java中的并发是用多线程来实现的， Tread

scala 中的并发，使用的是akka框架，基于Actor模型

# 并发编程模型Akka

## Akka 介绍

处理并发的框架， 基于scala开发的

多线程开发要处理并发，锁，线程同步等一系列问题，所以一般都不愿意自己动手写，能不能换一种思路，可以不可以搞一种更高级的抽象模型，让我们需要实现多线程应用的时候，不用再考虑这些底层问题呢？

Akka是JVM平台上构建高并发、分布式和容错应用的工具包和运行平台。Akka是用 Scala语言编程的一个并发编程框架，该框架基于Actor编程模型。

Akka实现并发，基于Actor模式，

相当于Java实现并发，基于Thread。

## Akka中Actor模型

与java的基于共享数据和锁的线程模型不同，scala的actor包则提供了另外一种不共享任何数据、依赖消息传递的模型。

在基于Actor的系统里，所有的事物都是Actor，就好像在面向对象设计里面所有的事物都是对象一样。但是有一个重要区别，那就是Actor模型是作为一个并发模型设计和架构的，而面向对象模式则不是。Actor与Actor之间只能通过消息通信。

可以将Actor当作是一群人，他们互相之间不会面对面地交流，而只是通过邮件的方式进行沟通。传递消息是actor模型的基础。

Actor是一种编程模型，通过发送消息的方式实现并发编程

必须有**多个**相同业务逻辑的Actor才可以实现并发编程，一个Actor相当于一个实例（老师/学生），消息是发送的数据（邮件）

Actor和Actor之间可以相互发送消息

每一个Actor中都有相应的业务逻辑，如果使用编程，定义一个类，在类中定义方法，通过new一个实例，即可得到一个Actor

Actor是ActorSystem创建的，ActorSystem的职责是负责创建并管理自己创建的Actor，ActorSystem的单例的，一个JVM进程中有一个即可，而Acotr是多例的。ActorSystem相当于老大，Actor相当于干活的小弟

## 使用Actor模型的好处：

事件模型驱动--Actor之间的通信是异步的，即使Actor在发送消息后也无需阻塞或者等待就能够处理其他事情；

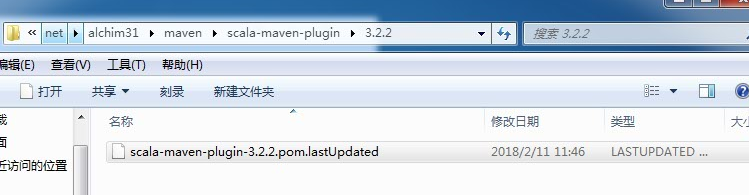
强隔离性--Actor中的方法不能由外部直接调用，所有的一切都通过消息传递进行的，从而避免了Actor之间的数据共享，想要观察到另一个Actor的状态变化只能通过消息传递进行询问；

轻量性--Actor是非常轻量的计算单机，单个Actor仅占400多字节，只需少量内存就能达到高并发；

|  |  |
| --- | --- |
| Java内置线程模型 | Scala actor模型 |
| “共享数据-锁”模型（share data and lock） | share nothing |
| 每个object有一个monitor，监视多线程对共享数据的访问 | 不共享数据，actor之间通过message通信 |
| 加锁的代码段用synchronized标识 |  |
| 死锁问题 |  |
| 每个线程内部是顺序执行的 | 每个actor内部是顺序执行的 |

