# Scala School

## Лекция 9: Akka

# План лекции

- Что такое акторы?
- Зачем?
- Примеры использования
- Основные принципы и термины
- Гарантии Akka
- Почтовые ящики
- Диспетчеры
- Маршрутизация
- Критика
- Мониторинг. Kamon

# Акторы

В компьютерных науках модель акторов представляет собой математическую модель параллельных вычислений, которая трактует понятие «актор» как универсальный примитив параллельного численного расчёта: в ответ на получаемые сообщения актор может принимать локальные решения, создавать новые акторы, посылать свои сообщения, а также устанавливать, как следует реагировать на последующие сообщения. Модель акторов возникла в 1973 году, использовалась как основа для понимания исчисления процессов и как теоретическая база для ряда практических реализаций параллельных систем.

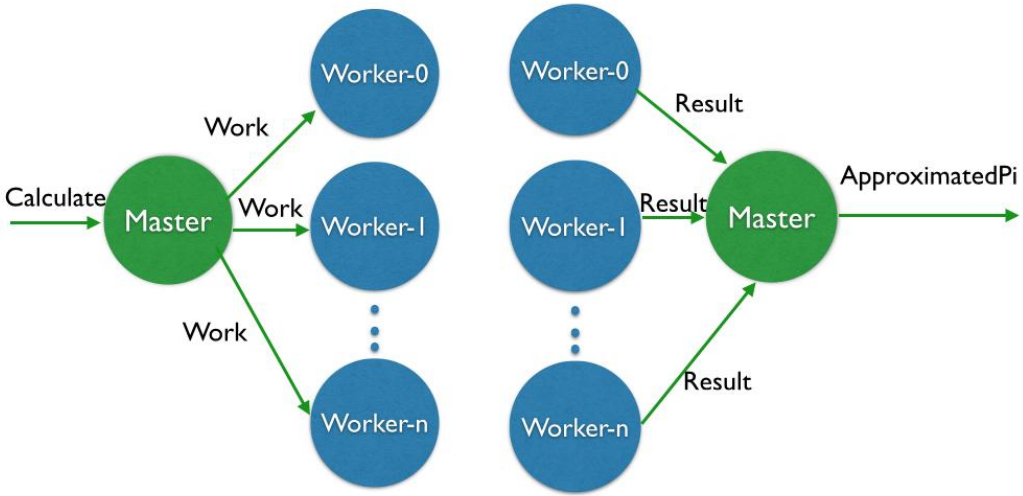
# Акторы

Актор - это сущность внутри системы акторов. Акторы могут создавать других акторов, посылать и получать сообщения, изменить свое поведение при обработке следующего сообщения. Акторы - это некие контейнеры логики обработки сообщений и внутреннего состояния.

# Зачем?

- Удобно при ассинхронной обработки сообщений (сетевое взаимодействие, event-based приложения)
- Удобно для параллельных вычислений (в теории)
- Удобно для создания распределенных приложений (в теории)
- Удобно, когда ложится на решаемую задачу
- Удобно для синхронизации внутреннего состояния (никаких больше блокировок, семафоров, race condition (внутри акторов!))

