配置

Akka 的所有配置信息装在 ActorSystem 的实例中，从外界看来，ActorSystem 是配置信息的唯一消费者.

在创建一个 ActorSystem 对象时，你可以传进来一个 Config 对象参数；

如果不传，就会使用 **ConfigFactory.load()**读取 classpath 根目录下的所有 application.conf、application.json、application.properties 这些文件。

然后 ActorSystem 会合并 classpath 根目录下的 reference.conf 来组成其内部使用的缺省配置。

注意：

如果你编写的是一个 Akka 应用，把配置放在 classpath 根目录下的 application.conf 中.

如果你编写的是一个基于 Akka 的库，把配置放在 jar 包根目录下的 reference.conf 中.

组织你的配置：

由于 ConfigFactory.load() 会合并 classpath 中所有匹配名称的资源，最简单的方式是利用这一功能而在配置树中区分 actor 系统:

```
myapp1 {
  akka.loglevel = WARNING
  my.own.setting = 43
}
myapp2 {
  akka.loglevel = ERROR
  app2.setting = "appname"
}
my.own.setting = 42
my.other.setting = "hello"

val config = ConfigFactory.load()
val app1 = ActorSystem("MyApp1", config.getConfig("myapp1").withFallback(config))
val app2 = ActorSystem("MyApp2", config.getConfig("myapp2").withOnlyPath("akka").withFallback(config))
```

第一种中，actor system 获得的配置是

```
akka.loglevel = WARNING
my.own.setting = 43
my.other.setting = "hello"
```

而在第二种中，只有 “akka” 子树被提升了，结果如下:

```
akka.loglevel = ERROR
my.own.setting = 42
my.other.setting = "hello"
```

可以通过代码来指定配置信息：

```
val customConf = ConfigFactory.parseString("""
  akka.actor.deployment {
    /my-service {
      router = round-robin
      nr-of-instances = 3
    }
  }
  """)
val system = ActorSystem("MySystem", ConfigFactory.load(customConf))
```

工具类

一、Duration

Duration 是表示时间长短的基础类，其可以是有限的或者无限的。

- 有限的 Duration 用 FiniteDuration 类来表示，并通过时间长度(length)和 java.util.concurrent.TimeUnit 来构造。
- 无限的 durations 同样扩展了 Duration，只在两种情况下存在，Duration.Inf 和 Duration.MinusInf。

Duration 能够用如下方法实例化：

- 隐式的通过 Int 和 Long 类型转换得来，例如 val d = 100 millis。
- 通过传递一个 Long length 和 java.util.concurrent.TimeUnit，例如 val d = Duration(100, MILLISECONDS)
- 通过传递一个字符串来表示时间区间，例如 val d = Duration("1.2 µs")

```
// instantiation
val d1 = Duration(100, MILLISECONDS)    // from Long and TimeUnit
val d2 = Duration(100, "millis")                // from Long and String
val d3 = 100 millis                                  // implicitly from Long, Int or Double
val d4 = Duration("1.2 µs")                   // from String
```

抽象的 Duration 类包含了如下方法：

- 到不同时间单位的转换(toNanos, toMicros, toMillis, toSeconds, toMinutes, toHours, toDays and toUnit(unit: TimeUnit))
- Duration 的比较(<，<=，>和>=)
- 算术运算符（+, -, *, / 和单值运算_-）
- Duration 的最大最小方法(min，max)
- 测试 Duration 是否是无限的方法(isFinite)

eg：

```
import scala.concurrent.duration._
val fivesec = 5 seconds
val threemillis = 3 millis
val diff = fivesec - threemillis
assert (diff < fivesec)
val fourmillis = threemillis * 4 / 3      // though you cannot write it the other way around
val n = threemillis / (1 millisecond)
```

Duration 也提供了 unapply 方法，因此可以被用于模式匹配中，例如：

```
// pattern matching
val Duration(length, unit) = 5 millis
```

二、日志

创建一个 Logging：

```
val log = Logging(context.system, this)
```

然后就可以使用它的 error, warning, info, debug 方法记录日志。

为了方便你可以向 actor 中混入 log 成员，而不是象上例一下定义它.

```
class MyActor extends Actor with ActorLogging {
  log.info(“”)    //直接可以使用 log
}
```

Logging 的第二个参数是这个日志通道的源. 这个源对象以下面的规则转换成字符串:

1) 如果它是 Actor 或 ActorRef，则使用它的路径
2) 如果是 String，就使用它自己
3) 如果是类，则使用它的 simpleName
4) 其它的类型，而且当前作用域中又没有隐式的 LogSource[T] 实例则会导致编译错误.

日志消息可以包含参数占位符 {}，你可以传入一个 Java 数组来为多个占位符提供数据：

```
val args = Array("The", "brown", "fox", "jumps", 42)
system.log.debug("five parameters: {}, {}, {}, {}, {}", args)
```

辅助的日志选项：

Akka 为非常底层的 debug 提供了一组配置选项
一定要将日志级别设为 DEBUG 来使用这些选项:

```
akka {
  loglevel = DEBUG
}
```

下面这个配置选项在你想知道 Akka 装载了哪些配置设置时非常有用:

```
akka {
  # 在 actor 系统启动时以 INFO 级别记录完整的配置
  # 当你不确定使用的是哪个配置时有用
  log-config-on-start = on
}
```

记录所有被使用 akka.event.LoggingReceive 的 Actor 处理的用户级消息的细节:

```
akka {
  debug {
    # 打开 LoggingReceive 功能，以 DEBUG 级别记录所有接收到的消息
    receive = on
  }
}
```

记录 Actor 所处理的所有自动接收的消息的细节：

```
akka {
  debug {
    # 为所有的 AutoReceiveMessages(Kill, PoisonPill 之类) 打开 DEBUG 日志
```

```
    autoreceive = on
  }
}
```

记录 Actor 的所有生命周期变化 (重启，死亡等) 的细节:

```
akka {
  debug {
    # 打开 actor 生命周期变化的 DEBUG 日志
    lifecycle = on
  }
}
```

记录所有继承了 LoggingFSM 的 FSM actor 的事件、状态转换和计时器的细节:

```
akka {
  debug {
    # 打开所有 LoggingFSMs 事件、状态转换和计时器的 DEBUG 日志
    fsm = on
  }
}
```

监控对 ActorSystem.eventStream 的订阅/取消订阅:

```
akka {
  debug {
    # 打开 eventStream 上订阅关系变化的 DEBUG 日志
    event-stream = on
  }
}
```

辅助的远程日志选项：

以 DEBUG 级别查看所有远程发送的消息: (这些日志是被传输层发送时所记录，而非 actor)

```
akka {
  remote {
    # 如果打开这个选项，Akka 将以 DEBUG 级别记录所有发出的消息， 不打开则不记录
    log-sent-messages = on
  }
}
```

以 DEBUG 级别查看接收到的所有远程消息: (这些日志是被传输层接收时所记录，而非 actor)

```
akka {
  remote {
    # 如果打开这个选项，Akka 将以 DEBUG 级别记录所有接收到的消息， 不打开则不记录
    log-received-messages = on
  }
}
```

三、定时器

ActorSystem 的 **scheduler 方法**返回一个 akka.actor.Scheduler 实例，这个实例在每个 Actor 系统里是唯一的，用来指定一段时间后发生的行为。

定时任务是使用 ActorSystem 的 MessageDispatcher 执行的。

1、Scheduler 接口

```
trait Scheduler {
  /**
   * 计划在一段初始的时间后及之后的某固定时间段重复发送消息。
   * 例如：如果你希望消息马上被发送，并每隔 500ms 后各发送一次，你应该设置
   * delay=Duration.Zero 和 frequency=Duration(500, TimeUnit.MILLISECONDS)
   */
  def schedule(initialDelay: Duration, frequency: Duration, receiver: ActorRef, message: Any): Cancellable

  /**
   * 计划一个函数在一段初始的时间后及之后的某固定时间段重复执行。
   */
  def schedule(initialDelay: Duration, frequency: Duration)(f: ⇒ Unit): Cancellable

  /**
   * 计划一个 Runnable 在一段初始的时间后及之后的某固定时间段重复执行。
   */
  def schedule(initialDelay: Duration, frequency: Duration, runnable: Runnable): Cancellable

  /**
   * 计划一个消息在一段时间后运行一次。
   */
  def scheduleOnce(delay: Duration, receiver: ActorRef, message: Any): Cancellable

  /**
   * 计划一个函数在一段时间后运行一次。
   */
  def scheduleOnce(delay: Duration)(f: ⇒ Unit): Cancellable

  /**
   * 计划一个 Runnable 在一段时间后运行一次, i.e. 在它执行之前必须经过一段时间。
   */
  def scheduleOnce(delay: Duration, runnable: Runnable): Cancellable
}
```

2、Cancellable 接口

它使你可以取消计划执行的任务。
注意：它不会中止已经启动的任务的执行。

```
trait Cancellable {
   def cancel(): Unit              // 取消当前 Cancellable
   def isCancelled: Boolean     // 返回当前 Cancellable 是否已经取消了
}
```

3、使用示例

```
//计划在 50ms 后将"foo"消息发送给 testActor
system.scheuler.scheduleOnce(50 milliseconds, testActor, "foo")

//计划 50ms 后执行函数 (发送当前时间给 testActor）
system.scheduler.scheduleOnce(50 milliseconds) {
   testActor ! System.currentTimeMillis
}

//这将会计划 0ms 后每 50ms 向 tickActor 发送 Tick 消息
val cancellable = system.scheduler.schedule(0 milliseconds, 50 milliseconds, tickActor, Tick)
//这会取消未来的 Tick 发送
cancellable.cancel()
```

