# 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. 不使用scala actor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. 使用scala actor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。

在模型中，程序由大量独立执行计算并发送消息通信的实体代表，这些实体就被称之为Actor

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

Actor模型的目标是通过彻底消除共享内存需求，解决与共享内存有关的问题，如数据争用和同步化。可变状态被限定在Actor模型的作用域内，而在Actor模型接受到消息时，可变状态有可能会被修改。Acotr模型收到的消息会以串行的方式执行，一个接一个地被处理，这确保了Actor模型的可变状态，永远不会以并发方式被访问。然而，独立的Actor模型可以通过并发方式处理收到的消息。在典型的基于Actor模型的程序中，Actor对象的数量可以比处理器的数量高出多个数量级。这与多线程程序中，处理器数量与线程数量的关系类似。Actor模型可以决定何时向指定Actor对象分配处理器时间，以便使这些对象能够处理消息。

* 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）而不是用锁机制的处理。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 Strategy）。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。

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

行为:actor处理消息的方法

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

子actor: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。

监管策略：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。

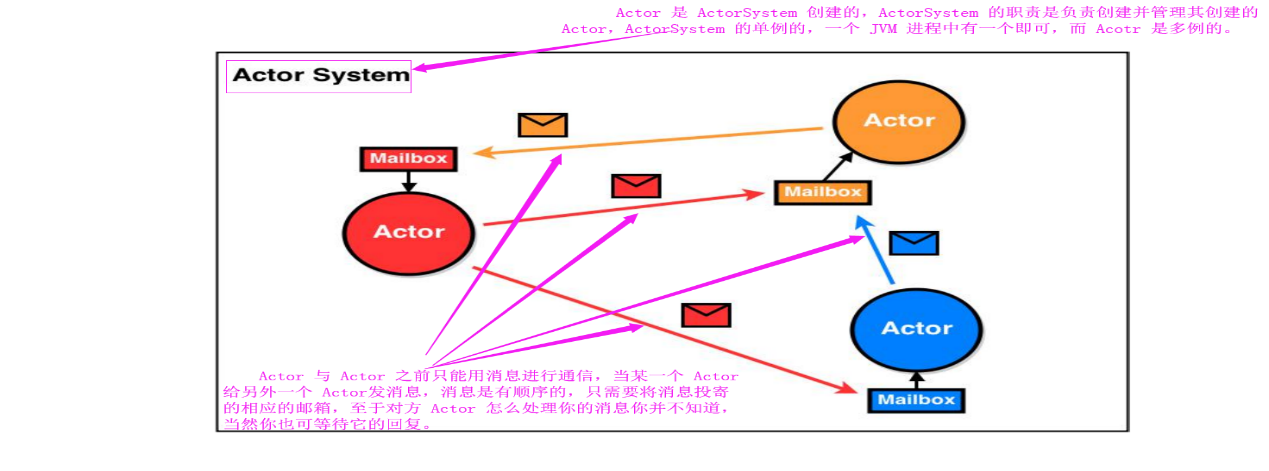
Actor引用: Actor引用是一种对象，使用这种对象可以向指定的Actor实列发送消息。Actor引用会对程序员隐藏Actor实例存储位置的信息。Actor 引用是ActorRef的一个子类型，其首要目的是支持将消息发送给它所代表的 Actor。每个 Actor 都可以通过self字段访问其规范（本地）引用；默认情况下，对于发送给其他 Actor 的所有消息，此引用也作为发送者引用包含在内。相应地，在消息处理期间，Actor 可以通过sender()方法访问表示当前消息发送者的引用。

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

* 1. 使用Actor模型

在Actor编程模型中，程序通过一系列并发方式执行的实体(Actor)执行。Actor模仿了人类的组织，如公司，政府，机构。下面通过介绍Actor系统与人类组织的相似之处，帮助理解。

Actor系统:Actor系统是一个具有层次结构的团体，在这个团体中每个角色都拥有通用的配置选项，Actor系统负责创建新的Actor对像，在Actor系统定位Actor对象和记录重要事件。就是一个软件公司的缩影

Actor类:用于描述角色内部状态和角色处理消息方式的模板。Actor类设置了Actor实例的行为，设置Actor实例回应消息和与其他Actor实例通信的方式，还封装了Actor实例所用对象的引用，这些对象代表了Actor实例的状态，定义了运行和关闭Actor实例的次序。通过同一个Actor类可以创建多个Actor实例。Actor类就像软件公司中的特定岗位，如软件工程师，经理和招聘专员。

Actor实例:运行程序时存在的实体，而且它们能接受消息。Actor实例会处于特定的可变状态并且能够向其它Acotr实例发送消息。Actor实例就像一个具体的员工

消息:一种通信数据单位，Ator对象使用消息进行通讯。在Akka中，任何对象都可以当作消息。消息就像软件公司员工发送的电子邮件。当Actor对象发送消息后，它会继续执行其他操作，而不会等到其他Actor对象收到该消息后才重新开始执行操作。与此类似，当员工发送邮件后，他会做其他工作，而不会等待其他员工收到这封邮件后才重新开始工作。员工由于工作繁忙而无法处于等待状态，因此，多封邮件可能会以并发方式发送给同一个员工。

邮箱:就像一块用于缓存消息的内存区域，每个actor实例都会有意块这样的专属区域，这个缓存区域是必要的，因为Actor实例同一时刻只能处理一条消息。在任意的时间点，电子邮箱中可能包含多封未读电子邮件，而且同一时刻一位员工只能阅读和回复一条消息。

Actor引用: Actor引用是一种对象，使用这种对象可以向指定的Actor实列发送消息。Actor引用会对程序员隐藏Actor实例存储位置的信息。Actor的实例可以在多个独立进程和多台计算机中运行。使用Actor引用可以将消息发送到运行Actor实例的位置。以软件公司为例，Actor引用就像员工的邮箱地址。通过邮件地址可以在无须知道员工具体位置的情况下，将电子邮件发送成功。员工可能在办公室，上班的路上，也可能在休假，但无论员工在哪，都能收到消息。

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

分配器：分配器是一种组件，它用于决定何时允许Actor实例处理消息，并为Actor实例分配相应的计算资源。在Akka框架，所有分配器同时也都是执行上下文。分配器确保了消息缓冲区非空的Actor实例，最终会由指定的线程运行，而且这些消息会以串行方式被处理。分配器就像软件公司中回应电子邮件的规章制度。一些员工(如技术支持)被要求在接到电子邮件时立刻做回复。软件工程师有更多的自由，可以先做别的事，然后再回复电子邮件。

* 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：

内置的main方法

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

创建ActorSystem

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

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())

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

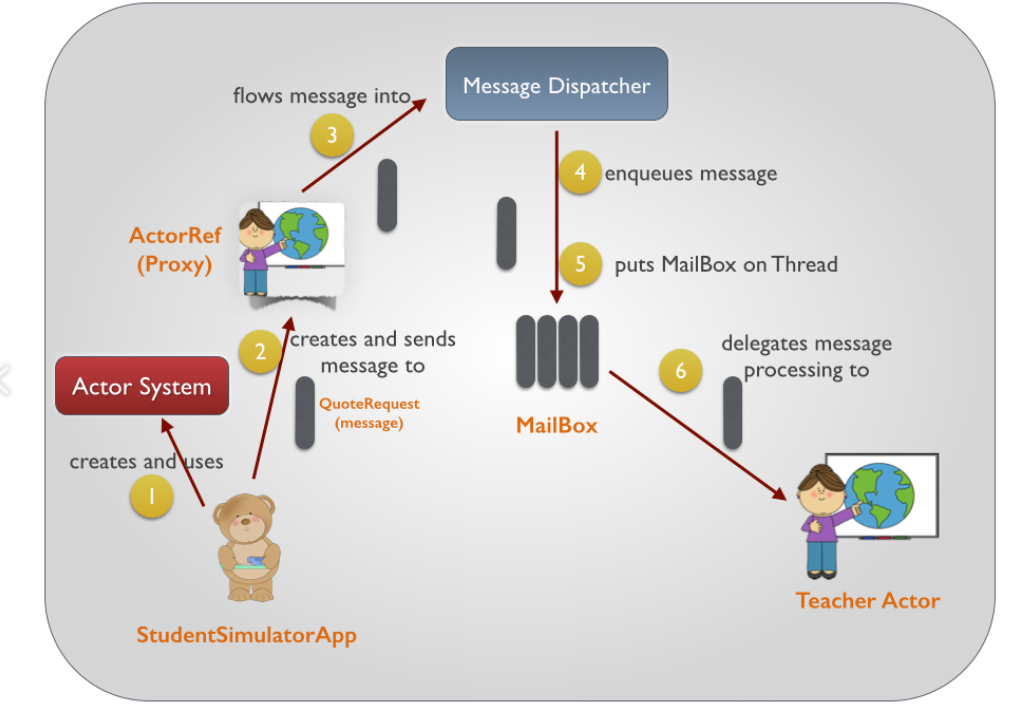
拥有重要数据的actor,找出所有可能丢失该数据的子actor,监控，并处理它们的错误。

