<PAAS 平台>

基于 AKKA 的后台应用开发手册

版本 <1.0>

修订历史

日期	版本	描述	修改人
2013-06-28	0.5	基于 AKKA 的后台应用开发手册	苏晓辉
2013-07-11	1.0	基于 AKKA 的后台应用开发手册	苏晓辉

基于 AKKA 的后台应用开发手册

1 概述

1.1 前言

随着计算机硬件技术和网络技术的飞速发展，计算机拥有越来越多数量的内核，分布式技术和集群技术的成熟使得一个应用程序可以被分块运行在多个独立的计算机上（可能安装不同的操作系统），这些技术使得程序可以真正的并行运行。但是，我们相信编写出正确的具有容错性、可扩展性和跨平台的并发程序太困难了。这多数是因为我们使用了错误的工具和错误的抽象级别。AKKA 就是为了改变这种状况而生的。通过使用 Actor 模型我们提升了抽象级别，为构建正确的可扩展并发应用提供了一个更好的平台。在容错性方面我们采取了“let it crash”（让它崩溃）模型，人们已经将这种模型用在了电信行业，构建出“自愈合”的应用和永不停机的系统，取得了巨大成功。Actor 还为透明的分布式系统以及真正的可扩展高容错应用的基础进行了抽象。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。

1.2 目的

本文档的目的是帮助 PAAS 平台开发人员快速使用 AKKA，开发基于 Actor 模式的应用程序。本文第二、三、四、五、六、七章介绍 AKKA 的基本使用方法。然后通过第八章介绍消息总线的实现和基于消息总线之上的开发方式，给出一个具体的使用例子。最后，第九、十、十一、十二章给出一些在开发时的注意事项、代码规范、附件代码和总结及展望。

1.3 范围

本文档适用于 PAAS 系统整个项目周期中对 Actor 模式应用程序开发的指导和约束。

要了解 AKKA 和 SCALA 的使用详情，参考 AKKA 2.1.4 官方文档和 SCALA 2.10 官方文档。

1.4 术语和缩写语

术语	说明
SCALA	函数式编程和面向对象式编程结合的语言。
AKKA	基于 Actor 模式，用于开发并发软件的框架，提供了完整的 API 和各类文档。
PAAS	Platform-as-a-Service 的缩写，意思是平台即服务。把服务器平台作为一种服务提供的商业模式。
SAAS	software-as-a-Service 的缩写。国外称为 SAAS，国内通常叫做软件运营服务模式，简称为软营模式。

2 Actor 的基本开发

本章首先通过一个简单的例子快速简单的介绍 AKKA actor 是如何实现和使用的，然后进一步介绍带参数构造器 actor 的实现方式、actor 的生命周期等内容。

2.1 创建简单 Actor

创建简单的 Actor 模式应用程序，定义 Actor1 如下代码 2-1 所示，Actor1 类继承 AKKA 中的 Actor 类，定义部分函数（Partial Function）receive，receive 函数中通过模式匹配（Pattern Match）实现程序逻辑。

```
import akka.actor.Actor
import akka.actor.ActorSystem
import akka.actor.Props
import akka.event.Logging

class Actor1 extends Actor {
  val log = Logging(context.system, this)
  def receive = {
    case "test" => log.info("received test")
    case _      => log.info("received unknown message")
  }
}
```

代码 2-1

如下代码 2-2 所示，通过 AKKA 中的 ActorSystem 对象的 actorOf 方法创建上面的 Actor1 类对象实例，返回 AKKA 中的 ActorRef 类型的 actor1 对象，actor1 对象是 Actor1 类对象实例的引用，通过 actor1 对象可以向 Actor1 类对象实例发送消息。

```
object Demo1 extends App{

  val system = ActorSystem("Demo1")
  val actor1 = system.actorOf(Props[Actor1], name = "Actor1")

  actor1 ! "test"
  actor1 ! "other"
}
```

代码 2-2

注意：

- 如果 Receive 方函数中不存在默认匹配，则会向 ActorSystem 的事件消息流（Event Stream）发送 akka.actor.UnhandledMessage(message,sender,recipient) 消息。（后续 2.5 节会详细说明 UnhandledMessage）
- ActorRef 类型的对象是不可变的和可序列化的，可以在网络中进行传输，作为远程对象使用，具体的操作还是在本地的 Actor 类对象。(后续章节会详细说明 ActorRef 类型的对象如何作为远程对象来使用)。
- Actor 对象的名称可以在创建时省略；如果不省略 actor 对象的名称，那么在同一个父 actor 对象下子 actor 的名称必须唯一。
- Actor 对象的名称不能为空，并且不能是以‘$’开头的字符串。

2.2 创建带参数构造器 Actor

如下代码 2-3 所示，Actor2 类带有参数的构造函数，这种情况下无法使用 Props[Actor2]的方式创建 actor2 对象。

```
class Actor2(name: String) extends Actor {
  val log = Logging(context.system, this)
  def receive = {
    case "test" => log.info(name + "received test")
    case _      => log.info(name + "received unknown message")
  }
}
```

代码 2-3

可以采用 call-by-name 块（参考 scala 相关内容）的方式创建 actor2 对象，如下代码 2-4 所示。

```
object Demo2 extends App{

  val system = ActorSystem("Demo2")
  val actor2 = system.actorOf(Props(new Actor2("Anything")), name = "Actor2")

  actor2 ! "test"
  actor2 ! "other"
}
```

def apply(creator: => akka.actor.Actor): akka.actor.Props

代码 2-4

注意：

- Props（...）中不能始终传入同一个 actor 对象，例如 val lazy 或 object extends Actor 等单例工厂实现方式，这会和 AKKA 中 actor restart 机制冲突。
- 构造器参数不能是可变的（var），因为 call-by-name 块可能被其它线程调用，引起条件竞争。

2.3 Actor 停止监视

通过 context 对象(AKKA 中的 ActorContext 类)的 watch/unwatch 方法可以注册/撤销对一个 actor 的监视，如下代码 2-5 所示，WatchActor 类在实例化时会注册监视 other，在某一时刻 other 停止后，WatchActor 的实例会收到 Terminated 类型的消息，来告知被监视 actor 对象已经停止了。Actor 的停止方式在 2.8 节介绍。

```
import akka.actor.{ Actor, Props, Terminated, ActorSystem, ActorRef }

class WatchActor(val other: ActorRef) extends Actor {

  context.watch(other) //context.unwatch(target)

  var lastSender = context.system.deadLetters

  def receive = {
    case Terminated(other) =>  lastSender ! other.toString+" finished!"
  }

}
```

代码 2-5

2.4 Actor 生命周期控制

Actor 的生命回调方法，如下代码 2-6 所示，启动前运行 preStart 方法（默认为空），重启前运行 preRestart 方法（默认停止所有子 actor，最后停止自己），重启后运行 postStart 方法（默认运行 preStart 方法），停止后调用 postStop 方法（默认为空）。

```
class LifeActor extends Actor {

  override def preStart() {}

  override def preRestart(reason: Throwable, message: Option[Any]) {
    super.preRestart(reason, message)
  }

  override def postRestart(reason: Throwable){preStart()}

  override def postStop() {}

  def receive = {
    case _ =>
  }

}
```

代码 2-6

注意：

- 在重启的同时，收件箱不会被影响，可以继续接收消息。
- PostStop 方法一定在收件箱停止后才运行，用于关闭资源。
- 消息发给已经停止的 actor 会被转发到系统的 deadLetters。
- Actor 的构造函数在第一次创建和每次重启时被调用，来初始化 actor 实体。
- PreStart 方法只在第一次创建时被调用，来初始化 actor 实体。

2.5 Actor 未定义消息处理

Actor 的 unhandled 方法对 receive 方法中未匹配成功的消息进行处理，如下代码 2-7 所示，默认情况有两种处理方式。当未处理消息类型是 akka.actor.Terminated 时，抛出 akka.actor.DeathPactException；当其它未处理消息时，先 akka.event.EventStream 发送 akka.actor.UnhandledMessage 类型消息。

```
class UnhandledActor extends Actor {

  override def unhandled(message : Any) {
    super.unhandled(message)
  }

  def receive = {
    case _ =>
  }

}
```

代码 2-7

2.6 Actor 消息发送

Actor 有两种发送消息的方式，“!”（tell）和“？”（ask）。“!”是 Fire-forget 方式；“？”是 Send-And-Receive-Future 方式，如下代码 2-8 所示，向 actor1 对象发送“hi”消息，由于是异步调用，消息发送后，程序没有阻塞，直接得到 future1 对象，通过 pipeTo 方法（异步调用）给 future1 装载消息答复完成处理器 actor2，由 actor2 处理“hi”请求后的答复（将答复打印到控制端），这个过程中所有方法的调用都是异步执行没有阻塞。Future 类的高级使用将在后续章节介绍。

```
import akka.actor._
import akka.pattern.{ask, pipe}
import akka.util.Timeout
import scala.concurrent.duration._
import scala.concurrent.Future

class ReplyActor extends Actor{

  def receive = {
    case "hi" => sender ! "hello"
  }
}

class FutureActor extends Actor{

  def receive = {
    case m => println(m)
  }
}

object ReplyDemo extends App{

  val system = ActorSystem("ReplyDemo")

  val actor1 = system.actorOf(Props[ReplyActor], name="ReplyActor")
  val actor2 = system.actorOf(Props[FutureActor], name="FutureActor")

  implicit val timeout = Timeout(5 seconds)
  val future1: Future[String] = ask(actor1, "hi").mapTo[String]
  //val future1: Future[String] = (actor1 ? "hi").mapTo[String]

  import system.dispatcher
  future1 pipeTo actor2
}
```

代码 2-8

注意：

- 如果消息不是来自 actor 对象，默认情况 sender 对象是 deathLetters actor 对象。
- Timeout 对象是 ask 方法的隐式参数，控制得到 Future 对象的最大时间，如果超过报 AskTimeoutException 异常。
- Dispatcher 对象是 pipe 隐式转换方法的隐式参数，指定任务分发对象。
- Actor 对象还可以进行消息转发，调用 forward 方法。

2.7 Actor 消息接收超时

Actor 可以设置接收消息超时，如下代码 2-9 所示，通过 context 的 setReceiveTimeout 方法启动消息接收超时功能，并设置时间间隔，超过该间隔时间没有收到消息产生 ReceiveTimeout 类型的消息，在下面代码中通过设置 Duration.Undefined 停止消息接收超时功能。

```
class TimeoutActor extends Actor {

  context.setReceiveTimeout(30 milliseconds)

  def receive = {
    case "Hello" => context.setReceiveTimeout(100 milliseconds)
    case ReceiveTimeout => context.setReceiveTimeout(Duration.Undefined)
      throw new RuntimeException("Receive timed out")
  }
}
```