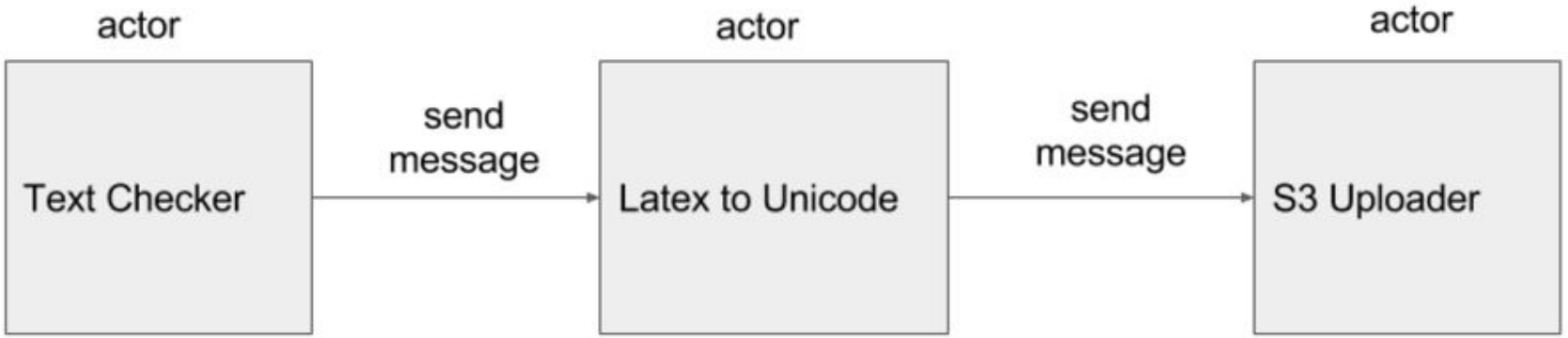
# Как это обычно работает



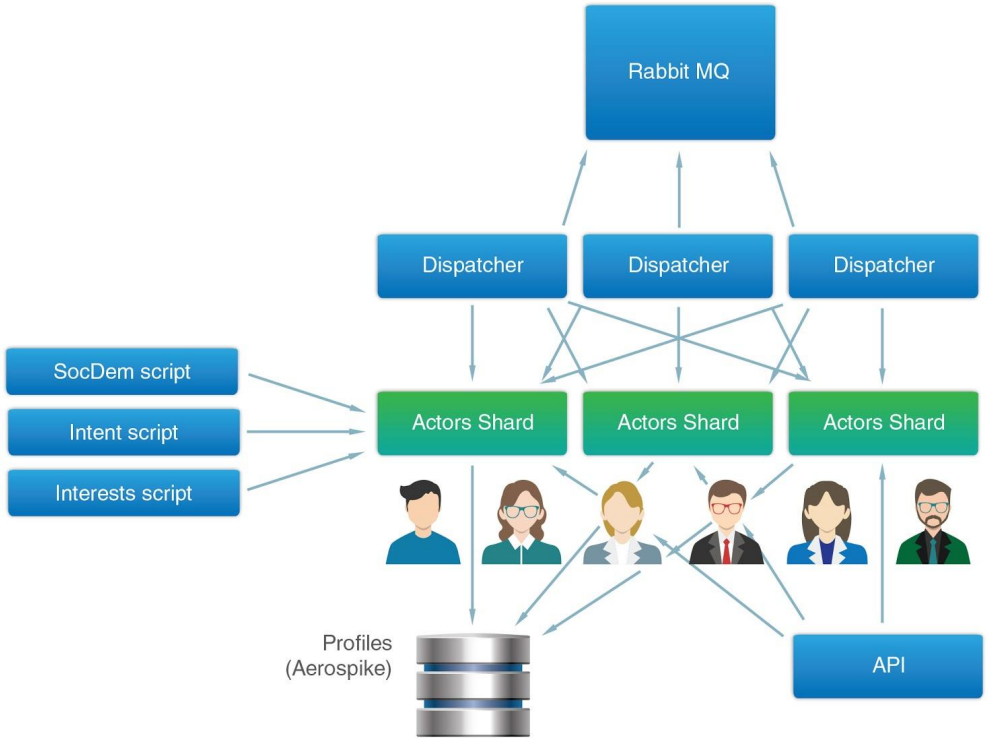

Computation starts…

Computation ends…

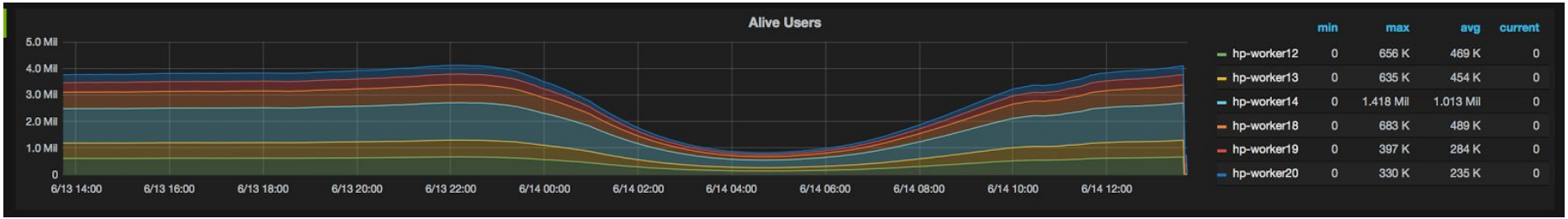
# Как это обычно работает



# Как это обычно работает



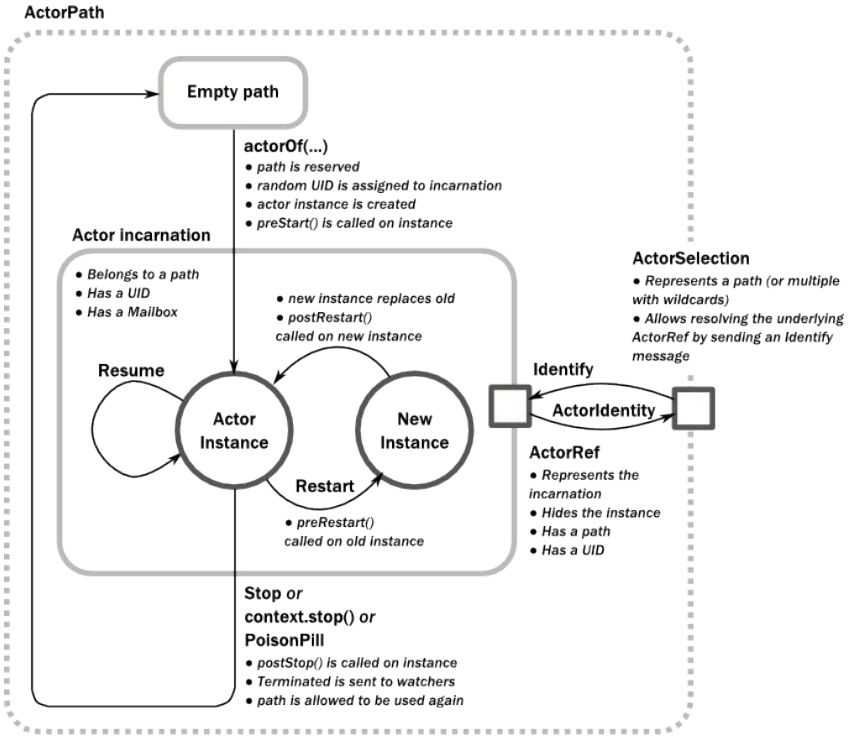
# Как это обычно работает



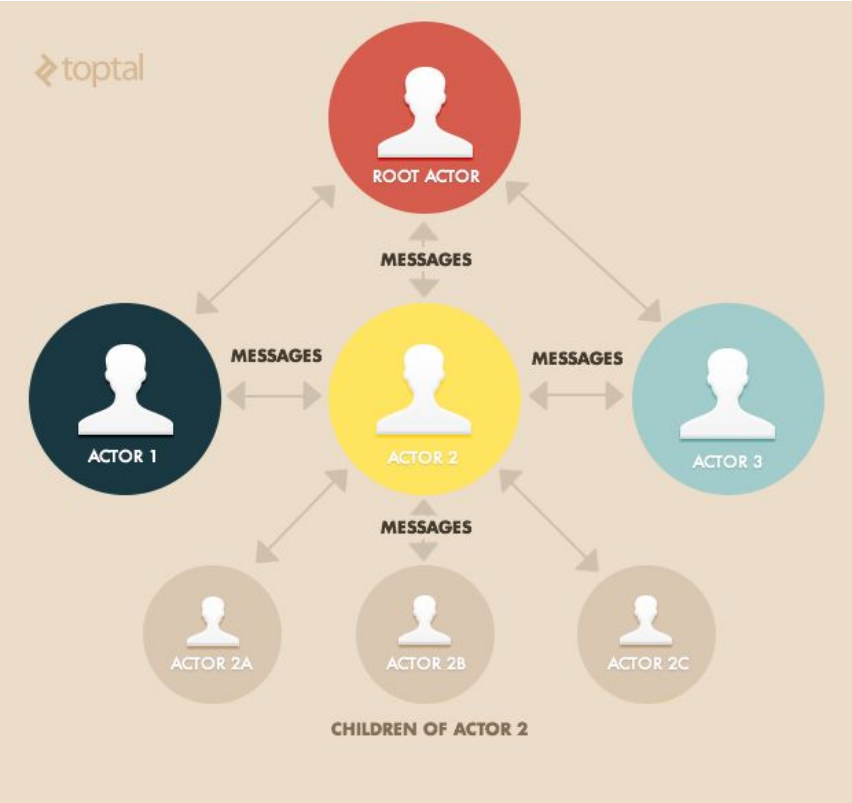
# Основные принципы

- Жизненный цикл
- Иерархия
- Supervision Strategy. Let it crash
- Location Transparency
- Иммутабельность
- Храним состояния внутри и пользуемся в том же потоке
- Все взаимодействия между акторами только через Akka API