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

* 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。

#### 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。

#### 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。

#### 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。

#### 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)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层次结构

与公司的组织层次类似，一系列Actor对象也能构成层次结构，层次较高的Actor对象负责处理通用性较高的任务，层次较低的Actor对象负责处理较为专业化的任务。这个层次结构模式非常适用将复杂的程序分解为简单的组件。在使用Actor对象的情况中，更具Actor对象之间的关系，选择正确的层次结构组织Actor对象，可以降低组件之间的耦合性，并且使替换程序中的失效组件更加容易。

在Akka框架中，Actor对象拥有隐式的层次结构。任何Actor对象都能够拥有一定数量的子Actor对象，而且父Actor对象可以使用context对象构建子Actor对象和停止子Actor对象的运行过程。

在AKka框架中，层次较高的Actotr对象被称为守护者Actor（guardian actor），它们负责处理各种内部任务，记录用户Actor对象的日志和重启用户Actor对象。所有在应用程序中创建的根Actor对象，都处于预定义的守护者Actor对象之下。这是另一种守护者Actor对象。

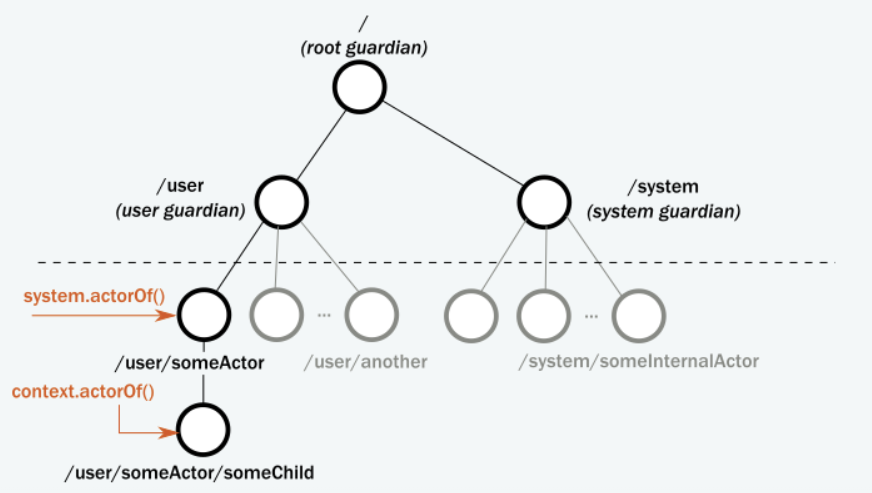
事实上，在你在代码中创建 Actor 之前，Akka 已经在系统中创建了三个 Actor 。这些内置的 Actor 的名字包含guardian，因为他们监督他们所在路径下的每一个子 Actor。监督者（守护者） Actor 包括：

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。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。根监督者在顶级范围内使用SupervisorStrategy.stoppingStrategy，它的目的是终止所有子集的所有异常。一个综合的实体ActorRef 会在它的子集出现麻烦时关闭，并设置Actor系统isTerminated状态为true然后根监督者将完全终止（所有子集递归停止）

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/。使用system.actorOf()方法创建的Actor都是它的children。这意味着只要这个Actor终止了，系统中所有常规的Actor都会被关掉。在Akka 2.1中，可以设置Supervisor Strategy，配置项为akka.actor.guardian-supervisor-strategy，对应类名为SupervisorStrategyConfigurator。倘若这个Guardian Actor扩大了失败，按照前面描述的Supervisor策略，它会使得root guardian终止该Actor，从而使得这个Actor下的所有子Actor都停止，即关掉了整个Actor系统

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，这个监督者是为了实现有序关闭序列日志，在所有普通Actor终止时日志仍是活跃的，尽管日志是由Actotr使用。它可以监控User Guardian，可以管理Top-Level的System Actor采用一种策略，可以在除了ActorInitializationException和ActorKilledException外的异常出现时. 无限制地重启它。

可以通过调用context().actorOf()来创建 Actor，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。



* 1. 识别Actor对象（Actor路径）

Actor对象是通过树形层次结构组织到一起的，在这种层次结构中，每个Actor对象都有父Actor对象，有些Actor对象还有子Actor对象。因此，在这种层次结构中每个Actor对象都有到达根Actor对象的唯一路径，而且能够被该路径中的唯一性的部分Actor对象名称识别。

Actor路径由Actor系统的名称，和从根守护者Actor对象到特定Actor对象的名称构成。与文件系统的文件路径非常相似，Actor路径也同样专门指定了特定Actor对象在Actor层次结构中的位置。确切的说，Actor路径是用于获取Actor对象引用的标识符。

Actor路径中的某些Actor对象名称也可以用通配符和..符号替换，这样就获得了路径选择。

例如: ..可以引用当前Actor对象的父对象。路径选择../\*可以引用当前Actor对象和该对象的所有同级对象。

Actor路径和Actor引用不同，无法通过Actor路径向Actor对象发送消息，但在发送消息前，必须使用Actor路径识别Actotr对象，通过Actor路径成功识别Actor对象后，才能向Actor对象发送消息。

要获取Actor路径标识的Actor对象的引用，可调用该对象中context对象中actorSelection方法，该方法能够接受Actor路径和路径选择。若标识的路径不存在，会调用0个Actor对象，否则会调用多个Actor对象。actorSelection方法不会返回ActorRef对象，而会返回0个，1个或多个Actor对象的ActorSelection对象。可以使用ActorSelection对象向这些Actor对象发送消息。（使用context对象中actorSelection，可以与Actor系统中的任意Actotr对象通信）

如果将ActorRef(Actor引用)对象比作具体的电子邮箱地址，那个可以将ActorSelecttion对象比作电子邮箱列表。将一封电子邮件发送给合法的电子邮件地址，能够确保这封电子邮件会被送给使用邮箱的人。另一方面，当向邮箱地址列表中的电子邮件地址发送电子邮件时，根据邮箱地址列表包含的电子邮箱地址数量，这封电子邮件可能会被发送给0个或多个人。

ActorSelection对象不会反应具体的Actor路径，这与邮箱地址列表将电子邮箱地址封装起来类似。因此，Akka框架定义了特殊的消息类型Identify，当Akka框架中的Actotr对象收到Identify消息时，就会自动发送一条带有ActorRef对象的ActorIdentify消息。如果路径选择中不存在Actor对象，这条ActorIdentify在发送给Identify消息的发送者时就不会带有ActorRef对象。向ActorSelecttion对象发送Identify消息，可以获取Actor系统中任意Actor对象的引用。