Actor 路径

1、什么是 Actor Path？

actor 是以一种严格的树形结构样式来创建的，沿着子 actor 到父 actor 的监管链一直到 actor 系统的根存在一条唯一的 actor 名字序列。

一个 **actor 路径**包含一个标识该 actor 系统的锚点，之后是各路径元素连接起来，从根到指定的 actor；
路径元素是路径经过的 actor 的名字，以"/"分隔。

actor 路径的锚点：
每一个 actor 路径都有一个 address component，它描述了协议和地址。紧接着后面是从“根”到“指定 actor”路径上各 actor 名字以”/”拼接起来的路径。 **【重要】**
例如：

```
"akka://my-sys/user/service-a/worker1"                // purely local
"akka.tcp://my-sys@host.example.com:5678/user/service-b"  // remote
```

路径中的 address component 用来做什么？
在跨网络传送 actor 引用时，是用它的路径来表示这个 actor 的。
所以，它的路径必须包括能够用来向它所代表的 actor 发送消息的完整的信息。这一点是通过在路径字符串的地址部分包括协议、主机名和端口来做到的。
当一个 actor system 从远程节点接收到一个 actor 路径，会检查它的地址部分是否与自己的地址相同，如果相同，那么会将这条路径解析为本地 actor 引用，否则解析为一个远程 actor 引用。

2、逻辑 Actor 路径、物理 Actor 路径 *

1. 逻辑 Actor 路径

沿着 actor 的“父监管”链一直到“根”的“唯一路径”被称为**逻辑 actor 路径**。 **【重要】**
这个路径与 actor 的创建祖先完全吻合，所以当 actor 系统的远程调用配置（和配置中路径的地址部分）设置好后它就是完全确定的了。

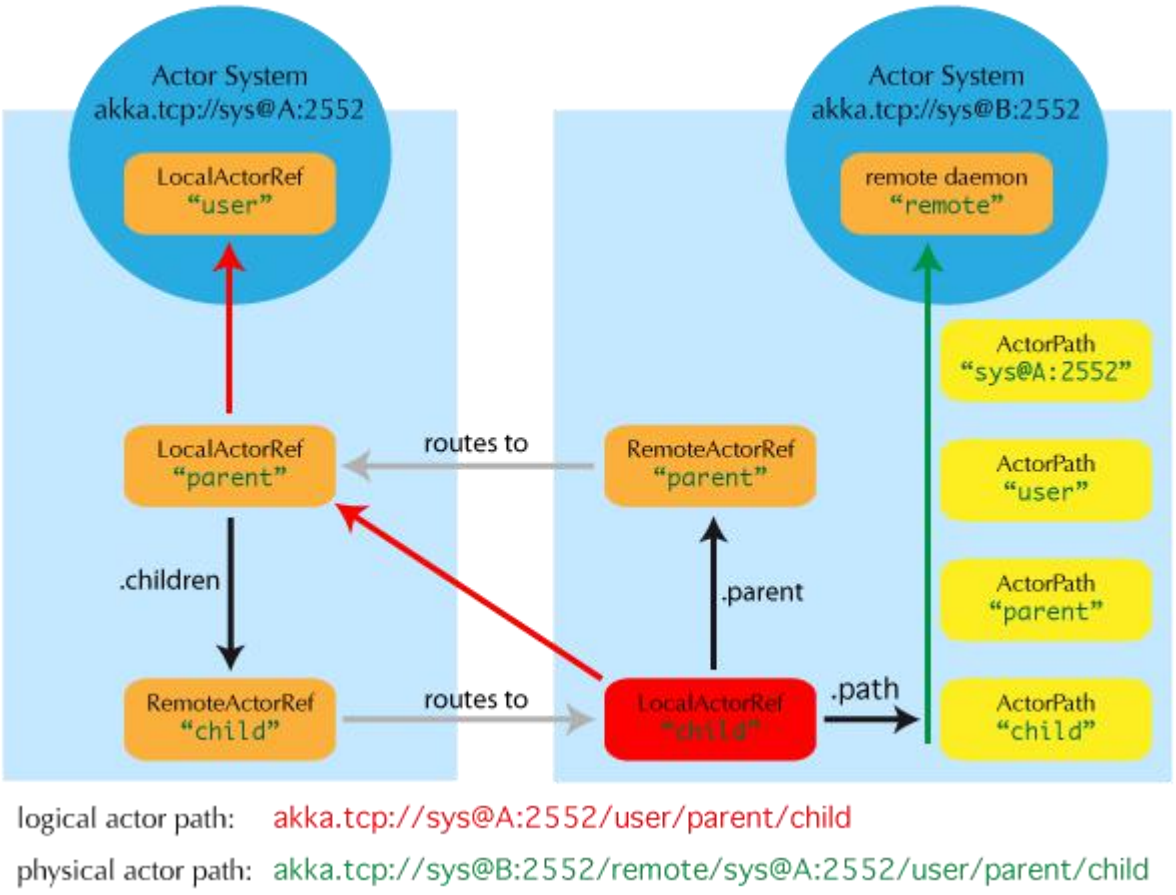
2. 物理 Actor 路径

基于配置的远程部署意味着一个 actor 可能在另外一台网络主机上被创建。i.e.在另一个 actor 系统中。
在这种情况下，从根穿过 actor 路径肯定要访问网络，这种开销比较大。因此，每一个 actor 同时还有一条**物理路径**，从“实际的 actor 对象所在的 actor system 的根”开始的。 **【重要】**
跟其它 actor 通信时使用物理路径作为发送方引用能够让接收方直接回复到这个 actor 上，将路由延迟降到最小。

3、远程 Actor 是怎么创建的？

当创建一个 actor 时，actor 系统的 deployer 会判断是在当前 JVM 中创建 actor 还是在另一个网络上的 JVM 上创建。
如果是后者的话，将会通过网络连接在另一个不同的 JVM 上触发这个新 actor 的创建，结果就是这个新 actor 在另一个不同的 actor system 中。

1. 远程 actor system 会把这个新 actor 放置在一个特殊的路径下；
   下面图里右侧部分黄颜色描述的路径就是这个新 actor 的路径。
2. 并且，这个新 actor 的监管者是一个远程 actor reference。
   在新 actor 中通过 **context.parent** (the supervisor reference) 和 **context.path.parent** (the parent node in the actor`s path)得到的 actor reference 并不指向相同的 actor。
3. 下面列出的 logical actor path 和 physical actor path 映证了上面提到的关于这两个路径的概念。



4、Akka 使用的特殊路径

在路径树的根上是根监管者，所有的的 actor 都可以从通过它找到。
在第二个层次的是以下这些：
"**/user**" 是由用户创建的顶级 actor 的监管者，用 ActorSystem.actorOf 创建的 actor 在其下一个层次。
"**/system**" 是由系统创建的顶级 actor（如日志监听器或由配置指定在 actor 系统启动时自动部署的 actor）的监管者
"**/deadLetters**" 是死信 actor，所有发往已经终止或不存在的 actor 的消息会被送到这里
"**/temp**" 是系统创建的短时 actor(i.e.那些用在 ActorRef.ask 的实现中的 actor)的监管者.
"**/remote**" 是一个人造的路径，用来存放所有其监管者是远程 actor 引用的 actor

ActorRef 介绍

1、Actor 引用是什么？

Actor 引用是 ActorRef 的子类，它的最重要功能是支持向它所代表的 actor 发送消息。
每个 actor 通过 self 来访问它的标准（本地）引用，在发送给其它 actor 的消息中也缺省包含这个引用。
反过来，在消息处理过程中，actor 可以通过 sender 来访问到当前消息的发送者的引用。

根据 actor 系统的配置，支持几种不同的 actor 引用：

1. **纯本地 actor 引用**
   使用在配置为不使用网络功能的 actor 系统中。这些 actor 引用不能在保持其功能的条件下从网络连接上向外传输。
2. **支持远程调用的本地 actor 引用**
   使用在支持同一个 jvm 中 actor 引用之间的网络功能的 actor 系统中。
   为了在发送到其它网络节点后被识别，这些引用包含了协议和远程地址信息。
3. **本地 actor 引用有一个子类是用在路由**（routers， i.e. mixin 了 Router trait 的 actor）.
   它的逻辑结构与之前的本地引用是一样的，但是向它们发送的消息会被直接重定向到它的子 actor。
4. **远程 actor 引用**
   代表可以通过远程通讯访问的 actor。i.e 从别的 jvm 向他们发送消息时，Akka 会透明地对消息进行序列化。
5. 有几种特殊的 actor 引用类型，在实际用途中比较类似本地 actor 引用：
   1.**PromiseActorRef** 表示一个 Promise，作用是从一个 actor 返回的响应来完成，它是由 ActorRef.ask 调用来创建的
   2.**DeadLetterActorRef** 是死信服务的缺省实现，所有接收方被关闭或不存在的消息都在此被重新路由。
   3.**EmptyLocalActorRef** 是查找一个不存在的本地 actor 路径时返回的：它相当于 DeadLetterActorRef，但是它保有其路径因此可以在网络上发送，以及与其它相同路径的存活的 actor 引用进行比较，其中一些存活的 actor 引用可能在该 actor 消失之前得到了。
6. 然后有一些内部实现，你可能永远不会用上：
   1.有一个 actor 引用并不表示任何 actor，只是作为根 actor 的伪监管者存在，我们称它为“时空气泡穿梭者”。
   2.在 actor 创建设施启动之前运行的第一个日志服务是一个伪 actor 引用，它接收日志事件并直接显示到标准输出上；它就是 Logging.StandardOutLogger。

