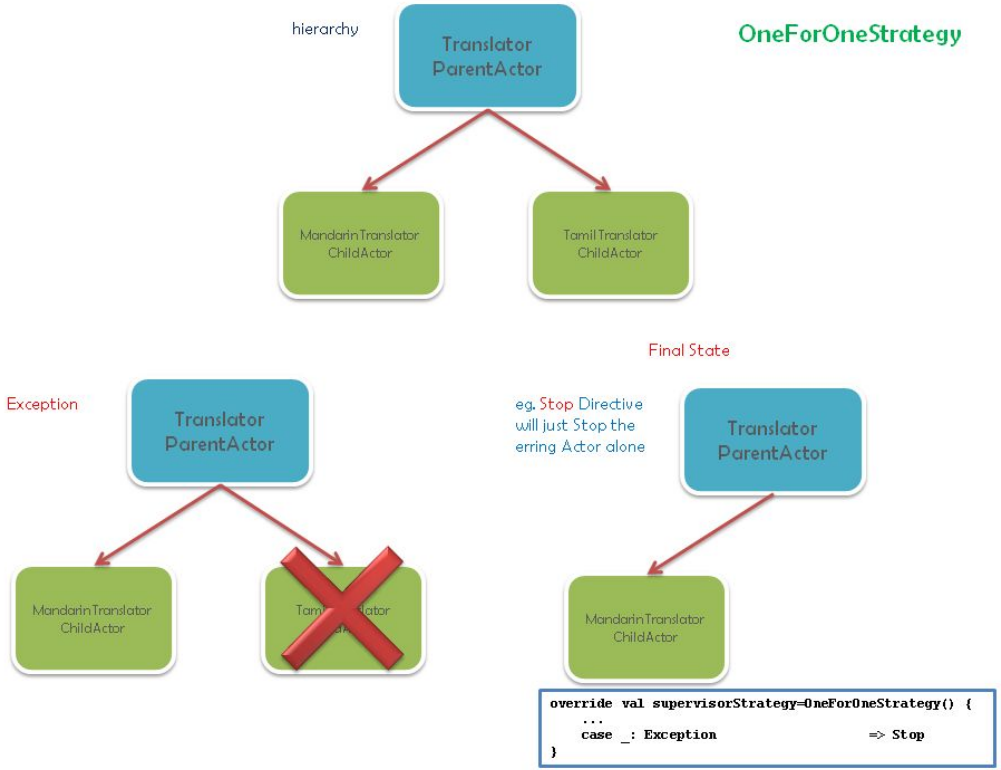
Scala School

Лекция 10: Akka

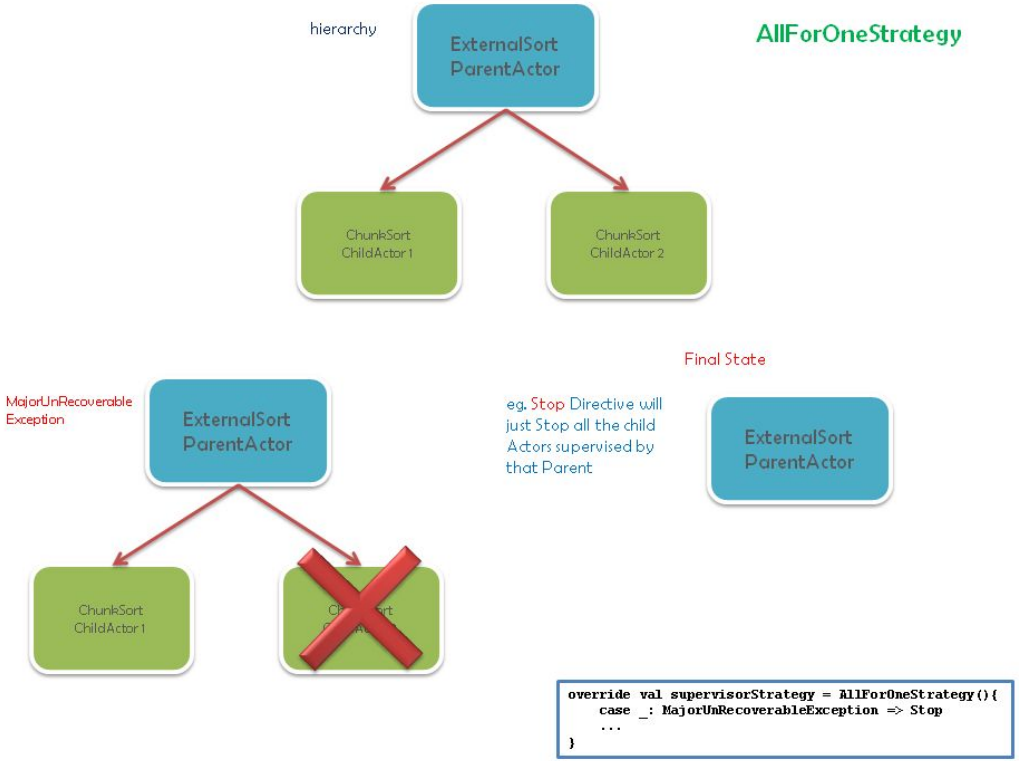
План лекции

- Supervision strategy: One-For-One Strategy vs. All-For-One Strategy
- Kill vs. Stop vs. Poison Pill
- Akka + Futures = Ask
- Event Bus
- Configuration
- Logging
- akka.pattern: Backoff, CircuitBreaker, etc
- TCP / UDP
- Remote
- Akka Extensions
- FSM
- Материалы

Supervision strategy: OFO vs. AFO



Supervision strategy: OFO vs. AFO



Kill vs. Stop vs. Poison Pill

- context.stop(self) останавливает актора прямо после обработки текущего сообщения
- сообщение PoisonPill останавливает актора, когда будет обработано
- сообщение Kill = ActorKilledException -> SupervisionStrategy. Есть шанс выжить

Event Bus

- Originally conceived as a way to send messages to groups of actors
- PubSub
- example: DeadLetters

Event Bus

```
import akka.actor.{Actor, ActorSystem, DeadLetter, PoisonPill, Props}
class Listener extends Actor {
  def receive: Receive = {
    case d: DeadLetter => println(d)
  }
}
object DeadLetterSubscriberApp extends App {
  val system = ActorSystem("mySystem")
  val listener = system.actorOf(Props(classOf[Listener]))
  system.eventStream.subscribe(listener, classOf[DeadLetter])
  val secondListener = system.actorOf(Props(classOf[Listener]))
  secondListener ! PoisonPill
  // it goes to dead letters
  secondListener ! "!11"
}
```

Configuration

- Typesafe Config Library
- Настройки определяются в ресурсах, в файле application.conf

Configuration

```
akka {
  loggers = ["akka.event.slf4j.Slf4jLogger"]
  loglevel = "DEBUG"
  logging-filter = "akka.event.slf4j.Slf4jLoggingFilter
  actor {
    provider = "cluster"
    default-dispatcher {
      # Throughput for default Dispatcher, set to 1 for as fair as possible
      throughput = 10
    }
  }
  remote {
    netty.tcp.port = 4711
  }
}
```

Configuration

```
myapp1 {
  akka.loglevel = "WARNING"
  my.own.setting = 43
}
myapp2 {
  akka.loglevel = "ERROR"
  app2.setting = "appname"
}
my.own.setting = 42
my.other.setting = "hello"
```