代码 2-9

注意：

- 时间间隔最小设置是 1 milliseconds。

2.8 Actor 停止

Actor 可以通过 ActorRefFactory 类型的对象，例如 ActorContext 和 ActorSystem 的 stop 方法来停止，stop 方式是异步执行的，可能在调用该方法后 actor 没有完全被停止，比如在调用 stop 方法时 actor 或子 actor 的 receive 方法正在运行，actor 要等到这些方法结束后才停止，stop 只是向 actor 发送了停止命令；Actor 也可以通过发送 akka.actor.PoisonPill 消息来停止 actor 对象，和 stop 的区别在于 PoisonPill 消息被发送到 actor 的邮箱，等到 actor 处理该消息时才开始停止 actor。如下代码 2-10 所示，利用 gracefulStop 方法和 future 特性来等到 actor 被完全停止。

```
import akka.pattern.gracefulStop
import scala.concurrent.Await

try {
  val stopped: Future[Boolean] = gracefulStop(actorRef, 5 seconds)(system)
  Await.result(stopped, 6 seconds)
} catch {
  case e: akka.pattern.AskTimeoutException =>
}
```

代码 2-10

注意：

- Actor 的停止和 Actor 名字的注销是异步进行的，有时 actor 停止了，但是名字还是存在，这时可以通过 supervisor 监测到 Terminated 消息来从新使用该名字，来确保安正确性。
- 向 actor 直接发送 Kill 消息停止 actor，默认情况 supervisor 会重启该 actor。

2.9 变换 Actor 消息匹配功能

Actor 消息匹配功能有两种变换方式，默认情况下使用 become 方法可以替换 Actor 的消息匹配功能，定义 HotSwapActor 代码 2-11 如下所示，angry 和 happy 方法相互替换。

```
class HotSwapActor extends Actor {
  import context._
  def angry: Receive = {
    case "foo" => sender ! "I am already angry?"
    case "bar" => become(happy)
  }

  def happy: Receive = {
    case "bar" =>sender ! "I am already happy :-)"
    case "foo" =>become(angry)
  }

  def receive = {
    case "foo" =>become(angry)
    case "bar" => become(happy)
  }
}
```

代码 2-11

另一种方式如代码 2-12 所示，通过设置 discardOld 参数为 false，这时使用 become 方法不会替换原来的消息匹配功能，而是在栈顶添加一个消息匹配功能，使用 unbecome 方法恢复到上一个消息匹配功能，这种方法可以无限制增加消息匹配功能直到内存溢出，所以要合理使用。

```
class Swapper extends Actor {
  import context._
  val log = Logging(system, this)

  def receive = {
    case Swap => log.info("Hi")
      become({
        case Swap =>log.info("Ho")
          unbecome()
      }, discardOld = false)
  }
}
```

代码 2-12

注意：

- Actor 重启后恢复到初始的消息匹配功能。

2.10 Actor 消息栈

Actor 通过混入 Stash Trait 实例可以实现消息压栈功能，如下代码 2-13 所示，调用 stash 方法，消息被压栈，不被当前消息匹配功能处理，调用 unstashAll 方法所有消息出栈加入到邮箱中，并且只有在变换消息匹配功能后该过程才会生效，

出栈消息的处理顺序和原来在邮箱中的一样。

```
class ActorWithProtocol extends Actor with Stash {
  def receive = {
    case "open" =>
      unstashAll()
      context.become({
        case "write" =>
        case "close" =>
          unstashAll()
          context.unbecome()
        case msg => stash()
      }, discardOld = false)
    case msg=> stash()
  }
}
```

代码 2-13

注意：

- 相同的消息被调用两次 stash，报 IllegalStateException 异常。
- 栈是有大小的过多的消息加入可能会导致 StashOverflowException 异常，可以同配置文件设置大小（stash-capacity）。
- 邮箱有大小的，在一次从栈返回过多消息添加到邮箱中可以会导致 MessageQueueAppendFailedException 异常。
- 不要在混入 Stash trait 后覆盖 preRestart 方法，会使该 Stash trait 的功能失效。

2.11 Actor 匹配消息功能扩展

如下代码 2-14 所示，通过继承的方式进行扩展，利用 ParitialFunction 类的 orElse 方法连接两个 ParitialFunction 类的函数。

```
abstract class GenericActor extends Actor {

  def specificMessageHandler: Receive

  def genericMessageHandler: Receive = {
    case event => printf("generic: %s\n", event)
  }

  def receive = specificMessageHandler orElse genericMessageHandler
}

class SpecificActor extends GenericActor {
  def specificMessageHandler = {
    case event: MyMsg => printf("specific: %s\n", event.subject)
  }
}

case class MyMsg(subject: String)
```

代码 2-14

如下代码 2-15 所示，通过代理的方式进行扩展，利用 ParitialFunction 类的 orElse 方法连接两个 ParitialFunction 类的函数。

```
class PartialFunctionBuilder[A, B] {
  import scala.collection.immutable.Vector

  type PF = PartialFunction[A, B]

  private var pfsOption: Option[Vector[PF]] = Some(Vector.empty)

  private def mapPfs[C](f: Vector[PF] => (Option[Vector[PF]], C)): C = {
    pfsOption.fold(throw new IllegalStateException("Already built"))(f) match {
      case (newPfsOption, result) => {
        pfsOption = newPfsOption
        result
      }
    }
  }

  def +=(pf: PF): Unit =
    mapPfs { case pfs => (Some(pfs :+ pf), ()) }

  def result(): PF =
    mapPfs { case pfs => (None, pfs.foldLeft[PF](Map.empty) { _ orElse _ }) }
}

trait ComposableActor extends Actor {
  protected lazy val receiveBuilder = new PartialFunctionBuilder[Any, Unit]
  final def receive = receiveBuilder.result()
}

trait TheirComposableActor extends ComposableActor {
  receiveBuilder += {
    case "foo" => sender ! "foo received"
  }
}

class MyComposableActor extends TheirComposableActor {
  receiveBuilder += {
    case "bar" => sender ! "bar received"
  }
}
```

代码 2-15

2.12 Actor 容错机制

Actor 系统中的每个 actor 都是其子 actor 们的监控者，每个 actor 都定义了自己的容错机制，在 actor 成为 Actor 系统的一部分后，这些机制就无法改变了。

如果 actor 没有定义容错机制，那么默认机制被执行，如下代码 2-16 所示，顶层的 actor 归 *User Guardian* 监控，只能通过配置来设置 *User Guardian* 的容错策略。匹配块决定策略，是一个部分函数 PartialFunction[Throwable, Directive]。

```
OneForOneStrategy() {
  case _: ActorInitializationException => Stop
  case _: ActorKilledException => Stop
  case _: Exception => Restart
  case _: Throwable => Escalate
}
```

代码 2-16

自定义容错机制，如下代码 2-17 所示，方法中的参数表示一段时间内多少次重试后采用 Restart 策略。

```
class SuperviseActor extends Actor {

  import akka.actor.OneForOneStrategy
  import akka.actor.SupervisorStrategy._
  import scala.concurrent.duration._

  override val supervisorStrategy =
    OneForOneStrategy(maxNrOfRetries = 10, withinTimeRange = 1 minute) {
      case _: ArithmeticException      => Resume
      case _: NullPointerException     => Restart
      case _: IllegalArgumentException => Stop
      case _: Exception                => Escalate
    }

  def receive = {
    case _ =>
  }

}
```

代码 2-17

除了代码中出现的 One For One 策略，还有 All For One 策略，前者只将以上某个策略作用于失败的那个子 actor 上，后者作用于所有子 actor。supervisorStrategy 是线程安全的方法。

3 Actor DSL

本章通过介绍 DSL 的方式来实现 actor 模式的应用程序开发。DSL 方式是本手册其它章节介绍的 AKKA actor 标准实现方式（以下简称‘标准方式’）的简化形式，DSL 能实现的功能标准方式也都可以实现，如果开发者只专注于程序逻辑功能的实现，而不在意技术实现方式的多样性可以跳过该章节。

3.1 创建 DSL Actor

创建 DSL 方式下的 actor 模式应用程序，定义 actor1 代码如下 3-1 所示，actor 方法的功能和 system.actorOf 或 context.actorOf 的功能相同，actor 方式需要一个隐式工厂参数，隐式工厂参数决定了使用 system 还是 context 来创建 actor1，例如 3-1 代码中使用 system 作为隐式工厂参数，所以 actor1 是系统下的 actor 对象。actor 方法的另一个参数是一个返回 Act Trait 的实例的函数参数，Act Trait 中的 become 方法类似 actor 标准方式中 receive 方法，实现消息匹配功能。

```
import akka.actor.ActorDSL._
import akka.actor.ActorSystem

object DSLDemo extends App{

  implicit val system = ActorSystem("DSLDemo")

  val actor1 = actor(new Act{
    become {
      case "hello" => sender ! "hi"
    }
  })

}
```

代码 3-1

3.2 切换 DSL Actor 消息匹配功能

通过使用 becomeStacked 和 unbecome 方法可以切换 DSL Actor 的消息匹配功能，定义 actor2 代码 3-2 如下所示，消息匹配功能块按堆栈方式存储，在 Act trait 实例初始时，空消息匹配块压入栈底，在实例 become 方法时，将 become 方法内的消息匹配块（后称‘become 块’）压入堆栈，此时栈顶是 become 块，当接收“switch”消息时使用 become 块来匹配，对应调用 becomeStacked 方法，将 becomeStacked 内的消息匹配块（后称‘becomeStacked 块’）压入堆栈，此时栈顶是 becomeStacked 块，当接收“switch”消息时使用 becomeStacked 块来匹配，对应调用 unbecome 方法，将 becomeStacked 块弹出堆栈，此时栈顶是 become 块，当接收“lobotomize”消息时使用 become 块来匹配，对应调用 unbecome 方法，将 become 块弹出堆栈，此时栈顶是空消息匹配块，不再匹配任何消息。

```
val actor2 = actor(new Act{
  become {
    case "info" => sender ! "A"
    case "switch" =>
      becomeStacked {
        case "info" =>  sender ! "B"
        case "switch" => unbecome()
      }
    case "lobotomize" => unbecome()
  }
})
```

代码 3-2

3.3 DSL Actor 生命周期控制

DSL Actor 的生命回调方法，如下代码 3-3 所示，启动前运行 whenStarting 方法，重启前运行 whenFailing 和 whenRestarted 方法，停止前调用 whenStopping 方法。

```
val child = actor(context, "child")(new Act{
    become {
      case "die" => throw new Exception("stop")
      case "restart" => throw new Exception("restart")
    }
    whenStarting { println("whenStarting") }
    whenFailing{(cause, msg) => println("whenFailing")}
    whenRestarted{ cause => println("whenRestarted")}
    whenStopping { println("whenStopping") }
})
```