使用actorSelection方法和Identify消息，是在同一Actor系统中查找未知Actor对象的基本方式。

* 1. Actor生命周期

要理解Actotr生命周期的重要性，可先思考如果Actor对象在处理消息时抛出异常，会出现怎样的情况。在Akka框架中，这类异常被视为非正常行为，因此默认情况下，抛出异常的根用户Actor对象会被重启。重启操作会创建一个全新的Actor对象，实际上意味着Actotr对象的状态也会被重新初始化。当Actotr对象被重启时，它的引用和Actor的路径会保持不变。因此，在逻辑上同一个Actor对象存在的时间段内，同一个ActorRef对象可能会引起在物理上不同的多个Actor对象。这也是绝不允许泄露Actor对象的this引用的原因之一。这样程序的其他部分就能够引用同一个Actor对象，从而实现Actor对象的透明性。此外，泄露Actor对象的this引用会泄露Actor对象的内部实现代码，会导致数据损坏。

Actor 在被创建时就会出现，然后在用户请求停止时被停止

通过调用actorOf方法，可以创建逻辑Actor实例。

使用Props对象可以实例化物理Actor对象，这样Actor对象就会被赋予消息缓冲区，并开始接受处理消息。

actorOf方法会向调用者线程返回Actor对象的引用，而且多个Actor对象能够以并发方式运行。

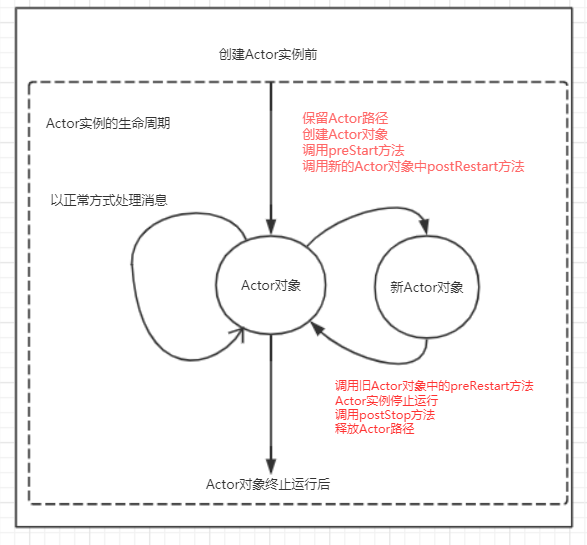
在Actor处理消息前，它的preStart方法就会被调用，preStart方法用于初始化逻辑Actor实例。

Actor对象被创建后，就开始处理消息。

遇到异常的情况时，Actor对象有时需要重启，当出现这种情况时，preRestart方法会被调用。重启的Actor对象的所有子Actor对象会被停止运行过程。然后，用于通过actorOf方法创建的Actor对象的Props对象会再次被使用，以创建新的Actor对象。新建的Actor对象中的postRestart方法会被调用。postRestart方法返回结果后，新的Actor对象就会被赋予与旧Actotr对象相同的消息缓冲区，并继续处理旧Actotr对象重启前缓存的消息。

默认情况下，postRestart方法调用前会调用preStart方法，有时需要重写这个行为。

一旦出现需要停止运行逻辑Actor实例的情况，就应该调用postStop方法。与该对象关联的Actor路径会被释放，并被返回Actor系统。默认情况下，preRestart方法会调用postStop方法



* 1. Acotor通讯

\* 自询目标模式: ！是一种非阻塞操作符，发送消息不会阻塞发送者线程，这种方式被称为自寻目标模式。因为这种模式无须等待消息接受者的回复，但也无法确保接受者是否会收到消息。通过这种方式可以提高Actor对象构建系统的吞吐量，但也会受限于某些情况。例如，有时需要在发送消息后等待接受者的回复。虽然自寻目标模式无法确保接受者能够收到消息，但它能确保发送者至多仅会发送消息一次，而且，这种模式海能确保按指定的Actor发送者和接受者对，对消息进行排序。（Actor对象B从Actor对象A接受消息的次序与Actor A发送消息的次序相同）。

\* 请求模式: 不在Actor对象内部使用阻塞机制，实现请求-回应通信模式。在这种通信模式中，想要获取某类消息的Actor对象，会向另一个Actor对象发送请求消息，该Actor需要等待另一个Actor对象发送的回应消息，Akka框架将这种通信模式成为请求模式。？操作符，会像接线员一样，将消息发送给目标Actor对象，此外，？操作符还会通过目标Actor对象的回应，返回一个Future对象。使用？操作符需要隐式的TimeOut限定时间，如果在限定时间内没有收到回应消息，那么Future对象就会因为AskTimeOutException异常而失效。

\* 转发模式: 某些Actor对象的作用仅仅是为其他Actor对象转发消息。例如，有些Actor对象专门负责多个Actor对象负载均衡，有些Actotr对象会将消息转发给它们的镜像对象，以确保更好的可用性。在这些情况中，Actor对象引用中的forward方法可以实现目标。转发模型通过用于路由器Actor对象，复制器Actor对象和负载均衡Actor对象。路由器 会根据指定的规则确定详细的目的地；复制器Actor对象用于将消息发送到多个目的地。负载均衡Actor用于确保多个Actor对象以负载均衡的方式工作。

* 1. Actor的停止

1.通过ActorSystem.shutdown方法停止所有 Actor的运行。从2.4版本开始不推荐使用此方式，使用ActorSystem.terminate方式停止

2.通过context.stop方法停止Actor的运行，它会使Actor对象在处理完当前消息后立刻停止运行。

3. 通过akka.actor.PoisonPill消息停止Actor的运行，与stop效果相同,但允许Actor处理完收到PoisonPill方法前存储在缓冲区中的消息

4. gracefulStop方法可以停止Actor的运行。akka.pattern的gracefullStop方法能够在接受到Actor对象引用，超时和关闭类型的消息，该方法会返回一个Future对象并以异步方式向Actor对象发送关闭消息。

在某些情况下，仅使用PosisonPill消息不足以达到目的。Actor对象在结束自己的运行过程前，可能不得不等待其他Actor对象停止运行。Akka框架提供了停止情况监控（Death Watch）模式，使用该模式可以令Actor对象能够跟踪其他Actor对象的停止情况。

当需要从Actor对象内部跟踪停止Actor对象时，可以使用停止监视模式。当需要在Actor外部等待Actor对象停止时，可使用正常停止模式(graceful stop pattern)。

通常在Actor对象内部使用停止情况监视模式，在应用程序的main线程中使用正常停止模式。只要能确保由gracefulStop方法返回的Fture对象中的回调函数不会捕捉Actor状态，那么也可以在Actor对象内部使用正常停止模式。

* 1. Acotor监督策略

在Akka框架中，所有的Actor对象都是其子Actor对象的监督者，当某个子Actor对象失效时，它会暂停处理消息，并向它的父Actor对象发送一条消息，以便决定在失效后执行哪些操作。在子Actor失效后，对父Actor对象和子Actor对象应用的处理策略称为监督策略。有两种基本的监管策略：OneForOneStrategy和AllForOneStrategy，前者意味着你只会处理你子Actor中的错误，并只会影响到出错的那个子Actor，而后者会影响到所有的子Actor。

父Actor对象可以选择执行下列操作(指令Directive)：

\*通过发送Restart消息，重启子Actor （子Actor异常导致无法正常运行）

\*通过发送Resume消息，在不重启子Actor对象的情况下恢复子Actor对象 (恢复子Actor，让子Actor继续运行，继续接受message，当且仅当子Actor还可以正常运行)

\*通过发送Stop消息，永久停止Actor对象 (关闭子Actor)

\*通过发送Escalate消息，使用同一个异常使本身也失效。（关闭自己，向上一级汇报错误）

默认情况下，用户级守护者Actor对象会自带重启失效子Actor对象的监督策略。用户Acotr对象在默认情况下会重启它们子Actor对象。

final val defaultStrategy: SupervisorStrategy = {

def defaultDecider: Decider = {

case \_: ActorInitializationException ⇒ Stop

case \_: ActorKilledException ⇒ Stop

case \_: Exception ⇒ Restart

}

OneForOneStrategy()(defaultDecider)

