# Scala当中的Actor并发编程

注：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。Akka也实现了类似Scala Actor的模型，其核心概念同样也是Actor

1. Scala中的并发编程
   1. java中的并发编程

java中的并发编程时基于共享数据和加锁的一种机制，即会有一个共享的数据，然后若干个线程去访问这个共享的数据（主要是对整个共享数据进行修改），同时java利用加锁的机制（synchronized）来确保同一时间只有一个线程对共享数据进行访问，以此来保证共享数据的一致性。

Java的并发编程存在资源争夺和死锁等多种问题

死锁：当两个或多个线程获取了某些资源，并且在不先释放自己拥有资源的情况下，反复尝试获取对方的线程所拥有的资源，死锁情况就会出现

* 1. scala中的并发编程

2.1 Scala中的并发编程思想与java中的并发编程思想完全不一样，它是基于事件模型的并发机制，Scala的Actor是一种不共享数据，依赖于消息传递的一种并发编程模式，避免死锁，资源竞争等情况。在具体实现过程中，Scala的Actor会不断的循环自己的邮箱，并通过receive函数进行消息的模式匹配，并进行相应的处理

2.2 如果Actor A和 Actor B要相互沟通的话，首先A要给B传递一个消息，B会有一个收件箱，然后B会不断的循环自己的收件箱， 若看见A发过来的消息，B就会解析A的消息并执行，处理完之后就有可能将处理的结果通过邮件的方式发送给A

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。

1. Scala中的Actor
   1. 什么是Actor模型

在计算机科学领域，Actor模型是一个并行计算（Concurrent Computation）模型，它把actor作为并行计算的基本元素来对待：为响应一个接收到的消息，一个actor能够自己做出一些决策，如创建更多的actor，或发送更多的消息，或者确定如何去响应接收到的下一个消息。

在Actor理论中，一切都被认为是actor，这和面向对象语言里一切都被看成对象很类似。但包括面向对象语言在内的软件通常是顺序执行的，而Actor模型本质上则是并发的，也就是Actor模型是作为一个并发模型设计和架构的。

Actor是一个运算实体，它遵循以下规则:

》接受外部消息，不占用调用方（消息发送者）的CPU时间片

》通过消息改变自身的状态

》创建有限数量的新Actor

》发送有限数量的消息给其他Actor

* 1. Actor方法的执行顺序

》首先调用start()方法启动Actor

》调用star()方法后其act()方法会被执行

》向Actor发送消息

》act()方法执行完后，程序会调用exit()方法

* 1. Actor发送消息的三种方式：

! : 异步发送消息，没有返回值

!! : 发送异步消息，有返回值 ，返回Future[Any]。如果要异步发送一个消息，但是在后续要获得消息的返回值，可以使用Future，即!!语法

!? : 发送同步消息，等待返回值。Scala默认情况下，消息都是以异步进行发送的；但是如果发送的消息是同步的，即对方接受后，一定要给自己返回结果，可以用!?的方式发送消息

注意：Future 表示一个异步操作的结果状态，可能还没有实际完成的异步任务的结果。

Any 是所有类的超类，Future[Any]的泛型是异步操作结果的类型。

1. Akka Actor
   1. Akka简介

Akka 基于 Actor 模型，提供了一个用于构建可扩展的（Scalable）、弹性的（Resilient）、快速响应的（Responsive）应用程序的平台。

Akka 是一个用 Scala 编写的库，用于简化编写容错的、高可伸缩性的 Java 和 Scala 的 Actor 模型应用。它已经成功运用在电信行业、银行和互联网行业。系统几乎不会宕机（高可用性 99.9999999 % 一年只有 31 ms 宕机）。

AKKA 是一款基于actor模型实现的并发处理框架。基于事件驱动的并发处理模型，每一个actor拥有自己的属性和操作，这样就避免了通常情况下因为多个线程之间要共享属性（数据）而是用锁机制的处理。这种机制在scala语言中应用的很好，将操作和属性放在一个独立的单元中进行处理，从而提高并发处理的能力。

