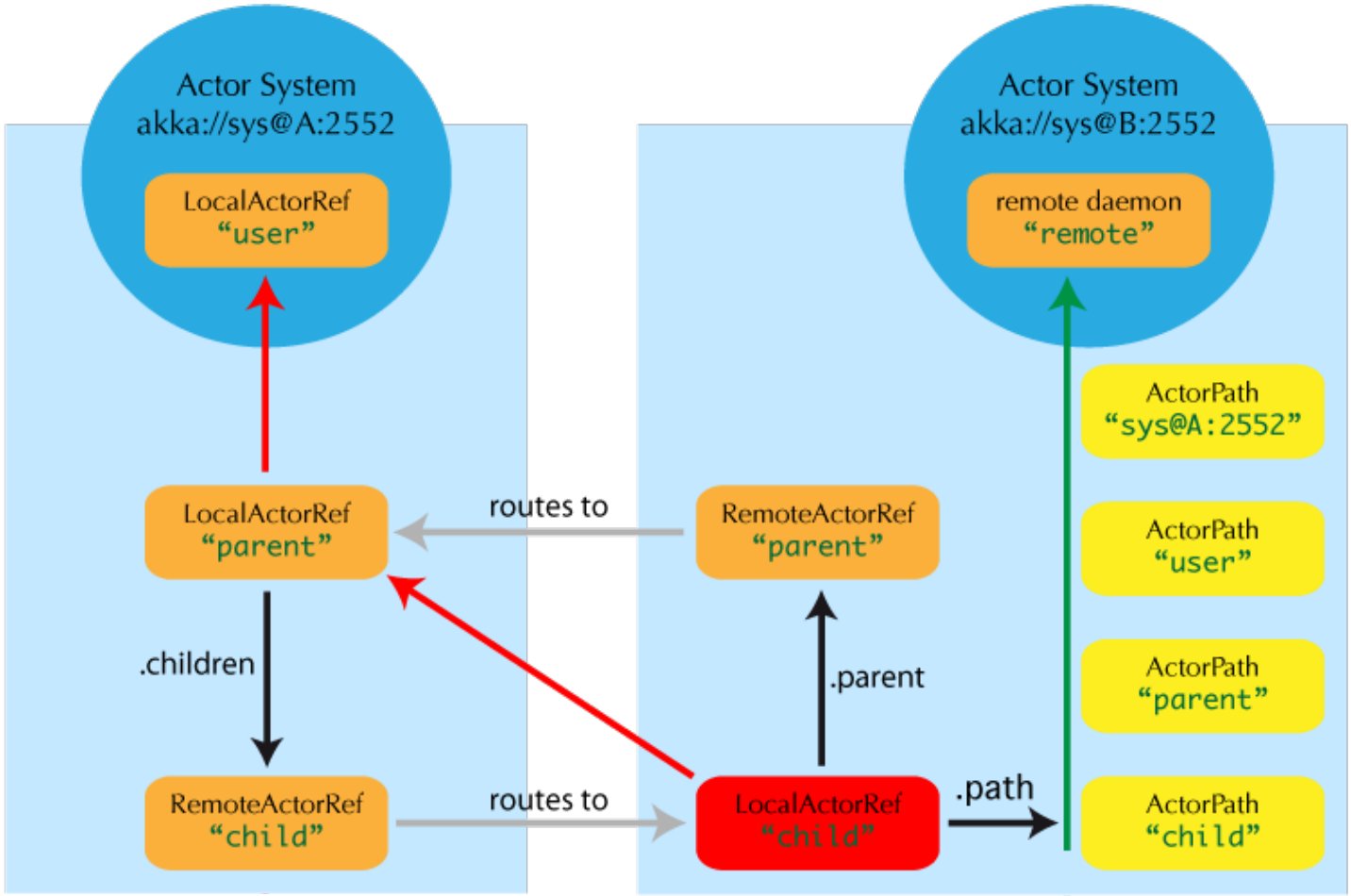
Типы путей:

- Логический
- Физический

Особые пути:

- `/user`
- `/system`
- `/deadLetters`
- `/temp`
- `/remote`

Удалённое развёртывание



logical actor path: akka://sys@A:2552/user/parent/child

physical actor path: akka://sys@B:2552/remote/sys@A:2552/user/parent/child

Actor Best Practices

- Не блокироваться
- Неизменяемые сообщения
- Нет разделяемому состоянию
- «Опасные» подзадачи в дочерние акторы
- События жизненного цикла