2、如何获得 Actor 引用

关于 actor 引用的获取方法分为两类：

创建 actor

通过对 actor 的拜访查找。这种又分两种：

通过具体的 actor 路径来查找 actor 引用

查询逻辑 actor 树

1. 创建 Actor

一个 actor 系统通常是在根 actor 上使用 ActorSystem.actorOf 创建 actor，然后使用 ActorContext.actorOf 从创建出的 actor 中生出 actor 树来启动的。

这些方法返回指向新创建的 actor 的引用。

每个 actor 都拥有到它的父亲，它自己和它的子 actor 的引用。这些引用可以与消息一直发送给别的 actor，以便接收方直接回复。

2. 查询逻辑 Actor 树

由于 actor 系统是一个类似文件系统的树形结构：可以将路径中的一部分用通配符(*和?)替换来组成对 0 个或多个实际 actor 的匹配。由于匹配的结果不是一个单一的 actor 引用，它拥有一个不同的类型 ActorSelection，这个类型不完全支持 ActorRef 的所有操作。

ActorSelection 可以用 ActorSystem.actorSelection 或 ActorContext.actorSelection 两种方式来获得，并且支持发送消息：

```
context.actorSelection("../*") ! msg        //将 msg 发送给包括当前 actor 在内的所有兄弟
context.actorSelection("/user/serviceA") ! msg      //以绝对路径来寻找 actor
```

定义一个 Actor 类

要定义自己的 Actor 类，需要继承 Actor 并实现 receive 方法.

receive 方法需要定义一系列 case 语句(类型为 PartialFunction[Any, Unit]) 来描述你的 Actor 能够处理哪些消息，以及实现对消息如何进行处理的代码。

如下例：

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

创建 Actor

2.1 使用缺省构造方法创建 Actor

```
object Main extends App {
  val system = ActorSystem("MySystem")
  val myActor = system.actorOf(Props[MyActor], name = "myactor")
```

在上面的例子中，actor 是从 system 创建的。也可以从其它的 actor 使用 actor 上下文（context）来创建:

```
class FirstActor extends Actor {
  val myActor = context.actorOf(Props[MyActor], name = "myactor")
```

注意：

1) name 参数是可选的, 但建议你为你的 actor 起一个合适的名字，因为它将在日志信息中被用于标识各个 actor. 名字不可以为空或以 $开头。

如果给定的名字已经被赋给了同一个父 actor 的其它子 actor，将会抛出 InvalidActorNameException 。

2) Actor 在创建后将自动异步地启动。

2.2 使用非缺省构造方法创建 Actor

如果你的 Actor 的构造方法带参数，那么你不能使用 actorOf(Props[TYPE]) 来创建它。这时你可以用 actorOf 的带有传名调用的变体，这样你可以用任意方式来创建 actor。

如下例:

```
// 允许传参数给 MyActor 构造方法
val myActor = system.actorOf(Props(new MyActor("...")), name = "myactor")
```

2.3. 使用匿名类创建 Actor

在从某个 actor 中派生新的 actor 来完成特定的子任务时，可能使用匿名类来包含将要执行的代码会更方便。

```
def receive = {
  case m: DoIt ⇒ context.actorOf(Props(new Actor {
      def receive = {
        case DoIt(msg) ⇒ val replyMsg = doSomeDangerousWork(msg)
          sender ! replyMsg
          context.stop(self)
      }
```

```
        def doSomeDangerousWork(msg: ImmutableMessage): String = { "done" }
    })) forward m
}
```

Props

Props is a configuration class to specify options for the creation of actors。

it is **immutable**, so it is **thread-safe** and **fully shareable**.

Here are some examples of how to create a Props instance.

```
val props1 = Props[MyActor]
val props2 = Props(new ActorWithArgs("arg"))          // careful, see below
val props3 = Props(classOf[ActorWithArgs], "arg")    // no support for value class arguments
```

第二种方式展示如何创建带类参数的 Actor。但是这种方式应该只用在 Actor 的外部。

第三种方式提供了一种在不关心上下文的情况下传递构造参数的可能性。

使用这种方式有两点需要注意：

1. 当 Actor 的构造函数接收是“值类型”的值时，就不能使用这种方式。

除非是采取下面的第二种和第三种方式来创建 Props 实例。

```
class Argument(val value: String) extends AnyVal
class ValueClassActor(arg: Argument) extends Actor {
  def receive = { case _ => () }
}
object ValueClassActor {
  def props1(arg: Argument) = Props(classOf[ValueClassActor], arg)          // fails at runtime
  def props2(arg: Argument) = Props(classOf[ValueClassActor], arg.value)    // ok
  def props3(arg: Argument) = Props(new ValueClassActor(arg))               // ok
}
```

2. 在构造 Props 对象的时候，会检查是否存在匹配的构造函数，如果没有或存在多个匹配的构造函数，则会导致 IllegalArgumentEception。

下面两个例子在创建 Props 实例时，都会抛出 IllegalArgumentEception 异常，说明没有发现匹配的构造函数。

```
class DefaultValueActor(a: Int, b: Int = 5) extends Actor {
  def receive = {
    case x: Int => sender() ! ((a + x) * b)
  }
}
val defaultValueProp1 = Props(classOf[DefaultValueActor], 2.0) // Unsupported

class DefaultValueActor2(b: Int = 5) extends Actor {
  def receive = {
    case x: Int => sender() ! (x * b)
  }
}
val defaultValueProp2 = Props[DefaultValueActor2]                 // Unsupported
val defaultValueProp3 = Props(classOf[DefaultValueActor2])      // Unsupported
```

危险的方式：

val props7 = Props(new MyActor(args))

这种方式不建议在另一个 Actor 种使用，因为它鼓励封闭作用域，会导致不可序列化的 Props 和可能的竞态条件(破坏了 Actor 的封闭性)。

警告：

将一个 Actor 声明在另一个 Actor 种是非常危险的。从来不要将 Actor 的 this 引用传递到 Props 中。

推荐的做法：

在 **Actor 的伴生对象**中提供一个**工厂方法**来**创建 Props 对象**，这是一种比较好的方式。因为在一个伴生对象中，给定的代码块不会保留指向其作用域的引用。

例如：

```
object DemoActor {
  def props(magicNumber: Int): Props = Props(new DemoActor(magicNumber))
}
class DemoActor(magicNumber: Int) extends Actor {
  def receive = {
    case x: Int => sender() ! (x + magicNumber)
  }
}
class SomeOtherActor extends Actor {
  // Props(new DemoActor(42)) would not be safe
  context.actorOf(DemoActor.props(42), "demo")
}
```

另一个好的实践是：在 Actor 的伴生对象中声明好此 Actor 接收哪些消息。这样就很容易知道这个 Actor 将接收什么消息。

例如：

```
object MyActor {
  case class Greeting(from: String)
  case object Goodbye
}
class MyActor extends Actor with ActorLogging {
  import MyActor._
  def receive = {
    case Greeting(greeter) => log.info(s"I was greeted by $greeter.")
    case Goodbye           => log.info("Someone said goodbye to me.")
  }
}
```

Actor API

Actor 特质只定义了一个抽象方法，就是上面提到的 receive，用来实现 actor 的行为。

如果当前 actor 的行为与收到的消息不匹配，则会调用 unhandled，它的缺省实现是向 actor 系统的事件流中发布一条 akka.actor.UnhandledMessage(message, sender, recipient)。

另外，它还包括:

self 代表本 actor 的 ActorRef

sender 代表最近收到的消息的发送 actor，通常用于返回回应消息

supervisorStrategy 用户可重写它来定义对子 actor 的监管策略

context 暴露 actor 和当前消息的上下文信息，如：

- 用于创建子 actor 的工厂方法 (actorOf)
- 父监管者
- 所监管的子 actor
- actor 所属的系统
- 生命周期监控
- hotswap 行为栈，见 Become/Unbecome

其余的可见方法是可以被用户重写的生命周期 hook，描述如下:

```
def preStart() {}
def preRestart(reason: Throwable, message: Option[Any]) {
  context.children foreach (context.stop(_))
  postStop()
}
def postRestart(reason: Throwable) { preStart() }
def postStop() {}
```

Actor 的生命周期 hook

6.1 启动 Hook

actor 启动后，它的 preStart 会被立即执行。

```
override def preStart() {
  someService ! Register(self)
}
```

6.2 重启 Hook

重启的原因：

当 actor 在处理消息时出现失败，失败的原因分成以下三类：

1.对收到的特定消息的系统错误（i.e.程序错误）

2.处理消息时一些外部资源的（临时性）失败

3.actor 内部状态崩溃了

重启过程：

1) 被重启的 actor 的 **preRestart** 被调用，携带着导致重启的异常以及触发异常的消息;
   如果重启并不是因为消息的处理而发生的，所携带的消息为 None，例如，当一个监管者没有处理某个异常继而被它自己的监管者重启时。
   这个方法是用来完成清理、准备移交给新的 actor 实例的最佳位置。
   preRestart 方法的缺省实现是终止所有的子 actor，并调用 postStop.
2) 最初 **actorOf** 调用的工厂方法将被用来创建新的实例。
3) 新的 actor 的 **postRestart** 方法被调用，携带着导致重启的异常信息。

注意:

- 触发异常的消息不会被重新接收;
- actor 的重启会替换掉原来的 actor 对象;
- 在 actor 重启过程中所有发送到该 actor 的消息将象平常一样被放进邮箱队列中。
- 重启不影响邮箱的内容，所以对消息的处理将在 postRestart hook 返回后继续;

6.3 终止 Hook

一个 Actor 终止后，它的 postStop hook 将被调用，这可以用来取消该 actor 在其它服务中的注册.