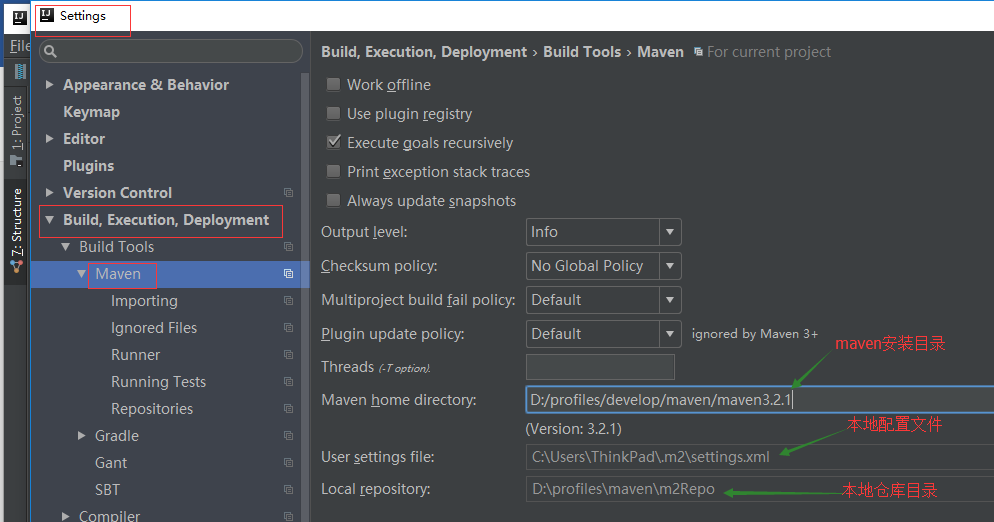
## maven工程：

如果本地仓库的jar包没有下载成功，就有一个.lastUpdate为后缀的文件，删除该文件，然后重新下载。

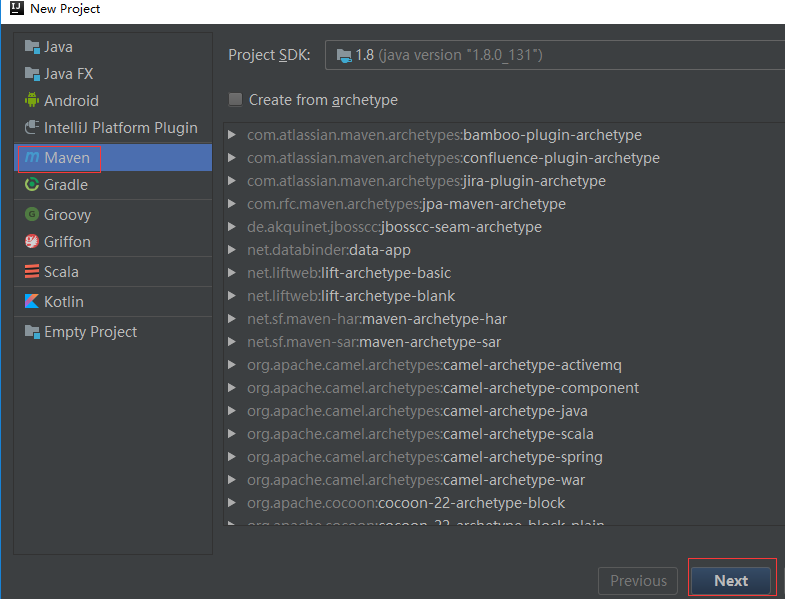


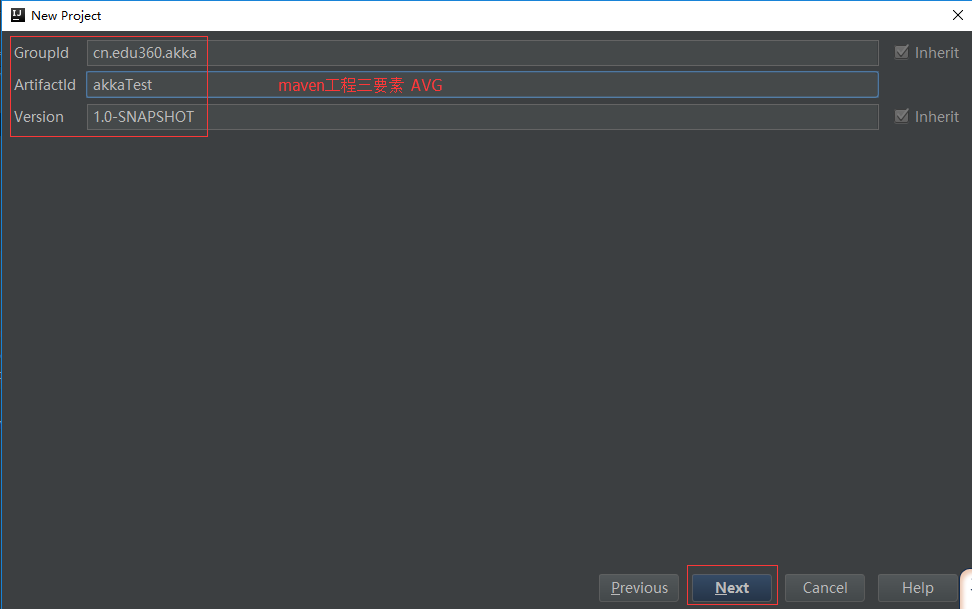
### IDEA中配置maven

默认加载本机的maven目录，所以正常不用配置



### 创建maven工程



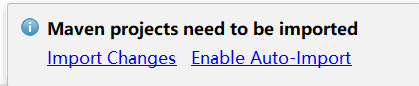


### Pom依赖：

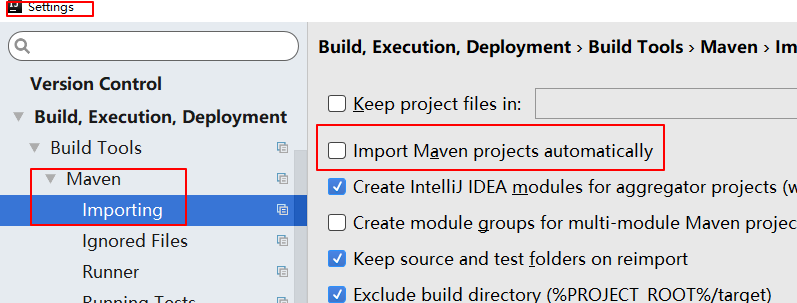
完整参考pom.xml文件

|  |
| --- |
| *<!-- 定义一下常量 -->* <**properties**>  <**maven.compiler.source**>1.8</**maven.compiler.source**>  <**maven.compiler.target**>1.8</**maven.compiler.target**>  <**encoding**>UTF-8</**encoding**>  <**scala.version**>2.11.8</**scala.version**>  <**scala.compat.version**>2.11</**scala.compat.version**>  <**akka.version**>2.4.17</**akka.version**> </**properties**>   <**dependencies**>  *<!-- 添加scala的依赖 -->* <**dependency**>  <**groupId**>org.scala-lang</**groupId**>  <**artifactId**>scala-library</**artifactId**>  <**version**>${scala.version}</**version**>  </**dependency**>   *<!-- 添加akka的actor依赖 -->* <**dependency**>  <**groupId**>com.typesafe.akka</**groupId**>  <**artifactId**>akka-actor\_${scala.compat.version}</**artifactId**>  <**version**>${akka.version}</**version**>  </**dependency**>   *<!-- 多进程之间的Actor通信 -->* <**dependency**>  <**groupId**>com.typesafe.akka</**groupId**>  <**artifactId**>akka-remote\_${scala.compat.version}</**artifactId**>  <**version**>${akka.version}</**version**>  </**dependency**>  </**dependencies**> |