}

要重写监控策略，可以重写Actor类中的supervisorStrategy

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. 远程Actor通讯

Akka的设计目标就是为分布式准备的，那么如何进程远程actor之间的通讯

实现远程Actotr通讯的主要步骤如下:

>>使用适当的配置声明Actor系统

>>在不同的进程或计算机中运行两个Actor系统

>>使用Actor选择获取Actor对象的引用

>>使用Actor对象的引用 ，以透明方式发送消息





* 1. Actor日志

当Actor实例收到无法由其Actor处理的消息时，该消息会被存储在UnhandledMessage对象中并转发给Actor系统的事件流(EventStream)，通常，Actor系统的事件流负责记录日志

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。。



* 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. 扩展
   1. Dispatcher

1.1、DisPatcher的理解:

DisPathcher负责调度并行任务，负责将工作分配给Actor，还可以分配资源用于处理Future的回调。在Akka中为了保证消息处理的及时性和线程的使用效率，DisPatcher对线程做了一些协调工作。简单来说，dispatcher相当于一个消息控制中心，负责对消息进行分发和派送。合理的运用Dispatcher可以提升系统的吞吐量。

Akka的MessageDispather是维持Akka actor运行的核心，它是整个机器的引擎，所有的MessageDispather实现同样也是一个ExecutionContext，这意味着它可以用来执行任何代码。如Future。

DisPatcher基于Executor（线程池），先介绍下Executor。

线程池： 提供一个线程队列，队列中保存着所有等待状态的线程。避免了创建与销毁的额外开销，提高了响应的速度。

线程池的体系结构：

java.util.concurrent.Executor 负责线程的使用和调度的根接口

|--ExecutorService 子接口： 线程池的主要接口

|--ThreadPoolExecutor 线程池的实现类

|--ScheduledExceutorService 子接口： 负责线程的调度

|--ScheduledThreadPoolExecutor : 继承ThreadPoolExecutor，实现了ScheduledExecutorService

Executor提供了异步任务的策略分为两种:ForkJoinPool,ThreadPool

|  |  |
| --- | --- |
| **线程池** | **描述** |
| thread-pool-executor | 有一个工作队列，队列中包含了分配给各个队列的工作，线程空闲时就从队列中认领工作，允许线程重用，减少线程分配和销毁的开销 |
| fork-join-executor | 使用一种分治算法，递归地将任务分隔为更小地子任务，然后把子任务分配给不同地线程运行，最后将运行结果组合起来。当某线程的任务队列没有任务时，会主动从其它线程的队列中获取任务，fork-join性能更佳，Akka的默认选项 |

1.2、默认分配器：

在没有为Actor配置的情况下，每一个ActorSystem都有一个默认的分配器。默认分配器是可配的，默认情况下是一个指定的default-executor的Dispatcher。如果通过传递ExecutionContext来创建ActorSystem，在ActorSystem中，这个ExecutionContext将作为所有分配器的默认执行者。

如果没有提供ExecutionContext，将回退到akka.actor.default-dispatcher.default-executor.fallback的executor，默认情况下是”fork-join-executor”,在大多数情况下给予非常高的性能。



1.3、DisPatcher的分类:

\* Dispatcher: 默认的Dispatcher类型，使用定义的Executor在Actor中处理消息，大多情况下可以提供最好的性能。这是一个基于事件的调度程序，它将一组Actor绑定到线程池，如果未指定分配器，则使用默认分配器。每个Actor都有自己的邮箱，分配器可以被任意数量的Actor共享，分配器可以由线程池或fork连接池支持，此调度器针对非阻塞代码进行了优化。

》可共享性：Unlimited

》邮箱：任意，为每个 Actor 创建一个

》用例：默认调度器，Bulkheading

》驱动：java.util.concurrent.ExecutorService使用fork-join-executor、thread-pool-executor，

或akka.dispatcher.ExecutorServiceConfigurator的FQCN指定的executor。

\* PinnedDispatcher: 这个分配器为每个使用它的Actor指定唯一的线程；即每个Actor将拥有自己的线程池，池中只有一个线程。这个分配器针对阻塞操作做了优化。（给每个Actor分配独有的线程，为每个Actor都创建一个ThreadPool Executor，每个Executor中包含一个线程，可以在单个Actor处理很重要工作的时候使用这种Dispatcher,否则不推荐使用，适合高优先级actor，为actor使用独立的线程）

》可共享性：None

》邮箱：任意，为每个 Actor 创建一个

》用例：Bulkheading

》驱动：任何akka.dispatch.ThreadPoolExecutorConfigurator。默认情况下为thread-pool-executor。

\* CallingThreadDispatcher: 此分配器仅在当前调用的线程上运行，这个分配器不创建任何新的线程，但是它可以从不同的线程并发地用于同一个Actor。（不创建Executor，在发起调用地线程上执行工作，主要用于测试或者调试。[使用参考网页15]）

》可共享性：Unlimited

》邮箱：任意，为每个 Actor 创建一个

》用例：Testing

》驱动：调用线程（duh）

\* BalancingDispatcher:Pool中的所有Actor都在共享同一个邮箱，并且为池中的每一个Actor都创建一个线程，只要有Actor处于空闲状态都会从共享邮箱中领取任务。只适合相同类型的actor使用，2.3版本之后被BalancingPool替代（现在已经不推荐使用此类DisPatcher了，在Router中可以使用BalancingPoolRouter）

》可共享性：与同种类型的Actor共享

》邮箱：1个，所有Actor公用

》用例：路由

》驱动：fork-join-executor、thread-pool-executor，

1.4、自定义DisPatcher

1.4.1. 在application.conf文件中定义一个DisPatcher，参数解析如下:

* my-dispatcher: 自定义dispatcher名称，Actor通过该名称配置dispatcher。
* type:DisPatcher的类型，选择符合场景的DisPatcher
* executor:定义Executor类型，如上面介绍的两种“fork-join-executor”、”thread-pool-executor“
* parallelism-min:单核最少线程数，Executor中最少线程数=parallelism-min\*parallelism-factor
* parallelism-factor:并发因子，此项最好配置为机器的CPU核数
* parallelism-max:单核最多线程数，Executor中最多线程数=parallelism-max\*parallelism-factor。

（parallelism-max这个参数不会在ForkJoinPool分配的线程总数上设置上限，这是一个设置，专门讨论池保持运行的热线程数，以减少处理新的传入任务的延迟。最大活跃线程数）

* throughput：跳到另一个Actor之前每个Actor最多处理的消息数量，这个参数是为了防止单个Actor一直占用线程，设置此Actor最多执行多少消息就要被调度。(这个参数可以让我们对actor在获取线程之后及进行处理的消息数量进行设置，设置为1则为公平的模式，如果设置得很大，则当前actor可以一直占用该线程直到消费完成指定数目的消息后才会让出线程。如果我们的cpu上下文切换过多可以考虑将该值设置得大一些，如果消息数量本身很少，设置过大的值会造成actor一直占据着线程空消息，这会影响其它actor的执行)



1.4.2. 使用自定义的Dispatcher来配置Actor

方式一：在配置文件中部署配置

akka.actor.deployment {  
 /genIDActor2 {  
 dispatcher = my-dispatcher  
 }  
}

方式2：代码中通过withDispatcher进行定义

**val** *actorSystem* = ActorSystem.*create*("genIdActorTest2")  
**val** *genIDActor* = *actorSystem*.actorOf(*Props*(**new** GenIdActor).withDispatcher("my-dispatcher"), "genIDActor2")

1.5、其它DisPatcher的配置示例:

1.5.1配置具有固定线程池大小的调度器，例如，对于执行阻塞 IO 的 Actor：

blocking-io-dispatcher {

type = Dispatcher

executor = "thread-pool-executor"

thread-pool-executor {

fixed-pool-size = 32

}

throughput = 1

}

1.5.2基于核（cores）数量使用线程池的示例，例如，对于 CPU 绑定的任务：

my-thread-pool-dispatcher {

# Dispatcher is the name of the event-based dispatcher

type = Dispatcher

# What kind of ExecutionService to use

executor = "thread-pool-executor"

# Configuration for the thread pool

thread-pool-executor {

# minimum number of threads to cap factor-based core number to

core-pool-size-min = 2

# No of core threads ... ceil(available processors \* factor)

core-pool-size-factor = 2.0

# maximum number of threads to cap factor-based number to

core-pool-size-max = 10

}

# Throughput defines the maximum number of messages to be

# processed per actor before the thread jumps to the next actor.

# Set to 1 for as fair as possible.

throughput = 100

}

1.5.3关联池

在保持某些内部状态的 Actor 数量相对较少的情况下，使用关联池（affinity pool）的不同类型的调度器可能会增加吞吐量。关联池尽可能的确保 Actor 总是被安排在同一线程上运行。这个 Actor 到线程的连接（pinning）旨在增加 CPU 缓存命中率，这可能使吞吐量显著提高。

affinity-pool-dispatcher {

# Dispatcher is the name of the event-based dispatcher

type = Dispatcher

# What kind of ExecutionService to use

executor = "affinity-pool-executor"

# Configuration for the thread pool

affinity-pool-executor {

# Min number of threads to cap factor-based parallelism number to

parallelism-min = 8

# Parallelism (threads) ... ceil(available processors \* factor)

parallelism-factor = 1

# Max number of threads to cap factor-based parallelism number to

parallelism-max = 16

}

# Throughput defines the maximum number of messages to be

# processed per actor before the thread jumps to the next actor.

# Set to 1 for as fair as possible.

throughput = 100

}

1.5.4配置一个PinnedDispatcher：

my-pinned-dispatcher {

executor = "thread-pool-executor"

type = PinnedDispatcher

}

注意：根据上面的my-thread-pool-dispatcher示例，thread-pool-executor配置不适用（NOT applicable）。不需要添加并发因子，最大最小线程等配置，这是因为每个 Actor 在使用PinnedDispatcher时都有自己的线程池，而该池只有一个线程。

注意:不能保证随着时间的推移使用相同的线程，因为核心池超时用于PinnedDispatcher，以在空闲 Actor 的情况下保持资源使用率低。要始终使用同一线程，需要添加PinnedDispatcher的配置thread-pool-executor.allow-core-timeout=off

* 1. Akka配置文件

在默认情况下，程序会加载classpath下application.conf、application.json、application.properties文件。

这里使用application.conf示例：

akka.log-config-on-start=on

 这里log-config-on-start配置成on，表示启动时（即调用ActorSystem.create()方法）会打印配置信息。

这是单文件的配置方式，还可以进行多文件配置。在大型项目中，配置文件可能需要拆分成多个文件，在Akka中，我们可以include的方式引入其它配置文件。

例如：现在我们有配置文件application.conf，system.conf文件，在system.conf文件中添加如下配置

**include** "application"  
akka.loglevel=debug

里system.conf引入了application配置文件，并增加了loglevel配置。在创建ActorSystem时，我们可以显示加载配置文件，例如：

**val** *system* = ActorSystem.*create*("system", ConfigFactory.*load*("system"))

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
     14. <https://www.cnblogs.com/gabry/p/9435471.html> Actor源码分析
     15. <https://doc.akka.io/docs/akka/current/testing.html#callingthreaddispatcher> CallingThreadDispatcher的使用
     16. [https://doc.akka.io/docs/akka/current/general/configuration.html application.conf](https://doc.akka.io/docs/akka/current/general/configuration.html%20application.conf)配置说明
     17. https://doc.akka.io/docs/akka/2.4/project/migration-guide-2.3.x-2.4.x.html actor版本迁移
2. 案例
   1. Scala-Actor实例:发送同步，异步消息

pom文件种导入scala-actors包

<properties>  
 <scala.version>2.11.11</scala.version>  
</properties>

<dependency>  
 <groupId>org.scala-lang</groupId>  
 <artifactId>scala-actors</artifactId>  
 <version>${scala.version}</version>  
</dependency>

**object** ActorTest2 **extends** App {  
 **val** *actor* = **new** SecondActor  
 *actor*.start()  
 //发送一个异步发送消息，无返回值消息  
 **val** *replay* = *actor* ! *AsyncMsg*(1, "hello actor")  
 *println*("异步发送消息完成")  
 *println*(*replay*)  
  
 //发送一个同步消息  
 **val** *replay2* = *actor* !? *SyncMsg*(2, "Hello Actor2")  
 *println*("同步消息发送完成")  
 *println*(*replay2*)  
  
 //发送一个异步带有返回值消息  
 **val** *replay3* = *actor* !! *AsyncMsg*(3, "Hello Actor3")  
 *println*("异步消息发送完成")  
 //检测返回结果是可用  
 *println*(*replay3*.isSet)  
 **val** *res* = *replay3*() //会阻塞，没有结果时会阻塞  
 *println*(*replay3*.isSet)  
 *println*(*res*)  
}  
  
**class** SecondActor **extends** Actor {  
 **override def** act(): Unit = {  
 **while** (**true**) {  
 receive {  
 **case** *SyncMsg*(id, msg) =>  
 *println*(s"**$**id SyncMsg **$**msg")  
 Thread.*sleep*(5000)  
 sender ! *ReplyMsg*(id, "finished")  
 **case** *AsyncMsg*(id, msg) =>  
 *println*(s"**$**id AsyncMsg **$**msg")  
 Thread.*sleep*(5000)  
 sender ! *ReplyMsg*(id, "finished")  
 }  
 }  
 }  
}

* 1. 创建Actor系统和Actor实例

pom文件种导入akka-actor包

<properties>  
 <actor.version>2.5.4</actor.version>  
</properties>

<dependency>  
 <groupId>com.typesafe.akka</groupId>  
 <artifactId>akka-actor\_2.11</artifactId>  
 <version>${actor.version}</version>  
</dependency>

*/\*\*  
 \* Actor类设置了Actor实例的行为:  
 \* 设置Actor实例回应消息和与其他Actor实例通信的方式，  
 \* 并封装了Actor实例的状态，  
 \* 定义了运行和关闭Actor实例的次序  
 \*/*

**class** HelloACtor **extends** Actor {  
 //重写接受消息的偏函数，其功能是接受消息并处理  
 //receive方法定义了一个类型为PartialFunction[Any, Unit]的偏函数，  
 //当Actor实例接受到Any类型的消息时，该偏函数就会被用到,如果没有为消息定义偏函数，那么消息就会被抛弃  
 **override def** receive: Receive = {  
 **case** "Hello" =>  
 *println*("你好!")  
 **case** "Stop" =>  
 //self是对自己的引用  
 *context*.stop(*self*) //停止自己的actorRef  
 //context.system 是Actor系统的引用  
 *context*.system.terminate() //关闭ActorSystem,即关闭其内部的线程池(ExcutorService)  
 **case** \_ =>  
 *println*("你是？")  
 }  
}