* 1. Akka特点

Actor具有如下特性：

1. 提供了一种高级抽象，能够简化在并发/并行应用场景下的编程开发
2. 提供异步、非阻塞的、高性能的事件驱动编程模型
3. 超级轻量级事件处理(每GB堆内存几百万Actor)

容错:  
拥有”let it crash “（让它崩溃）语义的管理层级  
管理层级可跨越多个JVM，实现真正的容错系统（AKKA是基于SCALA的软件框架，SCALA可以运行在JVM上，并且可以完全使用JAVA的类库，这样AKKA也拥有了JAVA跨平台的能力。）  
非常适合编写可自我修复且永不停机的高容错能力的系统

Akka 提供可伸缩的实时事务处理能力。  
Akka在以下方面提供了一致的运行时和编程模型：  
纵向扩展性（并发）  
横向扩展性（远程调用）  
容错性

* 1. Akka中的Actor模型

Actor是Akka中最核心的概念，它是一个封装了状态和行为的对象，Actor之间可以通过交换消息的方式进行通信，每一个actor都有自己的收件箱。通过Actor能够简化锁及线程管理，可以非常容易的开发出正确的并发程序和并行系统。

在Akka中，可以认为一个actor是一个容器，它包含 状态， 行为，信箱（MailBox），子Actor 和 监管策略（Supervisor）。Actor之间并不直接通信，而是通过Mail来互通。所有这些包含在一个ActorReference（Actor引用）里。一个actor需要与外界隔离才能从actor模型中获益，所以actor是以actor引用的形式展现给外界的。

State:actor拥有的一组变量，即actor的state。state是可恢复的，详见Persistence

Behavior:actor处理消息的方法

Mailbox：所有收到的消息，会进入actor的mailbox队列。默认是FIFO。

Children:每一个actor,都是潜在的监控者。actor会自动监控执行子任务的actor.children被放在context中，通过context.actorof(...))或者context.stop(child))操作children.这些操作都是异步的，所以相应非常快。

Supervisor Strategy：actor一旦创建，监控策略是不可修改的。Akka自动帮我们处理错误故障。

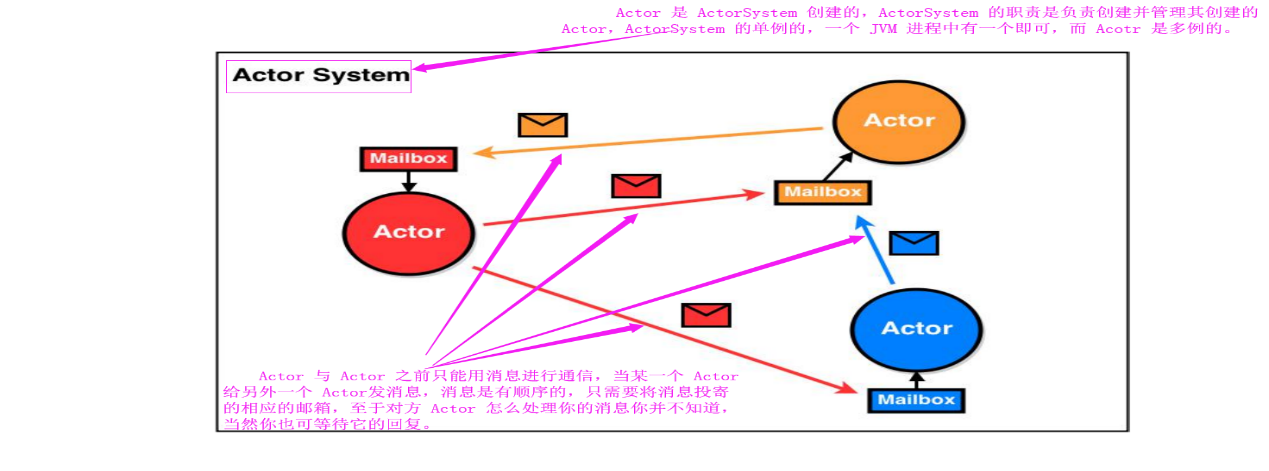
每个Actor都有一个(恰好一个)Mailbox。Mailbox相当于是一个小型的队列，一旦Sender发送消息，就是将该消息入队到Mailbox中。入队的顺序按照消息发送的时间顺序。Mailbox有多种实现，默认为FIFO（默认情况下，邮箱里面的邮件是按照他们先后达到的次序进行阅读和处理的**）**。但也可以根据优先级考虑出队顺序，实现算法则不相同。