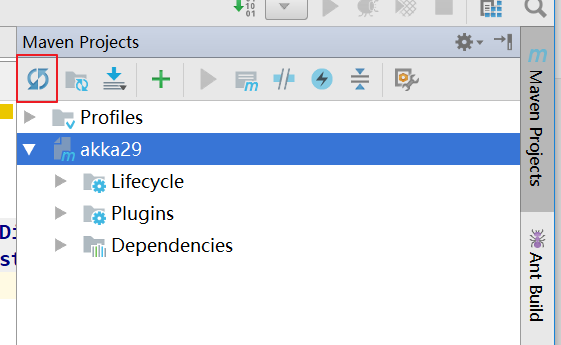
修改过pom.xml文件后，需要重新加载生效：

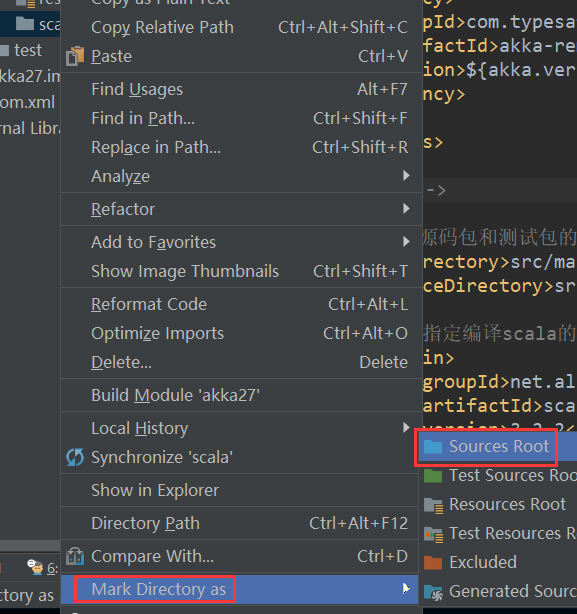


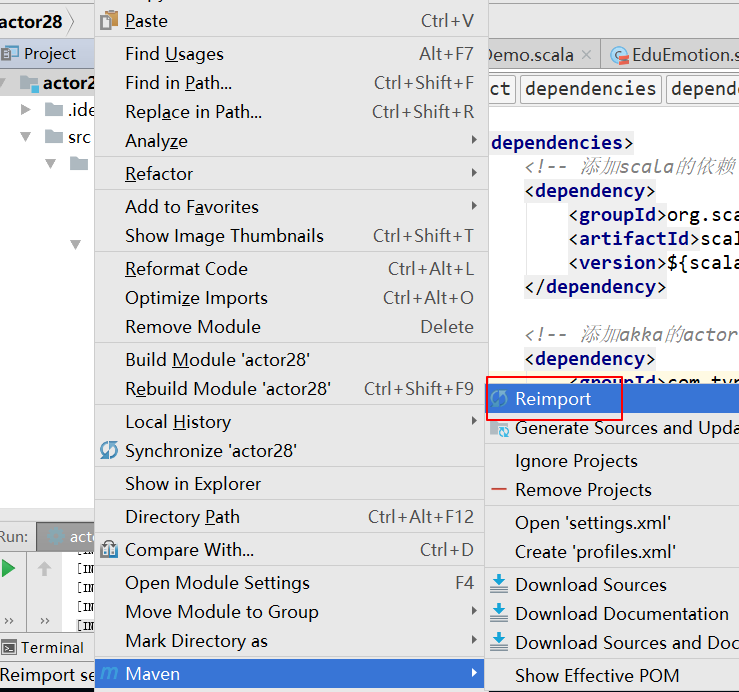
配置 自动导入改变：



创建src/main/scala包之后，重新加载





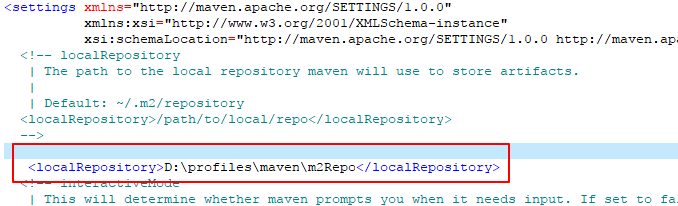


在pom.xml文件中，指定源码包路径

*<!-- 在build中，指定源码的路径-->*<**sourceDirectory**>src/main/scala</**sourceDirectory**>

### 可选(仅针对外部maven配置)：

如果使用内置maven，直接跳过本节



修改为自己指定的目录即可。

<localRepository>/path/to/local/repo</localRepository>

可在settings.xml中添加国内阿里云镜像：

|  |
| --- |
| <mirrors>  <mirror>  <id>alimaven</id>  <name>aliyun maven</name>  <url>http://maven.aliyun.com/nexus/content/groups/public/</url>  <mirrorOf>central</mirrorOf>  </mirror>  </mirrors> |

首先去本地仓库查找jar包，就去配置的 阿里云的镜像仓库中查找，找到之后，就下载到本地仓库；再去中央仓库（maven的官方仓库）

可去官网搜索jar包的配置：

<http://search.maven.org>

## 案例1：Actor入门案例

一个Actor就是一个类，要想实现actor功能，需要实现Actor特质

需要重写receive方法，进行消息匹配接收

程序运行需要执行入口，所以需要有main方法。

actor必须通过ActorSystem来创建

|  |
| --- |
| *// 通过字符串添加配置信息* **val** str =  **"""  |akka.actor.provider = "akka.remote.RemoteActorRefProvider"  |akka.remote.netty.tcp.hostname = "127.0.0.1"  |akka.remote.netty.tcp.port = "8888"  """**.stripMargin  *// 创建配置工厂*  **val** conf = ConfigFactory.*parseString*(str) *// 创建actorsystem* **val** as: ActorSystem = ActorSystem.*create*(**"actorsystem-test"**,conf)  *// 通过类型反射创建actor的引用对象*  **val** actor: ActorRef = as.actorOf(*Props*[HelloActor],**"actor-test"**)  *// 向actor引用对象发送异步消息*  actor ! **"hello"** |