**object** Demo1 **extends** App {  
 //1.新建一个ActorSystem,创建线程池对象system，用来创建actor的对象的  
 //HelloSystem是线程池的名称  
 **val** *system* = *ActorSystem*("HelloSystem")  
 // 2.通过actorOf方法来创建一个actor，通过name参数设置Actor实例的唯一名字，当没有明确设置时，Actor会自动为新建的Actor实例赋予具有唯一性的名字  
 // 3.配置对象，Actor配置对象中含有Actor类相关的信息：构造器参数，缓存区和分配器的实现代码,  
 // 在Akka系统中，Actor配置对象是由Props类所代表的。Props对象封装了创建Actor实例所需的所有信息，  
 // 而且该对象能够被序列化并且能够通过网络传输  
 //apply[T <: Actor: ClassTag](creator: ⇒ T) 在创建HelloACtor实例时接受代码块,Props(new HelloACtor)  
 //apply(clazz: Class[\_], args: Any\*)-通过Class对象和一系列构造参数创建Actor实例 Props(classOf[HelloACtor]  
 **val** *helloACtorRef* = *system*.actorOf(*Props*(**new** HelloACtor), name = "helloActor")  
 // val helloACtorRef = system.actorOf(Props(classOf[HelloACtor]), name = "helloActor")  
 **var** *flag* = **true  
 while** (*flag*) {  
 *println*("输入你想发送的消息:")  
 **val** consoleLine = StdIn.readLine()  
 *helloACtorRef* ! consoleLine  
 **if** (consoleLine.equals("Stop")) {  
 *flag* = **false** *println*("程序即将结束")  
 }  
 }  
 //为了不让while的运行速度在receive方法之上，我们可以让他休眠0.1秒  
 Thread.*sleep*(1000)  
}

* 1. Actor管理无法正常处理消息

*/\*\*  
 \* 管理无法处理正常处理的消息  
 \* ------当Actor实例收到无法由其Actor处理的消息时，该消息会被存储在UnhandledMessage对象中并转发给Actor系统的事件流  
 \* 通常，Actor系统的事件流负责记录日志  
 \*/***class** DeafActor **extends** Actor {  
  
 //receive方法返回空的偏函数，这个空的偏函数不是为任何类型的消息定义的，  
 // 因此DefActor收到任何消息都会被发送给unhandled方法  
 **override def** receive: Receive = PartialFunction.*empty* //默认情况下，unhandled方法会在Actor系统的事件流中发布无法正常处理的消息  
 // 通过重写这个unhandled方法可以修改默认处理方式  
 **override def** unhandled(message: Any): Unit = message **match** {  
 **case** msg: String => *println*(s"I do not hear **$**msg")  
 **case** msg => **super**.unhandled(msg)  
 }  
}

4、Actor层次结构

**class** ChildActor **extends** Actor {  
  
 */\*\* \*  
 \* ChildActor通过将sayHi消息中的存储的引用发送给它的父类实例，对sayHi小心做出回应，  
 \* 父Actor类实例回通过调用context对象中的parsent方法，获取这个引用  
 \*/* **override def** receive: Receive = {  
 **case** "sayHi" =>  
 **val** parent = *context*.parent  
 *println*(s"parent **$**parent to me say hi")  
 }  
  
 //重写postStop方法，在ChildActor对象停止运行后被调用  
 **override def** postStop(): Unit = {  
 *println*("child stopped!")  
 }  
}  
  
  
**class** ParentActor **extends** Actor {  
 **override def** receive: Receive = {  
 **case** "create" =>  
 //当收到create消息时,调用context对象的actorOf创建新的子对象  
 *context*.actorOf(*Props*[ChildActor])  
 *println*(s"create a kid;children =**$**{*context*.children}")  
 **case** "sayHi" =>  
 //当收到sayhi消息时，通过遍历context.children列表并向每个子对象发送消息，将消息传递给它的子对象  
 *println*("kids,say hi!")  
 **for** (c <- *context*.children) {  
 c ! "sayHi"  
 }  
 **case** "stop" =>  
 *println*("parent stopping!")  
 //当收到stop消息时，停止自己的运行过程  
 *context*.stop(*self*)  
 }  
}  
  
  
**object** ActorsHierarchy **extends** App {  
 **val** *system* = *ActorSystem*("hierarchyActorTest")  
 //先创建父Actor引用，向parent发送两条create消息  
 **val** *parentActor* = *system*.actorOf(*Props*[ParentActor], name = "parent")  
 *parentActor* ! "create"  
 *parentActor* ! "create"  
 Thread.*sleep*(1000)  
 *parentActor* ! "sayHi"  
 Thread.*sleep*(1000)  
 *parentActor* ! "stop"  
 Thread.*sleep*(1000)  
 *system*.terminate()  
  
  
 //运行结果  
 */\*\* \*  
 \* create a kid;children =ChildrenContainer(Actor[akka://hierarchyActorTest/user/parent/$a#-874564128])  
 \* create a kid;children =ChildrenContainer(Actor[akka://hierarchyActorTest/user/parent/$a#-874564128], Actor[akka://hierarchyActorTest/user/parent/$b#-1812241398])  
 \* kids,say hi!  
 \* parent Actor[akka://hierarchyActorTest/user/parent#-842633715] to me say hi  
 \* parent Actor[akka://hierarchyActorTest/user/parent#-842633715] to me say hi  
 \* parent stopping!  
 \* child stopped!  
 \* child stopped!  
 \*/* //分析  
 */\*\*\*  
 \* 1.当父Actor停止运行后，子Actor也会停止运行  
 \* 2.Actor对象引用反应了Actor对象在Actor层次结构中的位置  
 \* akka://hierarchyActorTest/user/parent/$a  
 \* -----akka:// 表明引用的是一个本地Actor对象构  
 \* -----hierarchyActorTest 本例Actor系统名称  
 \* -----parent/$a 父actor对象的名称,和以自动方式为子Actor对象生成的子actor名称 $a  
 \*/*}

5、识别Actor对象

*/\*\*  
 \* Acotor路径  
 \* 在收到含有Actor路径的消息时，检查并显示Actor对象的引用。  
 \* 当CheckActor收到含有Actor路径或路径选择的字符串时，或获取ActorSelection对象，并向对象发送一条Identify消息  
 \* 这条消息会发送给路径选择中的所有Actor对象，而这些对象都会回复一条ActorIdentify消息  
 \*  
 \* 当CheckActor收到Identify消息时不是显示获得的Actor对象的引用，就是报告路径中不含该对象  
 \*/***class** CheckActor **extends** Actor {  
 **override def** receive: Receive = {  
 **case** path: String =>  
 *println*(s"checking path **$**path")  
 *context*.actorSelection(path) ! *Identify*(path)  
 **case** *ActorIdentity*(path, *Some*(ref)) =>  
 *println*(s"found actor **$**ref at **$**path")  
 **case** *ActorIdentity*(path, None) =>  
 *println*(s"could not found actor at **$**path")  
  
 }  
}  
  
  
**object** CheckActor **extends** App {  
 **val** *system* = *ActorSystem*("checkActorTest")  
 **val** *checker* = *system*.actorOf(*Props*[CheckActor], "check")  
 // ../\* 获取checker对象的父Actor对象的所有子Actor对象的引用:  
 // cheker对象本省的引用及它同级Actor对象的引用  
 // checker ! "../\*"  
 //found actor Actor[akka://checkActorTest/user/check#-1851702317] at ../\* -----本身的引用  
  
 //检查比checker对象高一级的所有Actor对象，守护者Actor对象user和system  
 // checker ! "../../\*"  
 // found actor Actor[akka://checkActorTest/system] at ../../\*  
 // found actor Actor[akka://checkActorTest/user] at ../../\*  
  
 //绝对路径选择--守护者Actor对象的system(系统内置的Actor对象)  
 *checker* ! "/system/\*"  
 // found actor Actor[akka://checkActorTest/system/log1-Logging$DefaultLogger#-21119779] at /system/\* ----log1-Logging 用于记录日志  
 // found actor Actor[akka://checkActorTest/system/deadLetterListener#-375529312] at /system/\* ----deadLetterListener --处置无法正常处理的消息  
 // found actor Actor[akka://checkActorTest/system/eventStreamUnsubscriber-1#-1128593966] at /system/\*  
  
 //不存在的路径  
 *checker* ! "/user/checker2"  
 // could not found actor at /user/checker2  
}