JVM中的Actor都有以下特点:

\*每个Actor都有对应一个邮箱

\*Actor是串行处理消息的

\*Actor中的消息是不可变的



Actor模型图

什么是 Actor 的引用？

Actor 引用是ActorRef的一个子类型，其首要目的是支持将消息发送给它所代表的 Actor。每个 Actor 都可以通过self字段访问其规范（本地）引用；默认情况下，对于发送给其他 Actor 的所有消息，此引用也作为发送者引用包含在内。相应地，在消息处理期间，Actor 可以通过sender()方法访问表示当前消息发送者的引用。

为了防止actor对象被直接的访问和操作，所以actor通过ActorRef来间接交流，消息可以通过如下方式传递

! (“tell”) - 发送message并马上返回

? (“ask”) -发送后会等待一段时间，并返回一个future

* 1. Actor层次结构

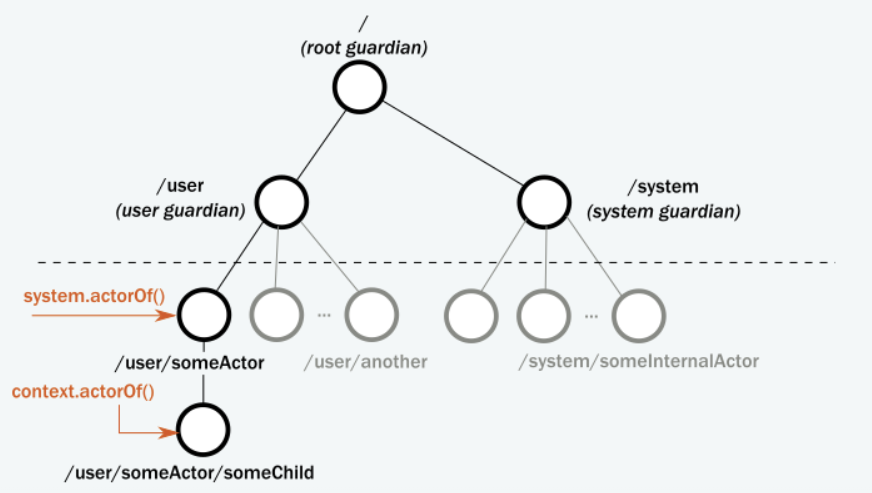
Akka 的 Actor 总是属于父 Actor。通常，您可以通过调用getContext().actorOf()来创建 Actor。与创建一个“独立的（freestanding）” Actor 不同，这会将新Actor 作为一个子节点注入到已经存在的树中：创建 Actor 的 Actor 成为新创建的子 Actor 的父级。如下图所示，所有的 Actors 都有一个共同的父节点，即用户监督者（the user guardian）。可以使用system.actorOf()在当前Actor下创建新的 Actor 实例。创建 Actor 将返回一个有效的 URL 引用例如，如果我们用system.actorOf(…, "someActor")创建一个名为someActor的 Actor，它的引用将包括路径/user/someActor。

