

Actor Model

Курс «Параллельные вычисления»

Вадим Цесько
incubos.org

Санкт-Петербургский политехнический университет Петра Великого

23 мая 2018 г.

Содержание

- 1 Introduction
- 2 Actor Model
- 3 Use Cases
- 4 Futures and Promises
- 5 Reactive Streams
- 6 Conclusion

О лекторе

- <http://kspt.icc.spbstu.ru/people/tsesko/>
- <http://digiteklabs.ru>
- <https://yandex.ru>
- <https://compscicenter.ru/teachers/12/>
- <https://v.ok.ru>

Disclaimer

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

Henry Ford

Jonas Bonér

State: You're Doing It Wrong — Alternative Concurrency Paradigms For The JVM^a.

^ahttp:

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

^aHerb Sutter. *The Free Lunch Is Over*. 2009

Проблема

Источник

Разделяемое состояние.

А точнее

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
 - Message-Passing Concurrency (Actor Model)
- Workflow-зависимые процессы
 - MapReduce, Spark, Tez, etc.
 - Dataflow Concurrency
- Консенсус и истинное разделяемое знание
 - Transactional RDBMS
 - Software Transactional Memory (STM)

Происхождение

- Carl Hewitt¹, 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:](http://letitcrash.com/post/20964174345/carl-hewitt-explains-the-essence-of-the-actor)

[//letitcrash.com/post/20964174345/carl-hewitt-explains-the-essence-of-the-actor](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 Concurrency²

Concurrency

Способность выполняться параллельно.

Parallelism

Действительно параллельное выполнение.

²Rob Pike. Concurrency is not Parallelism: <https://player.vimeo.com/video/49718712>

Реализации

Языки³ с «родной» поддержкой:

- Erlang
- Scala
- ...

Библиотеки для языков:

- Scala
- Java
- F#
- ...

³http://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.

Акка

Будем использовать
Акка (Scala API).

Некоторые пользователи:

- The Guardian
- CISCO
- Ebay
- Groupon
- BBC
- Amazon.com
- Яндекс 😊

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-SNAPSHOT⁴:

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

Akka 2.0⁵:

- **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

Примеры

```
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(classOf[Partitioner], partitionStorage)
9         .withDispatcher("dispatcher.cpu")),
10    "partitioner")
```

Конфигурация диспетчера

Конфигурация в HOCON⁶:

```
1 dispatchers.cpu {
2   type = Dispatcher
3   executor = "fork-join-executor"
4
5   fork-join-executor {
6     parallelism-min = 4
7     parallelism-factor = 1.0
8   }
9 }
```

⁶Human-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-transport = ["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

```
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 }
```

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 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)
```


Выпало из рассмотрения

- FSM
- Persistence
- Typed Actors
- Cluster
- Event Bus
- Streams
- HTTP
- Experimental

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 ящика⁷

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.

Задача

- Запросы приходят по одному
- Общаться с хранилищем эффективнее пачками
- Допустима фиксированная задержка

Решение