6、Actor生命周期

**object** ActorsLifecycle **extends** App {  
 **val** *system* = *ActorSystem*("lifecycleTest")  
 **val** *testy* = *system*.actorOf(*Props*[LifecycleActor], "testy")  
// testy ! math.Pi  
// Thread.sleep(1000)  
// testy ! 7  
// Thread.sleep(1000)  
// testy ! "hi there!"  
// Thread.sleep(1000)  
 *testy* ! *Nil* // Thread.sleep(1000)  
 //// testy ! "sorry about that"  
 //// Thread.sleep(1000)  
 *system*.stop(*testy*)  
 Thread.*sleep*(1000)  
 *system*.terminate()  
}  
  
  
**class** StringPrinter **extends** Actor {  
 **def** receive: Receive = {  
 **case** msg => *println*(s"child got message '**$**msg'")  
 }  
  
 **override def** preStart(): Unit = *println*(s"child about to start.")  
  
 **override def** postStop(): Unit = *println*(s"child just stopped.")  
}  
  
  
**class** LifecycleActor **extends** Actor {  
 **var** *child*: ActorRef = \_  
  
 **def** receive: Receive = {  
 **case** num: Double => *println*(s"got a double - **$**num")  
 **case** num: Int => *println*(s"got an integer - **$**num")  
 **case** lst: List[\_] => *println*(s"list - **$**{lst.head}, ...")  
 **case** txt: String => *child* ! txt  
 }  
  
 **override def** preStart(): Unit = {  
 *println*("about to start")  
 *child* = *context*.actorOf(*Props*[StringPrinter], "kiddo")  
 }  
  
 **override def** preRestart(reason: Throwable, msg: Option[Any]): Unit = {  
 *println*(s"about to restart because of **$**reason, during message **$**msg")  
 **super**.preRestart(reason, msg)  
 }  
  
 **override def** postRestart(reason: Throwable): Unit = {  
 *println*(s"just restarted due to **$**reason")  
 **super**.postRestart(reason)  
 }  
  
 **override def** postStop(): Unit = *println*("just stopped")  
}

7、Actor监督策略

**class** Naughty **extends** Actor {  
  
 **def** receive: Receive = {  
 **case** s: String => *println*(s"receive **$**s")  
 **case** \_ => **throw new** RuntimeException  
 }  
  
 **override def** postRestart(t: Throwable): Unit = {  
 *println*("naughty restarted")  
 }  
}  
  
**class** Supervisor **extends** Actor {  
 **val** *child* = *context*.actorOf(*Props*[Naughty], "victim")  
  
 **def** receive: Receive = PartialFunction.*empty* **override val** *supervisorStrategy* =  
 //OneForOneStrategy监控策略，当Actor对象失效时，根据导致该对象失效的异常，该对象会被恢复，重启，或停止  
 //AllForOneStrategy也有类似效果，但是应用于所有的子Actor对象  
 *OneForOneStrategy*() {  
 **case** \_: ActorKilledException => Restart  
 **case** \_ => Escalate  
 }  
}  
  
  
**object** SupervisionKill **extends** App {  
 **val** *ourSystem* = *ActorSystem*("supervisorTest")  
 **val** *s* = *ourSystem*.actorOf(*Props*[Supervisor], "super")  
 *ourSystem*.actorSelection("/user/super/\*") ! Kill //发送kill消息，让Naughty对象失效，但super会根据重启策略来重启actor  
 *ourSystem*.actorSelection("/user/super/\*") ! "sorry about that" //发送字符串消息,能正常处理  
 *ourSystem*.actorSelection("/user/super/\*") ! "kaboom".toList // 不能正常处理，两个对象都会停止  
 Thread.*sleep*(1000)  
 *ourSystem*.stop(*s*)  
 Thread.*sleep*(1000)  
 *ourSystem*.terminate()  
}

8、Actor通讯-请求模式

**class** Pongy **extends** Actor {  
  
 **def** receive: Receive = {  
 **case** "ping" =>  
 *println*("Got a ping -- ponging back!")  
 sender ! "pong"  
 *context*.stop(*self*)  
 }  
  
 **override def** postStop() = {  
 *println*("pongy going down")  
 }  
}  
  
  
**class** Pingy **extends** Actor {  
 **def** receive: Receive = {  
 **case** pongyRef: ActorRef =>  
 **implicit val** timeout = Timeout(2 seconds)  
 **val** future = pongyRef ? "ping"  
 //用pipeTo组合子，将Future对象的值发送给Actor对象的引用pongyRef的发送者  
 pipe(future) to sender  
 }  
}  
  
  
**object** CommunicatingAsk **extends** App {  
 **val** *ourSystem* = *ActorSystem*("communicatingAskTest")  
 **val** *masta* = *ourSystem*.actorOf(*Props*[Master], "masta")  
 *masta* ! "start"  
 Thread.*sleep*(1000)  
 *ourSystem*.terminate()  
  
  
 **class** Master **extends** Actor {  
 **val** *pingy* = *context*.actorOf(*Props*[Pingy], "pingy")  
 **val** *pongy* = *context*.actorOf(*Props*[Pongy], "pongy")  
  
 **def** receive: Receive = {  
 **case** "start" =>  
 *pingy* ! *pongy* **case** "pong" =>  
 *println*("got a pong back!")  
 *context*.stop(*self*)  
 }  
  
 **override def** postStop(): Unit = {  
 *println*("master going down")  
 }  
 }  
  
}

9、Actor通讯-转发模式

**class** Router **extends** Actor {  
 **var** *i* = 0  
 **val** *children*: immutable.IndexedSeq[ActorRef] = **for** (\_ <- 0 until 4) **yield** *context*.actorOf(*Props*[StringPrinter])  
  
 **override def** receive: Receive = {  
 **case** msg =>  
 //转发消息  
 *children*(*i*).forward(msg)  
 *i* = (*i* + 1) % 4  
 **case** "stop" => *context*.stop(*self*)  
 }  
}  
  
**object** CommunicatingRouter **extends** App {  
 **val** *ourSystem* = *ActorSystem*("communicatingAskTest")  
 **val** *router* = *ourSystem*.actorOf(*Props*[Router])  
 *router* ! "Hi."  
 *router* ! "I'm talking to you!"  
 Thread.*sleep*(1000)  
 *router* ! "stop"  
 *ourSystem*.terminate()  
}

10、Actor停止-PoisonPill

**object** CommunicatingPoisonPill **extends** App {  
 **val** *ourSystem* = *ActorSystem*("communicatingAskTest")  
 **val** *masta* = *ourSystem*.actorOf(*Props*[Master], "masta")  
 *masta* ! akka.actor.PoisonPill  
 Thread.*sleep*(1000)  
 *ourSystem*.terminate()  
}

11、Actor停止-gracefulStop

*/\*\*  
 \* Actor停止  
 \* 当需要从Actor对象内部跟踪停止Actor对象时，可以使用停止监视模式  
 \*/***class** GracefullPingy **extends** Actor {  
  
 **val** *pongy* = *context*.actorOf(*Props*[Pongy], "pongy")  
 //调用context对象的watch方法，这可以确保Pongy对象停止运行和postStop方法执行完毕后，  
 //GracefullPingy能收集到Pongy对象引用的Terminated消息  
 *context*.watch(*pongy*)  
  
 **override def** receive: Receive = {  
 **case** "Die,Pingy!" =>  
 *println*("get msg Die,Pingy!")  
 *context*.stop(*pongy*)  
 **case** *Terminated*(*`pongy`*) =>  
 //收到Terminated消息后会停止本身的运行  
 *println*("stop GracefullPingy")  
 *context*.stop(*self*)  
 }  
}  
  
  
**object** CommunicationGracefullStop **extends** App {  
 **val** *ourSystem* = *ActorSystem*("communicationGracefullStopTest")  
 **val** *grace* = *ourSystem*.actorOf(*Props*[GracefullPingy], "grace")  
 //akka.pattern的gracefullStop方法能够在接受到Actor对象引用，超时和关闭类型的消息，  
 // 该方法会返回一个Future对象并以异步方式向Actor对象发送关闭消息。  
 **val** *stopped* = gracefulStop(*grace*, 3.seconds, "Die,Pingy!")  
 *stopped* onComplete {  
 **case** *Success*(\_) =>  
 *println*("graceful shutdown successful")  
 *ourSystem*.terminate()  
 **case** *Failure*(\_) =>  
 *println*("grace not stopped!")  
 *ourSystem*.terminate()  
 }  
}