Configuration

- akka.loggers - Loggers to register at boot time
- akka.loglevel - Log level used by the configured loggers (see "loggers") as soon as they have been started; before that, see "stdout-loglevel". Options: OFF, ERROR, WARNING, INFO, DEBUG
- stdout-loglevel - Log level for the very basic logger activated during ActorSystem startup. This logger prints the log messages to stdout (System.out). Options: OFF, ERROR, WARNING, INFO, DEBUG
- akka.log-config-on-start - Log the complete configuration at INFO level when the actor system is started. This is useful when you are uncertain of what configuration is used.
- akka.actor.guardian-supervisor-strategy - The guardian "/user" will use this class to obtain its supervisorStrategy. It needs to be a subclass of akka.actor.SupervisorStrategyConfigurator.
- akka.actor.default-dispatcher
- akka.actor.default-dispatcher.throughput - Throughput defines the number of messages that are processed in a batch before the thread is returned to the pool. Set to 1 for as fair as possible.
- akka.actor.default-mailbox

Configuration

```
akka.actor.debug {
    # enable function of Actor.loggable(), which is to log any received messageat DEBUG level
    receive = off
    # enable DEBUG logging of all AutoReceiveMessages (Kill, PoisonPill et.c.)
    autoreceive = off
    # enable DEBUG logging of actor lifecycle changes
    lifecycle = off
    # enable DEBUG logging of all LoggingFSMs for events, transitions and timers
    fsm = off
    # enable DEBUG logging of subscription changes on the eventStream
    event-stream = off
    # enable DEBUG logging of unhandled messages
    unhandled = off
    # enable WARN logging of misconfigured routers
    router-misconfiguration = off
  }
```

Configuration

- akka.io
- akka.cluster
- akka.persistence
- akka.remote

Logging

```
import akka.event.Logging
class MyActor extends Actor {
  val log = Logging(context.system, this)
  override def preStart() = {
    log.debug("Starting")
  }
  override def preRestart(reason: Throwable, message: Option[Any]) {
    log.error(reason, "Restarting due to [{}] when processing [{}]",
      reason.getMessage, message.getOrElse(""))
  }
  def receive = {
    case "test" => log.info("Received test")
    case x      => log.warning("Received unknown message: {}", x)
  }
}
```

Logging

```
class MyActor extends Actor with akka.actor.ActorLogging {
  ...
}
```

Logging

```
akka {
  log-dead-letters = 10
  log-dead-letters-during-shutdown = on
}
```

Logging: SLF4J

build.sbt:

```
libraryDependencies += "ch.qos.logback" % "logback-classic" % "1.1.3"
```

application.conf:

```
akka {
  loggers = ["akka.event.slf4j.Slf4jLogger"]
  loglevel = "DEBUG"
  logging-filter = "akka.event.slf4j.Slf4jLoggingFilter"
}
```

akka.pattern

- ask
- pipe
- BackOff
- CircuitBreaker
- gracefulStop
- after

Akka + Futures = Ask

```
class AnswerActor extends Actor {
  override def receive: Receive = {
    case message => sender() ! message
  }
}
```

Akka + Futures = Ask

```
import akka.pattern.ask
object AskPattern extends App {
  val system = ActorSystem("mySystem")
  val actor = system.actorOf(Props[AnswerActor])
  implicit val timeout = Timeout(2.seconds)
  implicit val ec = ExecutionContext.global
  val f = for {
    answer <- actor ? "one"
    answerTwo <- actor ? "two"
    _ <- system.terminate()
  } yield {
    println(answer)
    println(answerTwo)
  }
  Await.result(f, Duration.Inf)
}
```

Akka + Futures = Ask

```
import akka.pattern.ask
object AskPattern extends App {
  val system = ActorSystem("mySystem")
  val actor = system.actorOf(Props[AnswerActor])
  implicit val timeout = Timeout(2.seconds)
  implicit val ec = ExecutionContext.global
  val f = for {
    answer <- actor ? "one"
    answerTwo <- (actor ? "two").mapTo[String]
    _ <- system.terminate()
  } yield {
    println(answer)
    println(answerTwo)
  }
  Await.result(f, Duration.Inf)
}
```

Akka + Futures = Ask (and pipe)

```
import akka.pattern.pipe
val f = Future.success(1)
f pipeTo actor
```