这个 hook 保证在该 actor 的消息队列被禁止后才运行，之后发给该 actor 的消息将被重定向到 ActorSystem 的 deadLetters 中。

(如果先调用 postStop，再禁止消息队列，那么在 postStop 方法执行期间，消息队列仍然在接受消息，但是 actor 已经不会处理消息，造成这些消息没有被处理。

所以优雅的处理是，在执行 postStop 方法之前，就禁止消息队列，不再接受消息，把发给该 actor 的消息将被重定向到 ActorSystem 的 deadLetters 中。

)

发送消息

向 actor 发送消息是使用下列方法之一。

! 意思是“fire-and-forget”，也称为 tell. 异步发送一个消息并立即返回。

? 意思是"Send-And-Receive-Future"，也称为 ask. 异步发送一条消息并返回一个 Future 代表一个可能的回应。

7.1 tell

这是发送消息的推荐方式。不会阻塞地等待消息。它拥有最好的并发性和可扩展性。

```
actor ! "hello"
```

- 如果是在一个 Actor 中调用，那么发送方的 actor 引用会被隐式地作为消息的 sender。目标 actor 可以使用它来向原 actor 发送回应，使用 sender ! replyMsg.
- 如果不是从 Actor 实例发送的，sender 成员缺省为 DeadLetters actor 引用。

7.2 ask

注意：

想使用 ask 方法，需要 **import akka.pattern.ask**

使用 ask 将会像 tell 一样发送消息给接收方，接收方必须通过 sender ! reply 发送回应来为返回的 Future 填充数据。

ask 操作创建一个内部 actor (实际是 PromiseActorRef) 来处理回应，必须为这个内部 actor 指定一个超时期限，过了超时期限内部 actor 将被销毁(以 AskTimeoutException 来结束)以防止内存泄露。

超时的时限是按下面的顺序和位置来获取的：

1) 显式指定超时：

```
val future = myActor.ask("hello")(5 seconds)
```

因为 Int 类型的值并没有 seconds 方法，因此 5 seconds 涉及到隐式转换。想使用工具 Duration，需要：import scala.concurrent.duration._

2) 提供类型为 akka.util.Timeout 的隐式参数，例如，

```
implicit val timeout = Timeout(5 seconds)
val future = myActor ? "hello"
```

如果要以异常来填充 future，你需要发送一个 Failure 消息给发送方。这个操作不会在 actor 处理消息发生异常时自动完成。

```
try {
  val result = operation()
  sender ! result
} catch {
  case e: Exception =>
    sender ! akka.actor.Status.Failure(e)
    throw e
}
```

转发消息

你可以将消息从一个 actor 转发给另一个。虽然经过了一个‘中转’，但最初的发送者地址/引用将保持不变。当实现功能类似路由器、负载均衡器、备份等的 actor 时会很有用。

myActor.forward(message)

接收消息

Actor 必须实现 receive 方法来接收消息：

protected def receive: PartialFunction[Any, Unit]

设置消息接收超时：

在接收消息时，如果在一段时间内没有收到第一条消息，可以使用超时机制。

要检测这种超时你必须设置 receiveTimeout 属性并声明一个处理 ReceiveTimeout 对象的匹配分支。

超时时间最小支持 1 毫秒。

```
class MyActor extends Actor {
  context.setReceiveTimeout(30 milliseconds)
  def receive = {
    case "Hello"           ⇒ //...
    case ReceiveTimeout ⇒ throw new RuntimeException("received timeout")
  }
}
```

如果想关掉消息的超时设置，可以这样：context.setReceiveTimeout(Duration.Undefined)

回应消息

使用 sender ! replyMsg 向发送方发送回应消息。

你也可以将 sender 保存起来将来再作回应。如果没有 sender(不是从 actor 发送的消息或者没有 future 上下文)那么 sender 缺省为 DeadLetterActorRef .

使用 DeathWatch 监控终止的 Actor

为了在其它 actor 永久终止时收到通知，**actor 可以将自己注册为其它 actor 在终止时所发布的 Terminated 消息的接收者**（见终止 Actor).

这个服务是由 actor 系统的 DeathWatch 组件提供的。

注册一个监控器很简单：

```
class WatchActor extends Actor {
  val child = context.actorOf(Props.empty, "child")
  context.watch(child)        // 这是注册所需要的唯一调用
  var lastSender = system.deadLetters

  def receive = {
    case "kill" => context.stop(child); lastSender = sender
    case Terminated(`child`) => lastSender ! "finished"   //接受所监控的 actor 结束时发布的 Terminated 消息，并进行处理
  }
}
```

注意：

要注意 Terminated 消息的产生与注册和终止行为所发生的顺序无关。

尤其是，the watching actor will receive a Terminated message even if the watched actor has already been terminated at the time of registration。

多次注册并不表示会有多个 Terminated 消息产生，也不保证有且只有一个这样的消息被接收到，原因是：

1. 如果在 Actor 终止之前调用了多次 watch()方法，只会在 Actor 终止时收到一个 Terminated 消息；
2. 如果被监控的 actor 已经终止并生成了 Terminated 消息进入了 watching actor 的邮箱队列，在这个消息被处理之前又发生了另一次注册，则会有第二个消息进入队列。
   因为对一个已经终止的 actor 注册监控器，会立刻导致 Terminated 消息的发生。

使用 context.unwatch(target)来停止对另一个 actor 的生存状态的监控。

注意：这不能保证不会接收到 Terminated 消息，因为该消息可能已经进入了队列。

终止 Actor

1. stop 方法

通过调用 ActorContext 或 ActorSystem 的 **stop 方法**来终止一个 actor，通常 context 用来终止子 actor，而 system 用来终止顶级 actor。
在 ActorSystem.**terminate** 被调用时，系统根监管 actor 会被终止。
实际的终止操作是异步执行的。

如果当前有正在处理的消息，对该消息的处理将在 actor 被终止之前完成，

- Actor 在终止之前会停止对邮箱的处理，之后发给该 actor 的消息将被重定向到 ActorSystem 的 **DeadLetter** 中。
- Actor 终止之后，邮箱中的后续消息将不会被处理，缺省情况下这些消息会被送到 ActorSystem 的 **DeadLetter**，但是这取决于邮箱的实现。

actor 的终止过程分两步：
第一步：向所有子 actor 发送终止命令，然后处理来自子 actor 的终止消息直到所有的子 actor 都完成终止；
第二步：最后终止自己(调用 postStop，销毁邮箱，向 DeathWatch 发布 Terminated 消息通知其监管者)。

注意：
由于 actor 的终止是异步的，你不能马上使用你刚刚终止的子 actor 的名字，这会导致 InvalidActorNameException。
你应该监视正在终止的 actor 而在最终到达的 Terminated 消息的处理中创建它的替代者。

2. PoisonPill

也可以向 actor 发送 PoisonPill 消息，这个消息处理完成后 actor 会被终止。
PoisonPill 与普通消息一样被放进队列，因此会在已经入队列的其它消息之后被执行。

3. 优雅地终止

如果你想等待终止过程的结束，或者组合若干 actor 的终止次序，可以使用 gracefulStop：

```
try {
  val stopped: Future[Boolean] = gracefulStop(actorRef, 5 seconds)(system)
  Await.result(stopped, 6 seconds)
  // actor 被成功终止
} catch {
  case e: ActorTimeoutException =>  // actor 没有在 5 秒内终止
}
```

杀死 Actor

你可以发送 Kill 消息来杀死 actor，这会造成 actor 抛出一个 ActorKilledException 异常。
actor 怎么被处理，由它的监管者来决定，可能恢复这个 actor，也可能重启或者终止这个 actor。

使用示例：

```
victim ! Kill    // 杀死名为 'victim' 的 actor
```

Actor 与异常

在消息被 actor 处理的过程中可能会抛出异常，例如数据库异常。

消息会怎样？

如果消息处理过程中发生了异常，这个消息将被丢失。它不会被放回到邮箱中。

如果你希望重试对消息的处理，需要自己抓住异常然后在异常处理流程中重试. 请确保你限制重试的次数，因为你不会希望系统产生活锁 (从而消耗大量 CPU 而于事无补)。

邮箱会怎样？

如果消息处理过程中发生异常，邮箱没有任何变化。如果 actor 被重启，邮箱会被保留。邮箱中的所有消息不会丢失。

actor 会怎样？

如果在一个 actor 中，代码抛出了一个异常，这个 actor 会被挂起，然后由它监管者的监管策略来决定如何处理这个 actor(恢复、重启或者终止它)。

Become/Unbecome

1) 升级

Akka 支持在运行时对 Actor 的行为进行实时替换：在 actor 中调用 context.become(partialFunction)方法。

Become 要求一个 PartialFunction[Any,Unit]参数作为新的消息处理实现。被替换的代码被存在一个栈中，可以被 push 和 pop。

使用 become 替换 Actor 的行为：

```
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)
  }
}
```

2) 降级

由于被热替换掉的代码存在栈中，你也可以对代码进行降级，只需要在 actor 中调用 context.unbecome()方法。

这将会用从栈顶取出的 PartialFunction[Any,Unit]作为 actor 的行为。

使用 unbecome 方法的例子：

```
def receive = {
  case "revert" => context.unbecome()
}
```

注意：如果 actor 被重启的话，那么 actor 将被重置为原来的行为。

使用 PartialFunction 链来扩展 actor

定义基础的消息处理器并通过继承或委托来对它进行扩展，使用 PartialFunction.orElse 链：

```
abstract class GenericActor extends Actor {
  // to be defined in subclassing actor
  def specificMessageHandler: Receive

  // generic message handler
  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)
  }
}
```

TypedActor

除了使用普通的 Actor 外，还可以使用 TypedActor。

TypedActor 由两部分组成：**一个公开的接口**和**一个实现**。

TypedActor 使用 **JDK Proxy** 来拦截方法的调用，然后**交给底层的 Actor** 来处理。

TypedActor 相对于普通 Actor 的优势在于 TypedActor 拥有静态的契约，你不需要定义你自己的消息，
它的劣势在于对你能做什么和不能做什么进行了一些限制，i.e.你不能使用 become/unbecome.

