Akka学习笔记：Actor生命周期

Akka学习笔记系列文章：
《Akka学习笔记：ACTORS介绍》
《Akka学习笔记：Actor消息传递(1)》
《Akka学习笔记：Actor消息传递(2)》
《Akka学习笔记：日志》
《Akka学习笔记：测试Actors》
《Akka学习笔记：Actor消息处理-请求和响应(1) 》
《Akka学习笔记：Actor消息处理-请求和响应(2) 》
《Akka学习笔记：ActorSystem(配置)》
《Akka学习笔记：ActorSystem(调度)》
《Akka学习笔记：Actor生命周期》
《Akka学习笔记：子Actor和Actor路径》

请注意：这里讲的生命周期(lifecycle)并不包含preRestart或postRestart方法，我们将在讨论监管(supervision)的时候来讲述他们。

Actor基本的生命周期非常的直观。实际上除了以下几点不一样，你完全可以将Actor的生命周期和Java servlet生命周期进行对比。
1、和其他普通类一样，我们得需要一个构造函数；
2、preStart函数在其之后被调用。在这里，你可以初始化一些资源，然后在postStop中清除；
3、servicing或receive方法里面的消息处理占用了绝大多数的时间。
让我们来看一下简单的Actor，他仅仅打印出生命周期

```
package me.rerun.akkanotes.lifecycle

import akka.actor.{ActorLogging, Actor}
import akka.event.LoggingReceive

class BasicLifecycleLoggingActor extends Actor with ActorLogging{

  log.info ("Inside BasicLifecycleLoggingActor Constructor")
  log.info (context.self.toString())
  override def preStart() ={
    log.info("Inside the preStart method of BasicLifecycleLoggingActor")
  }

  def receive = LoggingReceive{
    case "hello" => log.info ("hello")
```

```
 }

 override def postStop()={
   log.info ("Inside postStop method of BasicLifecycleLoggingActor")
 }

}
```

LifecycleApp类仅仅用于初始化Actor，并给它发送消息，然后关闭ActorSystem。

```
import akka.actor.{ActorSystem, Props}

object LifecycleApp extends App{

 val actorSystem=ActorSystem("LifecycleActorSystem")
 val lifecycleActor=actorSystem.actorOf(Props[BasicLifecycleLoggingActor],"lifecycleActor")

 lifecycleActor!"hello"

 //wait for a couple of seconds before shutdown
 Thread.sleep(2000)
 actorSystem.shutdown()

 }
```

结果

```
Inside BasicLifecycleLoggingActor Constructor
Actor[akka://LifecycleActorSystem/user/lifecycleActor#-2018741361]
Inside the preStart method of BasicLifecycleLoggingActor
hello
Inside postStop method of BasicLifecycleLoggingActor
```

Servlet生命周期和基本的Actor有什么不同呢？

Actor中的构造方法和preStart方法并没有什么不同——基本是一样的。

为什么我要在构造方法里打印出context.self？这是因为Actor跟Servlet不同，即使是在构造方法里，它也能够访问得到ActorContext。preStart和构造方法之间的边界变得非常模糊。后面

说到监督的时候我们会再说一下它们的区别。不过如果你好奇的话我这里可以先给你介绍一点：Actor重启的时候（比如失败重启）可以通过调用preStart方法来重新初始化；而在构造方法里面是实现不了的。

postStop方法什么时候被调用？

我们从程序中看出，ActorSystem关闭的时候，postStop方法被调用。其实还有其他几种情况可以调用postStop。

1、ActorSystem.stop()

我们可以调用ActorSystem或者ActorContext的stop方法来关闭Actor。

```
object LifecycleApp extends App{

 val actorSystem=ActorSystem("LifecycleActorSystem")
 val lifecycleActor=actorSystem.actorOf(Props[BasicLifecycleLoggingActor],"lifecycleActor")

 actorSystem.stop(lifecycleActor);

 ...
 ...

}
```

2、ActorContext.stop

1)、通过消息（内部的或外部的）

```
class BasicLifecycleLoggingActor extends Actor with ActorLogging{
...
...
 def receive = LoggingReceive{
   case "hello" => log.info ("hello")
   case "stop" => context.stop(self)
 }
```

和

```
 object LifecycleApp extends App{

 val actorSystem=ActorSystem("LifecycleActorSystem")
 val lifecycleActor=actorSystem.actorOf(Props[BasicLifecycleLoggingActor],"lifecycleActor")
```

```
  lifecycleActor!"stop"
...
...
```

2）、或者没任何理由自己杀死自己（这里仅仅是开个玩笑）

```
class BasicLifecycleLoggingActor extends Actor with ActorLogging{

  log.info ("Inside BasicLifecycleLoggingActor Constructor")
  log.info (context.self.toString())
  context.stop(self)
  ...
  ...
```