代码 3-3

3.4 DSL Actor 异常监控

DSL Actor 的异常监控方法，如下代码 3-4 所示，检查到子 actor 异常，进入 supervisewith 方法匹配异常。

```
superviseWith(OneForOneStrategy() {
    case e: Exception if e.getMessage == "stop" =>println("stop"); Stop
    case e: Exception if e.getMessage == "restart" =>println("restart"); Restart
})
```

代码 3-4

3.5 DSL Actor 消息栈

DSL Actor 通过创建 ActWithStash Trait 实例可以实现消息压栈功能，如下代码 3-5 所示，actor1 在收到消息“1”，“2”后消息压栈，收到消息“3”后消息出栈，同时切换到 becomeStacked 块，消息匹配后输出“5”，“6”。

```
val actor1 = actor(new ActWithStash {
become {
  case 1 => stash()
  case 2 => stash()
  case 3 =>
    unstashAll()
    becomeStacked {
      case 1 => b ! 5;
      case 2 => b ! 6;
    }
  }
})

val b = actor(new Act{
  become {
    case 5 => println("5")
    case 6 => println("6")
  }
})
```

代码 3-5

4 Typed Actor

本章通过介绍 Typed 的方式来实现 actor 模式的应用程序开发。Typed 方式是本手册其它章节介绍的 AKKA actor 标准实现方式（以下简称‘标准方式’）的方法调用形式，TypedActor 没有变化匹配消息的功能，Typed 能实现的功能标准方式也都可以实现，如果开发者只专注于程序逻辑功能的实现，而不在意技术实现方式的多样性可以跳过该章节。

4.1 创建 Typed Actor

创建 Typed Actor 模式的应用程序，如下代码 4-1 所示，首先定义类似于 JAVA 的接口和实现。squareDontCare 方法的定义实现 actor 的“！”发送消息机制；square 方法的定义实现 actor “？”的发送消息机制；squareNowPlease 方法的定义实现 actor 同步调用机制，超过调用时间返回 None；squareNow 方法的定义实现 actor 同步调用机制，超过调用时间抛出 java.util.concurrent.TimeoutException 异常。

```
trait Square {
  def squareDontCare(i: Int): Unit

  def square(i: Int): Future[Int]

  def squareNowPlease(i: Int): Option[Int]

  def squareNow(i: Int):Int
}

class SquareImpl (val name: String) extends Square{

  def this() = this("default")

  import TypedActor.dispatcher

  def squareDontCare(i: Int): Unit = println(i * i)

  def square(i: Int): Future[Int] = Promise.successful(i * i).future

  def squareNowPlease(i: Int): Option[Int] = Some(i * i)

  def squareNow(i: Int):Int = i * i
}
```

代码 4-1

4.2 Typed Actor 代理对象

创建 Typed Actor 的代理对象，如下代码 4-2 所示，第二行代码是不带参数构造器的 Typed Actor 代理对象的实现方法，第三行代码是带参数构造器的 Typed Actor 代理对象的实现方法。

```
val system = ActorSystem("System")

val mySquarer: Square = TypedActor(system).typedActorOf(TypedProps[SquareImpl]())

val otherSquarer: Square = TypedActor(system).typedActorOf(TypedProps(classOf[Square],
    new SquareImpl("foo")), "name")
```

代码 4-2

4.3 Typed Actor 停止

停止 Typed Actor，如下代码 4-3 所示，stop 方法标准 actor 中的 stop 方法功能类似，poisonPill 方法同标准 actor 中的 akka.actor.PoisonPill 消息功能类似。

```
TypedActor(system).stop(mySquarer)

TypedActor(system).poisonPill(otherSquarer)
```

代码 4-3

4.4 Typed Actor 其它

Typed Actor 的异常监控、生命周期控制、除方法调用外的消息接收和子 Typed Actor 的创建，如下代码 4-4 所示。

```
class otherImpl (val name: String) extends TypedActor.Receiver
    with TypedActor.Supervisor
    with TypedActor.PreStart
    with TypedActor.PostStop
    with TypedActor.PreRestart
    with TypedActor.PostRestart{

  val childSquarer: Square = TypedActor(TypedActor.context).typedActorOf(TypedProps[SquareImpl]())

  def supervisorStrategy = OneForOneStrategy(maxNrOfRetries = 10, withinTimeRange = 1 minute) {
    case _ => Resume
  }

  def onReceive(message: Any, sender: ActorRef) {

  }

  def postRestart(reason: Throwable) {

  }

  def postStop(){

  }

  def preRestart(reason: Throwable, message: Option[Any]){

  }

  def preStart() {

  }

}
```

代码 4-4

Typed Actor 远程调用，如下代码 4-5 所示。

```
val typedActor: Foo with Bar =
  TypedActor(system).
    typedActorOf(
      TypedProps[FooBar],
      system.actorFor("akka://SomeSystem@somehost:2552/user/some/foobar"))
```

代码 4-5

Typed Actor 方法连接，类似标准 actor 扩展匹配消息功能，如下代码 4-6 所示。

```
trait Foo {
  def doFoo(times: Int): Unit = println("doFoo(" + times + ")")
}

trait Bar {
  import TypedActor.dispatcher
  def doBar(str: String): Future[String] =
    Promise.successful(str.toUpperCase).future
}

class FooBar extends Foo with Bar
val awesomeFooBar: Foo with Bar =
  TypedActor(system).typedActorOf(TypedProps[FooBar]())

awesomeFooBar.doFoo(10)
val f = awesomeFooBar.doBar("yes")

TypedActor(system).poisonPill(awesomeFooBar)
```

代码 4-6

注意：

- 在标准 actor 中也可以创建子 Typed Actor，传入 ActorContext 参数到 TypedActor 的 get 方法。

5 Actor 日志

在一般的系统中 log 是同步的，这意味着 IO 和锁的开销和等待，但是 Actor 的 log 是异步执行的，这样可以提高系统的性能。

5.1 Actor Log 基本功能

Actor Log 的基本功能和使用情况，如下代码 5-1 所示，通过 Logging 对象创建 LoggingAdapter 类的对象，该对象有 error、warning、info 和 debug 等记录日志的方法，这些方法的第一个参数是记录的内容，内容中可以包含“{}”占位符，后面的参数是用来填充占位符的数据源，数据源会被转换成 String，也可以用一个数组参数来替换多个数据源参数。

```
import akka.event.Logging
import akka.actor.Actor

class LogActor extends Actor {

  val log = Logging(context.system, this)

  override def preStart() = {
    log.debug("Starting")
  }

  override def preRestart(reason: Throwable, message: Option[Any]) {
    log.error(reason, "Restarting due to [{}] when processing [{}]",
      reason.getMessage, message.getOrElse(""))
  }

  def receive = {
    case "test" => log.info("Received test")
    case x  => log.warning("Received unknown message: {}", x)
  }

}
```

代码 5-1

注意：

- 如果数据源是 Actor 或 ActorRef 类的对象，转换成路径。
- 如果数据源是 String 类的对象，直接使用。
- 如果是其它类的对象，转换成简单类名。
- 如果是其它情况，会编译出错，除非 implicit LogSource[T]在范围内。

6 Actor 任务

6.1 Actor schedule 基本功能

Actor 提供了简单的任务调度功能，如下代码 6-1 所示，通过 system 的 scheduler 方法得到 AKKA 中 Scheduler 类的对象，该对象的所有任务方法都需要传入一个 ExecutionContext 类的隐式参数，默认使用 system.dispatcher 对象。Cancellable 对象是任务方法调用后的一个返回对象，用来取消任务。

```
import system.dispatcher

system.scheduler.scheduleOnce(50 milliseconds, sActor, "foo")

system.scheduler.scheduleOnce(50 milliseconds){
  sActor ! System.currentTimeMillis()
}

val cancellable = system.scheduler.schedule(0 milliseconds, 50 milliseconds, sActor, "tick")

cancellable cancel
```

代码 6-1

7 Actor Future 使用

本节的内容涉及 Future 的高级应用，提供了一种编写并发程序和避免阻塞的方式。7.1 节是 Future 大多数情况下使用的方式；7.2 节以后是 Future 的一些特殊高级使用，大家可以根据应用程序的需要来决定是否阅这些内容。

7.1 Future 和 Actor 配合使用

Future 配合 Actor 一起是使用，如下代码 7-1 所示，在前面章节中也介绍过类似使用，ExecutionContext 对象作为分发器首先分发任务，然后得到 Future 对象，该 ExecutionContext 对象需要作为隐式对象，在使用 Future 的地方被引用，获取 ExecutionContext 对象的方法有 improt system.dispatcher，也可以直接使用 ExecutionContext 类的工厂方法。Await.result and Await.ready 方法会阻塞程序，直到返回 future 对象的结果；Pipeto 方法将 future1 对象委托给 actor 去处理。

```
val system = ActorSystem("futureSystem")

import system.dispatcher

val actor = system.actorOf(Props[FutureActor], name="futureActor")

implicit val timeout = Timeout(5 seconds)

val future = actor ? "hi"
val result = Await.result(future, timeout.duration).asInstanceOf[String]

println(result)

val future1: Future[String] = ask(actor, "hi").mapTo[String]

future1 pipeTo actor
```

代码 7-1

7.2 直接使用 Future

通过 Future 对象直接构造一个 Future 类的对象 future2，创建过程是通过分发器运行任务，同时返回 future2 对象，如下代码 7-2 所示，以下对 future2 的操作都是非阻塞的，不要等到构造完“Hello World”字符串和打印 future2 的结果后才运行后面的语句，print（“end”）语句可能在 future2 运行完之前就被执行。

```
val future2 = Future {
    println("future2 start")
    Thread.sleep(2000)
    "Hello" + "World"
}

future2 foreach println

println("end")
```

代码 7-2

7.3 Future 连接方法

Future 有类似 Collection 集合类的方法，如下代码 7-3 所示，这些方法都是并发非阻塞运行的，在后台每个 Future 对象并发运行时会等待其它 Future 对象的结果完成后再运行。

```
val f1 = Future {
    "Hello" + "World"
}
val f2 = Future.successful(3)
val f3 = f1 flatMap { x =>
    f2 map { y =>
      x.length * y
    }
}
f3 foreach println

val future4 = Future.successful(4)
val future5 = future4.filter(_ % 2 == 0)

future5 foreach println

val failedFilter = future4.filter(_ % 2 == 1).recover {
  case m: NoSuchElementException => 0
}

failedFilter foreach println
```

代码 7-3

7.4 Future 和 for 配合使用

通过 for 可以使多个 Future 对象配合使用，而且不产生阻塞，如下代码 7-4 所示，后面两个 Future 要等第一个 Future 完成后才执行，但是这些等待都在后台异步的执行进行。

```
val f = for {
  a <- Future(10 / 2) // 10 / 2 = 5
  b <- Future(a + 1) //  5 + 1 = 6
  c <- Future(a - 1) //  5 - 1 = 4
  if c > 3 // Future.filter
} yield b * c //  6 * 4 = 24

// Note that the execution of futures a, b, and c
// are not done in parallel.

f foreach println
```

