Akka

Actor based framework для разработки параллельных систем

Зачем нам нужна Akka ?

- Недостатки классической модели параллельных приложений
  - Использование несколькими потоками разделяемой памяти
  - Возможность поймать deadlock
  - Race conditions
  - Сложность организации общего доступа к данным
  - Сложно проконтролировать полную загрузку системы
  - Невозможно выйти за рамки одного компьютера – требуется смена модели разработки
  - Непонятно как обрабатывать ошибки

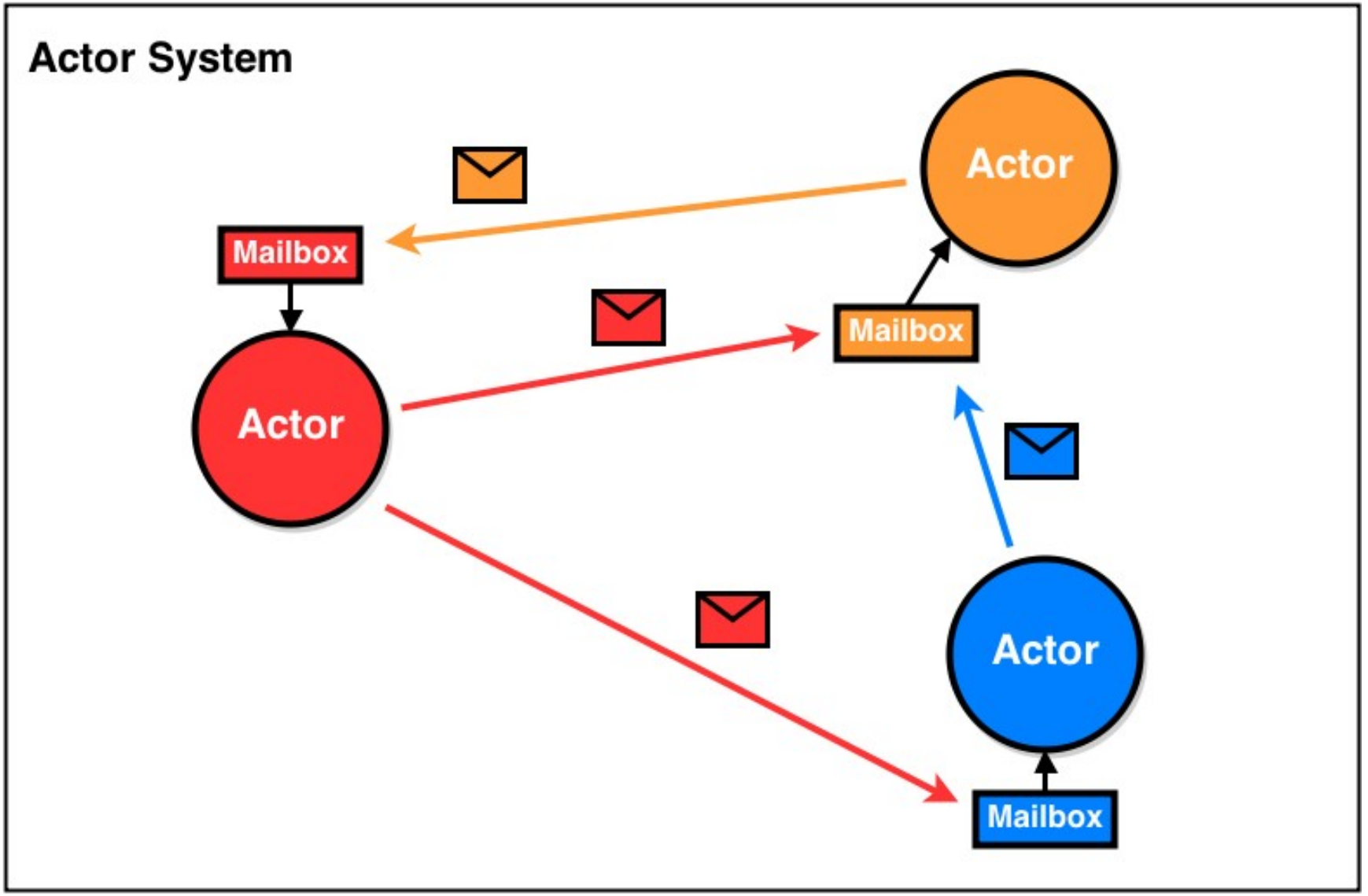
Что предлагает Akka ?