TypedActor 的工具方法 【重要】

```
//返回 TypedActor 扩展
val extension = TypedActor(system)          //system 是一个 ActorSystem 实例

//判断一个引用是否是 TypedActor 代理
TypedActor(system).isTypedActor(someReference)

//返回一个外部 TypedActor 代理所代表的 Akka actor
TypedActor(system).getActorRefFor(someReference)

//返回当前 TypedActor 的外部代理,
val s: Squarer = TypedActor.self[Squarer]     //此方法仅在一个 TypedActor 实现的方法中有效

//返回当前的 ActorContext,
val c: ActorContext = TypedActor.context      //此方法仅在一个 TypedActor 实现的方法中有效

//返回一个 TypedActor 扩展的上下文实例
//这意味着如果你用它创建其它的有类型 actor，它们会成为当前有类型 actor 的子 actor
TypedActor(TypedActor.context)
```

创建 TypedActor

要创建 TypedActor，需要一个或多个接口，和一个实现。

定义一个接口：

```
trait Squarer {
   def squareDontCare(i: Int): Unit              //fire-forget

   def square(i: Int): Future[Int]               //非阻塞 send-request-reply

   def squareNowPlease(i: Int): Option[Int]   //阻塞 send-request-reply

   def squareNow(i: Int): Int                    //阻塞 send-request-reply
}
```

接口的实现：

```
class SquarerImpl(val name: String) extends Squarer {
  def this() = this("default")

  import TypedActor.dispatcher     //这样我们才有一个隐式的 Promise 派发器

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

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

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

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

创建我们的 Squarer TypedActor 实例的最简单方法 ：

```
val mySquarer: Squarer = TypedActor(system).typedActorOf(TypedProps[SquarerImpl]())
```

如果要调用某特定的构造方法要这样做：：

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

方法派发语义 【重要】

方法返回:

1) Unit 返回类型 会以 fire-and-forget 语义进行派发，与 ActorRef.**tell** 完全一致。

2) akka.dispatch.Future[_]返回类型 会以 send-request-reply 语义进行派发，与 ActorRef.**ask** 完全一致。

3) scala.Option[_]返回类型 或者 akka.japi.Option<?>返回类型

会以 send-request-reply 语义派发，但是会阻塞等待应答，

如果在超时时限内没有应答则返回 None，否则返回包含结果的 scala.Some/akka.japi.Some .

在这个调用中发生的异常将被重新抛出。

4) 任何其它类型的返回值

以 send-request-reply 语义进行派发，但会阻塞地等待应答，

如果超时会抛出 java.util.concurrent.TimeoutException，

如果发生异常则将异常重新抛出。

接收任意消息

如果你的 TypedActor 的实现类扩展了 akka.actor.TypedActor.Receiver，**所有非方法调用的消息**会被传递到 onReceive 方法.
这使你能够对 DeathWatch 的 Terminated 消息或其它类型的消息进行处理

终止有类型 Actor

由于 TypedActor 底层还是 Akka actor，所以在不需要的时候要终止它。

```
TypedActor(system).stop(mySquarer)          // 这将会尽快地异步终止与指定的代理关联的有类型 Actor。
TypedActor(system).poisonPill(otherSquarer)  // 这将会在有类型 actor 完成所有在当前调用之前对它的调用后异步地终止它。
```

生命周期回调

通过使你的有类型 actor 实现类实现以下方法:

TypedActor.PreStart

TypedActor.PostStop

TypedActor.PreRestart

TypedActor.PostRestart

你可以 hook 进你的有类型 actor 的整个生命周期。

监管策略

通过让你的有类型 Actor 实现类实现 TypedActor.**Supervisor** 方法，你可以定义用来监管子 actor 的策略

监管与容错

监管

监管描述的是 actor 之间的关系：

监管者将任务委托给下属并对下属的失败状况进行响应。当一个下属出现了失败（i.e. 抛出一个异常），它自己会将自己和自己所有的下属挂起然后向自己的监管者发送一个提示失败的消息。

取决于所监管的工作的性质和失败的性质，监管者可以有 4 种基本选择：

1.让下属继续执行，保持下属当前的内部状态

2.重启下属，清除下属的内部状态

3.永久地终止下属

4.将失败沿监管树向上传递

因为一个监管者同时也是其上方监管者的下属, 并且隐含在前 3 种选择中：让 actor 继续执行同时也会继续执行它的下属，重启一个 actor 也必须重启它的下属，相似地终止一个 actor 会终止它所有的下属。

Akka 实现的是一种叫“父监管”的形式。Actor 只能由其它的 actor 创建，而顶部的 actor 是由库来提供的——每一个创建出来的 actor 都是由它的父亲所监管。

容错 【重要 4parts】

有两种监管策略：

- OneForOneStrategy：把指定的监管策略行为应用到失败的孩子上
- AllForOneStrategy：把指定的监管策略行为应用到**所有**子孙上

在定义的一个 Actor 类中自定义一个监管策略行为：

```
// 这里对重启的频率作了限制，最多每分钟能进行 10 次重启。如果重启次数超过限制，child cator 将被终止。
override val supervisorStrategy = OneForOneStrategy(maxNrOfRetries = 10, withinTimeRange = 1 minute) {
   case _: ArithmeticException => Resume      //如果被监控的子 actor 抛出 ArithmeticException，子 actor 会从异常崩溃中恢复，
                                                并且状态不会丢失，可以继续正常工作
   case _: NullPointerException => Restart    //如果被监控的子 actor 抛出 NullPointerException，子 actor 将会被重启，
                                                清除下属的内部状态
   case _: IllegalArgumentException => Stop   //如果被监控的子 actor 抛出 IllegalArgumentException，子 actor 将被终止
   case _: Exception => Escalate              //如果被监控的子 actor 抛出时其他的 Exception，监管者会将失败上溯
}
```

缺省的监管策略行为：

1. **如果定义的监管策略没有覆盖抛出的异常**，将使用**上溯**机制.

2. **如果 actor 没有定义监管策略，下列异常将被缺省地处理：**
   1) ActorInitializationException 将**终止**出错的子 actor
   2) ActorKilledException 将**终止**出错的子 actor
   3) DeathPactException 将**终止**出错的子 actor
   3) Exception 将**重启**出错的子 actor
   4) 其它的 Throwable 将被**上溯**传给父 actor

3. **如果异常一直被上溯到根监管者，在那儿也会用上述缺省方式进行处理**。
   顶级 actor 的缺省策略是对所有的 Exception 情况(注意 ActorInitializationException 和 ActorKilledException 是例外)进行重启。

由于缺省的重启行为会终止所有的子 actor。但有时候我们不希望发生这种情况，那么我们可以在监管者在 preRestart 方法中覆盖这一行为：

```
class Supervisor2 extends Actor {
   // 覆盖在重启时杀死所有子 actor 的缺省行为
   override def preRestart(cause: Throwable, msg: Option[Any]) {}
}
```

那么，在这个父 actor 之下的所有子 actor 会在重启中会得以幸免。

MessageDispatcher 是维持 Akka Actor 运作的部分，可以说它是整个机器的引擎。
所有的 MessageDispatcher 实现也同时是一个 ExecutionContext，这意味着它们可以用来执行任何代码。

一、缺省派发器

在没有为 Actor 作配置的情况下，ActorSystem 将有一个缺省的派发器。
缺省派发器是可配置的。**缺省情况下是一个使用“fork-join-executor”的 Dispatcher**，在大多数情况下拥有非常好的性能。

二、为 Actor 指定派发器

如果你希望为你的 Actor 设置非缺省的派发器，你需要做两件事：
第一步：在配置文件中指定 dispatcher：

```
my-dispatcher {
  type = Dispatcher                  # Dispatcher 是基于事件的派发器的名称
  executor = "fork-join-executor"    # 使用何种 ExecutionService
  # 配置 fork join 池
  fork-join-executor {
    parallelism-min = 2              # 基于 factor 的最小并行线程数量
    parallelism-factor = 2.0         # 并行数（线程） ceil(可用 CPU 数*倍数）
    parallelism-max = 10             # 基于 factor 的最大并行线程数量
  }
  # Throughput 定义了线程切换到另一个 actor 之前处理的消息数上限
  # 设置成 1 表示尽可能公平.
  throughput = 100
}
```

如果不想使用“fork-join-executor”，而是使用“thread-pool-executor”，可以像下面这样配置。这样的 dispatcher 通常用于执行 CPU 密集任务的 actor：