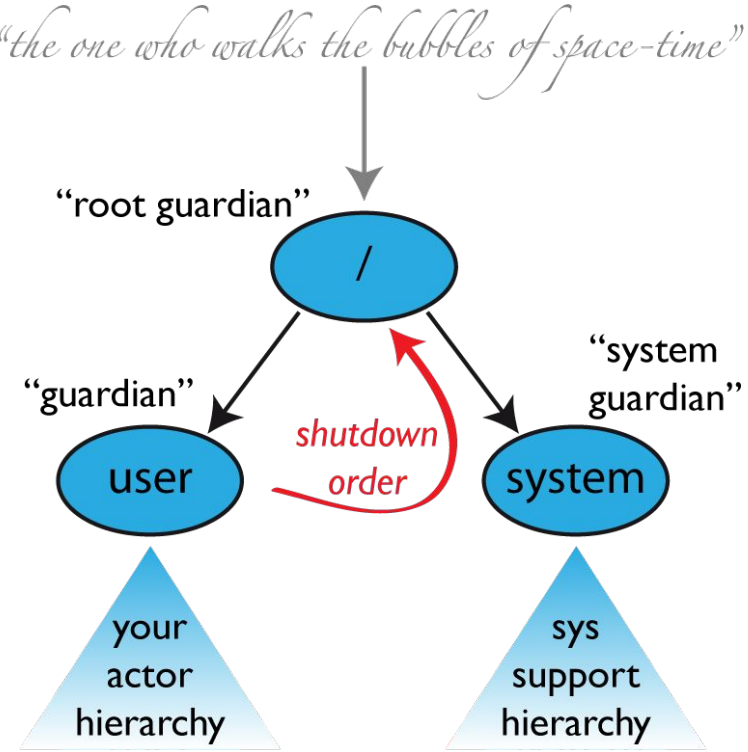
# Жизненный цикл



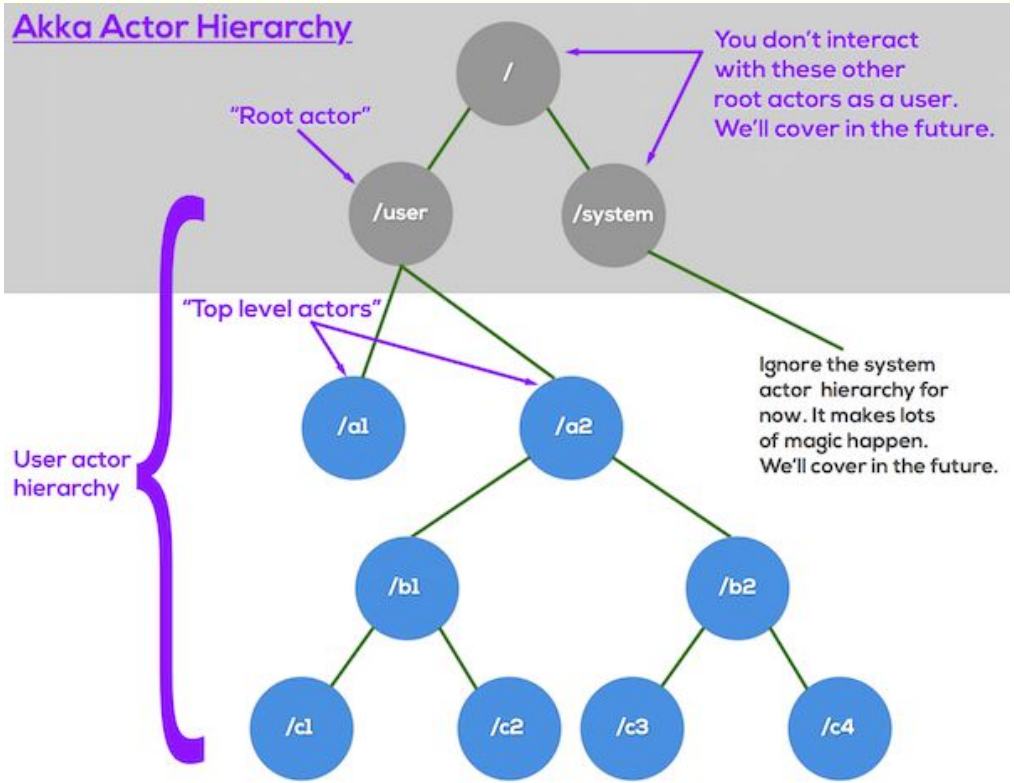
# Иерархия


toptal
ROOT ACTOR
MESSAGES
MESSAGES
MESSAGES
ACTOR 1
ACTOR 2
ACTOR 3
MESSAGES
ACTOR 2A
ACTOR 2B
ACTOR 2C
CHILDREN OF ACTOR 2

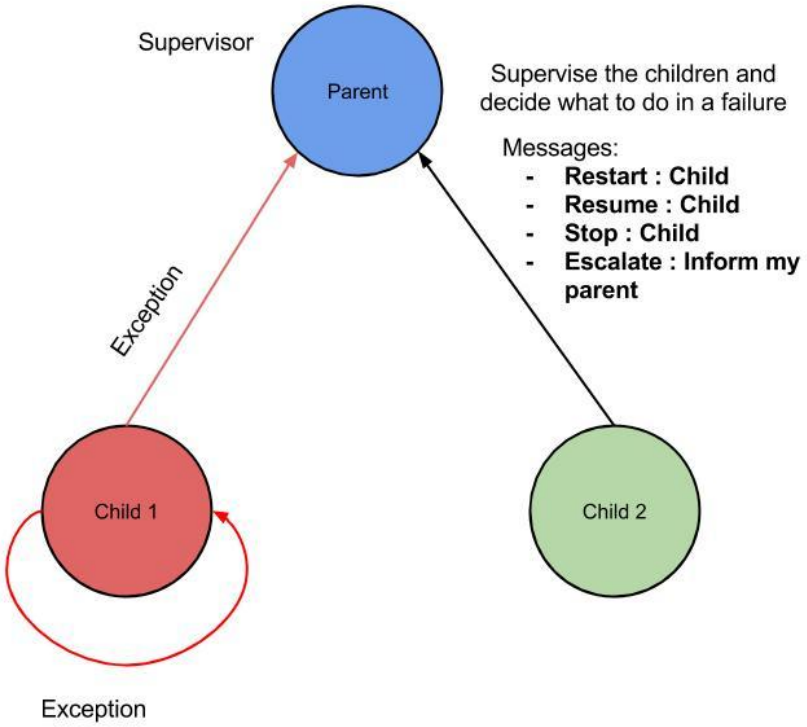
# Иерархия


"the one who walks the bubbles of space-time"
"root guardian"
/
"guardian"
"system guardian"
shutdown order
user
system
your actor hierarchy
sys support hierarchy