- Actor based framework :
  - Программа состоит из независимых акторов взаимодействующих между собой.
  - Каждый актор в один момент времени обрабатывает одну задачу
  - Сообщения между акторами передаются асинхронно с помощью очередей
  - Акторы образуют иерархию. В случае сбой вышестоящий актор принимает решение что делать.
  - Каждый актор изолированно хранит свое состояние

Происхождение

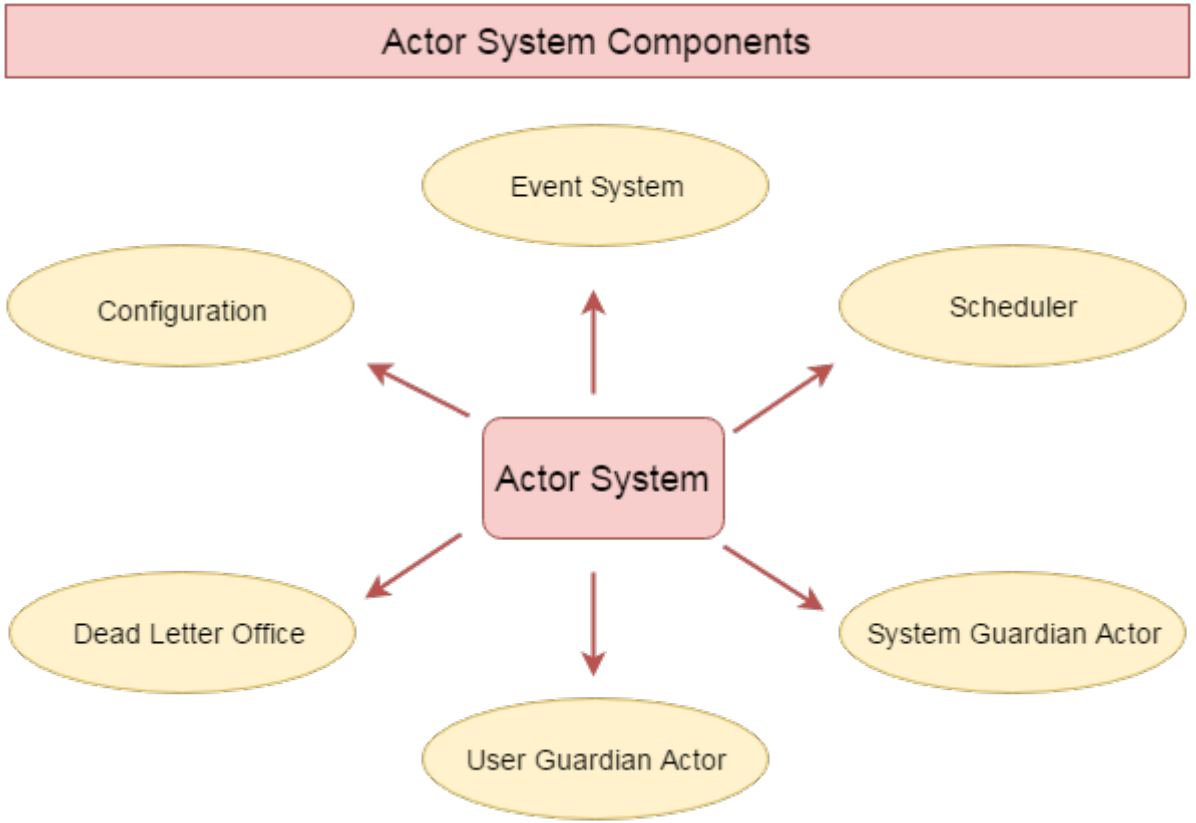
- Язык Erlang и фреймворк Erlang OTP были разработаны рабораторией фирмы Ericsson в начале 90-х
- Erlang – язык с сильной динамической типизацией специально разработанный для программирования распределенных отказоустойчивых программ.
- Erlang OTP – framework которые реализует распределенность и отказоустойчивость с помощью идеи иерархии акторов обменивающихся сежду собой сообщениями.
- Akka – реализация идей использованных в Erlang OTP в экосистеме Java