BackoffSupervisor

- BackoffSupervisor.props
- BackoffSupervisor.propsWithSupervisorStrategy
- Backoff.onFailure(

```
childProps:   Props,
childName:    String,
minBackoff:   FiniteDuration,
maxBackoff:   FiniteDuration,
randomFactor: Double): BackoffOptions
```

- Backoff.onStop

BackoffSupervisor

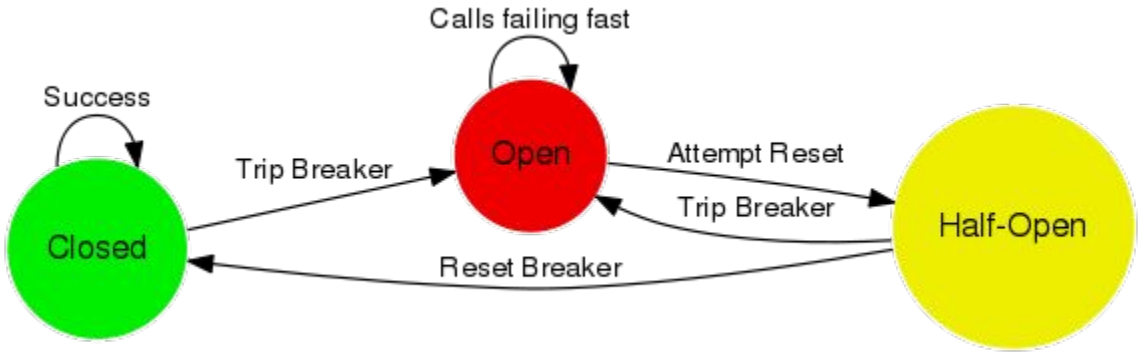
```
class StupidActor extends Actor with ActorLogging {
  override def receive: Receive = {
    case "oops" => throw new RuntimeException("oops")
  }

  override def preStart(): Unit = {
    log.info(s"prestart!")
  }
}
```

BackoffSupervisor

```
val actorSupervisorProps = BackoffSupervisor.props(
    Backoff.onFailure(
      Props[StupidActor],
      childName = "stupid",
      minBackoff = 1.seconds,
      maxBackoff = 30.seconds,
      randomFactor = 0.2
    ))
  val actorSupervisor = system.actorOf(actorSupervisorProps, name = "actorSupervisor")
  actorSupervisor ! "oops"
  actorSupervisor ! "oops"  // it goes to dead letters
  Thread.sleep(5000)
  actorSupervisor ! "oops"
  Thread.sleep(15000)
  actorSupervisor ! "oops"
```

CircuitBreaker



CircuitBreaker

- During normal operation, a circuit breaker is in the Closed state:
  - Exceptions or calls exceeding the configured callTimeout increment a failure counter
  - Successes reset the failure count to zero
  - When the failure counter reaches a maxFailures count, the breaker is tripped into Open state
- While in Open state:
  - All calls fail-fast with a CircuitBreakerOpenException
  - After the configured resetTimeout, the circuit breaker enters a Half-Open state
- In Half-Open state:
  - The first call attempted is allowed through without failing fast
  - All other calls fail-fast with an exception just as in Open state
  - If the first call succeeds, the breaker is reset back to Closed state and the resetTimeout is reset
  - If the first call fails, the breaker is tripped again into the Open state (as for exponential backoff circuit breaker, the resetTimeout is multiplied by the exponential backoff factor)
- State transition listeners:
  - Callbacks can be provided for every state entry via onOpen, onClose, and onHalfOpen
  - These are executed in the ExecutionContext provided.