事实上，在你在代码中创建 Actor 之前，Akka 已经在系统中创建了三个 Actor 。这些内置的 Actor 的名字包含guardian，因为他们监督他们所在路径下的每一个子 Actor。监督者 Actor 包括：

1)、/，根监督者（root guardian）。这是系统中所有 Actor 的父 Actor，也是系统本身终止时要停止的最后一个 Actor。

2)、/user，监督者（guardian）。这是用户创建的所有 Actor 的父 Actor。不要让用户名混淆，它与最终用户和用户处理无关。使用 Akka 库创建的每个 Actor 都将有一个事先准备的固定路径/user/。

3）、/system，系统监督者（system guardian）。这是除上述三个 Actor 外，系统创建的所有 Actor 的父 Actor，



* 1. Actor监督和监控

supervision表示actors之间的关系。监督者分配任务给下属，因此需要处理反馈的错误。根据不同的错误，监督者supervisor可以做如下操作：

1. 恢复下属，让下属继续运行，继续接收message。当且仅当下属还可以正常运行
2. 重启下属，清空status。一般是第一条情况中，child的异常导致无法正常运行。
3. 关闭下属
4. 关闭自己，向上一级汇报错误
5. 上面的操作都是递归的。

注意: supervision发送的是system messages，用的mailbox也和普通的messages不一样。

由于每个真正的Actor都有一个supervisor，因此，root guardian的supervisor不是一个真正的Actor。

当出现如下三类失败错误时，就可能Restart Actor：

收到特定消息时，发生系统错误，如编程的错误；

在处理消息时，因为一些外部资源的原因出现错误；

Actor的内部状态出现问题

Restart的过程：

暂停Actor（这意味着在Restart期间，不会处理常规的消息，直到它被Resume）。同时，还会递归地暂停所有的children；

调用旧实例的preRestart钩子方法（默认情况下，会发送终止消息给所有children，调用children的postStop()）。

等待所有的children被终止（调用context.stop()）。这个过程是非阻塞的；

通过调用原来提供的工厂去创建新的Actor实例；

调用新实例的postRestart()方法（默认情况下，仍然要先调用preStart()）；

将restart的请求发送给执行第3步时没有被kill掉的children；然后遵循第2步递归地对children执行restart；

resume actor。

* 1. Actor的启动:

Akka中，用ActorSystem来管理所有的Actor，包括其生命周期及交互。

启动Actor，有两种方式：

1. 内置的main方法

akka.Main.*main*(*Array*(*classOf*[HelloWordActor].getName))

1. 创建ActorSystem

**val** system = *ActorSystem*("helloApp")  
system.actorOf(*Props*[HelloWordActor], "helloWordActor")

两种方法本质上其实是一样的，只不过第一种里面把创建ActorSystem等工作封装。

在Akka中，ActorSystem是一个重量级的结构，它需要分配多个线程，所以在实际应用中，ActorSystem通常是一个单例对象（object），我们可以用整个ActorSystem创建很多的Actor，它的职责就是创建并管理其创建的Actor，并且还需要监控Actor。在同一个ActorySystem中，Actor不能重名。

如果一个Actor的业务逻辑非常复杂，为了降低代码的复杂度，可以将其拆分成多个子任务(在一个actor内部可以创建一个或多个actor，actor的创建者也是该actor的监控者)