代码 7-4

7.5 大量 Future 配合使用

前面两节都是 Future 对象数量比较少的情况下配合使用，如果 Future 数量很

多，要采用 sequence 辅助方法来进行 Future 的配合使用，如下代码 7-5 所示，将通过 sequence 方法将 List[Future[Int]]转换成 Future[List[Int]]，就是将许多个 Future 转换成一个 Future。

```
val actor1 = system.actorOf(Props[Actor1], name="actor1")

val listOfFutures = List.fill(10)(ask(actor1, GetNext).mapTo[Int])

val futureList = Future.sequence(listOfFutures)

val sum = futureList.map(_.sum)

sum foreach println
```

代码 7-5

Traverse 方法和 sequence 类似，区别在于需要传入两个参数，如下代码 7-6 所示，传入 List[Int]类型的参数和返回 Future[Int]的方法参数，比起 sequence，少了一个 List[Future[Int]]的中间变量。Future 的 fold 和 reduce 方法可以对传入的 List[Future[Int]]中的大量 Future 对象做并发处理，区别在于 reduce 方法用第一个完成的 Future 对象作为开始计算值，fold 需要传入初始计算值。

```
val futureList1 = Future.traverse((1 to 100).toList)(x => Future(x * 2 - 1))
val oddSum = futureList.map(_.sum)
oddSum foreach println

val futures = for (i <- 1 to 1000) yield Future(i * 2)
val futureSum1 = Future.fold(futures)(0)(_ + _)
futureSum1 foreach println

val futureSum2 = Future.reduce(futures)(_ + _)
futureSum2 foreach println
```

代码 7-6

7.6 Future 回调

有时只想对 Future 对象的结果产生副作用，而不是得到返回结果，可以采用如下代码 7-7 所示的回调函数来实现。

```
future onSuccess {
  case "bar"       =>println("Got my bar alright!")
  case x: String => println("Got some random string: " + x)
}

future onFailure {
  case ise: IllegalStateException if ise.getMessage == "OHNOES" =>
  case e: Exception =>
}

future onComplete {
  case Success(result)  =>doSomethingOnSuccess(result)
  case Failure(failure) =>doSomethingOnFailure(failure)
}
```

代码 7-7

有时希望顺序执行多个 Future 对象，前一个影响后一个，如下代码 7-8 所示，

```
val result = Future { 1+1 } andThen {
    case v => v
} andThen {
  case v => v
}
```

代码 7-8

有时希望当一个 Future 失败后，使用另一个成功 Future 的结果，如下代码 7-9 所示，

```
val future4 = future1 fallbackTo future2 fallbackTo future3
```

代码 7-9

有时希望对两个 Future 的成功结果一起做处理，如下代码 7-10 所示，

```
val future3 = future1 zip future2 map { case (a, b) => a + " " + b }
```

代码 7-10

对 Future 对象的异常处理，如下代码 7-11 所示，recover 和 recoverWith，他们区别如同 map 和 flatMap。

```
val future = akka.pattern.ask(actor, msg1) recover {
  case e: ArithmeticException ⇒ 0
}
val future = akka.pattern.ask(actor, msg1) recoverWith {
  case e: ArithmeticException ⇒ Future.successful(0)
  case foo: IllegalArgumentException ⇒
    Future.failed[Int](new IllegalStateException("All br0ken!"))
}
```

代码 7-11

akka.pattern.after 使 Future 要么得到一个值，要么得到一个异常在超时后，如下代码 7-12 所示。

```
val delayed = after(200 millis, using = system.scheduler)(Future.failed(
  new IllegalStateException("OHNOES")))
val future = Future { Thread.sleep(1000); "foo" }
val result = Future firstCompletedOf Seq(future, delayed)
```

代码 7-12

7.7 Dataflow 并发

AKKA 提供了一种 Dataflow 并发的方式开发并发和非阻塞程序，这种方式可以用看上去像同步代码的格式写程序，实际却实现了异步功能，如下代码 7-13 所示。Dataflow 并发方式开发的代码的优点在于无论运行多少次，其结果都是一样的，比如，一个方法运行第一次是死锁，以后每次运行都是死锁，结果是不会变化的。

```
val v1, v2 = Promise[Int]()
flow {
    v1 << v2() + 10
    v1() + v2()
} onComplete println
flow { v2 << 5 }

val f1, f2 = Future { 1 }

val usingFor = for { v1 <- f1; v2 <- f2 } yield v1 + v2
val usingFlow = flow { f1() + f2() }

usingFor onComplete println
usingFlow onComplete println
```

代码 7-13

注意：

- Futures are readable-many, using the apply method, inside flow blocks.
- Promises are readable-many, just like Futures.
- Promises are writable-once, using the << operator, inside flow blocks. Writing to an already written Promise throws a java.lang.IllegalStateException, this has the effect that races to write a promise will be deterministic, only one of the writers will succeed and the others will fail.
- 该部分内容可以参阅 Scala's Continuations。

8 消息总线实例

8.1 消息总线的功能和作用

‘总线’顾名思义，消息运行的路径。消息总线的功能和作用是在整个 PAAS 平台中对各种消息进行路由和转发，使 PAAS 平台中的各个模块整合到一起；消息总线具有负载均衡的功能，使 PAAS 平台中的各个模块集群化，每个模块不再是单一结点，而是具有多个功能相同的结点组成的群。集群化的系统有更好的并发性、更大的吞吐量和更强的抗灾性。

8.2 PAAS 平台架构

PAAS 平台架构如下图 8-1 所示，com.beyond.paas.messagecenter 消息总线包实现了消息总线功能，其中消息总线的实现依赖于 com.beyond.paas.model 模块包中的具体某个模块的 service 包中的服务来生成相应的 router 和 routee，com.beyond.paas.tool 工具包向模块提供提供底层支持，比如数据存取和公式解析等功能。

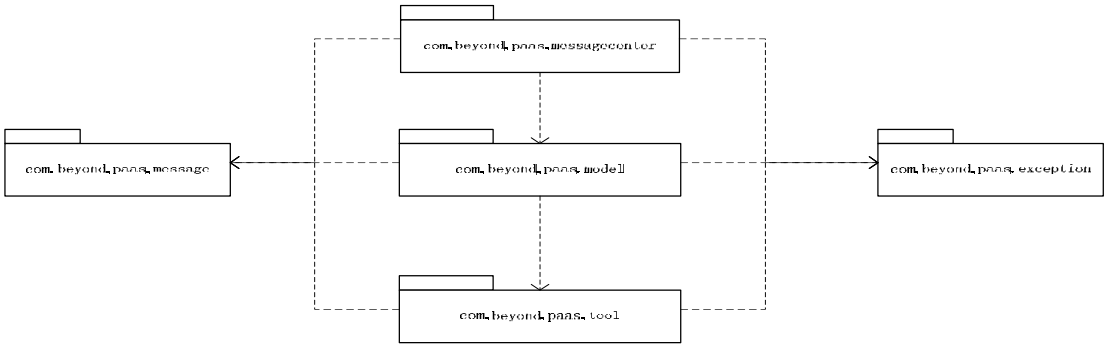

图 8-1 PAAS 平台架构图

消息总线架构如下图 8-2 所示，当前版本的消息总线 com.beyond.pass.messagecenter.akka 依赖于 AKKA-1.4.1 API 实现。

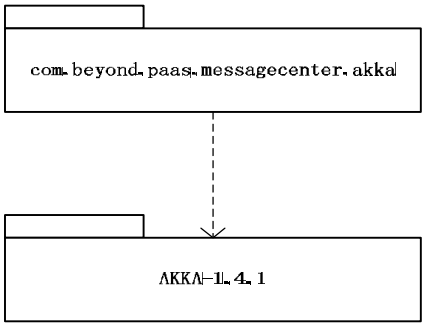

图 8-2 消息总线架构图

模块架构如下图 8-3 所示，目前 PAAS 模块有 com.beyond.pass.model.metadata

元数据包、com.beyond.pass.model.multilanguage 多国语言包、com.beyond.pass.model.report 报表包、com.beyond.pass.model.uiengine 界面引擎包、com.beyond.pass.model.workflow 工作流包、com.beyond.pass.model.authority 权限包和 com.beyond.pass.model.log 日志包，它们都依赖于 com.beyond.pass.model.akka 包，每个模块相互独立，不能直接调用，模块需要通过 com.beyond.pass.model.akka 包和消息总线进行通信，来实现调用其它模块的功能。

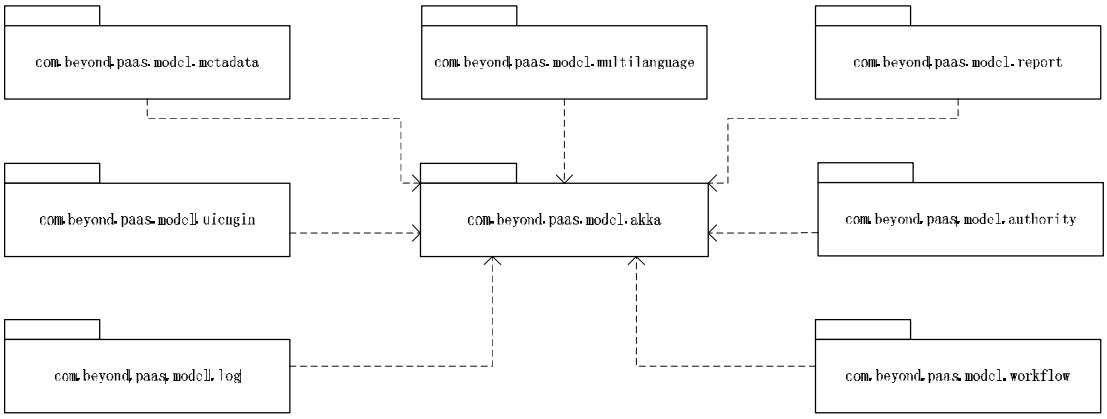

图 8-3 模块架构图

模块内部架构如下图 8-4 所示，Service 包依赖于 DAO 包实现数据的存取，并且 Service 和 DAO 包都依赖于 Entity 包来是实现数据的转递，Service 包是模块的对外接口。

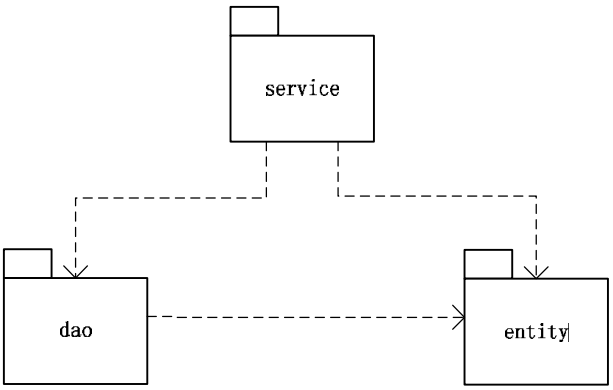

图 8-4 模块内部架构

8.3 消息总线类

消息总线 com.beyond.paas.messagecenter.akka 包下括三个类，有 MessCentImplAkka、RouterImplAkka 和 NodeStateImplAkka。MessCentImplAkka 实现对集群中结点和路由 Actor 的管理，通过引用 RouterImplAkka 创建路由 Actor；RouterImplAkka 现实路由功能，通过引用 ModelActorImplAkka 实现不同的 router 和 routee；NodeStateImplAkka 实现集群中结点状态的维护。