## 案例2 人机Ping Pang大战

思路：

需要两个Actor，通过发送消息进行通信。

一个Actor在启动时，需要向另一个actor发送连接请求。

连接请求在preStart()方法中实现。

|  |
| --- |
| **class** AlphGo **extends** Actor{  *// 表示接受的方法，进行模式匹配 会被调用多次* **override def** receive: Receive = {  } }  **object** AlphGo{  **def** main(args: Array[String]): Unit = {} } |

|  |
| --- |
| **class** MaLong **extends** Actor{  *// 在构造器之后，receive方法之前执行* **override def** preStart(): Unit = {  *// worker向master建立连接，并向master发送消息*  *// 四个关键参数： actorSystem的名称 actor的名称 ip地址，端口号* **val** proxy: ActorSelection = *context*.actorSelection(**"akka.tcp://acs-go@127.0.0.1:8888/user/ac-go"**)  proxy ! **"request get connect"** }  **override def** receive: Receive = {  } }  **object** MaLong{  **def** main(args: Array[String]): Unit = {  }} |

## 案例3 Spark Master Worker进程通信示例

### 业务流程：

1,master worker 都要启动

2,worker在启动之后，需要向master发送注册请求 附带信息 workerId, cores 内存 可以使用 case calss 封装数据

3,master接收到worker的请求信息之后，保存worker的注册信息，向worker发送响应信息（注册成功）

4，worker收到注册成功的信息之后，要定时发送心跳（报活） 定时任务 case class workerId

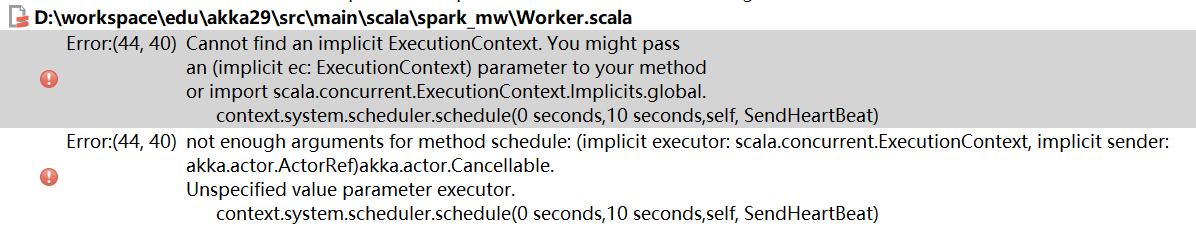
5，master收到worker发送的心跳信息之后，就要更新worker的心跳时间

0，master启动之后，定时检测worker的状态 （如果检测出来worker挂掉了，那 删除该worker的注册信息）

### 业务实现

|  |
| --- |
| **class** Master **extends** Actor{  **val** *workersMap* = mutable.HashMap[String,WorkerInfo]()  *// master 启动之后，会启动一个定时任务，作用是检测活着的worker信息* **override def** preStart(): Unit = {  **import** *context*.dispatcher  **import** scala.concurrent.duration.\_  *context*.system.scheduler.schedule(0 millis,15000 millis,*self*,CheckWorkerStatus)  }  *// 表示接受的方法* **override def** receive: Receive = {  } } **object** Master{  } } |

定时任务报错： 需要导入 **import** *context*.dispatcher



|  |
| --- |
| **class** Worker(**val** master\_name:String,**val** master\_port:Int,**var** memory:Int,**var** cores:Int) **extends** Actor{  **var** *master\_proxy*: ActorSelection=\_  **val** *workerId* = UUID.randomUUID().toString  *// 在构造器之后，receive方法之前执行* **override def** preStart(): Unit = {  *// worker向master建立连接，并向master发送消息  master\_proxy* = *context*.actorSelection(**s"akka.tcp://$**{Master.*MASTER\_ACTOR\_SYSTEM*}**@$**master\_name**:$**master\_port**/user/$**{Master.*ACTOR\_NAME*}**"**)  *// 向master发送注册消息 封装数据 workerid memory cores  master\_proxy* ! *Register2Master*(*workerId*,memory,cores)  *println*(**"connect to master"**)  }   **override def** receive: Receive = {  } }  **object** Worker{  } } |

需要传递的消息：

|  |
| --- |
| *// 数据在网络间传输，需要实现序列化特质 // worker向master发送的注册消息*  *// master 向worker回复的注册成功信息* **case object** RegisteredMaster **extends** Serializable *// worker向master发送的心跳信息（带workerid）*  *// worker定时器 发送的心跳信息*  *// master定时器 检测超时worker* |