```
my-thread-pool-dispatcher {
  type = Dispatcher
  executor = "thread-pool-executor"
  #配置线程池
  thread-pool-executor {
    core-pool-size-min = 2           # 基于 factor 的最小并行线程数量
    core-pool-size-factor = 2.0      # 核心线程数 .. ceil(可用 CPU 数*倍数）
    core-pool-size-max = 10          # 基于 factor 的最大并行线程数量
  }
  throughput = 100
}
```

或者，配置一个固定线程数量的 dispatcher。这样的 dispatcher 通常用于执行 I/O 阻塞任务的 actor：

```
blocking-io-dispatcher {
  type = Dispatcher
  executor = "thread-pool-executor"
  thread-pool-executor {
    fixed-pool-size = 32
```

```
  }
  throughput = 1
}
```

上面三个例子都是使用 Dispatcher 类型的 dispatcher，如果想使用 PinnedDispatcher，可以像下面这样配置：

```
my-pinned-dispatcher {
  type = PinnedDispatcher
  executor = "thread-pool-executor"
}
```

第二步：在创建 actor，指定想使用的 dispatcher

①一种方式，是在配置文件 application.conf 中为某个 actor 指定 dispatcher

application.conf：

```
akka.actor.deployment {
  /myactor {
    dispatcher = my-dispatcher
  }
}
```

然后，在代码中：

```
val myActor = context.actorOf(Props[MyActor], "myactor")
```

②第二种方式，是在代码中在创建 actor 时指定 dispatcher

```
val myActor = context.actorOf(Props[MyActor].withDispatcher("my-dispatcher"), "myactor1")
```

注意：

在 withDispatcher 中指定的“dispatcherId”其实是配置中的一个路径。

所以在这种情况下它位于配置的顶层，但你可以把它放在下面的层次，用.来代表子层次，象这样: "foo.bar.my-dispatcher"

三、Dispatcher 的种类

一共有 3 种类型的消息派发器：

Dispatcher

这是一个基于事件的 dispatcher，它将一个 actors 的集合绑定在一个线程池中。

如果没有显示指定 dispatcher，这会是默认的 dispatcher。

- 可共享性: 无限制
- **邮箱：为每一个 Actor 创建一个邮箱**
- **使用场景：缺省派发器，Bulkheading(防水墙/防水层)**
- 底层使用: java.util.concurrent.ExecutorService。可以通过 executor 属性指定是使用“fork-join-executor”还是“thread-pool-executor”。

PinnedDispatcher

凡是使用这个 dispatcher 的 actor，都会被分配一个唯一的线程。也就是说每个 actor 将会拥有一个它属于自己的线程池，这个线程池中只有一个线程。

- 可共享性: 无
- **邮箱：为每个 Actor 创建一个邮箱**
- **使用场景: Bulkheading**

底层使用: 任何 akka.dispatch.ThreadPoolExecutorConfigurator。缺省为一个“thread-pool-executor”

CallingThreadDispatcher

这个 dispatcher 仅仅在当前线程中执行调用。并不会创建新的线程。

- 可共享性: 无限制
- **邮箱：每 Actor 每线程创建一个邮箱（需要时）**
- **使用场景：测试**
- 底层使用: 调用的线程 (duh)

四、小心地管理阻塞操作

一定要注意：
最好不要将同步的操作和异步的操作放在同一个 dispatcher 中执行。

一种合适的解决方案是：专门配置一个额外的 dispatcher 用来执行阻塞操作。这种技术也经常被称作：**bulk-heading**。

在配置文件中专门配置一个用来执行阻塞操作的 dispatcher，像下面这样：

```
(固定线程数量的 dispatcher 用于执行 I/O 阻塞任务的 actor)
my-blocking-dispatcher {
  type = Dispatcher
  executor = "thread-pool-executor"
  thread-pool-executor {
    fixed-pool-size = 16
  }
  throughput = 1
}
```

然后，在代码中使用我们配置的这个额外 dispatcher：

```
class SeparateDispatcherFutureActor extends Actor {
  implicit val executionContext: ExecutionContext = context.system.dispatchers.lookup("my-blocking-dispatcher")
  def receive = {
    case i: Int =>
      println(s"Calling blocking Future: ${i}")
      Future {
        Thread.sleep(5000)          //block for 5 seconds
        println(s"Blocking future finished ${i}")
      }
  }
}
```

总结：This is the recommended way of dealing with any kind of blocking in reactive applications.

Mailbox

Mailbox 保存发往某 Actor 的消息。

通常每个 Actor 拥有自己的邮箱，
但是如果是使用 BalancingDispatcher 类型的消息派发器，则使用同一个 BalancingDispatcher 的所有 Actor 共享同一个邮箱实例。

一、为 Actor 指定 Mailbox

当一个 Actor 被创建的时候，ActorRefProvider 首先确定哪一个 dispatcher 被执行，然后按照下面的步骤确定使用哪个 mailbox。

1. 如果在 actor 的“deployment”配置部分包含一个“mailbox”属性，则这个属性指定的 mailbox type 将被使用；

```
prio-mailbox {
    mailbox-type = "docs.dispatcher.DispatcherDocSpec$MyPrioMailbox"
}
akka.actor.deployment {
    /actorName {
        mailbox = prio-mailbox
    }
}
val myActor = context.actorOf(Props[MyActor], "actorName")
```

2. 如果在创建 actor 时，调用了 Props 的 withMailbox 方法，这个方法中指定的 mailbox type 将被使用；

```
prio-mailbox {
    mailbox-type = "docs.dispatcher.DispatcherDocSpec$MyPrioMailbox"
}
val myActor = context.actorOf(Props[MyActor].withMailbox("prio-mailbox"))
```

3. 如果在“dispatcher”配置部分包含一个“mailbox-type”属性，则这个属性指定的 mailbox type 将被使用；

```
prio-dispatcher {
    mailbox-type = "docs.dispatcher.DispatcherDocSpec$MyPrioMailbox"
}
val a = system.actorOf(Props[ActorClass].withDispatcher("prio-dispatcher"))
```

4. 使用默认邮箱 akka.actor.default-mailbox。

如果没有按照上面的方式指定使用 mailbox 时，默认的 mailbox 将被使用。
默认情况它是无界的邮箱，由 java.util.concurrent.ConcurrentLinkedQueue 实现。

二、内置的邮箱实现

1. UnboundedMailbox

- 默认的邮箱
- 底层是一个 java.util.concurrent.ConcurrentLinkedQueue
- 阻塞: 否
- 有界: 否
- 配置名称："unbounded" 或 "akka.dispatch.UnboundedMailbox"

2. SingleConsumerOnlyUnboundedMailbox

- 底层是一个非常高效的 Multiple-Producer Single-Consumer 队列，不能被用于 BalancingDispatcher
- 阻塞: 否
- 有界: 否
- 配置名称："akka.dispatch.SingleConsumerOnlyUnboundedMailbox"

3. NonBlockingBoundedMailbox

- 底层依赖一个非常高效的 Multiple-Producer Single-Consumer 队列
- 阻塞：否
- 有界：是 (discards overflowing messages into deadLetters)
- 配置名称："akka.dispatch.NonBlockingBoundedMailbox"

4. UnboundedControlAwareMailbox

- 继承自 akka.dispatch.ControlMessage 的消息会被高优先级传递。最佳的使用场景是：想立即处理某些特定的消息，不管邮箱的队列中已经有了多少消息。
- 底层是两个 java.util.concurrent.ConcurrentLinkedQueue
- 阻塞：否
- 有界：否
- 配置名称：“akka.dispatch.UnboundedControlAwareMailbox”

5. UnboundedPriorityMailbox

- 拥有相同优先级的消息被传输的顺序是不确定的，与之形成对比的是 UnboundedStablePriorityMailbox。
- 底层是一个 java.util.concurrent.PriorityBlockingQueue
- 阻塞：否
- 有界：否
- 配置名称：“akka.dispatch.UnboundedPriorityMailbox”

6. UnboundedStablePriorityMailbox

- 拥有相同优先级的消息按照 FIFO 的顺序被传输
- 底层依赖一个 akka.util.PriorityQueueStabilizer，是针对 java.util.concurrent.PriorityBlockingQueue 的一个包装。
- 阻塞：否
- 有界：否
- 配置名称：“akka.dispatch.UnboundedStablePriorityMailbox”

上面介绍的邮箱都是非阻塞的。当然也有一些邮箱在有界且阻塞的。当邮箱的容量已经满了，会阻塞发送者，还可以配置一个往邮箱中添加消息的超时时间：mailbox-push-timeout-time。

三、邮箱配置示例

下面演示如何创建一个 PriorityMailbox：

第一步：继承 UnboundedStablePriorityMailbox，并创建一个优先级生成器

```
class MyPrioMailbox(settings: ActorSystem.Settings, config: Config) extends UnboundedStablePriorityMailbox(
    PriorityGenerator {
      // 'highpriority messages should be treated first if possible
      case 'highpriority => 0

      // 'lowpriority messages should be treated last if possible
      case 'lowpriority    => 2

      // PoisonPill when no other left
      case PoisonPill      => 3

      // We default to 1, which is in between high and low
      case otherwise       => 1
    })
```

第二步：将它添加到配置文件中

```
prio-dispatcher {
  mailbox-type = "docs.dispatcher.DispatcherDocSpec$MyPrioMailbox"
}
```

第三步：演示如何使用我们创建的 MyPrioMailbox