一个ActorSystem应该被正确规划，例如哪一个Actor负责监控，监控什么等等：

负责分发工作的actor,管理接受任务的actor,因为包工头知道所有任务出错了该怎么解决

拥有重要数据的actor,找出所有可能丢失该数据的子actor,监控，并处理它们的错误。

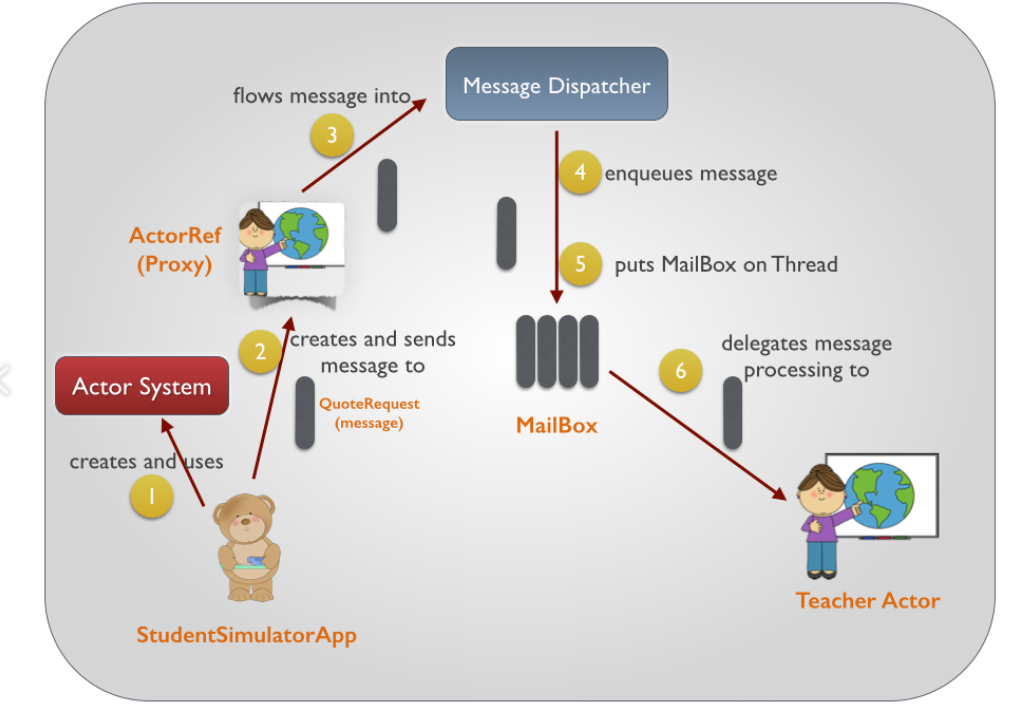
如果一个actor需要其他的actor传递其需求或职责，必将监控之。它应该查看其他actor的存活状态，已经是否完成完成任务，当然这不是监控范畴了，不涉及到监控策略

在上面的例子里，我们使用的是system.actorOf来创建Actor，actorOf返回的并不是Actor自身，而是一个ActorRef，它屏蔽了Actor的具体物理地址(可能是本jvm，也可以是其他jvm或另一台机器)。通过直接打印ActorRef看到Actor的path，比如本例是/app/user/echoActor。  
像这种直接由system创建出来的Actor被称为顶层Actor，一般系统设计的时候，顶层Actor数量往往不会太多，大都由顶层Actor通过getContext().actorOf()派生出来其他的Actor。

* 1. Actor消息传递

以Teacher-Student为例说明消息传递。消息从StudentSimulatorApp单独发送到TeacherActor。

这里所说的StudentSimulatorApp只不过是一个简单的主程序。



这副图片解释如下：  
　　1.Student创建了一些东西，称为ActorSystem；  
　　2.用ActorSystem来创建一个ActorRef，并将QuoteRequest message发送到ActorRef 中（到达TeacherActor的一个代理）；  
　　3.ActorRef 将message单独传输到Dispatcher；  
　　4.Dispatcher将message按照顺序保存到目标Actor的MailBox中；  
　　5.然后Dispatcher将Mailbox放在一个Thread 中；  
　　6.MailBox按照队列顺序取出消息，并最终将它递给真实的TeacherActor接受方法中。

5.1、创建ActorSystem

**val** *actorSystem* = *ActorSystem*("MessagesSystem")

ActorSystem是进入Actor世界的切入点，通过ActorSystem你可以创建和停止Actors，甚至关掉整个Actor环境！  
　　另一方面，Actor是一个体系，ActorSystem类似于java.lang.Object or scala.Any，能够容纳所有的Actor，它是所有的Actor的父类。当你创建一个Actor，你可以用ActorSystem的actorOf方法。