```
1 class Batcher[T: ClassTag](persistor: ActorRef, batchSize: Int, flushPeriod:
  FiniteDuration) extends Actor {
2   var batch = Seq.empty[T]
3   context.setReceiveTimeout(flushPeriod)
4   def receive = {
5     case event: T =>
6       batch = batch :+ event
7       if (batch.size == batchSize) flush()
8     case ReceiveTimeout => if (batch.nonEmpty) flush()
9   }
10  def flush() {
11    persistor ! BatchRequest(batch)
12    batch = Seq.empty[T]
13  }
14 }
```

Задача

- На входе поток запросов
- WebSockets/HTTP persistence

Решение

- Актор на соединение (Akka IO⁸)
- Актор на запрос (Spray⁹/Akka HTTP¹⁰)

⁸<http://doc.akka.io/docs/akka/2.4.4/scala/io.html>

⁹<http://spray.io/documentation/1.2.3/spray-can/http-server/>

¹⁰<http://doc.akka.io/docs/akka/2.4.4/scala/http/introduction.html>

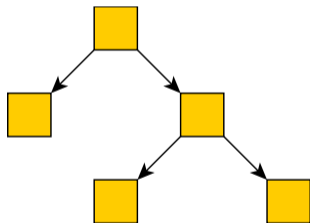
Задача

- Обработка запроса — общение с другими акторами
- Взаимодействия запрос-ответ
- Необходимо восстановить контекст перед ответом пользователю

Решение

- Трансформируем контекст `HttpContext` в `AnyRef`
- Контракт: каждый актер копирует непрозрачный контекст из запроса в ответ
- Корневой узел трансформирует контекст `AnyRef` из ответа в `HttpContext`

Задача



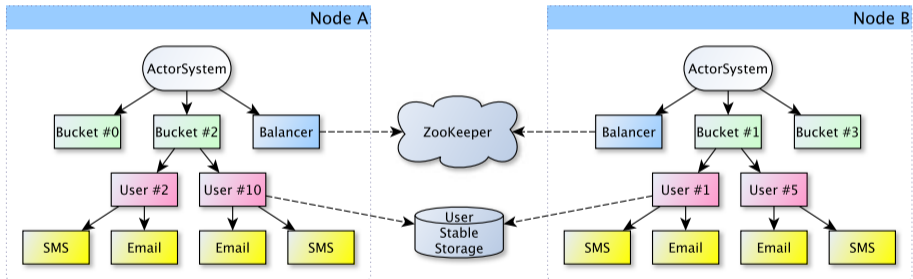
- Естественные подсистемы
- Масштабируемость
- Устойчивость к сбоям
- Реактивность
- Сессионность

Решение

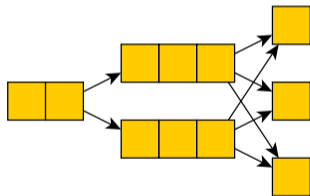
- User pinning (DC, node) + proxy
- Иерархия акторов для каждого активного пользователя/подписки/визита/запроса/whatever¹¹
- Eviction
- State flush and/or persistence

¹¹<http://2016.secr.ru/program/submitted-presentations/streaming-matching-of-events>

Компонент сервиса уведомлений



Задача



- Масштабируемость
- Устойчивость к сбоям
- Ограниченное потребление памяти

Решение

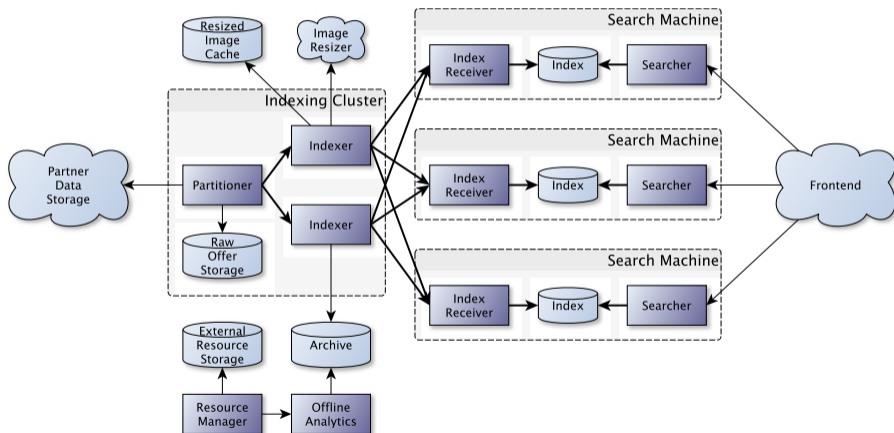
- Каждая стадия — актор¹²
- Ограниченные очереди
- Back pressure
- LoadBalancingDispatchers
- Графики размеров очередей
- Декомпозиция тяжёлых стадий
- Масштабирование пулами
- Загрузка CPU 100%
- InstrumentedUnboundedSkipClonesMailbox

¹²<http://2014.jpoint.ru/talks/07/>

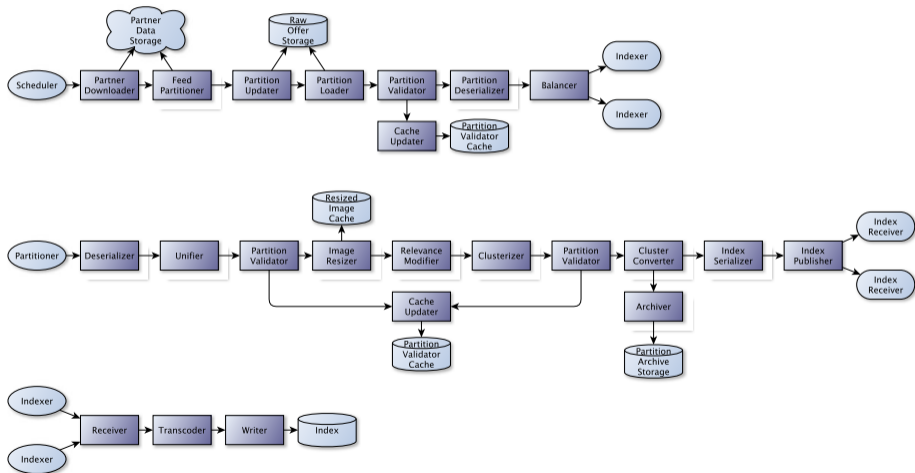
Проблемы

- Ёмкость зависит от структуры конвейера
- Затор приводит к отбрасыванию случайных задач

Архитектура



Гиперконвейер



О чём приходится думать

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

Futures and Promises

David Ross

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

Зачем

- Future возникают, когда появляется асинхронный IO
- Т.е. повсеместно

Происхождение

- Henry Baker and Carl Hewitt. The Incremental Garbage Collection of Processes¹³. 1977.

Future

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

¹³http://en.wikipedia.org/wiki/Futures_and_promises

Реализации

- Java (`java.util.concurrent.Future`)
- Scala¹⁴ (**Akka**¹⁵)
- C#
- Python
- Haskell
- Clojure
- Oz
- R
- Scheme
- Node.js

¹⁴<http://docs.scala-lang.org/overviews/core/futures.html>

¹⁵<http://doc.akka.io/docs/akka/2.4.4/scala/futures.html>

Future