CircuitBreaker

```
class DangerousActor extends Actor with ActorLogging {

  val breaker =
    new CircuitBreaker(
      context.system.scheduler,
      maxFailures = 5,
      callTimeout = 10.seconds,
      resetTimeout = 1.minute).onOpen(notifyMeOnOpen())

  def notifyMeOnOpen(): Unit =
    log.warning("My CircuitBreaker is now open, and will not close for one minute")
```

CircuitBreaker

```
def dangerousCall: String = "This really isn't that dangerous of a call after all"

def receive = {
  case "is my middle name" =>
    breaker.withCircuitBreaker(Future(dangerousCall)) pipeTo sender()
  case "block for me" =>
    sender() ! breaker.withSyncCircuitBreaker(dangerousCall)
}
```

gracefulStop

```
import akka.pattern.gracefulStop

Await.result(gracefulStop(actorSupervisor, 1 second), 1 second)
```

after

```
import akka.pattern.after

implicit val ec = ExecutionContext.global
val afterFuture = after(1 second, system.scheduler) {
   Future {
    Thread.sleep(300)
    1
   }
}

Await.result(afterFuture, 2 seconds)
```

TCP: Connecting

```
object Client {
  def props(remote: InetSocketAddress, replies: ActorRef) =
    Props(classOf[Client], remote, replies)
}

class Client(remote: InetSocketAddress, listener: ActorRef) extends Actor {

  import Tcp._
  import context.system
  IO(Tcp) ! Connect(remote)

  def receive = {
    case CommandFailed(_: Connect) =>
      listener ! "connect failed"
      context stop self
```

TCP: Connecting

```
case c @ Connected(remote, local) =>
   listener ! c
   val connection = sender()
   connection ! Register(self)
   context become {
     case data: ByteString =>
       connection ! Write(data)
     case CommandFailed(w: Write) =>
       // O/S buffer was full
       listener ! "write failed"
     case Received(data) =>
       listener ! data
     case "close" =>
       connection ! Close
     case _: ConnectionClosed =>
       listener ! "connection closed"
       context stop self}}}
```

TCP: Accepting connections

```
class Server extends Actor {
  import Tcp._
  import context.system
 IO(Tcp) ! Bind(self, new InetSocketAddress("localhost", 0))
  def receive = {
    case b @ Bound(localAddress) =>
      // do some logging or setup ...
    case CommandFailed(_: Bind) => context stop self
    case c @ Connected(remote, local) =>
      val handler = context.actorOf(Props[SimplisticHandler])
      val connection = sender()
      connection ! Register(handler)
  }
}
```

UDP: Sender

```
class SimpleSender(remote: InetSocketAddress) extends Actor {
  import context.system
  IO(Udp) ! Udp.SimpleSender
  def receive = {
    case Udp.SimpleSenderReady =>
      context.become(ready(sender()))
  }
  def ready(send: ActorRef): Receive = {
    case msg: String =>
      send ! Udp.Send(ByteString(msg), remote)
  }
}
```

Remote

build.sbt:
libraryDependencies += "com.typesafe.akka" %% "akka-remote" % "2.4.17"

application.conf:

```
akka {
  actor {
    provider = remote
  }
  remote {
    enabled-transports = ["akka.remote.netty.tcp"]
    netty.tcp {
      hostname = "127.0.0.1"
      port = 2552
    }
 }
}
```

Remote

build.sbt:
libraryDependencies += "com.typesafe.akka" %% "akka-remote" % "2.4.17"

application.conf:

```
akka {
  actor {
    provider = remote
  }
  remote {
    enabled-transports = ["akka.remote.netty.tcp"]
    netty.tcp {
      hostname = "127.0.0.1"
      port = 2552
    }
 }
}
```