```
class Logger extends Actor {
  val log: LoggingAdapter = Logging(context.system, this)

  self ! 'lowpriority
  self ! 'lowpriority
  self ! 'highpriority
  self ! 'pigdog
  self ! 'pigdog2
  self ! 'pigdog3
  self ! 'highpriority
  self ! PoisonPill

  def receive = {
    case x => log.info(x.toString)
  }
}

val a = system.actorOf(Props(classOf[Logger], this).withDispatcher("prio-dispatcher"))
```

通过打印的日志信息，可以看到消息是以什么样的顺序被处理的：

```
/*
 * Logs:
 * 'highpriority
 * 'highpriority
 * 'pigdog
 * 'pigdog2
 * 'pigdog3
 * 'lowpriority
 * 'lowpriority
 */
```

Router

路由 Actor 是将收到的消息路由到目的 actor 的 actor。路由 actor 将消息发送给它所管理的称为‘routees’的 actor.

- 通过 router 向 routee 发送消息，就像向普通的 actor 发送消息一样，即通过 ActorRef。
- router 转发消息到 routee，并不会改变原始的发送者。
- 当 routee 答复消息时，回复将发送到原始的发送者，而不是 router。
- 并且，默认情况下，routee 会隐式的将自己作为消息的发送者。但是，有时候将 router 置为消息的发送者是非常有用的，例如，你想隐藏 routee 的细节。下面的代码展示如何将 router 置为 routee 回复消息的发送者：

  sender().tell("reply", context.parent) // replies will go back to parent

  sender().!("reply")(context.parent) // alternative syntax (beware of the parens!)

路由 actor 有两种不同的模式：

- Pool：Router 创建 Routees 作为它的子 actors，并在子 actor 终止时将它从 Router 中删除；
- Group：Routees 在 Router 的外部创建，并且 Router 通过使用 actor selection 将消息发送到指定的 actor path。
  Router 不会 watch Routee 的终止。

一、Pool

1、创建 RoundRobinPool 类型的 Router

作用：这种类型的 router 在发送消息时会**轮询**所有的 routee

①在配置文件中配置 router 的相关信息，然后在代码中创建 Router

```
akka.actor.deployment {
  /parent/router1 {
    router = round-robin-pool
    nr-of-instances = 5
  }
}
```

创建 router：

```
val router1: ActorRef = context.actorOf(FromConfig.props(Props[Worker]), "router1")
```

②以编程的方式配置 router，并创建 router actor

```
val router2: ActorRef = context.actorOf(RoundRobinPool(5).props(Props[Worker]), "router2")
```

2、创建 RandomPool 类型的 Router

作用：这种类型的 router 在发送消息时会**随机**选取一个 routee 作为转发消息的目标

①在配置文件中配置 router 的相关信息，然后在代码中创建 Router

```
akka.actor.deployment {
  /parent/router1 {
    router = randon-pool
    nr-of-instances = 5
  }
}
```

创建 router：

```
val router1: ActorRef = context.actorOf(FromConfig.props(Props[Worker]), "router1")
```

②以编程的方式配置 router，并创建 router actor

```
val router2: ActorRef = context.actorOf(RandonPool(5).props(Props[Worker]), "router2")
```

3、创建一个 SmallestMailboxPool 类型的 Router

作用：这种类型的路由 actor 在发送消息选择 routee 的方式是(优先级从高往低)：

- 选取任何一个空闲的（没有正在处理的消息）邮箱为空的 routee
- 选择任何邮箱为空的 routee
- 选择邮箱中等待的消息最少的 routee
- 选择任何一个远程 routee, 由于邮箱大小未知，远程 actor 被认为具有低优先级

①在配置文件中配置 router 的相关信息，然后在代码中创建 Router

```
akka.actor.deployment {
  /parent/router1 {
    router = smallest-mailbox-pool
    nr-of-instances = 5
  }
}
```

创建 router：

```
val router1: ActorRef = context.actorOf(FromConfig.props(Props[Worker]), "router1")
```

②以编程的方式配置 router，并创建 router actor

```
val router2: ActorRef = context.actorOf(SmallestMailboxPool(5).props(Props[Worker]), "router2")
```

4、创建一个 BroadcastPool 类型的 Router

作用：这种类型的路由 actor 会将消息转发给所有的 routee。类似于消息广播。

①在配置文件中配置 router 的相关信息，然后在代码中创建 Router

```
akka.actor.deployment {
  /parent/router1 {
    router = broadcast-pool
    nr-of-instances = 5
  }
}
```

创建 router：

```
val router1: ActorRef = context.actorOf(FromConfig.props(Props[Worker]), "router1")
```

②以编程的方式配置 router，并创建 router actor

```
val router2: ActorRef = context.actorOf(BroadcastPool(5).props(Props[Worker]), "router2")
```

5、创建 ScatterGatherFirstCompletedPool 类型的 Router

作用：

这种类型的 Router 会在发送消息时将消息作为一个 Future 发送给**所有**的 Routees。

然后等待回送的第一个结果，这个结果将被发还给最初的发送者。其它的回复将被丢弃。

在指定的时间内，Router 期望至少收到一个回复。如果在指定的时间之内没有收到任何一个回复，Router 将会收到包装了 akka.pattern.AskTimeoutException 的 akka.actor.Status.Failure。

①在配置文件中配置 router 的相关信息，然后在代码中创建 Router

```
akka.actor.deployment {
  /parent/router1 {
    router = scatter-gather-pool
    nr-of-instances = 5
    within = 10 seconds
  }
}
```

创建 router：

```
val router1: ActorRef = context.actorOf(FromConfig.props(Props[Worker]), "router1")
```

②以编程的方式配置 router，并创建 router actor

```
val router2: ActorRef = context.actorOf(ScatterGatherFirstCompletedPool(5, within = 10.seconds).props(Props[Worker]), "router2")
```

6、创建一个 TailChoppingPool 类型的 Router

作用：

这种类型的 Router 首先把消息发送到随机选取的一个 Routee，然后在一段时间延迟之后，在剩余的 Routees 再随机选取一个 Routee 把消息发给它。

Router 等待第一个回复，将它转发给原始的发送者。其它的回复被丢弃。

①在配置文件中配置 router 的相关信息，然后在代码中创建 Router

```
akka.actor.deployment {
  /parent/router1 {
    router = tail-chopping-pool
    nr-of-instances = 5
    within = 10 seconds
    tail-chopping-router.interval = 20 milliseconds
  }
}
```

创建 router：

```
val router1: ActorRef = context.actorOf(FromConfig.props(Props[Worker]), "router1")
```

②以编程的方式配置 router，并创建 router actor

```
val router2: ActorRef = context.actorOf(TailChoppingPool(5, within = 10.seconds, interval = 20.millis).props(Props[Worker]), "router2")
```

二、Group

有时候也期望单独创建 routees，并将它们提供给 router 供其使用。
通过将 routee 的路径传递给 router 的配置来实现。消息将通过 ActorSelection 发送到这些路径。

1、创建一个 RoundRobinGroup 类型的 Router

①在配置文件中配置 routee paths，然后在在代码中创建 router

```
akka.actor.deployment {
  /parent/router3 {
    router = round-robin-group
    routees.paths = ["/user/workers/w1", "/user/workers/w2", "/user/workers/w3"]
  }
}
```

创建 router：

```
val router3: ActorRef = context.actorOf(FromConfig.props(), "router3")
```

②在代码中配置 routee paths，同时创建 router

```
val router4: ActorRef = context.actorOf(RoundRobinGroup(paths).props(), "router4")
```

2、创建一个 RandomGroup 类型的 Router

①在配置文件中配置 routee paths，然后在在代码中创建 router

```
akka.actor.deployment {
  /parent/router3 {
    router = random-group
    routees.paths = ["/user/workers/w1", "/user/workers/w2", "/user/workers/w3"]
  }
}
```

创建 router：

```
val router3: ActorRef = context.actorOf(FromConfig.props(), "router3")
```

②在代码中配置 routee paths，同时创建 router

```
val router4: ActorRef = context.actorOf(RandomGroup(paths).props(), "router4")
```

3、创建一个 BroadcastGroup 类型的 Router

①在配置文件中配置 routee paths，然后在在代码中创建 router

```
akka.actor.deployment {
  /parent/router3 {
    router = broadcast-group
    routees.paths = ["/user/workers/w1", "/user/workers/w2", "/user/workers/w3"]
  }
}
```

创建 router：

```
val router3: ActorRef = context.actorOf(FromConfig.props(), "router3")
```

②在代码中配置 routee paths，同时创建 router

```
val router4: ActorRef = context.actorOf(BroadcastGroup(paths).props(), "router4")
```

4、创建一个 ScatterGatherFirstCompletedGroup 类型的 Router

①在配置文件中配置 routee paths，然后在在代码中创建 router

```
akka.actor.deployment {
   /parent/router3 {
      router = scatter-gather-group
      routees.paths = ["/user/workers/w1", "/user/workers/w2", "/user/workers/w3"]
      within = 10 seconds
   }
}
```

创建 router：

```
val router3: ActorRef = context.actorOf(FromConfig.props(), "router3")
```

②在代码中配置 routee paths，同时创建 router

```
val router4: ActorRef = context.actorOf(ScatterGatherFirstCompletedGroup(paths, within = 10.seconds).props(), "router4")
```

5、创建一个 TailChoppingGroup 类型的 Router

①在配置文件中配置 routee paths，然后在在代码中创建 router