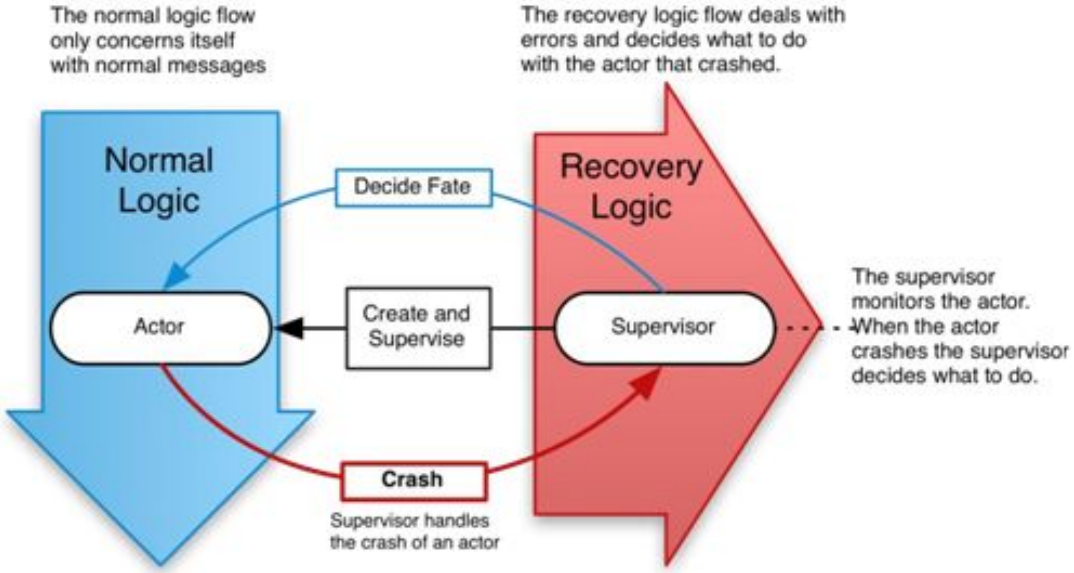
# Иерархия


Akka Actor Hierarchy
/
"Root actor"
You don't interact with these other root actors as a user. We'll cover in the future.
/user
/system
"Top level actors"
Ignore the system actor hierarchy for now. It makes lots of magic happen. We'll cover in the future.
User actor hierarchy
/a1
/a2
/b1
/b2
/c1
/c2
/c3
/c4

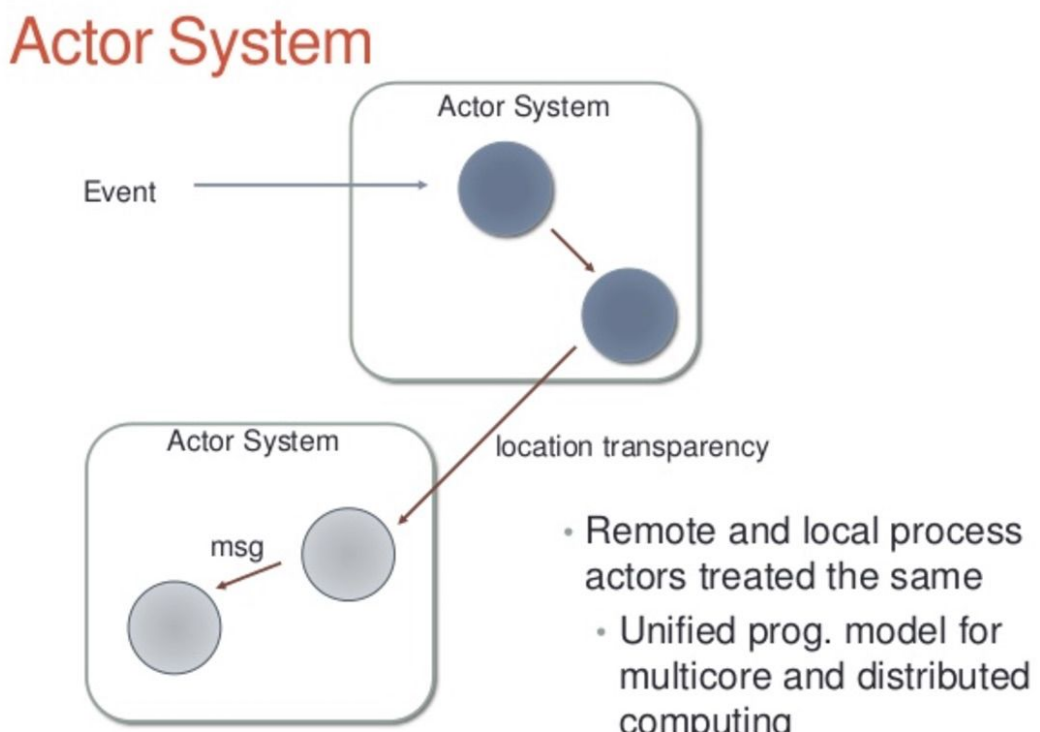
# Supervision Strategy



# Let it crash


The normal logic flow only concerns itself with normal messages
The recovery logic flow deals with errors and decides what to do with the actor that crashed.
Normal Logic
Recovery Logic
Decide Fate
Actor
Create and Supervise
Supervisor
The supervisor monitors the actor. When the actor crashes the supervisor decides what to do.
Crash
Supervisor handles the crash of an actor
The reason for the crash is sent to the supervisor.
The supervisor decides what to do with the crashed actor.
The flows are orthogonal to each other, the recovery flow does not have to be opposite to the normal flow as with exception handling.