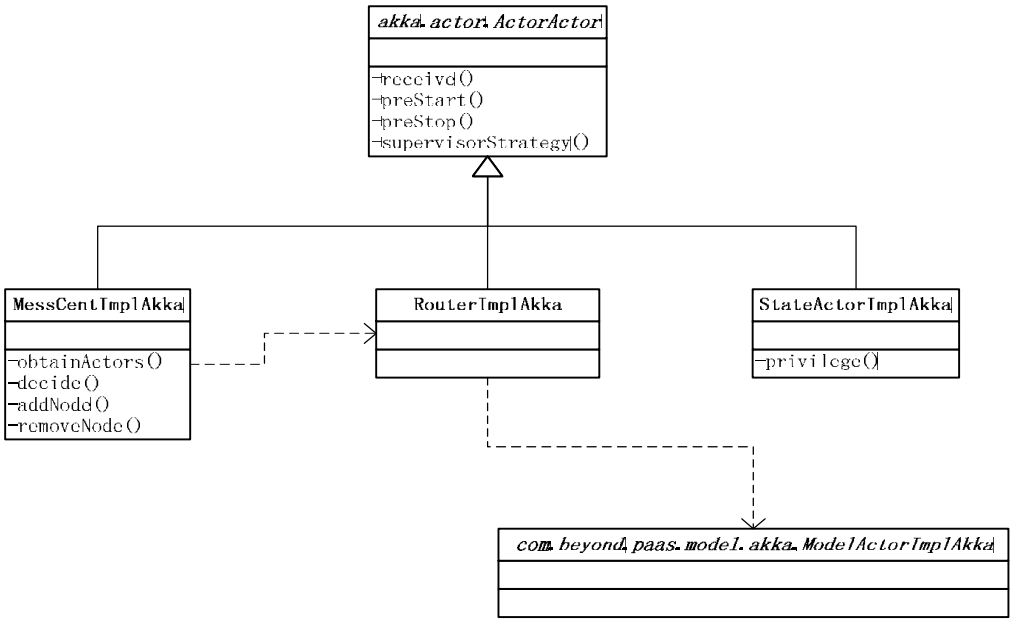

图 8-5 消息总线类

8.4 消息总线时序图

消息总线初始化序列图如下图 8-6 所示。

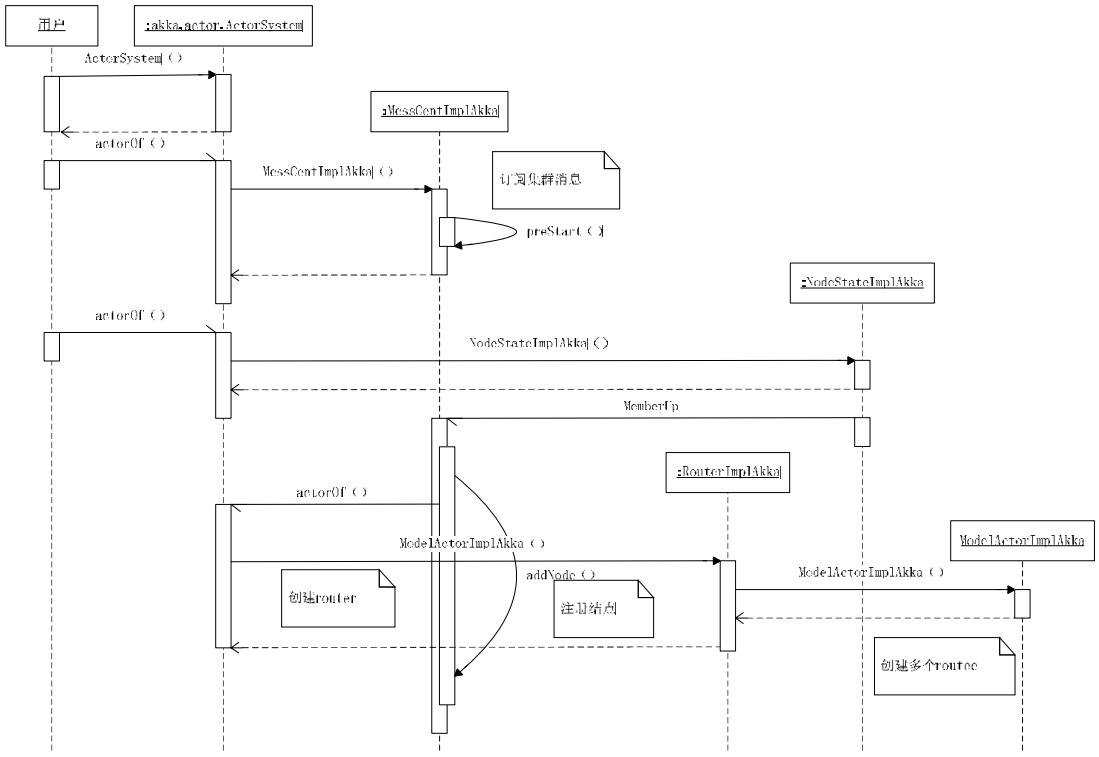

图 8-6 初始化序列图

消息总线调用序列图如下图 8-7 所示。

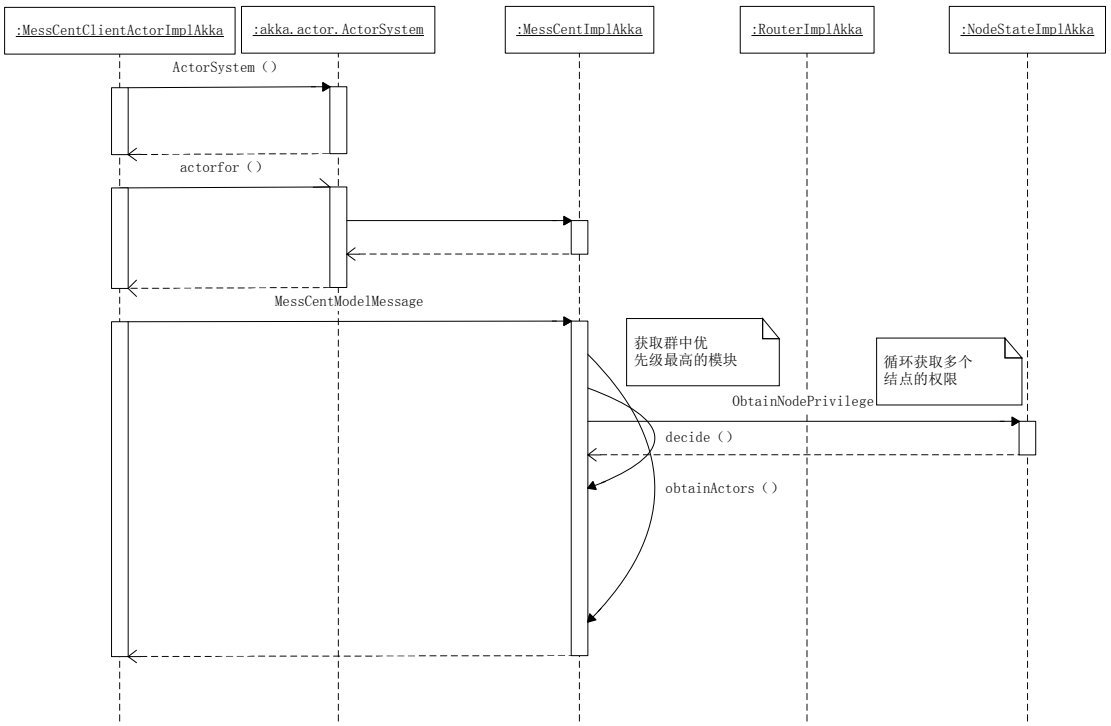

图 8-7 获取模块序列图

消息总线结点退出集群序列图如下图 8-8 所示。

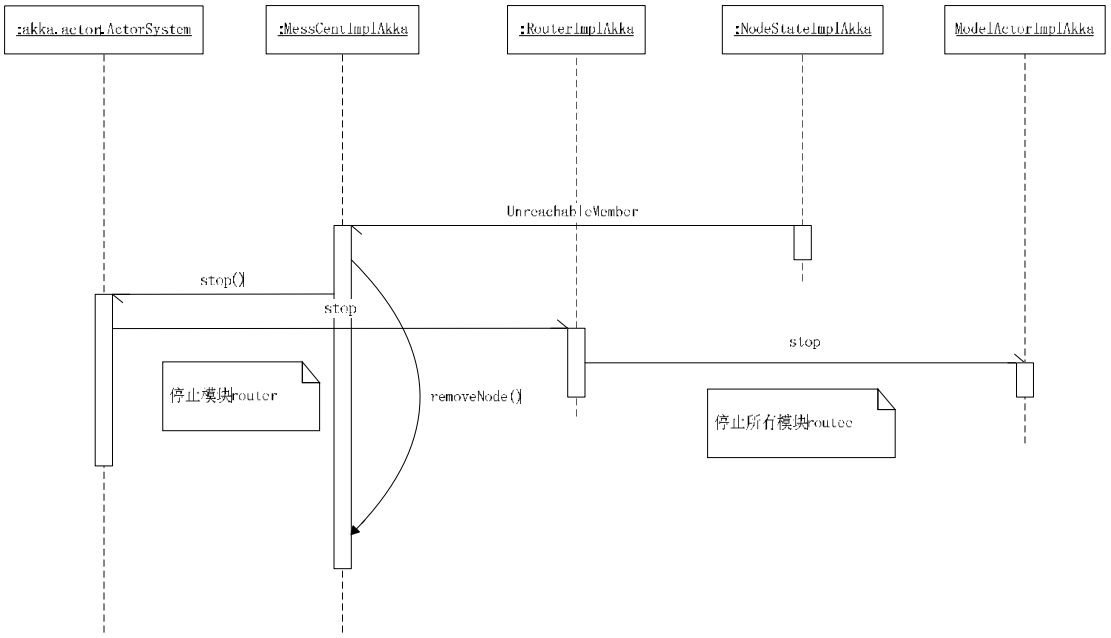

图 8-8 结点退出集群序列图

8.5 消息总线客户代码调用机制

客户端代码通过使用 com.beyond.paas.model.akka 包中的 MessCentClientActorImplAkka 类提供的方法实现和模块的通讯。客户代码调用机制序列图如下图 8-9 所示。