Основные фичи

- Actors
- Logging
- Scheduler
- Dispatchers
- Routing
- Remoting
- Serialization
- Testing
- FSM
- Fault Tolerance

Дополнительные фичи

- Typed Actors
- Event Bus
- Futures
- Dataflow Concurrency
- STM
- Agents
- Transactors
- Durable Mailboxes (file, Redis, ZooKeeper, Mongo)
- **Akka Cluster**

Определение актора

```
1  class Partitioner(partitionStorage: ActorRef) extends Actor {
2
3    lazy val log = Logging(context.system, this)
4
5    def receive = {
6      case PartitionFeed(partner, offers) =>
7          partition(partner, offers)
8      case msg =>
9          log.error("Unsupported message received: {}", msg)
10   }
11
12   def partition(partner: Partner, offers: Traversable[Offer]) {
13     ...
14
15     partitionStorage ! UpdatePartitions(partner, partitioning)
16   }
17 }
```

Создание актора

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

Конфигурация диспетчера

Конфигурация в HOCON[7]:

```
dispatcher {
  cpu {
    type = Dispatcher
    executor = "fork-join-executor"
    mailbox-capacity = 4
    mailbox-push-timeout-time = 5m

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

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

Диспетчеры и ящики

Диспетчеры:

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

Почтовые ящики:

- `UnboundedMailbox`
- `BoundedMailbox`
- `UnboundedPriorityMailbox`
- `BoundedPriorityMailbox`
- Durable Mailboxes

Конфигурация маршрутизатора

```
akka.actor.deployment {
  /partitioner {
    router = smallest-mailbox
    nr-of-instances = 4
  }
}
```

```
akka.actor.deployment {
  /unifier {
    router = round-robin
    resizer {
      lower-bound = 2
      upper-bound = 16
    }
  }
}
```

Маршрутизаторы

- `RoundRobinRouter`
- `RandomRouter`
- `SmallestMailboxRouter`
- `BroadcastRouter`
- `ScatterGatherFirstCompletedRouter`

Конфигурирование из кода

```
1 val shardActors =
2   shards.map(system.actorFor(_)).toIndexedSeq
3
4 val shard = system.actorOf(
5   Props[PartitionReceiver]
6       .withRouter(RoundRobinRouter(shardActors))
7       .withDispatcher("dispatcher.shard"),
8   "shard")
```

Распределённые акторы

```
akka {
  actor {
    provider = "akka.remote.RemoteActorRefProvider"
  }

  remote {
    transport = "akka.remote.netty.NettyRemoteTransport"

    netty {
      hostname = "server.yandex.ru"
      port = 2553
    }
  }
}
```

Пример теста

```
1  val probe = TestProbe()
2  val burstScaler = TestActorRef(new BurstScaler(probe.ref))
3
4  before {
5    burstScaler.underlyingActor.sent =
6        burstScaler.underlyingActor.sent.empty
7  }
8
9  "A BurstScaler" should {
10   "always forward the first message" in {
11     probe.within(1 second) {
12       burstScaler ! 1
13       probe.expectMsg(1)
14     }
15   }
16 }
```

Тестирование акторов

- Модульное тестирование с `TestActorRef`
- Интеграционное тестирование с `Probe`
- Проверки с сопоставлением по шаблону
- Замедление времени
- Детализированные логи `akka.actor.debug.*`
- `CallingThreadDispatcher`

Супервизор

Решение при сбое:

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

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

Супервизор по умолчанию

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

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[8]

[8] 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[9]. 1977.

Future

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

[9] 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

```
1 val future = Future {
2   "Hello" + "World"
3 }
4 val result = Await.result(future, 1 second)
```

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

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

Future — это монада: map (1)

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

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

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

```
future onComplete {
  case Right(result) => doSomethingOnSuccess(result)
  case Left(failure) => doSomethingOnFailure(failure)
}
```

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():

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

Future.zip():