На что это похоже



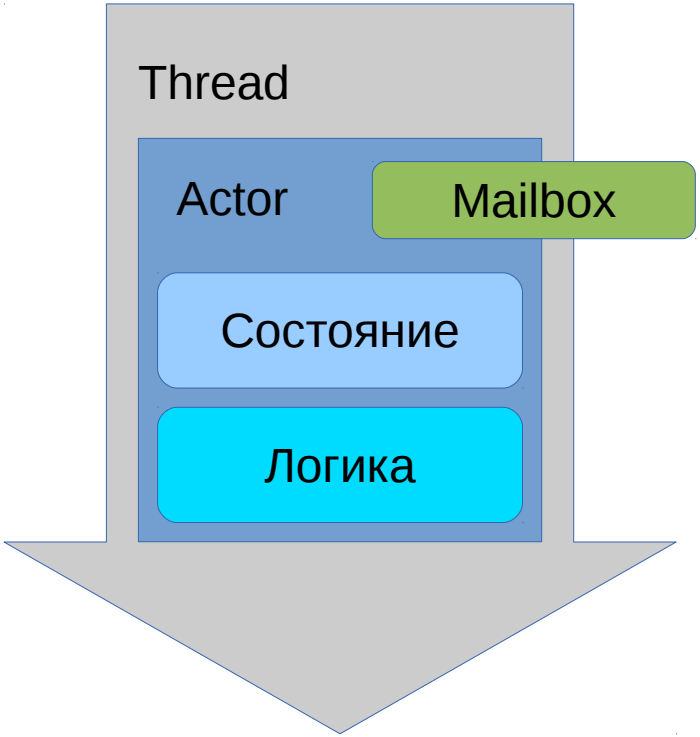
Actor System

- Множество действующих акторов запускается и работает в рамках запущенной Actor system
- Actor system обеспечивает запуск акторов пересылку сообщений и т.д.



Актор

- Актор – основной “Кирпичик” akka
- Инкапсулирует состояние, поведение
- Асинхронно в одном потоке обрабатывает входящие сообщения которые берет из mailbox
- Все акторы образуют систему акторов




```
public class Greeter extends AbstractActor {
  public static enum Msg {
    GREET, DONE;
  }
  @Override
  public Receive createReceive() {
    return receiveBuilder()
      .matchEquals(Msg.GREET, m -> {
        System.out.println("Hello World!");
        sender().tell(Msg.DONE, self());
      })
      .build();
  }
}
```

Пример простого actor-a

```
import akka.actor.AbstractActor;
import akka.japi.pf.ReceiveBuilder;
import java.util.HashMap;
import java.util.Map;
public class StoreActor extends AbstractActor {
    private Map<String, String> store = new HashMap<>();
    @Override
    public Receive createReceive() {
        return ReceiveBuilder.create()
                .match(StoreMessage.class, m -> {
                    store.put(m.getKey(), m.getValue());
                    System.out.println("receive message! "+m.toString());
                })
                .match(GetMessage.class, req -> sender().tell(
                        new StoreMessage(req.getKey(), store.get(req.getKey())), self())
                ).build();
    }
}
```

