Actor

Russell Winder

Version 1.0, 2015-10-01

Table of Contents

Concepts

Actor

Actors are independent isolated active objects, which mutually share no data and communicate solely by messages passing. Avoiding shared mutable state relieves developers from many typical concurrency problems, like live-locks or race conditions. The body (code) of each **actor** gets executed by a random thread from a thread pool and so **actors** can proceed concurrently and independently. Since **Actors** can share a relatively small thread pool, they avoid the threading limitations of the JVM and don't require excessive system resources even in cases when your application consists of thousands of **actors**.

Actors typically perform three basic types of operations on top of their usual tasks:

- Create a new **actor**
- Send a message to another **actor**
- Receive a message

Actors can be created as subclasses of an particular **actor** class or using a factory method supplying the **actor**'s body as a closure parameter. There are various ways to send a message, either using the >> operator or any of the *send()* , *sendAndWait()* or *sendAndContinua()* methods.

Receiving a message can be performed either in a blocking or a non-blocking way, when the physical thread is returned to the pool until a message is available.

Actors can be orchestrated into various sorts of algorithms, potentially leveraging architectural patterns similar to those known from the enterprise messaging systems.

Lifecycle

Creating an Actor Using Factory Methods

Creating an **Actor**

```
Actors.actor {
    println "actor1 has started"
    delegate.metaClass {
        afterStop = {List undeliveredMessages ->
            println "actor1 has stopped"
        }
        onInterrupt = {InterruptedException e ->
            println "actor1 has been interrupted"
        }
        onTimeout = {->
            println "actor1 has timed out"
        }
        onException = {Exception e ->
            println "actor1 threw an exception"
        }
    }
    println("Running actor1")
    ...
}
```

Sub-classing the DefaultActor class

Sub-class an **Actor**

```
class PooledLifeCycleSampleActor extends DefaultActor {
    protected void act() {
        println("Running actor2")
        ...
    }
    private void afterStart() {
        println "actor2 has started"
    }
    private void afterStop(List undeliveredMessages) {
        println "actor2 has stopped"
    }
    private void onInterrupt(InterruptedException e) {
        println "actor2 has been interrupted"
    }
    private void onTimeout() {
        println "actor2 has timed out"
    }
    private void onException(Exception e) {
        println "actor2 threw an exception"
    }
}
```

Usage

Creating an Actor Using a Factory Method

An **Actor** *from The Factory*

```
import static groovyx.gpars.actor.Actors.actor

def console = actor {
    loop {
        react {
            println it
        }
    }
    ...
}
```

Sub-classing the DefaultActor class

Sub-class a **DefaultActor**

```
class CustomActor extends DefaultActor {
    @Override protected void act() {
        loop {
            react {
                println it
            }
        }
    }
}

def console=new CustomActor()
console.start()
```

Sending Messages

Messages for **Actors**

```
console.send('Message')
console << 'Message'
console.sendAndContinue 'Message', {reply -> println "I received reply: $reply"}
console.sendAndWait 'Message'
```

Timeouts

How To Handle Timing Issues

```
import static groovyx.gpars.actor.Actors.actor

def me = actor {
    friend.send('Hi')
    react(30.seconds) {msg ->
        if (msg == Actor.TIMEOUT) {
            friend.send('I see, busy as usual. Never mind.')
            stop()
        } else {
            //continue conversation
        }
    }
}
me.join()
```

When a timeout expires when waiting for a message, the *Actor.TIMEOUT* message arrives instead. Also the *onTimeout()* handler is invoked, if present on the **actor**:

What Happens When an Actor Times-out

```
import static groovyx.gpars.actor.Actors.actor

def me = actor {
    delegate.metaClass.onTimeout = {->
        friend.send('I see, busy as usual. Never mind.')
        stop()
    }
    friend.send('Hi')
    react(30.seconds) {
        // Continue conversation.
    }
}
me.join()
```

Actor Groups

A Group of **Actors** *Is Called What ?*

```
def coreActors = new NonDaemonPGroup(5)  //5 non-daemon threads pool
def helperActors = new DefaultPGroup(1)  //1 daemon thread pool
def priceCalculator = coreActors.actor {
...
}
def paymentProcessor = coreActors.actor {
...
}
def emailNotifier = helperActors.actor {
...
}
def cleanupActor = helperActors.actor {
...
}

// Increase size of the core actor group.
coreActors.resize 6

// Shutdown the group's pool once you no longer need the group to release resources.
helperActors.shutdown()
```

DynamicDispatchActor

Dynamic Dispatch

```
final Actor actor = new DynamicDispatchActor({
    when {String msg -> println 'A String'; reply 'Thanks'}
    when {Double msg -> println 'A Double'; reply 'Thanks'}
    when {msg -> println 'A something ...'; reply 'What was that?'}
})
actor.start()
```

Reactor

When **Actors** *React*

```
import groovyx.gpars.actor.Actors

final def doubler = Actors.reactor {
    2 * it
}.start()
```