```
1 val future3 =
2   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 brOken!"))
7 }
```

Ссылки

- Peter Van Roy and Seif Haridi. Concepts, Techniques and Models of Computer Programming[10].

[10] http://www.info.ucl.ac.be/~pvr/book.html

Реализация в Akka

- В стиле языка Oz[11]
- Декларативное описание
- Ленивое вычисление «по требованию»
- Базируется на Futures
- Вычисление **детерминировано**
- Нет разницы между последовательным и параллельным кодом
- Использует Scala Delimited Continuations

[11] http://www.mozart-oz.org/documentation/tutorial/node8.html#chapter.concurrency

Ограничения

- Код должен быть **без побочных эффектов** — только «чистые» функции
- Подход не общего назначения
- Подходит для четко определенных изолированных модулей

Примитивы

- **Создание** переменной потока данных
- **Ожидание** «связывания» переменной
- **Связывание** переменной

Dataflow Delimiter (1)

```
1  import Future.flow
2
3  implicit val dispatcher = ...
4
5  val a = Future( ... )
6  val b = Future( ... )
7  val c = Promise[Int]()
8
9  flow {
10   c << (a() + b())
11 }
12
13 val result = Await.result(c, timeout)
```

Dataflow Delimiter (2)

```
1  import Future.flow
2
3  implicit val dispatcher = ...
4
5  val a = Future( ... )
6  val b = Future( ... )
7
8  val c = flow {
9    a() + b()
10 }
11
12 val result = Await.result(c, timeout)
```

Пример

```
1  import akka.dispatch._
2  import Future.flow
3
4  implicit val dispatcher = ...
5
6  val x, y, z = Promise[Int]()
7
8  flow {
9    z << x() + y()
10   println("z = " + z())
11 }
12
13 flow { x << 40 }
14
15 flow { y << 2 }
```

Происхождение

- Tom Knight. An Architecture for Mostly Functional Languages. 1986.
- Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. 1993.
- Nir Shavit and Dan Touitou. Software Transactional Memory. 1997.

- Вначале как идея аппаратной поддержки транзакций
- В дальнейшем развитие в сторону чисто программной транзакционной памяти

Идея

- Взгляд на память (Java Heap) как на транзакционный набор данных
- Похоже на БД: `begin`, `commit`, `abort/rollback`
- Транзакции **автоматически перезапускаются** при коллизиях
- `rollback` при ошибках
- Свойства ACI (из ACID)

Реализации

- Clojure STM[12]
- **Akka** — войдет в SDK Scala
- Haskell
- etc

Ссылки:

- Rich Hickey. Persistent Data Structures and Managed References[13]

[12] http://clojure.org/state

[13] http://www.infoq.com/presentations/Value-Identity-State-Rich-Hickey

Свойства

Достоинства

Транзакции могут быть вложенными.

Недостатки

Все операции в области действия транзакции должны быть **идемпотентными** и **не иметь побочных эффектов**.

Akka Transactional Agents

```
1  def transfer(from: Agent[Int], to: Agent[Int], amount: Int):
       Boolean = {
2    atomic { txn =>
3      if (from.get < amount) false
4      else {
5        from send (_ - amount)
6        to send (_ + amount)
7        true
8      }
9    }
10 }
11
12 val from = Agent(100)
13 val to = Agent(20)
14 val ok = transfer(from, to, 50)
15 val fromValue = from.await // -> 50
16 val toValue = to.await // -> 70
```

Потенциальные проблемы

Множественные **коллизии**:

- Низкая производительность и большие задержки
- Отсутствие прогресса (live locking)
- Несправедливость (unfairness)

Заключение

- Параллельные программы — это неизбежно
- Нужны более простые способы создания **корректных параллельных** программных систем
- Низкоуровневый параллелизм — это чрезвычайно сложно
- Но есть альтернативы[14]:
  - Message-Passing Concurrency (Actor Model)
  - Dataflow Concurrency
  - Software Transactional Memory (STM)
- No Silver Bullet

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

Куда двигаться дальше

- Typesafe Activator[15]
- Reactive Manifesto[16]
- Principles of Reactive Programming[17]
- Книги:
  - Jamie Allen. *Effective Akka*. 2013
  - Derek Wyatt. *Akka Concurrency*. 2013
  - Series: The Neophyte's Guide to Scala[18]
  - EAI Patterns Series[19]
  - Chris Okasaki. *Purely Functional Data Structures*. 1999
  - JCIP 2nd edition + JMM

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

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

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

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

Вопросы?

- http://incubos.org/contacts/
- Общие вопросы — в Twitter: @incubos
- Вопросы по лекциям — в комментариях: http://incubos.org/blog/
- Частные вопросы — в почту vadim.tsesko@gmail.com