3、POISONPILL

在前面的例子里面，我们通过在LifecycleApp发送带有stop的消息到Actor，Actor收到这个消息，并通过context.stop来杀死自己。其实我们可以通过发送PoisonPill消息来达到同样的功能。PoisonPill和前面的那个stop消息类似，它也会被扔进到一个普通邮箱里面排队，只有当轮到它的时候才会进行处理。

```
object LifecycleApp extends App{

  val actorSystem=ActorSystem("LifecycleActorSystem")
  val lifecycleActor=actorSystem.actorOf(Props[BasicLifecycleLoggingActor],"lifecycleActor")

  lifecycleActor!PoisonPill
  ...
  ...
```

4.、KILL

如果你不想用PoisonPill，你可以通过发送Kill消息给目标Actor

```
lifecycleActor ! Kill
```

PoisonPill消息和Kill消息之间的区别很小，但是很重要：
1、如果用了PoisonPill消息，将给所有的Watcher发送Terminated消息（后面将会介绍）。
2、如果用了Kill消息，宿主Actor将会抛出ActorKilledException，并传播到监管者哪里去（后面会介绍）。

停止阶段

Actor一旦停止，他也就进入到Terminated状态。你脑海里肯定会想，那些发送给已经停止的Actor的消息将会发生什么？让我们来看下：

```
object LifecycleApp extends App{

  val actorSystem=ActorSystem("LifecycleActorSystem")
  val lifecycleActor=actorSystem.actorOf(Props[BasicLifecycleLoggingActor],"lifecycleActor")

  lifecycleActor!"hello"
  lifecycleActor!"stop"
  lifecycleActor!"hello" //Sending message to an Actor which is already stopped

}
```

Actor---和前面的一样

```
class BasicLifecycleLoggingActor extends Actor with ActorLogging{

  def receive = LoggingReceive{
    case "hello" => log.info ("hello")
    case "stop" => context.stop(self)

  }
}
```

输出

```
BasicLifecycleLoggingActor - hello

akka.actor.RepointableActorRef - Message 1 from Actor[akka://LifecycleActorSystem/deadLetters] to Actor[akka://LifecycleActorSystem/user/lifecycleActor#-569230546] was not delivered. [1] dead letters encountered. This logging can be turned off or adjusted with configuration settin
```

gs 'akka.log-dead-letters' and 'akka.log-dead-letters-during-shutdown'.

从日志中你可以看到好几个deadletters，所有发送到已终止的Actor的消息都会被转发给一个叫做DeadLetterActor的内部Actor。

好奇后面会发生什么?

DeadLetter Actor会处理自己邮箱里的消息，并把每条消息都封装成一个DeadLetter，然后再发布到EventStream里面去。

另一个叫做DeadLetterListener的Actor会去消费所有这些DeadLetter消息并把它们作为一条日志消息发布出去。看下这里。

还记得吧，Akka日志
那篇文章中曾经说过，所有的日志消息都会发布到EventStream里面，我们可以随便订阅—仅仅需要保证订阅者也是一个Actor。我们来试一下。

对于我们这个示例而言，我们要订阅到EventStream上并监听所有的DeadLetter消息然后将它们打印到控制台上。其实我们可以做很多事情，比如生成一个警告，或者把它存储到数据库里，甚至用它来进行分析。

```
import akka.actor.ActorSystem
import akka.actor.Props
import akka.actor.PoisonPill
import akka.actor.DeadLetter
import akka.actor.Actor

object LifecycleApp extends App {

  val actorSystem = ActorSystem("LifecycleActorSystem")
  val lifecycleActor = actorSystem.actorOf(Props[BasicLifecycleLoggingActor], "lifecycleActor")

  val deadLetterListener = actorSystem.actorOf(Props[MyCustomDeadLetterListener])
  actorSystem.eventStream.subscribe(deadLetterListener, classOf[DeadLetter])

  lifecycleActor ! "hello"
  lifecycleActor ! "stop"
  lifecycleActor ! "hello"

}

class MyCustomDeadLetterListener extends Actor {
  def receive = {
    case deadLetter: DeadLetter => println(s"FROM CUSTOM LISTENER $deadLetter")
```

```
 }
}
```

输出

```
164  [LifecycleActorSystem-akka.actor.default-
dispatcher-4] INFO  BasicLifecycleLoggingActor - hello

167  [LifecycleActorSystem-akka.actor.default-dispatcher-4] INFO  akka.actor.RepointableActor
Ref - Message 1 from Actor[akka://LifecycleActorSystem/deadLetters] to Actor[akka://Lifecycle
ActorSystem/user/lifecycleActor#-782937925] was not delivered. [1] dead letters encountered.
 This logging can be turned off or adjusted with configuration settings 'akka.log-dead-
letters' and 'akka.log-dead-letters-during-shutdown'.

FROM CUSTOM LISTENER DeadLetter(hello,Actor[akka://LifecycleActorSystem/deadLetters],Act
or[akka://LifecycleActorSystem/user/lifecycleActor#-782937925])
```

本文翻译自：http://rerun.me/2014/10/21/akka-notes-actor-lifecycle-basic/