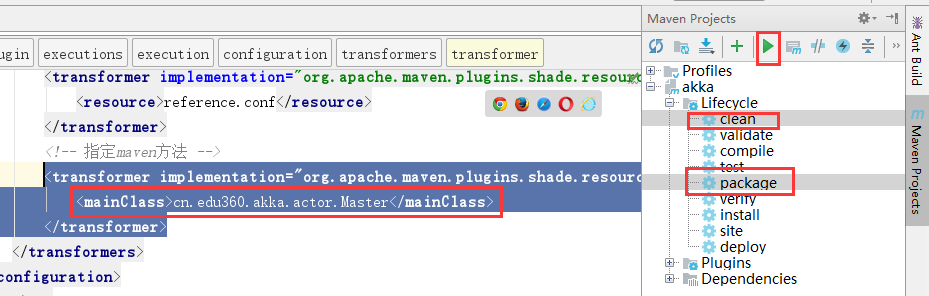
|  |
| --- |
| override def receive: Receive = {  // worker 接收master的注册成功消息 然后启动定时任务  case SuccessRegister => {  println("register success")  // worker需要启动定时任务 定时发送心跳  /\*\*  \* initialDelay: FiniteDuration, 是否延迟执行 不延迟 0  \* interval: FiniteDuration, 间隔时间 每隔多长时间调用一次定时任务 10s  \* receiver: ActorRef, 发送给谁 Master  \* message: Any 发送什么消息 心跳消息  \*/  // master 接收到worker 的心跳消息之后，要做什么事？ master需要保存worker的心跳时间 最后一次心跳的时间  import scala.concurrent.duration.\_  import context.dispatcher  // context.system.scheduler.schedule(0 seconds,10 seconds,selection, HeartBeat2Master(workerId))  context.system.scheduler.schedule(0 seconds,10 seconds,self, SendHeartBeat)  // 曲线救国 不能直接向master发送心跳信息 ----》 先发给自己，接收该消息之后，再发送给Master  }  // 接收发给自己的消息，目的，该报活了  case SendHeartBeat =>{  // 向Master发送心跳信息  selection ! HeartBeat2Master(workerId)  println("send heart beat to master")  }  } |

### windows机器测试

### Linux并发测试

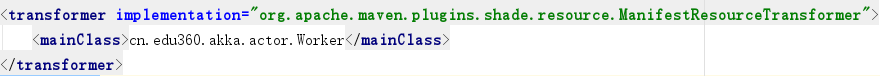
选择设置Master主类，并打成jar包

|  |
| --- |
| <**transformer implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer"**>  <**mainClass**>cn.edu360.akka.actor.Master</**mainClass**> </**transformer**> |



修改jar包名称。

选择设置Worker主类，并打成jar包



上传master jar包到hdp-01机器上

执行启动：

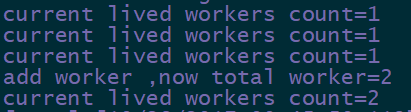


上传WORKER jar包到其他节点上

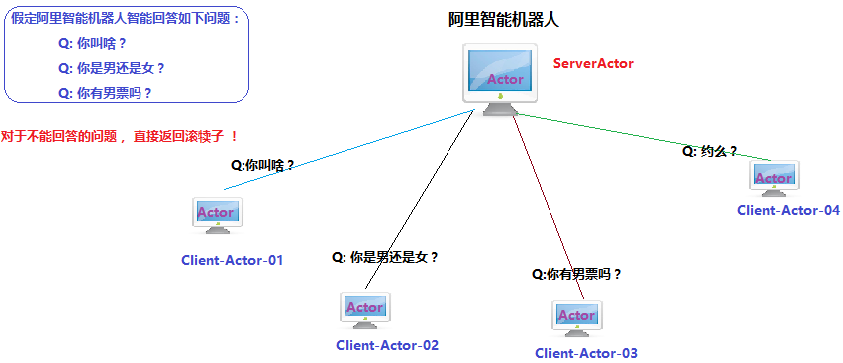
执行启动：注意worker的启动参数



验证：



## 案例4 基于Actor的聊天模型



创建一个Server端用于服务客户端发送过来的问题，并作处理并返回信息给客户端！

创建一个Client端，用于向服务端发送问题，并接收服务端发送过来的消息！

读取键盘输入：val line = StdIn.readLine()