Создаем экземпляр актора

- Актор создается с помощью вспомогательного класса Props :
  - Props props1 = Props.create(MyActor.class);
  - Props props2 = Props.create(ActorWithArgs.class,
  - () -> new ActorWithArgs("arg"));
  - Props props3 = Props.create(ActorWithArgs.class, "arg");
- Класс Props нужен из-за того что Актор может быть создан на уделенном сервере кластера а также пересоздан в случае ошибок.
- Второй метод с использованием интерфейса Creator потенциально опасен из-за возможности непредсказуемо “сломать” контекст лямбда функции
- Непосредственно создание актора :

```
ActorRef storeActor = system.actorOf(Props.create(StoreActor.class));
```

- Все взаимодействие с актором после его создания происходит с помощью ActorRef

Зачем нам ActorRef ?

- Работать напрямую с java объектом Актором в Акка нельзя.
- Жизненный цикл актора не гарантирует сохранения одного и того же java объекта. В случае остановки или рестарта – конкретный java объект будет уничтожен.
- Акка может работать в составе кластера и в этом случае актор может физически работать в составе другой java машины

Посылка сообщений актору

- С помощью ActorRef мы можем послать сообщение актору с помощью следующих методов :
  - ActorRef.tell отправляет сообщение

```
testPackageActor.tell(msg, ActorRef.noSender());
```

  - ActorRef.forward предназначен для реализации маршрутизации сообщений. Сохраняет исходный адрес отправителя.
  - PatternCS.ask позволяет отправить сообщение и получить Future с ответным сообщением

```
CompletionStage<Object> result = PatternsCS.ask(
                                    storeActor,
                                      newStoreActor.GetMessage(key), 5000
                                   )
```

Пример создания актора и отправки сообщения

```
ActorSystem system = ActorSystem.create("test");
ActorRef storeActor = system.actorOf(
    Props.create(StoreActor.class)
);
storeActor.tell(
   new StoreActor.StoreMessage("test", "test"),
   ActorRef.noSender()
);
```