```
1 import scala.concurrent.ExecutionContext.Implicits.global

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

1 val future = Future.successful("Yay!")

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

Promise

```
1 import scala.concurrent.{Future, Promise}
2 import scala.concurrent.ExecutionContext.Implicits.global
3
4 val p = Promise[T]()
5 val f = p.future
6 val producer = Future {
7   val r = produceSomething()
8   p success r
9   continueDoingSomethingUnrelated()
10 }
11 val consumer = Future {
12   startDoingSomething()
13   f onSuccess {
14     case r => doSomethingWithResult()
15   }
16 }
```


Futures and Actors

Общение с акторами вне Actor System:

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

Или так:

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

И обратно:

```
1 import akka.pattern.pipe
2 future pipeTo actor
```

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(Success("HelloWorld")))
```

map (2)

```
1 val f1 = Future {  
2   "Hello" + "World"  
3 }  
4 val f2 = Future.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]

flatMap

```
1 val f1 = Future {
2   "Hello" + "World"
3 }
4 val f2 = Future.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)
```

filter

```
1 val future1 = Future.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 _: NoSuchElementException => 0
8 }
9
10 val result2 = Await.result(failedFilter, 1 second)
11 result2 must be(0)
```

Await.result()

```
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.

For-comprehension

```
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)
```

Блокировки

Уже лучше, но усложняется с ростом числа акторов.

Комбинаторы

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]`

Базовые

```
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)
```

Порядок

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 }
```

Полезное

`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 }
```

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 }
```

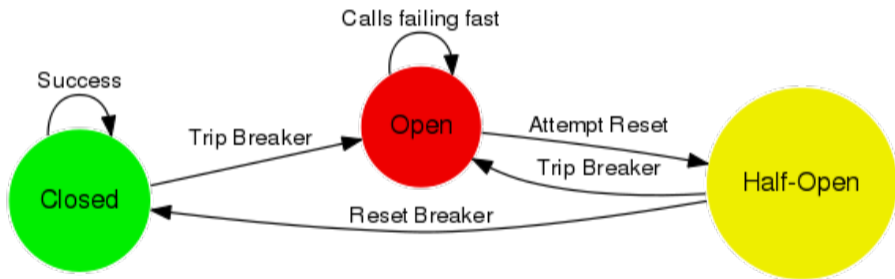
Или так:

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

After

```
1 val delayed =
2   akka.pattern.after(
3     200 millis,
4     using = system.scheduler)(
5     Future.failed(
6       new IllegalStateException("OHNOES")))
7 val future = Future { Thread.sleep(1000); "foo" }
8 val result = Future firstCompletedOf Seq(future, delayed)
```

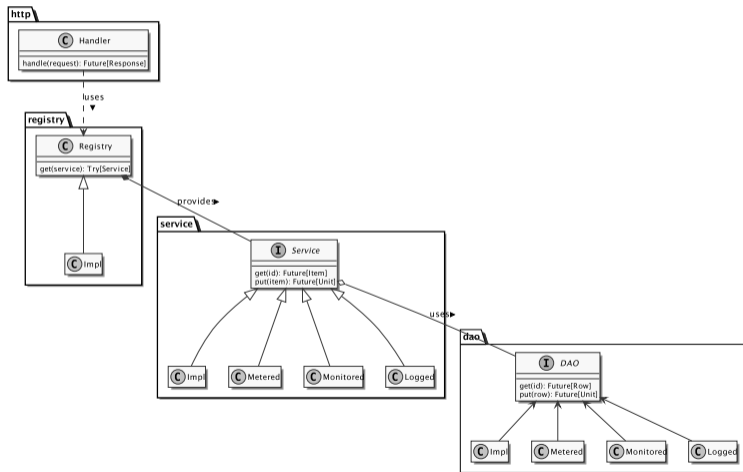
Circuit Breaker: States



Circuit Breaker: Code

```
1 class DangerousActor extends Actor with ActorLogging {
2   import context.dispatcher
3
4   val breaker =
5     new CircuitBreaker(
6       context.system.scheduler,
7       maxFailures = 5,
8       callTimeout = 10.seconds,
9       resetTimeout = 1.minute)
10      .onOpen(notifyMeOnOpen())
11
12   def notifyMeOnOpen(): Unit =
13     log.warning(
14       "My CircuitBreaker is now open, and will not close for one minute")
```

Типичный CRUD Web-сервис



Внимание

Проблема

Можно получить OOM из-за неограниченного накопления Future (FJP).

Решение

- Throttling на верхнем уровне
- Но легко пропустить callback и перестать отвечать на запросы
- Покрывайте 100% кода тестами

Reactive Streams

*Ever-newer waters flow on those
who step into the same rivers.*

Heraclitus

Потоки данных¹⁶:

- Real-time источники
- Массовое копирование
- Batch-обработка больших объёмов
- Мониторинг/аналитика

¹⁶Roland Kuhn. Reactive Streams: Handling Data-Flows the Reactive Way:
<http://www.infoq.com/presentations/reactive-streams-akka>

Проблема

Back Pressure

Consumer не справляется с потоком данных от producer.

Демаркация:

- Приложения
- Узлы
- Процессоры
- Потоки
- Акторы
- ...