ActorSystem的配置: 当我们创建ActorSystem，用的是ActorSystem对象的apply方法，而该并没有注明任何的配置，它将在classpath中寻找 application.conf、 application.json和application.properties，并自动的加载他们。所以上述方式与下等同:

ActorSystem("UniversityMessagingSystem", ConfigFactory.load())

5.2、创建ActorRef  
**val** *teacherActorRef*: ActorRef = *actorSystem*.actorOf(*Props*[TeacherActor], "teacherActor")

actorOf是ActorSystem中创建Actor的方法。但是正如你所看到的，它并不返回我们所需要的TeacherActor对象，它的返回类型为ActorRef。

ActorRef为真实Actor的充当代理，客户端并不直接和Actor进行通信。这就是Actor Model中的处理方式，该方式避免直接进入TeacherActor或者任何Actor中的任何custom/private方法或者变量。

5.3、将message 发送到ActorRef(代理)中

*teacherActorRef* ! QuoteRequest

QuoteRequest是用来给TeacherActor发送消息的；而Actor将会用QuoteResponse来响应。

5.4、ActorRef取出消息并放到Dispatcher中

当我们创建了ActorSystem 和ActorRef,Dispatcher和MailBox也将会创建

MailBox: 每个Actor都有一个MailBox（后面我们将看到一个特殊情况）。在我们之前的模型中，每个Teacher也有一个MailBox。Teacher需要检查MailBox并处理其中的message。MailBox中有个队列并以FIFO方式储存和处理消息。

Dispatcher：Dispatcher做一些很有趣的事。从图中可以看到，Dispatcher好像只是仅仅将message从ActorRef 传递到MailBox中。但是在这背后有件很奇怪的事：Dispatcher 包装了一个 ExecutorService (ForkJoinPool 或者 ThreadPoolExecutor).它通过ExecutorService运行 MailBox

5.5、Actoc取出消息receive方法处理

 当MailBox的run方法被运行，它将从队列中取出消息，并传递到Actor进行处理。该方法最终在你将消息tell到ActorRef中的时候被调用，在目标Actor其实是个receive 方法。receive 方法是用来处理消息的

为了更好的控制多线程，JDK提供了一套线程框架Executor，帮助开发人员有效地进行线程控制。它们都在java.util.concurrent包中，是JDK并发包的核心。ExecutorService是Executor直接的扩展接口，也是最常用的线程池接口，我们通常见到的线程池定时任务都是它的实现类。Java API堆ExecutorService接口的实现有两个（ThreadPoolExecutor，ScheduleThreadPoolExecutor）

* 1. Actor生命周期

Actor 在被创建时就会出现，然后在用户请求时被停止。每当一个 Actor 被停止时，它的所有子 Actor 也会被递归地停止。这种行为大大简化了资源清理，并有助于避免诸如由打开的套接字和文件引起的资源泄漏。事实上，在处理初级多线程代码时，一个通常被忽视的困难是各种并发资源的生命周期管理。

在Akka中，Actor负责通信，在Actor中有一些重要的生命周期方法

1.preStart()方法：该方法在Actor对象构造方法执行后执行，整个Actor生命周期中仅执行一次。在 Actor启动之后但在处理其第一条消息之前调用

2.receive()方法，该方法在Actor的preStart()方法执行后执行，用于接受消息，会被反复执行。

3.postStop()在 Actor 停止之前调用，在此时之后将不再处理任何消息

* 1. 错误恢复

父 Actor 和子 Actor 在他们的生命周期中是相互联系的。当一个 Actor 失败（抛出一个异常或从接收中冒出一个未处理的异常）时，它将暂时挂起。如前所述，失败信息被传播到父 Actor，然后父 Actor 决定如何处理由子 Actor 引起的异常。这样，父 Actor 就可以作为子 Actor 的监督者（supervisors）。默认的监督策略是停止并重新启动子 Actor。如果不更改默认策略，所有失败都会导致重新启动。