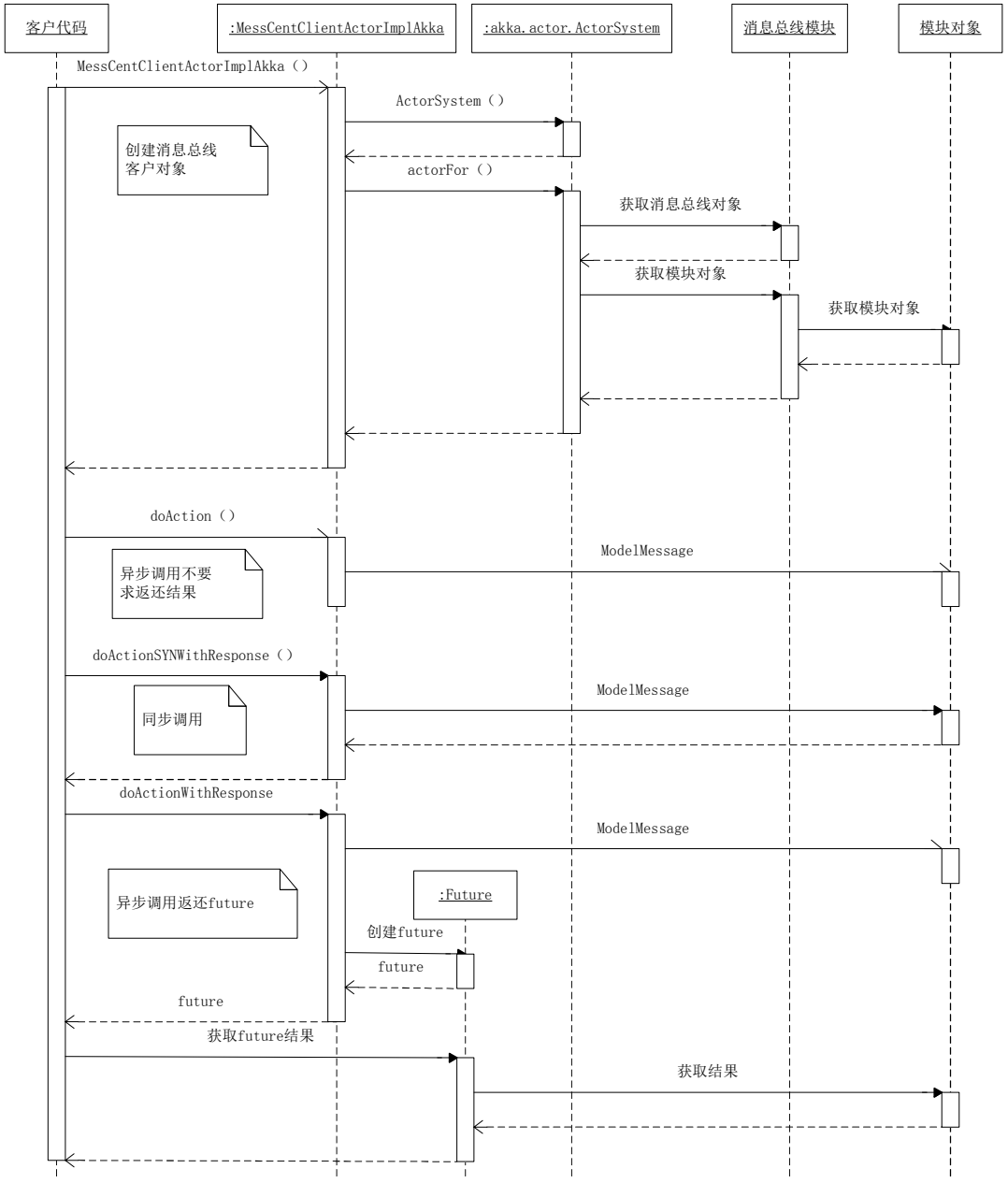

图 8-9 客户代码调用机制序列图

8.6 消息总线服务代码实现机制

服务端代码通过继承 com.beyond.paas.model.akka 包中的 ModelActorImplAkka 类实现各自的业务逻辑。服务端代码实现机制类图如下图 8-10 所示。

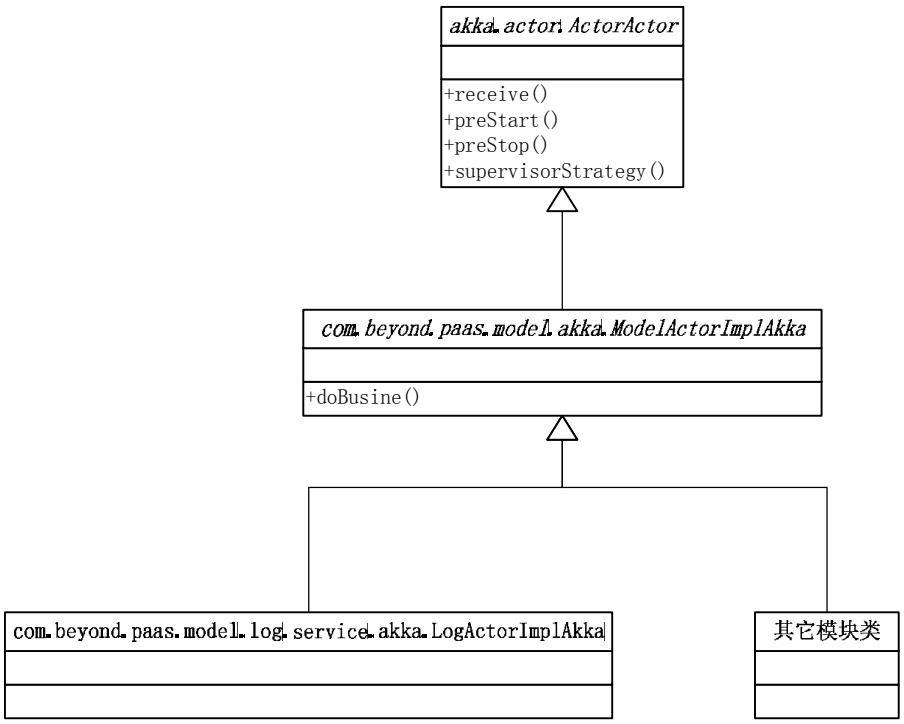

图 8-10 服务端代码实现机制类图

8.7 PAAS 平台消息定义机制

PAAS 平台消息定义类图如下图 8-11 所示。

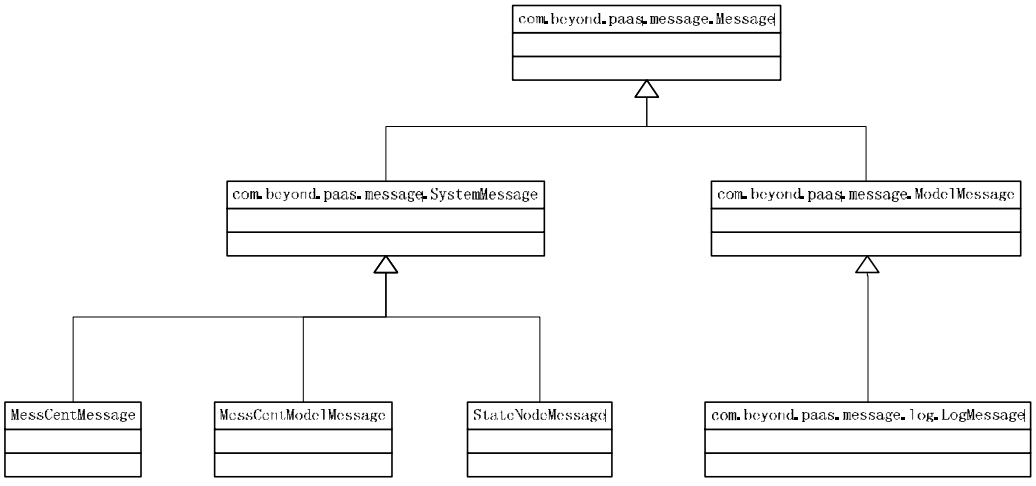

图 8-11 PAAS 平台消息定义类图

8.8 PAAS 平台异常定义机制

PAAS 平台异常定义类图如下图 8-12 所示。

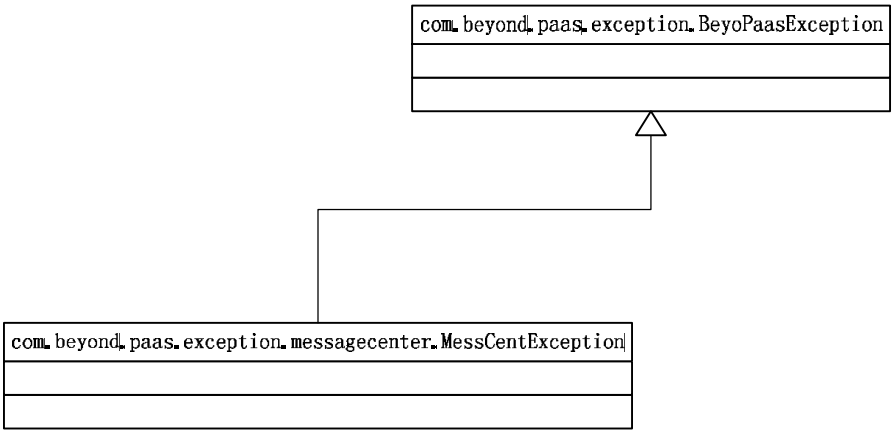

图 8-12 PAAS 平台异常定义类图

8.9 PAAS 平台包结构

PAAS 平台包结构图如下 8-13 所示。

图 8-13 包结构图

9 注意事项

- 要尽量避免在一个 actor 中关闭另一个 actor，这可能会引起 Bug 和条件竞争。
- 消息一定要采用不可变的对象，在 scala 中不可变对象有 String、Int、Double 等的对象，还有 case class 类型的对象，以及 scala.Tuple2、scala.List 和 scala.Map 类型的对象。
- 出于性能的考虑，发送消息尽量采用“!”（tell），除非有必要才使用“？”（ask）。
- 由于 AKKA 邮件先进先出的机制，任何系统或配置产生的系统消息，未必会第一时间被 receive 方法处理，可能排在其它消息后面被处理。
- 初始化 Actor 可以通过消息的方式，配合 stack、become 等方法，但是要尽量避免使用。
- 开发 Actor 程序时，一般采用 one-for-one 容错机制，有特殊需求才使用 all-for-one 容错机制。
- 尽量避免使用除 actor 以外的同步机制，例如 sychronized、thread 和 lock 等。
- Actor 类最好是无状态的。