Традиционные решения

- Блокирующие вызовы
- Push: буферизация и/или отбрасывание
- **Reactive way**¹⁷

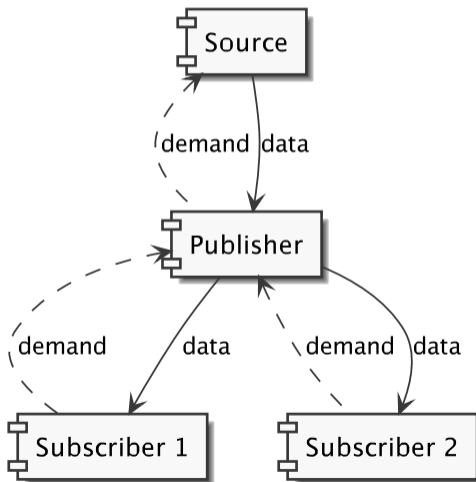
¹⁷<http://www.reactive-streams.org>

Основная идея

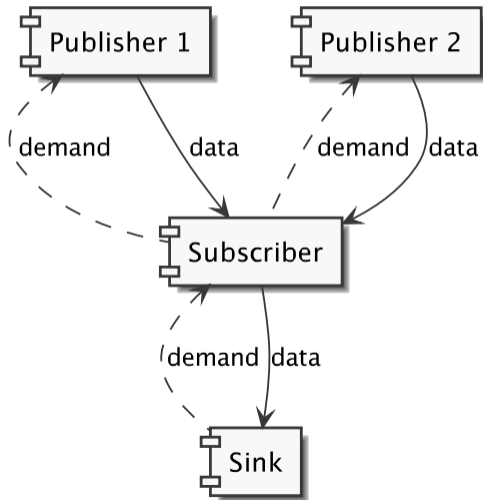


- Получатель контролирует скорость потока данных
- Потребление памяти зависит от потребности
- Consumer быстрее — push
- Producer быстрее — pull
- Все вызовы неблокирующие

Split Data = Merge Demand



Merge Data = Split Demand



The Reactive Streams Project

Консорциум¹⁸:

- Netflix
- Twitter
- Red Hat
- Pivotal
- Typesafe

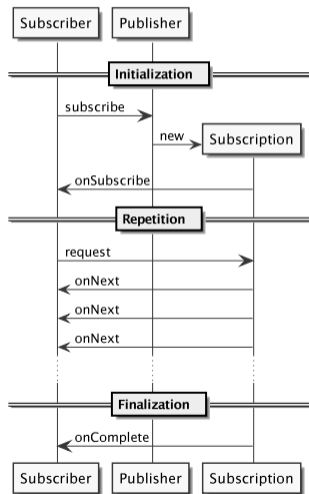
Секрет успеха

- Минимальные интерфейсы
- Строгая спецификация семантики
- ТСК
- Свобода для идиоматичных API

API

```
1 public interface Publisher<T> {
2     public void subscribe(Subscriber<? super T> s);
3 }
4
5 public interface Subscriber<T> {
6     public void onSubscribe(Subscription s);
7     public void onNext(T t);
8     public void onError(Throwable t);
9     public void onComplete();
10 }
11
12 public interface Subscription {
13     public void request(long n);
14     public void cancel();
15 }
```

Sequence



Реализации

- RxJava
- Project Reactor
- Vert.x
- **Akka Streams**
- **Slick**
- **Java 9**
- etc.¹⁹

¹⁹https://en.wikipedia.org/wiki/Reactive_Streams

Ссылки

- `http://www.reactive-streams.org`
- `https://github.com/reactive-streams/reactive-streams-jvm`
- `https://github.com/typesafehub/activator-akka-stream-scala`
- JSR 166²⁰ in Java 9 by Doug Lea

²⁰<http://gee.cs.oswego.edu/dl/jsr166/dist/docs/java/util/concurrent/Flow.html>

Conclusion

- Параллельные программы — это неизбежно
- Нужны более простые способы создания **корректных параллельных** программных систем
- Низкоуровневый параллелизм — это чрезвычайно сложно
- Но есть альтернативы²¹:
 - Message-Passing Concurrency (Actor Model)
 - Futures
 - Reactive Streams
- No Silver Bullet

²¹ScalaDays 2013. Concurrency — The good, the bad, the ugly:
<http://www.parleys.com/play/51c0bc58e4b0ed877035680a/>

Куда двигаться дальше

- <http://www.reactivemanifesto.org>
- Jamie Allen. *Effective Akka*. 2013
- Derek Wyatt. *Akka Concurrency*. 2013
- Series: The Neophyte's Guide to Scala²²
- EAI Patterns Series²³
- Chris Okasaki. *Purely Functional Data Structures*. 1999
- JCIP 2nd edition + JMM
- Transactional Memory and Beyond²⁴

²²<http://letitcrash.com/post/64667109914/series-the-neophytes-guide-to-scala>

²³<http://letitcrash.com/post/59190266995/eai-patterns-series-by-vaughnvernon>

²⁴<https://jug.ru/talks/meetups/transactional-memory-and-beyond/>

Вопросы?

- <https://incubos.org/blog/>
- <https://twitter.com/incubos>
- mail@incubos.org