1.会有一个Actor池，每个Actor会处理不同的事件。  
　　2.Actor做的事情可能会抛出异常，而它自己无法从中恢复。在这种情况下，需要再生成（created ）一个新的Actor来顶替它。换句话说，这个新的Actor会忽略刚才那条消息，继续处理剩余的消息。这些也被称为指令（Directive），后面我们会再讲到它们。

一旦actor意外终止，比如失败了。它会释放所有资源，将未处理的messages发送给system的dead letter mailbox.这些message将由EventStream处理。

* 1. Actor日志

1.[Akka](https://www.iteblog.com/archives/tag/akka/)提供了一个非常小的trait 来打印日志，称为 ActorLogging。当我们以日志记下来一条消息时，ActorLogging 中的logging 方法已经将该消息publishes（发布）到了EventStream。

EventStream就像一个我们用来发布及接受消息的代理。它与常见的消息中间件的根本区别是EventStream的订阅者（subscribers）只能是Actor。默认情况下，订阅这些消息的Actor是DefaultLogger ，它只是简单的将消息打印到标准输出。(Akka提供的默认日志系统只输出到控制台，这种日志系统不可以用到产品环境，当然你可以整合SLF4J这样的日志系统，下面介绍如何在Akka中使用Logback记录日志)

class DefaultLogger extends Actor with StdOutLogger {

    override def receive: Receive = {

        ...

        case event: LogEvent ⇒ print(event)

    }

}

EventStream不仅仅用来打印日志。它是Actor在同一个虚拟机内的一个通用的public-subscribe（发布-订阅）机制

2.配置Akka来支持SLF4J

在application.conf中配置如下信息:

akka{

    loggers = ["akka.event.slf4j.Slf4jLogger"]

    loglevel = "DEBUG"

    logging-filter = "akka.event.slf4j.Slf4jLoggingFilter"

}

1、loggers属性指定订阅日志的Actor。 Slf4jLogger所做的仅仅是消费日志，并委托给slf4j日志接口去处理。  
　　2、loglevel 属性配置日志打印的最小级别。（OFF, ERROR, WARNING, INFO, DEBUG）  
　　3、logging-filter会将配置的loglevel和传进来的日志消息的级别进行比较，把低于loglevel的日志都给过滤掉，然后发布到EventStream中。

在application.conf中除了日志参数，还有许多很棒的参数以供使用。这里是一个[详细的说明](http://doc.akka.io/docs/akka/2.0/scala/logging.html" \t "_blank)。

* 1. Actor测试

Akka-testkit 是 Akka 官方推出的 Akka 测试工具包，用于减轻 Akka 程序的测试难度。

编写单元测试基本范例如下:

class TeacherPreTest extends TestKit(ActorSystem("testSystem"))

with WordSpecLike

with MustMatchers

with BeforeAndAfterAll {

"A actor" must {

"actors in this way" in {

…………

}

}

}

Akka-testkit 的主要工具包括,

1. testProbe 用于测试被测 Actor 回应和发送消息

对于被测 Actor 是否正确地发出消息，也可以用 testProbe 测试。首先将 testProbe 设置为被测 Actor 发出消息的目标，然后让被测 Actor 发出消息，再看 testProbe 是否接受到期望的消息

"A teacher" must {  
 "reponse a correct answer test" in {  
 //将 testProbe 设置为被测 Actor 发出消息的目标  
 **val** otherActor = *TestProbe*()  
 **val** helloACtor = *system*.actorOf(*Props*(**new** HelloACtor(otherActor.ref)))  
 helloACtor ! "Add"  
 //测试是否收到预期消息  
 otherActor.expectMsg("is continue ?")  
 }  
}