10 代码规范

- 为了代码便于理解，类必须有注解。
- 为了代码便于理解，公共方法和属性必须有注解。
- 为了代码便于理解，私有方法和属性如果比较复杂要有注解。
- 为了代码可读，代码行缩进 2 个字符。
- 为了代码可读，如果行和行之间没有必要联系，应该在两行之间空一行。
- 为了代码可读，方法内第一行应该是空行。
- 为了代码可读，类名、方法名和属性名应该采用可以理解的英语单词。
- 为了代码可读，类名中每个单词首字母大写。
- 为了代码可读，方法名和属性每个单词首字母大写，除了一个单词。
- 为了代码可读，不可变属性可以采用全部大写字符，表示为常量。
- 为了代码可读，最好使用类伴随对象来创建实例。
- 类的方法定义在类伴随对象中，起到 Java 静态方法的作用。
- 使用集合时尽量采用结合类的方法来代替 for。
- 一个文件中；要么每句都用，要么都不用。
- 尽量使用 val 代替 var，private 代替 public。
- 类和类之间的复杂关系，要有相关静态类图和序列图进行说明。
- 复杂算法要求有伪代码说明。
- 尽量使用引用代替继承。
- 类的设计要做到高内聚低耦合，可以采用各种设计模式解耦。
- 尽量采用局部变量代替全局变量。

- 尽量不要把未构造完的对象作为参数传给另一个对象的构造函数。
- 消息必须是 case class 实例、case object 对象和 ActorRef 类对象，不要使用其它类型。
- 消息必须是可序列化的对象，防止远程调用时的异常。
- 模块对外接口不能轻易修改，要和使用者沟通。

11 总结和展望

目前 PAAS 平台只实现了消息总线的功能，其它模块需要进行设计和开发。消息总线只实现了基本功能，后续还有异常处理机制、管理机制和管理工具、Actor 模下 actor 和 actor 之间的同步机制（事务功能）、PAAS 平台编译机制和 actor 测试机制等设计和开发工作。

12 附件代码

BeyoPaasException.scala

```
package com.beyond.paas.exception

/**
 * All Beyond Paas's parent exception.
 */
trait BeyoPaasException extends Exception
```

MessCentException.scala

```
package com.beyond.paas.exception.messagecenter

import com.beyond.paas.exception.BeyoPaasException

/**
  * All Message Center's parent exception.
  */
trait MessCentException extends BeyoPaasException

/**
  * Stop Message Center Exception
  */
class StopException extends MessCentException

/**
  * Resume Message Center Exception
  */
class ResumeException extends MessCentException

/**
  * Restart Message Center Exception
  */
class RestartException extends MessCentException

/**
  * Escalate Message Center Exception
  */
class EscalateException extends MessCentException
```

```
/**
  * Bad Parameter Exception in Message Center
  */
class BadParameterException(val para: String) extends MessCentException

/**
  * Empty Parameter Exception in Message Center
  */
class EmptyParameterException extends MessCentException

/**
  * Can not obtain Actor Exception in Message Center
  */
class CannotObtainActorException extends MessCentException
```

BeyondPaasMessage.scala

```
package com.beyond.paas.message

/**
  * All Beyond Paas's parent message
  */
trait Message

/**
  * All System's parent message
  */
trait SystemMessage extends Message

/**
  * All Model's parent message
  */
trait ModelMessage extends Message
```

LogMessage.scala

```
package com.beyond.paas.message.log

import com.beyond.paas.message.ModelMessage

/**
  * All log model's parent message
  */
sealed trait LogMessage extends ModelMessage
```

```
case class TestLogMessage extends LogMessage
```

MessCentMessage.scala

```
package com.beyond.paas.message.messagecenter

import com.beyond.paas.message.SystemMessage

/**
  * All Message Center's parent Message
  */
sealed trait MessCentMessage extends SystemMessage

case object RestartMessageCenter extends MessCentMessage

case object StopMessageCenter extends MessCentMessage

case class RestartModel(nodeAdress: String) extends MessCentMessage

case class StopModel(nodeAdress: String) extends MessCentMessage

/**
  * All Message Center model's parent Message
  */
sealed trait MessCentModelMessage extends SystemMessage

case object ObtainMetaDataActor extends MessCentModelMessage

case object ObtainUIEngineActor extends MessCentModelMessage

case object ObtainAuthorityActor extends MessCentModelMessage

case object ObtainReportActor extends MessCentModelMessage

case object ObtainWorkFlowActor extends MessCentModelMessage

case object ObtainMultiLanguageActor extends MessCentModelMessage

case object ObtainLogActor extends MessCentModelMessage

/**
  * All State Node Actor's parent Message
  */
```

```
sealed trait StateNodeMessage extends SystemMessage

case object ObtainNodeType extends StateNodeMessage

case object ObtainNodePrivilege extends StateNodeMessage

case object OpenNode extends StateNodeMessage

case object CloseNode extends StateNodeMessage
```

MessCentImplAkka.scala

```
package com.beyond.paas.messagecenter.akka

import akka.actor.Actor
import akka.actor.ActorLogging
import akka.actor.ActorRef
import akka.actor.ActorSystem
import akka.actor.Address
import akka.actor.OneForOneStrategy
import akka.actor.Props
import akka.actor.SupervisorStrategy._
import akka.pattern.ask
import akka.cluster.Cluster
import akka.cluster.ClusterEvent.ClusterDomainEvent
import akka.cluster.ClusterEvent.MemberUp
import akka.cluster.ClusterEvent.UnreachableMember
import akka.cluster.Member
import akka.util.Timeout
import com.typesafe.config.ConfigFactory
import com.beyond.paas.exception.messagecenter._
import com.beyond.paas.message.messagecenter._
import scala.concurrent.duration._
import scala.concurrent.Await
import scala.concurrent.Future
import akka.actor.Deploy
import akka.remote.RemoteScope

/**
 * Message Center Actor
 * The MessCentImplAkka class implements Akka's Actor. The MessCentImplAkka instance serves as message bus that only dispatches
 * and routes messages from client program call to model actors, then return model actors to client program. In another words,
```

```
 * after client programs obtain model actors, client programs send message to model actors directly.
 */
class MessCentImplAkka extends Actor with ActorLogging{

  //Message Center Actor's address
  private[this] val selfAddress = Cluster(context.system).selfAddress

  //State Actor's Path
  private[this] val StateActorPath = "/user/StateActor"

  //Timeout for unsynchronized call
  private[this] implicit val timeout = Timeout(2 seconds)

  //To save started nodes
  private[this] var logNodes: Map[Address, ActorRef] = Map()

  private[this] var metadataNodes: Map[Address, ActorRef] = Map()

  private[this] var uiEnginelogNodes: Map[Address, ActorRef] = Map()

  private[this] var authorityNodes: Map[Address, ActorRef] = Map()

  private[this] var reportNodes: Map[Address, ActorRef] = Map()

  private[this] var workFlowNodes: Map[Address, ActorRef] = Map()

  private[this] var multiLanguageNodes: Map[Address, ActorRef] = Map()
  //To save started nodes

  /**
   * Subscribe cluster event.
   */
  override def preStart(): Unit =
    Cluster(context.system).subscribe(self, classOf[ClusterDomainEvent])

  /**
   * Unsubscribe cluster event.
   */
  override def postStop(): Unit =
    Cluster(context.system).unsubscribe(self)

  /**
   * Handle cluster event, MessCentModelMessage message and MessCentMessage.
```

```
  */
  def receive = {
    case MemberUp(member) =>
      log.info("jvm is Up: {}", member.address)
      if(member.address!=selfAddress) addNode(member.address)

    case UnreachableMember(member) =>
      log.info("jvm is down: {}", member.address)
      if(member.address!=selfAddress) removeNode(member.address)

    case message : MessCentModelMessage => sender ! obtainActors(message)

    case RestartMessageCenter => throw new RestartException // this exception will let akka's root to restart message center actor

    case StopMessageCenter => context.system.shutdown // this action will let akka's system to shut down.
  }

  /**
   * Supervise children actors and handle exceptions from children actors
   */
  override val supervisorStrategy =
    OneForOneStrategy() {
        case _: StopException => Stop
        case _: ResumeException => Resume
        case _: RestartException => Restart
        case _: EscalateException => Escalate
        case _ => Escalate
  }

  /**
   * Obtain a model actor and a state actor.
   */
  private[this] def obtainActors(message : MessCentModelMessage): (ActorRef, ActorRef) = {

    val nodes = message match {
      case ObtainMetaDataActor => metadataNodes
      case ObtainUIEngineActor => uiEnginelogNodes
      case ObtainAuthorityActor => authorityNodes
      case ObtainReportActor => reportNodes
      case ObtainWorkFlowActor => workFlowNodes
      case ObtainMultiLanguageActor => multiLanguageNodes
      case ObtainLogActor => logNodes
```

```
        case _ => null
    }

    if(nodes==null || nodes.isEmpty){
        (null, null)
    }else{
        val (router, stateActor) = decide(nodes)

        val future = (router ? message).mapTo[ActorRef]
        val modelActor = Await.result(future, timeout.duration)
        (modelActor, stateActor)
    }

}

/**
  * To choose a model actor on a state actor's privilege.
  */
private[this] def decide(nodes: Map[Address, ActorRef]): (ActorRef, ActorRef)    = {

    var minStateActor : ActorRef = null
    var router : ActorRef = null

    for(address <- nodes.keys) {
        val stateActor = context.actorFor(address+StateActorPath)

        if(minStateActor==null) {
            minStateActor = stateActor
            router = nodes(address)
        }else{
            val future1: Future[Int] = (minStateActor ? ObtainNodePrivilege).mapTo[Int]
            val future2: Future[Int] = (stateActor ? ObtainNodePrivilege).mapTo[Int]

            val privilege1 = Await.result(future1, timeout.duration)
            val privilege2 = Await.result(future2, timeout.duration)

            if(privilege1>privilege2) {
                minStateActor = stateActor
                router = nodes(address)
            }
        }
    }

    (router, minStateActor)
```

```
  }

  /**
   * When a node start up, message center register the node and create a children actor (a router) on the node.
   */
  private[this] def addNode(address: Address) {

    val stateActor = context.actorFor(address+StateActorPath)

    val future: Future[MessCentModelMessage] = (stateActor ? ObtainNodeType).mapTo[MessCentModelMessage]

    val nodeType = Await.result(future, timeout.duration)

    val nodes = nodeType match {
      case ObtainMetaDataActor => metadataNodes
      case ObtainUIEngineActor => uiEnginelogNodes
      case ObtainAuthorityActor => authorityNodes
      case ObtainReportActor => reportNodes
      case ObtainWorkFlowActor => workFlowNodes
      case ObtainMultiLanguageActor => multiLanguageNodes
      case ObtainLogActor => logNodes
      case _ => null
    }

    if(nodes!=null && !nodes.contains(address) ) {
      val actor = context.actorOf(Props(new RouterImplAkka(nodeType)).
          withDeploy(Deploy(scope = RemoteScope(address))))

      nodeType match {
        case ObtainMetaDataActor => metadataNodes = nodes + (address -> actor)
        case ObtainUIEngineActor => uiEnginelogNodes = nodes + (address -> actor)
        case ObtainAuthorityActor => authorityNodes = nodes + (address -> actor)
        case ObtainReportActor => reportNodes = nodes + (address -> actor)
        case ObtainWorkFlowActor => workFlowNodes = nodes + (address -> actor)
        case ObtainMultiLanguageActor => multiLanguageNodes = nodes + (address -> actor)
        case ObtainLogActor => logNodes = nodes + (address -> actor)
        case _ =>
      }
    }

  }
```

```
  /**
   * When a node is unreachable , message center unregister the node and stop a children actor.
   */
  private[this] def removeNode(address: Address) {

    if(metadataNodes.contains(address)){
      context.stop(metadataNodes.get(address).get)
      metadataNodes = metadataNodes - address
    }else if(uiEnginelogNodes.contains(address)){
      context.stop(uiEnginelogNodes.get(address).get)
      uiEnginelogNodes = uiEnginelogNodes - address
    }else if(authorityNodes.contains(address)){
      context.stop(authorityNodes.get(address).get)
      authorityNodes = authorityNodes - address
    }else if(reportNodes.contains(address)){
      context.stop(reportNodes.get(address).get)
      reportNodes = reportNodes - address
    }else if(workFlowNodes.contains(address)){
      context.stop(workFlowNodes.get(address).get)
      workFlowNodes = workFlowNodes - address
    }else if(multiLanguageNodes.contains(address)){
      context.stop(multiLanguageNodes.get(address).get)
      multiLanguageNodes = multiLanguageNodes - address
    }else if(logNodes.contains(address)){
      context.stop(logNodes.get(address).get)
      logNodes = logNodes - address
    }
  }

}

/**
 * To create and run Message Center Actor.
 */
object MessCentImplAkka extends App{

  val config = ConfigFactory.parseString("akka.remote.netty.port=2551")
      .withFallback(ConfigFactory.load())

  val system = ActorSystem("BeyondPaasFramework", config)

  system.actorOf(Props[MessCentImplAkka], "MessageCenter")
```

```
}
```