# Location Transparency




- Remote and local process actors treated the same
  - Unified prog. model for multicore and distributed computing

# Location Transparency

# Иммутабельность

## Immutable messages

```
// define the case class
case class Register(user: User)

// create and send a new case class message
actor ! Register(user)

// tuples
actor ! (username, password)

// lists
actor ! List(“bill”, “bob”, “alice”)
```

# Храним состояния внутри и пользуемся в том же потоке

# Все взаимодействия между акторами только через Akka API

# Deadlock vs. Starvation vs. Livelock

Deadlock - ничего не происходит, все ждут

Starvation - когда какая-то работа в системе происходит, но она полностью блокирует работу другой части программы

Livelock - как Deadlock, все делают какой-то прогресс, но оно не приводит к нужным результатам

# Основные термины

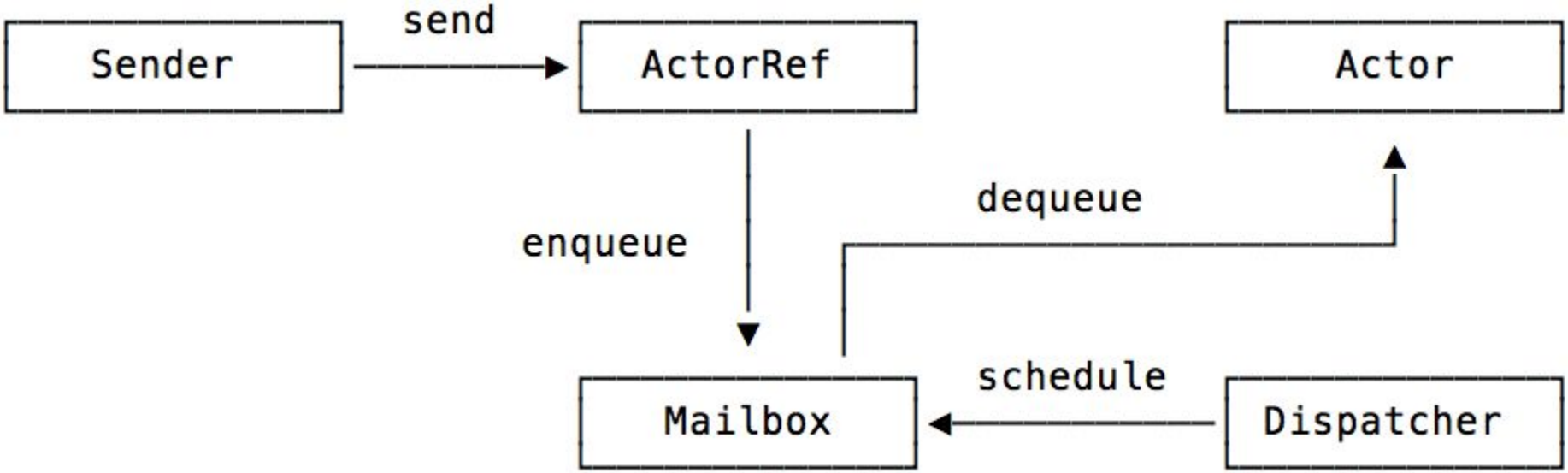
Ссылки, пути и адреса

Почтовые ящики

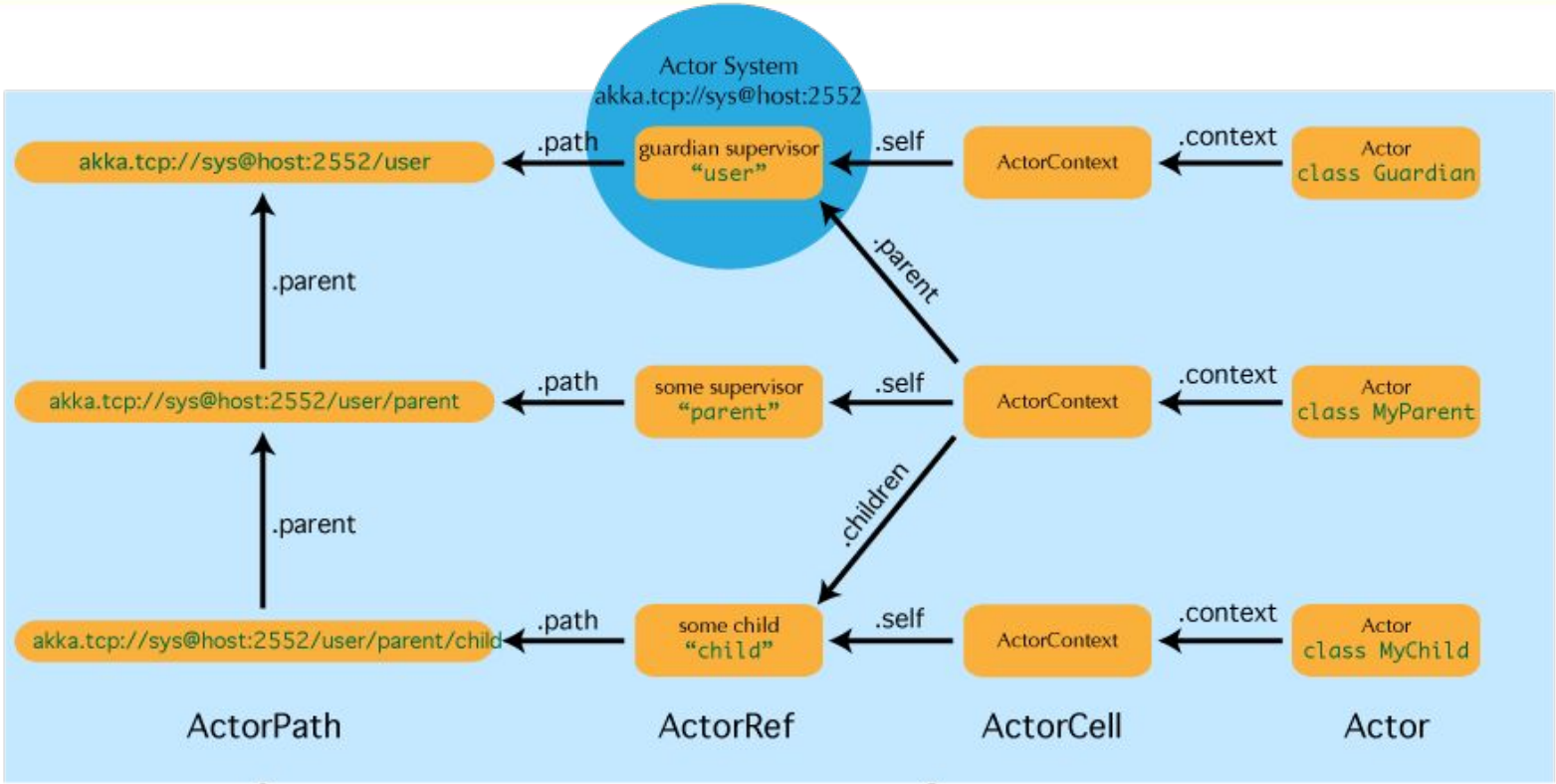
Диспетчеры

Маршрутизация

# Основные термины



# Ссылки, пути и адреса



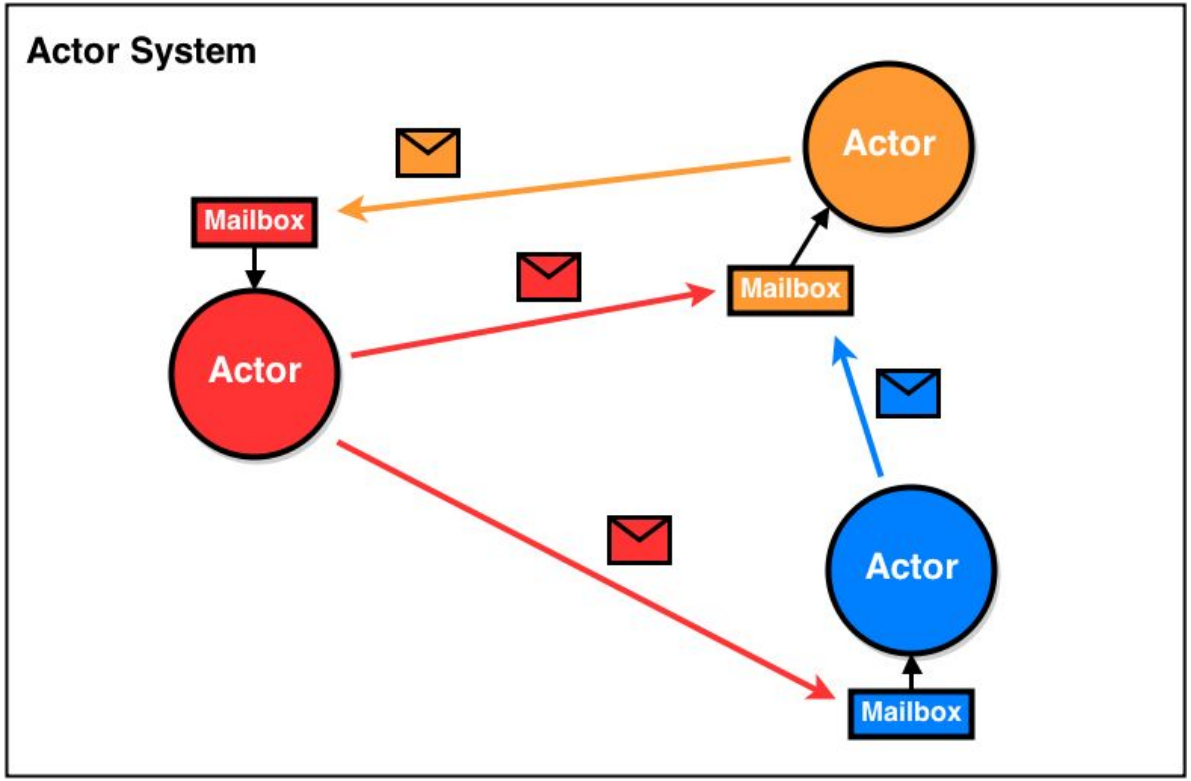
# Ссылки, пути и адреса

- Объекты класса ActorRef - для отправки сообщений актору. Соответствует одному живому актору, после смерти объект неактуален.
- Объекты класса ActorPath - для попытки поиска актора по абсолютному или относительному пути с помощью Identity. Могут быть абсолютными и относительными в иерархии.
- Address - строчное представление пути до актора.

# ActorRef

- метод tell  (!) - отправить
- метод forward - переслать. sender сообщения не меняется

# Почтовые ящики



# Почтовые ящики

- UnboundedMailbox (default)
  - The default mailbox
  - Backed by a java.util.concurrent.ConcurrentLinkedQueue
  - Blocking: No
  - Bounded: No
  - Configuration name: "unbounded" or "akka.dispatch.UnboundedMailbox"

- SingleConsumerOnlyUnboundedMailbox
  This queue may or may not be faster than the default one depending on your use-case—be sure to benchmark properly!
  - Backed by a Multiple-Producer Single-Consumer queue, cannot be used with BalancingDispatcher
  - Blocking: No
  - Bounded: No
  - Configuration name: "akka.dispatch.SingleConsumerOnlyUnboundedMailbox"

# Почтовые ящики

- NonBlockingBoundedMailbox
  - Backed by a very efficient Multiple-Producer Single-Consumer queue
  - Blocking: No (discards overflowing messages into deadLetters)
  - Bounded: Yes
  - Configuration name: "akka.dispatch.NonBlockingBoundedMailbox"

- UnboundedControlAwareMailbox
