Scala Actor

# 课程目标

## 目标一：熟悉Scala Actor并发编程

## 目标二：为学习Akka做准备

注：我们现在学的Scala Actor是scala 2.10.x版本及以前版本的Actor。

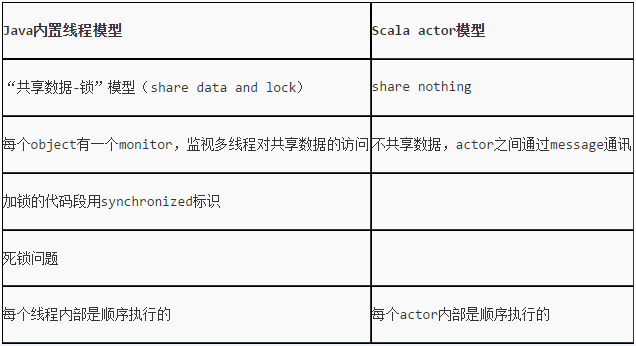
Scala在2.11.x版本中将Akka加入其中，作为其默认的Actor，老版本的Actor已经废弃

# 什么是Scala Actor

## 概念

Scala中的Actor能够实现并行编程的强大功能，它是基于事件模型的并发机制，Scala是运用消息（message）的发送、接收来实现多线程的。使用Scala能够更容易地实现多线程应用的开发。

## 传统java并发编程与Scala Actor编程的区别



对于Java，我们都知道它的多线程实现需要对共享资源（变量、对象等）使用synchronized 关键字进行代码块同步、对象锁互斥等等。而且，常常一大块的try…catch语句块中加上wait方法、notify方法、notifyAll方法是让人很头疼的。原因就在于Java中多数使用的是可变状态的对象资源，对这些资源进行共享来实现多线程编程的话，控制好资源竞争与防止对象状态被意外修改是非常重要的，而对象状态的不变性也是较难以保证的。 而在Scala中，我们可以通过复制不可变状态的资源（即对象，Scala中一切都是对象，连函数、方法也是）的一个副本，再基于Actor的消息发送、接收机制进行并行编程

## Actor方法执行顺序

1. 首先调用start()方法启动Actor
2. 调用start()方法后其act()方法会被执行
3. 向Actor发送消息

## 发送消息的方式

|  |  |
| --- | --- |
| ! | 发送异步消息，没有返回值。 |
| !? | 发送同步消息，等待返回值。 |
| !! | 发送异步消息，返回值是 Future[Any]。 |

# Actor实战

## 第一个例子

|  |
| --- |
| **package** cn.itcast.actor //注意导包是scala.actors.Actor  **import** scala.actors.Actor  **object** MyActor1 **extends** Actor{  *//重新act方法* **def** act(){  **for**(i <- 1 to 10){  *println*(**"actor-1 "** + i)  Thread.*sleep*(2000)  }  } }  **object** MyActor2 **extends** Actor{  *//重新act方法* **def** act(){  **for**(i <- 1 to 10){  *println*(**"actor-2 "** + i)  Thread.*sleep*(2000)  }  } }  **object** ActorTest **extends** App{  *//启动Actor* MyActor1.start()  MyActor2.start() } |

说明：上面分别调用了两个单例对象的start()方法，他们的act()方法会被执行，相同与在java中开启了两个线程，线程的run()方法会被执行

注意：这两个Actor是并行执行的，act()方法中的for循环执行完成后actor程序就退出了

## 第二个例子（可以不断地接收消息）

|  |
| --- |
| **package** cn.itcast.actor  **import** scala.actors.Actor  */\*\*  \* Created by ZX on 2016/4/4.  \*/* **class** MyActor **extends** Actor {   **override def** act(): Unit = {  **while** (**true**) {  receive {  **case "start"** => {  *println*(**"starting ..."**)  Thread.*sleep*(5000)  *println*(**"started"**)  }  **case "stop"** => {  *println*(**"stopping ..."**)  Thread.*sleep*(5000)  *println*(**"stopped ..."**)  }  }  }  } }  **object** MyActor {  **def** main(args: Array[String]) {  **val** actor = **new** MyActor  actor.start()  actor ! **"start"** actor ! **"stop"** *println*(**"消息发送完成！"**)  } } |