NodeStateImplAkka.scala

```
package com.beyond.paas.messagecenter.akka

import akka.actor.Actor
import akka.actor.ActorSystem
import akka.actor.Props
import com.typesafe.config.ConfigFactory
import com.beyond.paas.exception.messagecenter._
import com.beyond.paas.message.messagecenter._

/**
 * Node State Actor
 * NodelStateImplAkka class implement AKKA Actor. The StateActor's instance contains the model actor's state in whole life.
 */
class NodeStateImplAkka(nodeType: MessCentModelMessage) extends Actor{

   //the number of node's connection
   private var volunm = 0

   def receive = {
      case ObtainNodeType => sender ! nodeType
      case ObtainNodePrivilege => sender ! privilege
      case OpenNode => volunm += 1
      case CloseNode => volunm -= 1
   }

   /**
    * To calculate node's privilege
    */
   private def privilege(): Int = {
      volunm
   }

}

/**
 * To start node and create node's state actor.
 */
object NodeStateImplAkka{
   def main(args: Array[String]): Unit = {
```

```
        if(args.size == 0) throw new EmptyParameterException

        val nodeType : MessCentModelMessage = args(0) match {
            case "MetaData" => ObtainMetaDataActor
            case "UIEngine" => ObtainUIEngineActor
            case "Authority" => ObtainAuthorityActor
            case "Report" => ObtainReportActor
            case "WorkFlow" => ObtainWorkFlowActor
            case "MultiLanguage" => ObtainMultiLanguageActor
            case "Log" => ObtainLogActor
            case _ => throw new BadParameterException(args(0))
        }

        val config = ConfigFactory.load()

        val system = ActorSystem("BeyondPaasFramework", config)

        system.actorOf(Props(new NodeStateImplAkka(nodeType)), "StateActor")

        println(args(0)+"'s node is running!")
    }
}
```

RouterImplAkka.scala

```
package com.beyond.paas.messagecenter.akka

import akka.actor.Actor
import akka.actor.ActorRef
import akka.actor.OneForOneStrategy
import akka.actor.Props
import akka.actor.SupervisorStrategy._
import akka.pattern.ask
import akka.routing.RoundRobinRouter
import akka.util.Timeout
import com.beyond.paas.exception.messagecenter._
import com.beyond.paas.message.messagecenter._
import com.beyond.paas.model.log.service.akka.LogActorImplAkka
import scala.concurrent.Await
import scala.concurrent.duration._

/**
 * Router Actor
```

```
 * RouterImplAkka's class implements AKKA Actor. RouterImplAkka's instance can be created according to routerType in the
 * constructor. RouterImplAkka's manages model actors.
 */
class RouterImplAkka(routerType: MessCentModelMessage) extends Actor{

  //Timeout for unsynchronized call
  private implicit val timeout = Timeout(1 seconds)

  private[this] val router = routerType match {
    case ObtainMetaDataActor => context.actorOf(Props.empty.withRouter(RoundRobinRouter(nrOfInstances = 20)))
    case ObtainUIEngineActor => context.actorOf(Props.empty.withRouter(RoundRobinRouter(nrOfInstances = 20)))
    case ObtainAuthorityActor => context.actorOf(Props.empty.withRouter(RoundRobinRouter(nrOfInstances = 20)))
    case ObtainReportActor => context.actorOf(Props.empty.withRouter(RoundRobinRouter(nrOfInstances = 20)))
    case ObtainWorkFlowActor => context.actorOf(Props.empty.withRouter(RoundRobinRouter(nrOfInstances = 20)))
    case ObtainMultiLanguageActor => context.actorOf(Props.empty.withRouter(RoundRobinRouter(nrOfInstances = 20)))
    case ObtainLogActor => context.actorOf(Props[LogActorImplAkka].withRouter(RoundRobinRouter(nrOfInstances = 20)))
    case _ => null
  }

  def receive = {
    case message: MessCentModelMessage => //choose one model actor in the router
      if(router!=null){
        val future = (router ? message).mapTo[ActorRef]
        val modelActor = Await.result(future, timeout.duration)
        sender ! modelActor
      }else{
        sender ! null
      }

    case RestartMessageCenter => throw new RestartException // this exception will let Message Center actor to restart Router actor

    case StopMessageCenter => throw new StopException // this exception will Message Center actor to stop Router actor
  }
```

```
    /**
     * Supervise children actors and handle exceptions from children actors
     */
   override val supervisorStrategy =
      OneForOneStrategy() {
         case _: StopException => Stop
         case _: ResumeException => Resume
         case _: RestartException => Restart
         case _: EscalateException => Escalate
         case _ => Escalate
   }

}
```

MessCentClientActorImplAkka.scala

```
package com.beyond.paas.model.akka

import akka.actor.ActorRef
import akka.actor.ActorSystem
import akka.pattern.ask
import akka.util.Timeout
import com.typesafe.config.ConfigFactory
import com.beyond.paas.exception.messagecenter.CannotObtainActorException
import com.beyond.paas.message.ModelMessage
import com.beyond.paas.message.messagecenter._
import com.beyond.paas.message.log._
import scala.concurrent.duration._
import scala.concurrent.Await
import scala.concurrent.Future

/**
  * MessCentClientActorImplAkka class serve as actor's agencies that is called by client programs.
  */
class MessCentClientActorImplAkka(actorType: MessCentModelMessage) {

   private val system = ActorSystem("MessageCenterClient", ConfigFactory.load("client"))

   private val messageCenter = system.actorFor("akka://BeyondPaasFramework@127.0.0.1:2551/user/MessageCenter")

   private implicit val timeout = Timeout(2 seconds)

   private val (modelActor, stateActor) = {
```

```
      val future = (messageCenter ? actorType).mapTo[(ActorRef, ActorRef)]
      Await.result(future, timeout.duration)
    }

    if(modelActor==null || stateActor==null) throw new CannotObtainActorException

    stateActor ! OpenNode

    /**
     * To handle synchronized message request.
     */
    def doActionSYNWithResponse(message: ModelMessage): Any ={
      val future = doActionWithResponse(message)
      Await.result(future, timeout.duration)
    }

    /**
     * To handle unsynchronized message request.
     */
    def doActionWithResponse(message: ModelMessage): Future[Any] = (modelActor ? message).mapTo[Any]

    /**
     * To handle unsynchronized message request, but don't obtain response.
     */
    def doAction(message: ModelMessage) = modelActor ! message

    /**
     * To close MessCentClientActorImplAkka's instance.
     */
    def close() {
      stateActor ! CloseNode
      system.shutdown
    }

}

/**
 * MessCentClientActorImplAkka's accompany object
 */
object MessCentClientActorImplAkka {

    def apply(actorType: MessCentModelMessage) = {
      new MessCentClientActorImplAkka(actorType)
```

```
    }
}

object test extends App {

    val client = MessCentClientActorImplAkka(ObtainLogActor)

    client.doAction(TestLogMessage())

    client.close()
}
```

ModelActorImplAkka.scala

```
package com.beyond.paas.model.akka

import akka.actor.Actor
import akka.actor.ActorRef
import com.beyond.paas.message.messagecenter.MessCentModelMessage
import com.beyond.paas.message.ModelMessage

abstract class ModelActorImplAkka extends Actor{

    def receive = {
        case _ : MessCentModelMessage => sender ! self
        case message : ModelMessage => doBusine(message, sender)

    }

    def doBusine(message: ModelMessage, sender: ActorRef)

}
```

LogActorImplAkka.scala

```
package com.beyond.paas.model.log.service.akka

import akka.actor.ActorRef
import com.beyond.paas.model.akka.ModelActorImplAkka
import com.beyond.paas.message.ModelMessage

/**
 * LogActorImplAkka class implement ModelActorImplAkka to handle log event.
 */
```

```
class LogActorImplAkka extends ModelActorImplAkka{

   def doBusine(msg: ModelMessage, sender: ActorRef) {
      println(self)
      println("log demo")
   }
}
```

application.conf

```
akka {
  actor {
    provider = "akka.cluster.ClusterActorRefProvider"
  }
  remote {
    transport = "akka.remote.netty.NettyRemoteTransport"
    log-remote-lifecycle-events = off
    netty {
      hostname = "127.0.0.1"
      port = 0
    }
  }
  cluster {
    seed-nodes = ["akka://BeyondPaasFramework@127.0.0.1:2551"]

    auto-down = on
  }
}
```

client.conf

```
akka {
  actor {
    provider = "akka.remote.RemoteActorRefProvider"
  }
  remote {
    transport = "akka.remote.netty.NettyRemoteTransport"
    log-remote-lifecycle-events = off
    netty {
      hostname = "127.0.0.1"
      port = 0
    }
  }
}
```

13 参考资料

- AKKA1.4.1 官方开发文档
- AKKA1.4.1 官方 API 文档
- SCALA2.10.1 官方 API 文档
- 《基于函数式语言的网络消息服务器的构建方法研究》
- 《快学 Scala》
- 《Scala in Program》