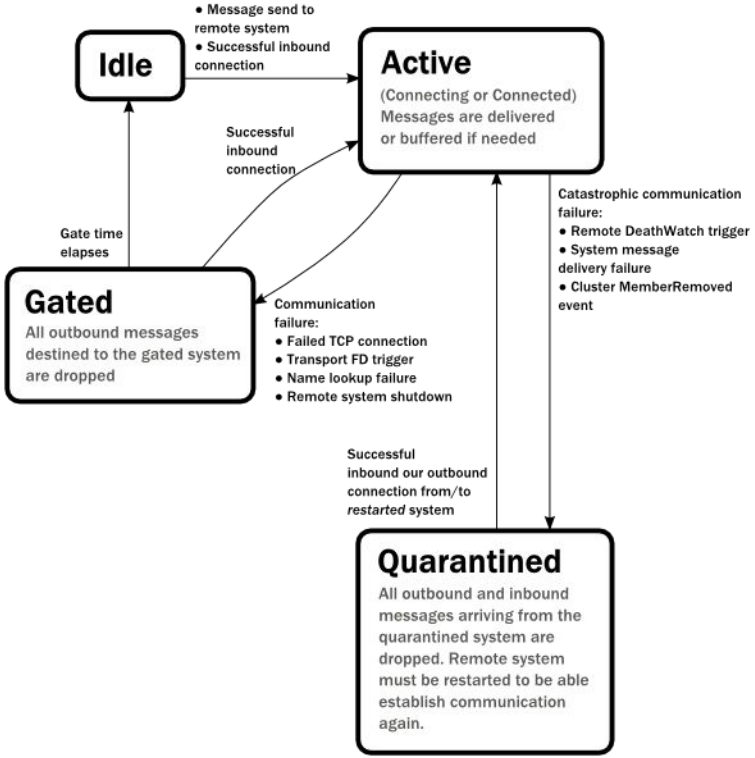
Remote

```
val selection =
  context.actorSelection("akka.tcp://actorSystemName@10.0.0.1:2552/user/actorName")
selection ! "Pretty awesome feature"
```

Remote

- Creating Actors Remotely
- Programmatic Remote Deployment
- Remote deployment whitelist
- Routers with Remote Destinations
- Remote Security (SSL)
- New Remoting (codename Artery, Aeron based)

Remoting Lifecycle and Failure Recovery Model



Artery

What is new in Artery
Artery is a reimplementation of the old remoting module aimed at improving performance and stability. It is mostly backwards compatible with the old implementation and it is a drop-in replacement in many cases. Main features of Artery compared to the previous implementation:

- Based on Aeron (UDP) instead of TCP
- Focused on high-throughput, low-latency communication
- Isolation of internal control messages from user messages improving stability and reducing false failure detection in case of heavy traffic by using a dedicated subchannel.
- Mostly allocation-free operation
- Support for a separate subchannel for large messages to avoid interference with smaller messages
- Compression of actor paths on the wire to reduce overhead for smaller messages
- Support for faster serialization/deserialization using ByteBuffers directly
- Built-in Flight-Recorder to help debugging implementation issues without polluting users logs with implementation specific events
- Providing protocol stability across major Akka versions to support rolling updates of large-scale systems

Akka Extensions

If you want to add features to Akka, there is a very elegant, but powerful mechanism for doing so. It's called Akka Extensions and is comprised of 2 basic components: an Extension and an ExtensionId.

Extensions will only be loaded once per ActorSystem, which will be managed by Akka. You can choose to have your Extension loaded on-demand or at ActorSystem creation time through the Akka configuration.

- Некий shared объект
- One per actor system
- Интеграция сторонних библиотек в вашу Actor System
    Внутри Akka многие библиотеки построены так

Akka Extensions

```
import akka.actor.Extension

class CountExtensionImpl extends Extension {
  //Since this Extension is a shared instance
  // per ActorSystem we need to be threadsafe
  private val counter = new AtomicLong(0)

  //This is the operation this Extension provides
  def increment() = counter.incrementAndGet()
}
```

Akka Extensions

```
object CountExtension
  extends ExtensionId[CountExtensionImpl]
  with ExtensionIdProvider {
  //The lookup method is required by ExtensionIdProvider, so we return ourselves here, this allows us
  // to configure our extension to be loaded when the ActorSystem starts up
  override def lookup = CountExtension

  //This method will be called by Akka
  // to instantiate our Extension
  override def createExtension(system: ExtendedActorSystem) = new CountExtensionImpl

  // Java API: retrieve the Count extension for the given system
  override def get(system: ActorSystem): CountExtensionImpl = super.get(system)
}
```