12、远程Actor通讯

pom文件种导入akka-actor包，及远程akka-remote包

<properties>  
 <actor.version>2.5.4</actor.version>  
</properties>

<dependency>  
 <groupId>com.typesafe.akka</groupId>  
 <artifactId>akka-actor\_2.11</artifactId>  
 <version>${actor.version}</version>  
</dependency>

<dependency>  
 <groupId>com.typesafe.akka</groupId>  
 <artifactId>akka-remote\_2.11</artifactId>  
 <version>${actor.version}</version>  
</dependency>

**package object** pratice12 {  
  
 //使用名为RemoteActorRefProvider的自定义ActorRef工厂对象，它可以创建用于实现远程通信的Actor对象引用  
 //Actor系统通过TCP网络层和指定的tcp端口，使用Netty网络库  
 **def** remotingConfig(port: Int) = ConfigFactory.*parseString*(  
 s"""  
akka {  
 actor.provider = "akka.remote.RemoteActorRefProvider"  
 remote {  
 enabled-transports = ["akka.remote.netty.tcp"]  
 netty.tcp {  
 hostname = "127.0.0.1"  
 port = **$**port  
 }  
 }  
}  
 """)  
  
 //1.在两个actotr进行通讯前，需要创建一个能够使远程Actor进行通讯的Actor系统  
 //创建哪一个自定义的Actor系统配置字符串，这个字符串可以用于配置一系列Actor属性  
 **def** remotingSystem(name: String, port: Int) = *ActorSystem*(name, *remotingConfig*(port))  
}

**import** akka.actor.Props  
**import** pratice10.Pongy  
  
//2.创建一个应用程序，使用24321实例化名文PongyDimension的Actor系统，运行15秒后关闭  
//启动该程序后，只能在限定的一小段时间内，启动另一个程序  
**object** RemotingPongySystem **extends** App {  
 **val** *system* = *remotingSystem*("PongyDimension", 24321)  
 **val** *pongy* = *system*.actorOf(*Props*[Pongy], "pongy")  
 Thread.*sleep*(15000)  
 *system*.terminate()  
}

//1.实例化ping对象，并获取pong对象的引用，将该引用交给ping对象，  
//Actor系统配置的Runner对象并不具备位置透明性，这是分布式系统中常见的情况;  
//程序中的某个部分负责寻找远程Actor系统中的Actor对象，并对这些Actor对象进行初始化，而应用程序本身的逻辑封装在其他独立运行的Actor对啊ing中  
**class** Runner **extends** Actor {  
  
 //创建ping的引用  
 **val** *pingy* = *context*.actorOf(*Props*[Pingy], "pingy")  
  
 **def** receive: Receive = {  
 **case** "start" =>  
 //收到消息后会为pong对象构造actor路径，使用tcp路径和actro系统的名称，一及远程计算机的地址，端口  
 //发送Identify消息，获取pong对象的引用  
 **val** path = *context*.actorSelection("akka.tcp://PongyDimension@127.0.0.1:24321/user/pongy")  
 path ! *Identify*(0)  
 **case** *ActorIdentity*(0, *Some*(ref)) =>  
 //当pong的引用发送给ping后，就可以通讯了  
 *pingy* ! ref  
 **case** *ActorIdentity*(0, None) =>  
 *println*("Something's wrong -- no pongy anywhere!")  
 *context*.stop(*self*)  
 **case** "pong" =>  
 *println*("got a pong from another dimension.")  
 *context*.stop(*self*)  
 }  
}

**object** RemotingPingySystem **extends** App {  
 **val** *system* = *remotingSystem*("PingyDimension", 24567)  
 **val** *runner* = *system*.actorOf(*Props*[Runner], "runner")  
 *runner* ! "start"  
 Thread.*sleep*(5000)  
 *system*.terminate()  
}

13、actor分配器-dispatcher

**class** GenIdActor **extends** Actor {  
 **var** *num* = 0  
  
 **def** genID: String = {  
 **val** time = **new** SimpleDateFormat("yyyyMMddHHmmss").format(**new** Date)  
 **if** (*num* > 999999999) *num* = 1 **else** *num* += 1  
 s"**$**time-**$**{*num*.formatted("%09d")}"  
 }  
  
 **def** receive: Receive = {  
 **case** "genID" =>  
 sender ! genID  
 }  
}

**object** GenIdApp **extends** App {  
 **val** *actorSystem* = *ActorSystem*("genIdActorTest")  
 **val** *genIDActor* = *actorSystem*.actorOf(*Props*(**new** GenIdActor))  
  
 **def** genID: Future[String] = {  
 //设置超时时间  
 **implicit val** timeout: Timeout = Timeout(20.seconds)  
 (*genIDActor* ? "genID").asInstanceOf[Future[String]]  
 }  
  
 //分配器实现了ExecutionContext接口，因此可以用来运行Future调用等待。  
 **implicit val** *ex* = *actorSystem*.dispatcher  
 **val** *getID* = **for** {  
 //Future执行需要一个隐式参数ExecutionContext，可以直接用actor的dispacher  
 \_ <- *Future*(*println*("======================"))  
 id <- *genID* } **yield** id  
 **val** *rs* = Await.*result*(*getID*, 1.seconds)  
 *println*(*rs*)  
}

14、自定义的Dispatcher

在application.conf中添加dispatcher配置

my-dispatcher {  
 # Dispatcher is the name of the event-based dispatcher  
 type = Dispatcher  
 # What kind of ExecutionService to use  
 executor = "fork-join-executor"  
 # Configuration for the fork join pool  
 fork-join-executor {  
 # Min number of threads to cap factor-based parallelism number to  
 parallelism-min = 2  
 # Parallelism (threads) ... ceil(available processors \* factor)  
 parallelism-factor = 2.0  
 # Max number of threads to cap factor-based parallelism number to  
 parallelism-max = 10  
 }  
 # Throughput defines the maximum number of messages to be  
 # processed per actor before the thread jumps to the next actor.  
 # Set to 1 for as fair as possible.  
 throughput = 100  
}

在配置文件中部署时指定dispatcher

akka.actor.deployment {  
 /genIDActor2 {  
 dispatcher = my-dispatcher  
 }  
}

或者代码中创建Actor时，使用自定义的dispatcher

**import** akka.actor.{ActorSystem, Props}  
**import** akka.pattern.\_  
**import** akka.util.Timeout  
**import** scala.concurrent.duration.\_  
**import** scala.concurrent.{Await, ExecutionContextExecutor, Future}

**object** GenIdApp2 **extends** App {  
 //通过配置文件来配置DisPatcher  
 **val** *actorSystem* = ActorSystem.*create*("genIdActor2")  
**val** *genIDActor* = *actorSystem*.actorOf(*Props*(**new** GenIdActor).withDispatcher("my-dispatcher"), "genIDActor2")

//查找分配器,  
 //分配器实现了ExecutionContext接口，因此可以用来运行Future调用等待。  
 **implicit val** *ex*: ExecutionContextExecutor = *actorSystem*.dispatchers.lookup("my-dispatcher")  
 **val** *getID* = **for** {  
 //Future执行需要一个隐式参数ExecutionContext，可以直接用actor的dispacher  
 \_ <- *Future*(*println*("======================"))  
 id <- *genID* } **yield** id  
 **val** *rs* = Await.*result*(*getID*, 1.seconds)  
 *println*(*rs*)  
 *actorSystem*.terminate()  
 **def** genID: Future[String] = {  
 //设置超时时间  
 **implicit val** timeout: Timeout = Timeout(20.seconds)  
 (*genIDActor* ? "genID").asInstanceOf[Future[String]]  
 }  
}