  - Delivers messages that extend akka.dispatch.ControlMessage with higher priority
  - Backed by two java.util.concurrent.ConcurrentLinkedQueue
  - Blocking: No
  - Bounded: No
  - Configuration name: "akka.dispatch.UnboundedControlAwareMailbox"

# Почтовые ящики

- UnboundedPriorityMailbox
  - Backed by a java.util.concurrent.PriorityBlockingQueue
  - Delivery order for messages of equal priority is undefined - contrast with the UnboundedStablePriorityMailbox
  - Blocking: No
  - Bounded: No
  - Configuration name: "akka.dispatch.UnboundedPriorityMailbox"

- UnboundedStablePriorityMailbox
  - Backed by a java.util.concurrent.PriorityBlockingQueue wrapped in an akka.util.PriorityQueueStabilizer
  - FIFO order is preserved for messages of equal priority - contrast with the UnboundedPriorityMailbox
  - Blocking: No
  - Bounded: No
  - Configuration name: "akka.dispatch.UnboundedStablePriorityMailbox"

# Почтовые ящики

- BoundedMailbox
  - Backed by a java.util.concurrent.LinkedBlockingQueue
  - Blocking: Yes if used with non-zero mailbox-push-timeout-time, otherwise No
  - Bounded: Yes
  - Configuration name: "bounded" or "akka.dispatch.BoundedMailbox"

- BoundedPriorityMailbox
  - Backed by a java.util.PriorityQueue wrapped in an akka.util.BoundedBlockingQueue
  - Delivery order for messages of equal priority is undefined - contrast with the BoundedStablePriorityMailbox
  - Blocking: Yes if used with non-zero mailbox-push-timeout-time, otherwise No
  - Bounded: Yes
  - Configuration name: "akka.dispatch.BoundedPriorityMailbox"

# Почтовые ящики

- BoundedStablePriorityMailbox
  - Backed by a java.util.PriorityQueue wrapped in an akka.util.PriorityQueueStabilizer and an akka.util.BoundedBlockingQueue
  - FIFO order is preserved for messages of equal priority - contrast with the BoundedPriorityMailbox
  - Blocking: Yes if used with non-zero mailbox-push-timeout-time, otherwise No
  - Bounded: Yes
  - Configuration name: "akka.dispatch.BoundedStablePriorityMailbox"

- BoundedControlAwareMailbox
  - Delivers messages that extend akka.dispatch.ControlMessage with higher priority
  - Backed by two java.util.concurrent.ConcurrentLinkedQueue and blocking on enqueue if capacity has been reached
  - Blocking: Yes if used with non-zero mailbox-push-timeout-time, otherwise No
  - Bounded: Yes
  - Configuration name: "akka.dispatch.BoundedControlAwareMailbox"

# Диспетчеры

Dispatchers >= ExecutionContext

Определяет то, на каком пуле потоков и по каким правилам акторы обрабатывают сообщения.