说明：在act()方法中加入了**while** (**true**) 循环，就可以不停的接收消息

注意：发送start消息和stop的消息是异步的，但是Actor接收到消息执行的过程是同步的按顺序执行

## 第三个例子（react方式会复用线程，比receive更高效）

|  |
| --- |
| **package** cn.itcast.actor  **import** scala.actors.Actor  */\*\*  \* Created by ZX on 2016/4/4.  \*/* **class** YourActor **extends** Actor {  **override def** act(): Unit = {  loop {  react {  **case "start"** => {  *println*(**"starting ..."**)  Thread.*sleep*(5000)  *println*(**"started"**)  }  **case "stop"** => {  *println*(**"stopping ..."**)  Thread.*sleep*(8000)  *println*(**"stopped ..."**)  }  }  }  } }  **object** YourActor {  **def** main(args: Array[String]) {  **val** actor = **new** YourActor  actor.start()  actor ! **"start"** actor ! **"stop"** *println*(**"消息发送完成！"**)  } } |

说明： react 如果要反复执行消息处理，react外层要用loop，不能用while

## 第四个例子（结合case class发送消息）

|  |
| --- |
| **package** cn.itcast.actor**package** cn.itcast.actor **import** scala.actors.Actor **class** AppleActor **extends** Actor {   **def** act(): Unit = {  **while** (**true**) {  receive {  **case "start"** => *println*(**"starting ..."**)  **case** *SyncMsg*(id, msg) => {  *println*(id + **",sync "** + msg)  Thread.*sleep*(5000)  sender ! *ReplyMsg*(3,**"finished"**)  }  **case** *AsyncMsg*(id, msg) => {  *println*(id + **",async "** + msg)  Thread.*sleep*(5000)  }  }  }  } }  **object** AppleActor {  **def** main(args: Array[String]) {  **val** a = **new** AppleActor  a.start()  *//异步消息* a ! *AsyncMsg*(1, **"hello actor"**)  *println*(**"异步消息发送完成"**)  *//同步消息  //val content = a.!?(1000, SyncMsg(2, "hello actor"))  //println(content)* **val** reply = a !! *SyncMsg*(2, **"hello actor"**)  *println*(reply.isSet)  *//println("123")* **val** c = reply.apply()  *println*(reply.isSet)  *println*(c)  } } **case class** SyncMsg(id : Int, msg: String) **case class** AsyncMsg(id : Int, msg: String) **case class** ReplyMsg(id : Int, msg: String) |

# 练习

用actor并发编程写一个单机版的WorldCount，将多个文件作为输入，计算完成后将多个任务汇总，得到最终的结果

|  |
| --- |
| **package** cn.itcast.actor  **import** java.io.File  **import** scala.actors.{Actor, Future} **import** scala.collection.mutable **import** scala.io.Source  */\*\*  \* Created by ZX on 2016/4/4.  \*/* **class** Task **extends** Actor {   **override def** act(): Unit = {  loop {  react {  **case** *SubmitTask*(fileName) => {  **val** contents = Source.*fromFile*(**new** File(fileName)).mkString  **val** arr = contents.split(**"\r\n"**)  **val** result = arr.flatMap(\_.split(**" "**)).map((\_, 1)).groupBy(\_.\_1).mapValues(\_.length)  *//val result = arr.flatMap(\_.split(" ")).map((\_, 1)).groupBy(\_.\_1).mapValues(\_.foldLeft(0)(\_ + \_.\_2))* sender ! *ResultTask*(result)  }  **case** StopTask => {  exit()  }  }  }  } }  **object** WorkCount {  **def** main(args: Array[String]) {  **val** files = *Array*(**"c://words.txt"**, **"c://words.log"**)   **val** replaySet = **new** mutable.HashSet[Future[Any]]  **val** resultList = **new** mutable.ListBuffer[ResultTask]   **for**(f <- files) {  **val** t = **new** Task  **val** replay = t.start() !! *SubmitTask*(f)  replaySet += replay  }   **while**(replaySet.size > 0){  **val** toCumpute = replaySet.filter(\_.isSet)  **for**(r <- toCumpute){  **val** result = r.apply()  resultList += result.asInstanceOf[ResultTask]  replaySet.remove(r)  }  Thread.*sleep*(100)  }  **val** finalResult = resultList.map(\_.result).flatten.groupBy(\_.\_1).mapValues(x => x.foldLeft(0)(\_ + \_.\_2))  *println*(finalResult)  } }  **case class** SubmitTask(fileName: String) **case object** StopTask **case class** ResultTask(result: Map[String, Int]) |