Akka Extensions

```
trait Counting { self: Actor =>
  def increment() = CountExtension(context.system).increment()
}
class MyCounterActor extends Actor with Counting {
  def receive = {
    case someMessage => increment()
  }
}
```

Akka Extensions

```
akka {
  extensions = ["docs.extension.CountExtension"]
}
```

```
akka.library-extensions += "docs.extension.ExampleExtension"
```

FSM (Детерминированный конечный автомат)

A FSM can be described as a set of relations of the form:

State(S) x Event(E) -> Actions (A), State(S')

These relations are interpreted as meaning:

If we are in state S and the event E occurs, we should perform the actions A and make a transition to the state S'.

FSM (Детерминированный конечный автомат)

```
// received events
final case class SetTarget(ref: ActorRef)
final case class Queue(obj: Any)
case object Flush

// sent events
final case class Batch(obj: immutable.Seq[Any])
```

FSM (Детерминированный конечный автомат)

```
// states
sealed trait State
case object Idle extends State
case object Active extends State

sealed trait Data
case object Uninitialized extends Data
final case class Todo(target: ActorRef, queue: immutable.Seq[Any]) extends Data
```

FSM (Детерминированный конечный автомат)

```
class Buncher extends FSM[State, Data] {
  startWith(Idle, Uninitialized)
  when(Idle) {
    case Event(SetTarget(ref), Uninitialized) =>
      stay using Todo(ref, Vector.empty)
  }
  // transition elided ...
  when(Active, stateTimeout = 1 second) {
    case Event(Flush | StateTimeout, t: Todo) =>
      goto(Idle) using t.copy(queue = Vector.empty)
  }
  // unhandled elided ...
  initialize()
}
```

FSM (Детерминированный конечный автомат)

```
whenUnhandled {
  // common code for both states
  case Event(Queue(obj), t @ Todo(_, v)) =>
    goto(Active) using t.copy(queue = v :+ obj)

  case Event(e, s) =>
    log.warning("received unhandled request {} in state {}/{}", e, stateName, s)
    stay
}
```

FSM (Детерминированный конечный автомат)

```
onTransition {
  case Active -> Idle =>
    stateData match {
      case Todo(ref, queue) => ref ! Batch(queue)
      case _                => // nothing to do
    }
}
```

FSM (Детерминированный конечный автомат)

- forMax(duration)

  This modifier sets a state timeout on the next state. This means that a timer is started which upon expiry sends a StateTimeout message to the FSM. This timer is canceled upon reception of any other message in the meantime; you can rely on the fact that the StateTimeout message will not be processed after an intervening message.

  This modifier can also be used to override any default timeout which is specified for the target state. If you want to cancel the default timeout, use Duration.Inf.

- using(data)

  This modifier replaces the old state data with the new data given. If you follow the advice *above*, this is the only place where internal state data are ever modified.

- replying(msg)

  This modifier sends a reply to the currently processed message and otherwise does not modify the state transition.

FSM (Детерминированный конечный автомат)

- forMax(duration)

  This modifier sets a state timeout on the next state. This means that a timer is started which upon expiry sends a StateTimeout message to the FSM. This timer is canceled upon reception of any other message in the meantime; you can rely on the fact that the StateTimeout message will not be processed after an intervening message.

  This modifier can also be used to override any default timeout which is specified for the target state. If you want to cancel the default timeout, use Duration.Inf.

- using(data)

  This modifier replaces the old state data with the new data given. If you follow the advice *above*, this is the only place where internal state data are ever modified.

- replying(msg)

  This modifier sends a reply to the currently processed message and otherwise does not modify the state transition.

Материалы

http://doc.akka.io/docs/akka/2.4/scala/howto.html

http://doc.akka.io/docs/akka/2.4/experimental/index.html

http://doc.akka.io/docs/akka/2.4/additional/books.html

http://rerun.me/2014/11/10/akka-notes-actor-supervision-8/

Спасибо