# Диспетчеры

- Dispatcher
  - This is an event-based dispatcher that binds a set of Actors to a thread pool. It is the default dispatcher used if one is not specified.
  - Sharability: Unlimited
  - Mailboxes: Any, creates one per Actor
  - Use cases: Default dispatcher, Bulkheading
  - Driven by: java.util.concurrent.ExecutorService
  - specify using "executor" using "fork-join-executor", "thread-pool-executor" or the FQCN of an akka.dispatcher.ExecutorServiceConfigurator

# Диспетчеры

PinnedDispatcher

- This dispatcher dedicates a unique thread for each actor using it; i.e. each actor will have its own thread pool with only one thread in the pool.
- Sharability: None
- Mailboxes: Any, creates one per Actor
- Use cases: Bulkheading
- Driven by: Any akka.dispatch.ThreadPoolExecutorConfigurator
- by default a "thread-pool-executor"

# Диспетчеры

BalancingDispatcher

- This is an executor based event driven dispatcher that will try to redistribute work from busy actors to idle actors.
- All the actors share a single Mailbox that they get their messages from.
- It is assumed that all actors using the same instance of this dispatcher can process all messages that have been sent to one of the actors; i.e. the actors belong to a pool of actors, and to the client there is no guarantee about which actor instance actually processes a given message.
- Sharability: Actors of the same type only
- Mailboxes: Any, creates one for all Actors
- Use cases: Work-sharing Driven by: java.util.concurrent.ExecutorService
- specify using "executor" using "fork-join-executor", "thread-pool-executor" or the FQCN of an akka.dispatcher.ExecutorServiceConfigurator
- Note that you can not use a BalancingDispatcher as a Router Dispatcher. (You can however use it for the Routees)

# Диспетчеры

- CallingThreadDispatcher
  - This dispatcher runs invocations on the current thread only. This dispatcher does not create any new threads, but it can be used from different threads concurrently for the same actor. See CallingThreadDispatcher for details and restrictions.
  - Sharability: Unlimited
  - Mailboxes: Any, creates one per Actor per Thread (on demand)
  - Use cases: Testing
  - Driven by: The calling thread (duh)

# Маршрутизация

Route - маршрут, определяет какому актору отправить сообщение.

- akka.routing.RoundRobinRoutingLogic
- akka.routing.RandomRoutingLogic
- akka.routing.SmallestMailboxRoutingLogic
- akka.routing.BroadcastRoutingLogic
- akka.routing.ScatterGatherFirstCompletedRoutingLogic
- akka.routing.TailChoppingRoutingLogic
- akka.routing.ConsistentHashingRoutingLogic

# Маршрутизация

Есть два вида Route:

- Pool - маршрутизатор сам создает дочерные акторы и следит за их завершением.
- Group - маршрутизатор шлет сообщения по переданным ему ActorPath и не следит сам за их завершением.

# Гарантии Akka

- Сообщение обрабатывается только после завершения отправки
- Следующее сообщение обрабатывается только после завершения обработки предыдущего
- Сообщения от одного адресата приходят в порядке отправления на одной машине

# Примеры

```
object MyActor {
  case class Greeting(from: String)
  case object Goodbye
}
class MyActor extends Actor with ActorLogging {
  import MyActor._
  def receive: Receive = {
    case Greeting(greeter) => log.info(s"I was greeted by $greeter.")
    case Goodbye           => log.info("Someone said goodbye to me.")
  }
}
```

# Примеры

```
import akka.actor.ActorSystem

// ActorSystem is a heavy object: create only one per application
val system = ActorSystem("mySystem")
val myActor = system.actorOf(Props[MyActor], "myactor")
```

# Примеры

```
case request =>
  val result = process(request)
  sender() ! result
```

# Примеры

```
class HotSwapActor extends Actor {
  import context._
  def angry: Receive = {
    case "foo" => sender() ! "I am already angry?"
    case "bar" => become(happy)
  }

  def happy: Receive = {
    case "bar" => sender() ! "I am already happy :-)"
    case "foo" => become(angry)
  }

  def receive = {
    case "foo" => become(angry)
    case "bar" => become(happy)
  }
}
```

# Внутри

```
object Actor
  type Receive = PartialFunction[Any, Unit]
  object emptyBehavior extends Receive {
    def isDefinedAt(x: Any) = false
    def apply(x: Any) = throw new UnsupportedOperationException("Empty behavior apply()")
  }
  object ignoringBehavior extends Receive {
    def isDefinedAt(x: Any): Boolean = true
    def apply(x: Any): Unit = ()
  }
  final val noSender: ActorRef = null

  ...
}
```

# Внутри

```
trait Actor {

  type Receive = Actor.Receive

  implicit val context: ActorContext = ...

  implicit final val self = context.self //MUST BE A VAL, TRUST ME

  final def sender(): ActorRef = context.sender()

  def receive: Actor.Receive
```

# Внутри

```
def supervisorStrategy: SupervisorStrategy = SupervisorStrategy.defaultStrategy

def preStart(): Unit = ()

def postStop(): Unit = ()

def preRestart(reason: Throwable, message: Option[Any]): Unit = {
  context.children foreach { child ⇒
    context.unwatch(child)
    context.stop(child)
  }
  postStop()
}
```

# Внутри

```
def postRestart(reason: Throwable): Unit = {
    preStart()
  }

  def unhandled(message: Any): Unit = {
    message match {
      case Terminated(dead) ⇒ throw new DeathPactException(dead)
      case _                ⇒ context.system.eventStream.publish(UnhandledMessage(message, sender(), self))
    }
  }
```

# Внутри

```
trait ActorContext extends ActorRefFactory {
  def self: ActorRef
  def props: Props
  def receiveTimeout: Duration
  def setReceiveTimeout(timeout: Duration): Unit
  def become(behavior: Actor.Receive): Unit = become(behavior, discardOld = true)
  def become(behavior: Actor.Receive, discardOld: Boolean): Unit
  def unbecome(): Unit
```