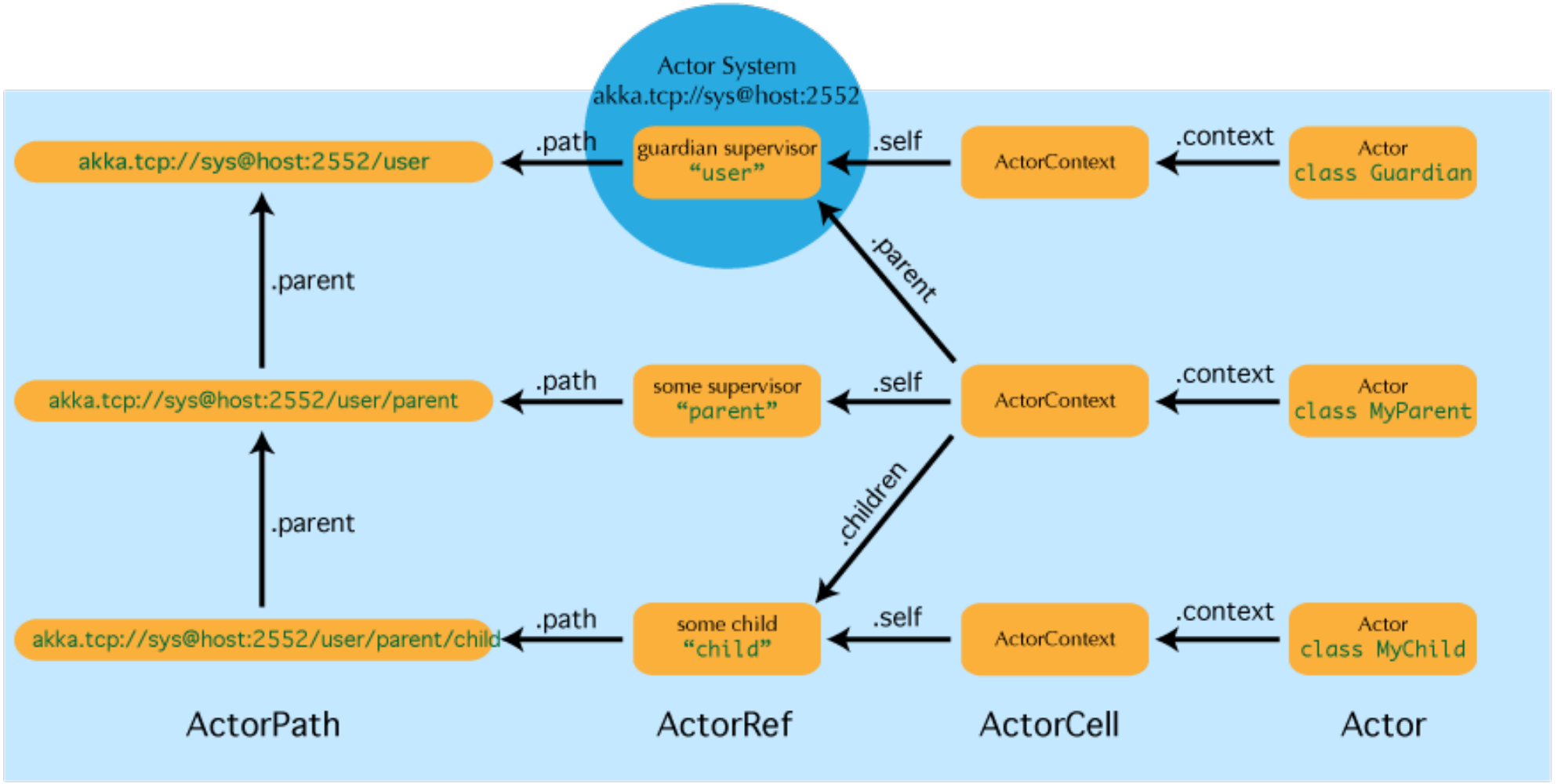
Иерархия акторов

- Акторы образуют иерархию где у каждого актора есть вышестоящий актор.
- Выше акторов верхнего уровня стоит системный актор Guardian
- Каждый актор имеет путь, который задает положение актора в иерархии
- Актор может быть обнаружен если мы знаем его путь
  - Future<ActorRef>ref=system

    .actorSelection("a/b")

    .resolveOne(Timeout.apply(10, TimeUnit.SECONDS));

Иерархия акторов



Actor Selection

- Представляет собой набор отобранных actor-ов соответствующих заданому пути в иерархии actor-ов
- Создается с помощью метода

  - ActorSelection actorSelection = ActorSystem.actorSelection(<path>)

  - Path может содержать ? И *
- ActorSelection можно использовать для следующих целей :
  - отправки сообщений всем actor-ам ActorSelection
    - actorSelection.tell(msg, ActorRef.noSender());
  - Получения ActorRef
    - Вызов метода resolveOne
    - Отправка сообщения Identify. На это сообщение каждый actor автоматически отвечает сообщением ActorIdentity в котором содержится его ActorRef

