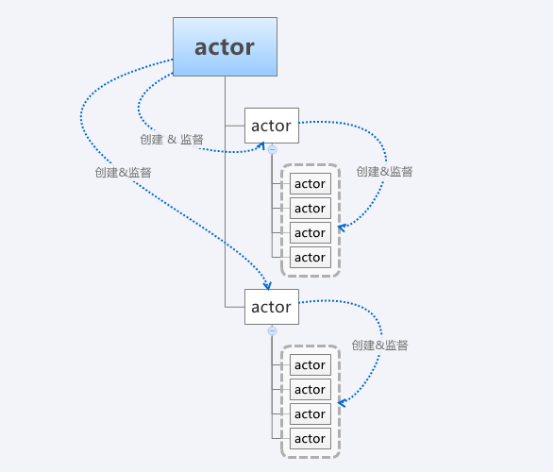
### **Akka**

#### **Akka和Actor介绍**

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。

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。

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。



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，每个Actor都有自己的收件箱（Mailbox）。通过Actor能够简化锁及线程管理，可以非常容易地开发出正确地并发程序和并行系统，Actor具有如下特性：

1.提供了一种高级抽象，能够简化在并发（Concurrency）/并行（Parallelism）应用场景下的编程开发

2.提供了异步非阻塞的、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）

为什么 Actor 模型是一种处理并发问题的解决方案？

　　处理并发问题就是如何保证共享数据的一致性和正确性，为什么会有保持共享数据正确性这个问题呢？无非是我们的程序是多线程的，多个线程对同一个数据进行修改，若不加同步条件，势必会造成数据污染。那么我们是不是可以转换一下思维，用单线程去处理相应的请求，但是又有人会问了，若是用单线程处理，那系统的性能又如何保证。Actor 模型的出现解决了这个问题，简化并发编程，提升程序性能。

#### ****重要类介绍****

**ActorSystem**

在Akka中，ActorSystem是一个重量级的结构，他需要分配多个线程，所以在实际应用中，ActorSystem通常是一个单例对象，我们可以使用这个ActorSystem创建很多Actor。

**Actor**

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，主要是用于初始化环境。

2.receive()方法：该方法在Actor的preStart方法执行完成后执行，用于接收消息，会被反复执行。

3.25.3 Maven依赖

|  |
| --- |
| <dependencies>  <!-- 添加scala的依赖 -->  <dependency>  <groupId>org.scala-lang</groupId>  <artifactId>scala-library</artifactId>  <version>2.11.8</version>  </dependency>  <!-- 添加akka的actor依赖 -->  <dependency>  <groupId>com.typesafe.akka</groupId>  <artifactId>akka-actor\_${scala.compat.version}</artifactId>  <version>2.4.17</version>  </dependency>  <!-- 多进程之间的Actor通信 -->  <dependency>  <groupId>com.typesafe.akka</groupId>  <artifactId>akka-remote\_${scala.compat.version}</artifactId>  <version>2.4.17</version>  </dependency>  </dependencies> |

#### 案例实现