# Внутри

```
  def sender(): ActorRef
  def children: immutable.Iterable[ActorRef]
  def child(name: String): Option[ActorRef]
  implicit def dispatcher: ExecutionContextExecutor
  implicit def system: ActorSystem
  def parent: ActorRef
  def watch(subject: ActorRef): ActorRef
  def unwatch(subject: ActorRef): ActorRef
}
```

# Внутри

```
abstract class ActorSystem extends ActorRefFactory {
  import ActorSystem._
  def name: String
  def settings: Settings
  def logConfiguration(): Unit
  def /(name: String): ActorPath
  def /(name: Iterable[String]): ActorPath
  val startTime: Long = System.currentTimeMillis
  def uptime: Long = (System.currentTimeMillis - startTime) / 1000
  def eventStream: EventStream
  def log: LoggingAdapter
  def deadLetters: ActorRef
  def scheduler: Scheduler
```

# Внутри

```
  def dispatchers: Dispatchers
  implicit def dispatcher: ExecutionContextExecutor
  def mailboxes: Mailboxes

  def registerOnTermination[T](code: ⇒ T): Unit

  def isTerminated: Boolean
  def terminate(): Future[Terminated]
  def whenTerminated: Future[Terminated]

  def registerExtension[T <: Extension](ext: ExtensionId[T]): T
  def extension[T <: Extension](ext: ExtensionId[T]): T
  def hasExtension(ext: ExtensionId[_ <: Extension]): Boolean
}
```

# Критика

- Не типизированы! (Akka typed)
- Сложно контролировать. Блокирующие операции требуют тщательного контроля.
- Под капотом достаточно магии
- Достаточно редко нужны на самом деле
- Профессионал знает исключения из правил

http://pchiusano.blogspot.ru/2010/01/actors-are-not-good-concurrency-model.html

http://pchiusano.blogspot.ru/2010/03/follow-up-to-actors-are-not-good.html

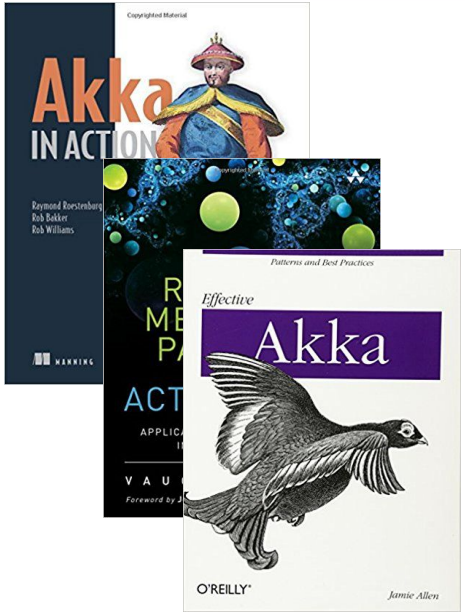
http://softwareengineering.stackexchange.com/questions/212754/when-is-it-not-good-to-use-actors-in-akka-erlang

# Материалы

- https://www.toptal.com/scala/concurrency-and-fault-tolerance-made-easy-an-intro-to-akka
- https://blog.codecentric.de/en/2015/08/introduction-to-akka-actors/
- http://doc.akka.io/docs/akka/current/scala.html

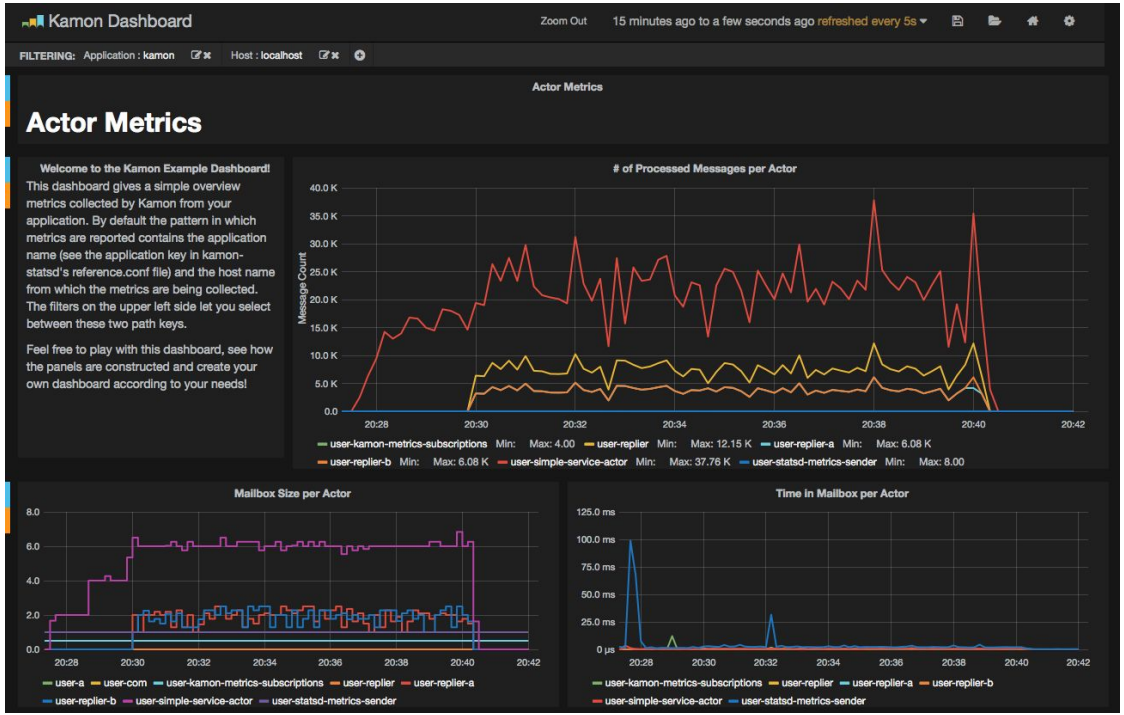
# Книги

- https://www.manning.com/books/akka-in-action
  - https://github.com/RayRoestenburg/akka-in-action
- https://www.packtpub.com/application-development/akka-cookbook
  - https://github.com/PacktPublishing/Akka-Cookbook
- https://www.amazon.com/Effective-Akka-Patterns-Best-Practices/dp/1449360076
  - https://github.com/amollenkopf/akka-effective
- https://www.amazon.com/Reactive-Messaging-Patterns-Actor-Model/dp/0133846830
  - https://github.com/VaughnVernon/ReactiveMessagingPatterns_ActorModel
- http://whatpixel.com/best-akka-books/



# Kamon

http://kamon.io/documentation/get-started/



# Спасибо