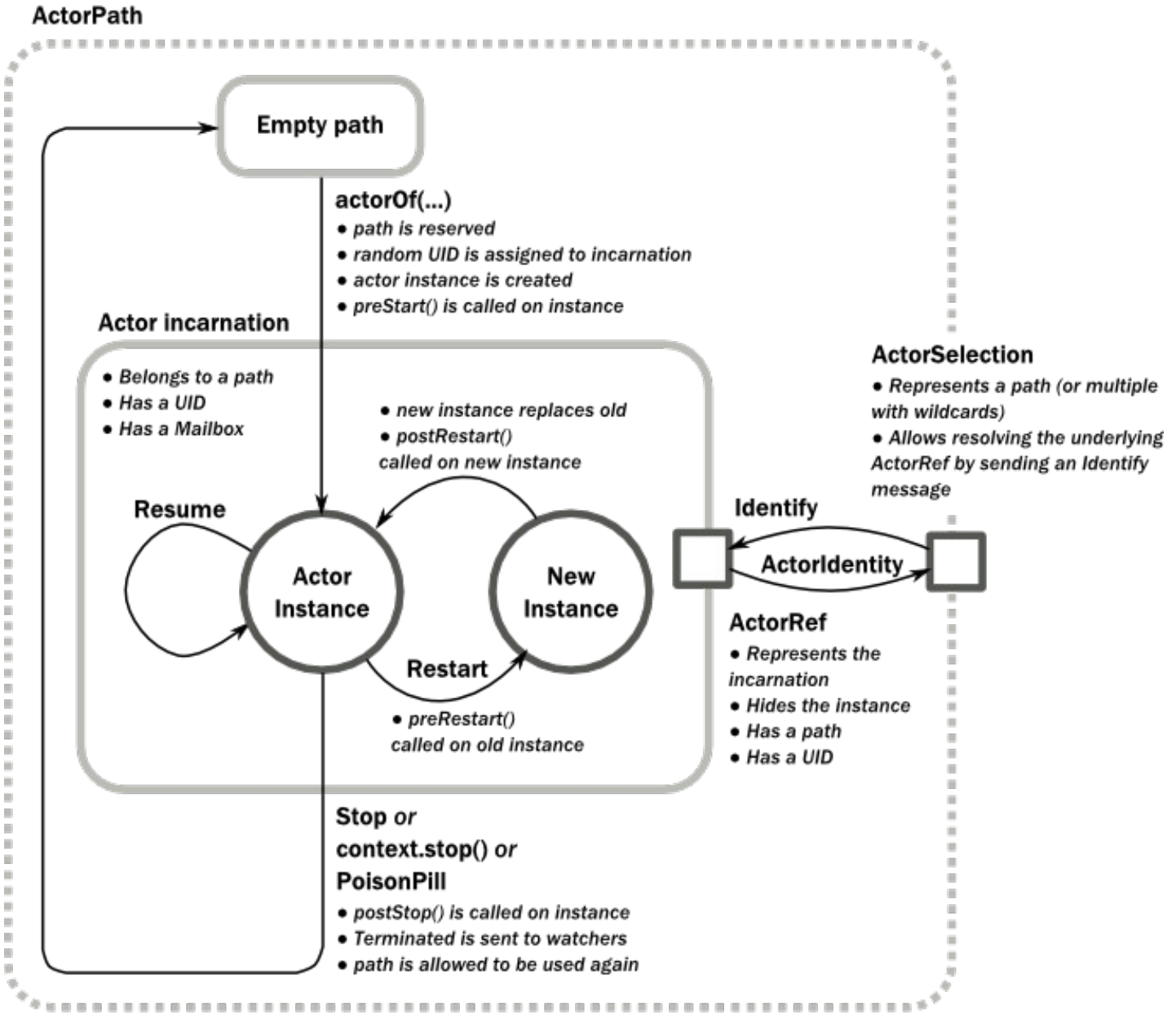
Отказоустойивость

- Ключевой вопрос параллельного программирования – что делать в случае ошибки - сломано ли состояние актора ?
- Акка использует иерархию акторов для решения этой проблемы.
- Решение о необходимом действии принимает вышестоящий актор с помощью SupervisorStrategy.
- Создавая SupervisorStrategy – мы указываем две вещи:
  - Политику по отношению ко всем дочерним акторам(OneForOneStrategy или AllForOneStrategy)
- Для каких exception (произошедших в нижестоящих actor) что нам надо делать
  - Resume – командует актору продолжать как будто ничего не было
  - Restart – сбрасывает состояние актора и создает его заново
  - Stop – останавливает актор
  - Escalate – доверяет решение вышестоящему актору