对于被测 Actor 是否正确地回应消息，可以用 testProbe 测试。首先将 testProbe 给被测 Actor 发送消息，再看 testProbe 是否接受到期望的回应消息。

"A teacher" must {  
 "reponse a correct answer" in {  
 //首先将 testProbe 给被测 Actor 发送消息，再看 testProbe 是否接受到期望的回应消息。  
 **val** testProb = **new** TestProbe(*system*)  
 **val** helloACtor = *system*.actorOf(*Props*[HelloACtor])  
 testProb.send(helloACtor, "Hello")  
 //测试是否收到预期消息  
 expectMsg("Hi")  
 }  
}

1. testActor 用于简便情况下测试被测 Actor 回应消息

除了使用 testProbe 之外，Akka-testkit 还提供一种简便方法: 使用 testActor。 如果测试类实现特质 ImplicitSender，就可以直接使用。使用 testActor 的代码比使用 testProbe 的简便。但是，一个东西的用法越是简便，功能便越缺失。testActor 最大的缺失是只能接受被测 Actor 发来的一个回应消息。

"A teacher" must {  
 "reponse correctly" in {  
 **val** helloACtor = *system*.actorOf(*Props*[HelloACtor])  
 //从testActor发来  
 helloACtor ! "Hello"  
 //测试是否符合预期  
 expectMsg("Hi")  
 }  
}

1. testActorRef 用于测试被测 Actor 内部状态的改变

对于被测 Actor 内部状态的改变，可以用 TestActorRef 进行测试。TestActorRef.underlyingActor 可以探测被测 Actor 的内部，用于测试被测 Actor 内部状态是否符合预期

"A teacher with ActorLogging" must {  
 "have a quote list of size 4" in {  
 **val** teacherLogRef = *TestActorRef*[TeacherLogActor]  
 //通过underlyingActor 方法进入到Actor的内部  
 teacherLogRef.underlyingActor.quoteListList must *have* size (4)  
 }  
}

1. 使用Akka的优势

1、AKKA提供一种Actor并发模型，其粒度比线程小很多，这意味着你可以在项目中使用大量的Actor。

2、Akka提供了一套容错机制，允许在Actor出错时进行一些恢复或者重置操作

3、AKKA不仅可以在单击上构建高并发程序，也可以在网络中构建分布式程序，并提供位置透明的Actor定位服务

任何系统只要是需要高吞吐量或低延迟率的都可以考虑使用Akka。

1. 参考网页
   * 1. <https://doc.akka.io/docs/akka/current/guide> Akka官网
     2. <https://blog.csdn.net/moshang_3377/article/details/92651497> scala学习第四篇之Actor&AKKA实战（线程）
     3. <https://blog.csdn.net/aimomo007/article/details/78814686#akka%E5%AE%98%E7%BD%91> Akka学习笔记
     4. <https://www.iteblog.com/archives/1154.html> Akka系列文章
     5. http://doc.akka.io/docs/akka/snapshot/scala/mailboxes.htm 包括了有界和基于优先级的MailBox实现
     6. <https://blog.csdn.net/lovezhaohaimig/article/details/80358471> ExecutorService深入理解
     7. <https://www.cnblogs.com/lixuwu/p/7979480.html> 多线程ForkJoinPool
     8. <https://www.cnblogs.com/zengbiaobiao2016/p/5922957.html> Akka使用Logback日志框架
     9. <https://doc.akka.io/docs/akka/2.0/scala/logging.html> Akka actor logging
     10. <https://blog.csdn.net/wsscy2004/article/details/38421399> Akka TestKit的使用
     11. <http://www.algorithmdog.com/akka-test> Akka测试
     12. <http://www.algorithmdog.com/akka-hierarchy-fault> akka层次结构
     13. <https://blog.csdn.net/qq_35246620/article/details/87003335> akaka的引用，路径，地址
2. 案例