Пример использования SupervisorStrategy

```
public class TestActor extends AbstractActor {
    private static final int MAX_RETRIES = 10;
    ...
    private static SupervisorStrategy strategy =
          new OneForOneStrategy(MAX_RETRIES,
                                Duration.create("1 minute"),
                                DeciderBuilder.
                                    match(ArithmeticException.class, e -> resume()).
                                    match(NullPointerException.class, e -> restart()).
                                    match(IllegalArgumentException.class, e -> stop()).
                                    matchAny(o -> escalate()).build());
    @Override
    public SupervisorStrategy supervisorStrategy() {
       return strategy;
    }
}
```

Жизненный цикл актора



Что происходит с сообщением и mailbox в случае ошибки ?

- В любом случае исходное сообщение как вызвавшее ошибку - удаляется
- В случае команды restart – Mailbox сохраняются.
- В случае команды stop – актор полностью пересоздается, включая mailbox
- В случае если нам нужно что-то сделать с сообщением которое вызвало ошибку – доступ к нему есть в методе актора - preRestart

Deathwatch

- Часто требуется предпринять действия в случае если актор “убит” - пересоздать его с другими параметрами, подать сигнал администраторам и т.д.
- Актор может подписаться на уведомления о смерти другого актора
  - this.context().watch(anotherActorRef);
- В случае если актор будет остановлен, всем кто подписан на его остановку – придет сообщение Termiated.

Роутеры

- Роутеры представляют собой типовые паттерны управления и использования акторов.
- Типичный пример – пул актров который распределяет входящие сообщения равномерно по всем участникам пула.
- Роутер сам является актором, который управляет созданием дочерних акторов и пересылкой сообщений
- Роутер может иметь свою SupervisorStrategy

```
testPerformerActor = getContext().actorOf(
        new RoundRobinPool(5)
            .withSupervisorStrategy(strategy)
            .props(Props.create(TestPerformerActor.class, logResultsActor)),
        "routerForTests"
    );
```

Роутеры - группы

- В случае если роутеру нельзя доверить самостоятельное создание дочерних акторов, используются роутеры группы
- В момент создания такого роутера требуется передать ему набор путей к дочерним роутерам.

```
getContext().actorOf(Props.create(TestPerformerActor.class, logResultsActor),
    "t1");
getContext().actorOf(Props.create(TestPerformerActor.class, logResultsActor),
    "t2");
getContext().actorOf(Props.create(TestPerformerActor.class, logResultsActor),
     "t3");
List<String> routeePaths = Arrays.asList("../t1", "../t2", "../t3");
testPerformerActor = getContext().actorOf(
       new RoundRobinGroup(routeePaths).props(), "testGroup");
```

Готовые роутеры AKKA

- RoundRobinPool, RoundRobinGroup
  - Распределяет сообщения равномерно по всему пулу котороы
- RandomPool, RandomGroup
  - Каждое сообщение уходит случайному актору пулу
- BalancingPool
  - Для всех дочерних авторов используется общий mailbox. В результате сообщения не “залипают” в случае плохой работы одного актора
- SmallestMailboxPool
  - Каждое сообщение отправляется актору с mailbox наименьшей длины
- BroadcastPool, BroadcastGroup
  - Отправляет сообщение всем дочерним акторам

Готовые роутеры Akka 2

- ScatterGatherFirstCompletedPool (Group)
  - Запрос уходит всем дочерним акторам. Первый ответ пересылается отправителю. Остальные ответы игнорируются
- TailChoppingPool, TailChoppingGroup
  - Запрос уходи случайному актору пула, через определенное время – второму т.д. Ответы обрабатываются аналогично ScatterGatherFirstCompletedPool
- ConsistentHashingPool, ConsistentHashingGroup
  - Запрос посылается актору исходя из хэша сообщения (вычислить хэ мжно с помощью использования метода withHashMapper или реализовав в сообщении интерфейсы – ConsistentHashable или ConsistentHashableEnvelope)