```
akka.actor.deployment {
   /parent/router3 {
      router = tail-chopping-group
      routees.paths = ["/user/workers/w1", "/user/workers/w2", "/user/workers/w3"]
      within = 10 seconds
      tail-chopping-router.interval = 20 milliseconds
   }
}
```

创建 router：

```
val router3: ActorRef = context.actorOf(FromConfig.props(), "router3")
```

②在代码中配置 routee paths，同时创建 router

```
val router4: ActorRef = context.actorOf(TailChoppingGroup(paths, within = 10.seconds, interval = 20.millis).props(), "router4")
```

三、特殊的消息

1、Broadcast 消息

有一类特殊的消息，不管是哪种类型的 Router，都会被广播给所有的 routee，这种消息被称为 Broadcast。

用以下的方法使用：

router ! Broadcast("hellor")

注意：

只有实际的消息被转发到 routees，例如上例中的“hello”。

是否要处理广播消息由 routee 的实现决定。

2、PoisonPill 消息

一个 Router 收到了 PoisonPill 消息，Router 会被终止。

Router 不会将 PoisonPill 消息转发给它的 Routees。尽管不会将这个消息转发给它的 Routees，但是如果有的 Routees 是它的子 Actor，这些 Routees 同样也会被终止(这是终止的正常行为)(不是 Router 子 Actor 的 Routees 不会被终止)。

1. Router 会被终止
2. Router 的所有子 Actors 都会被终止
3. Router 的所有 Routees 都会被终止；
4. 如果在收到 PoisonPill 消息之前，邮箱中还有消息，则会继续处理，直到处理到 PoisonPill。

router ! Broadcast(PoisonPill)

1. Router 会被终止
2. Router 的所有子 Actors 都会被终止
3. 只有是 Router 的子 Actor 的 Routees 才会被终止；
4. 被终止的 Routee 会立即终止，即使邮箱中还有消息也不会再处理。

router ! PoisonPill

3、Management 消息

Sending akka.routing.GetRoutees to a router actor will make it send back its currently used routees in a akka.routing.Routees message.

Sending akka.routing.AddRoutee to a router actor will add that routee to its collection of routees.

Sending akka.routing.RemoveRoutee to a router actor will remove that routee to its collection of routees.

Sending akka.routing.AdjustPoolSize to a pool router actor will add or remove that number of routees to its collection of routees.

四、远程部署 Routees

- 除了可以创建本地 actors 作为 routees，也可以让 router 将它创建的子 actors 部署在一系列远程主机上。
- routees 将会以**轮询**的方式被部署。
- 为了远程部署 routees，需要将 router 的配置包装在 RemoteRouterConfig 中，并附加远程主机的地址信息。
- 远程部署需要引入 akka-remote 模块

代码示例：

```
val addresses = Seq(
  Address("akka.tcp", "remotesys", "otherhost", 1234),
  AddressFromURIString("akka.tcp://othersys@anotherhost:1234")
)
val routerRemote = system.actorOf(RemoteRouterConfig(RoundRobinPool(5), addresses).props(Props[Echo]))
```

五、配置 Router 的监管策略

- **如果没有配置监管策略，默认的策略**是“always escalate”。由 router 的监管者来决定如何处理错误。
  router 的监管者将会把错误视为 router 本身的错误(实际上这些错误是由 routee 产生的)。因此 router 的监管者在发起 Stop(终止)或 Restart(重启)指令时，将会造成 router 本身终止或重启，相应的，router 也会终止或重启它的子 actors。

设置 router 的监管策略是很容易的，可以使用 supervisorStrategy 属性来为 router 配置监管策略。

代码示例：

```
val escalator = OneForOneStrategy() {
  case e ⇒ testActor ! e; SupervisorStrategy.Escalate
}
val router = system.actorOf(RoundRobinPool(1, supervisorStrategy = escalator).props(routeeProps = Props[TestActor]))
```

注意：

如果 pool 模式的 router 的子 actor 终止了，router 不会自动的产生一个新的子 actor。

若 router 的所有子 actor 都已经终止了，则 router 将会终止它自身。除非它是一个动态的路由器，例如使用了大小调整。

六、配置 Dispatcher

用于创建 Router 的子 actor 的 dispatcher 取自 Props。

为 routees 定义 Dispatcher 是很容易的：

```
akka.actor.deployment {
  /poolWithDispatcher {
    router = random-pool
    nr-of-instances = 5
    pool-dispatcher {
      fork-join-executor.parallelism-min = 5
      fork-join-executor.parallelism-max = 5
    }
  }
}
```

七、动态改变 routee 数量(与 Pool 模式的 Router 相关)

Router 可以拥有固定数量的 routees 或者也可以有一个改变大小的策略来动态调整 routees 的数量.

有两种类型的 resizer：default Resizer 和 OptimalSizeExploringResizer。

1、Default Resizer

default resizer 根据 routees 的繁忙程度来增加或者减少 routees 的数量。

① 在配置文件配置一个带有 default resizer 的 Router

```
akka.actor.deployment {
  /parent/router29 {
    router = round-robin-pool
    resizer {
      lower-bound = 2
      upper-bound = 15
      messages-per-resize = 100
    }
  }
}
```

创建 Router：

```
val router29: ActorRef = context.actorOf(FromConfig.props(Props[Worker]), "router29")
```

② 使用编程的方式创建一个带有 default resizer 的 Router

```
val resizer = DefaultResizer(lowerBound = 2, upperBound = 15)
val router30: ActorRef = context.actorOf(RoundRobinPool(5, Some(resizer)).props(Props[Worker]), "router30")
```

远程调用 Akka-remote

一、使你的 ActorSystem 作好远程调用的准备

Akka 远程调用功能在一个单独的 jar 包中，在使用 AKKA 的远程调用之前需要引入依赖 akka-remote 模块。

同时，要在 Akka 项目中使用远程调用，最少要在 application.conf 文件中加入以下内容:

```
akka {
  actor {
    provider = "akka.remote.RemoteActorRefProvider"
  }
  remote {
    enabled-transports = ["akka.remote.netty.tcp"]
    netty.tcp {
      hostname = "127.0.0.1"
      port = 2552
    }
  }
}
```

解释：

- 将 provider 从 akka.actor.LocalActorRefProvider 改为 akka.remote.RemoteActorRefProvider
- 增加远程主机名：你希望运行 actor 系统的主机。
- 增加端口号：actor 系统监听的端口号。0 表示让它自动选择。

二、远程交互的类型

1、查找远程 Actor

客户端在寻找远程 ActorRef 时，也需要在配置文件中进行配置：

```
akka {
  actor {
    provider = "akka.remote.RemoteActorRefProvider"
  }
}
```

然后，客户端可以这样需要远程 ActorRef：

```
val remoteActor = context.actorSelection("akka.tcp://RpcServerSystem@127.0.0.1:2552/user/serverActor")
```

一旦得到了 actor 的引用，你就可以象与本地 actor 通讯一样与它进行通迅，例如：

```
actor ! "Pretty awesome feature"
```

2、创建远程 Actor

(1) 在配置文件中配置创建远程 Actor 的配置信息

首先，在 Akka 中要使用远程创建 actor 的功能，需要对 application.conf 文件进行以下修改 (只显示 deployment 部分):

```
akka.actor.deployment {
        /sampleActor {
          remote = "akka://sampleActorSystem@127.0.0.1:2553"
        }
}
```

解释：

这个配置告知 Akka 当一个路径为 /sampleActor 的 actor 被创建时进行响应。例如：调用 system.actorOf(Props(...), sampleActor)时，指定的 actor 不会被直接实例化，而是远程 actor 系统的 daemon 被要求创建这个 actor。

一旦配置了以上属性你可以在代码中进行如下操作：

```
class SampleActor extends Actor {
        def receive = { case _ => println("Got something") }
}
val actor = context.actorOf(Props[SampleActor], "sampleActor")
actor ! "Pretty slick"
```

(2) 用代码进行远程部署

要允许动态部署系统，也可以在用来创建 actor 的 Props 中包含 deployment 配置，这一部分信息与配置文件中的 deployment 部分是等价的，如果两者都有，则外部配置拥有更高的优先级。

有两种方式创建远程系统地址：

```
val one = AddressFromURIString("akka://sys@host:1234")
```

或

```
val two = Address("akka", "sys", "host", 1234)
```

然后就可以用代码进行远程部署：

```
val ref = system.actorOf(Props[Echo].withDeploy(Deploy(scope = RemoteScope(address))))
```

三、有远程目标的路由 actor

将 *远程调用* 与 *路由* 进行组合是非常实用的. 也要用配置文件:

```
akka.actor.deployment {
        /serviceA/aggregation {
          router = "round-robin"
          nr-of-instances = 10
          target {
            nodes = ["akka://app@10.0.0.2:2552", "akka://app@10.0.0.3:2552"]
          }
        }
}
```

这个配置文件会将“aggregation”actor 复制 10 份并将其均匀地部署到给定的两个远程结点上.

序列化

Akka 提供内置的序列化支持扩展，你可以选择使用内置的序列化功能，也可以自己写一个。
内置的序列化功能被 Akka 内部用来序列化消息，你也可以用它做其它的序列化工作。

一、通过配置使用序列化

首先，配置一些 Serializer，将 akka.serialization.Serializer 的不同实现类与指定的名字绑定，这部分的配置写在“akka.actor.serializers”中。
然后，在配置中指定哪些类的序列化需要使用哪种 Serializer，这部分配置写在“akka.actor.serialization-bindings”部分
示例：

```
val config = ConfigFactory.parseString("""
  akka {
    actor {
      serializers {
        java = "akka.serialization.JavaSerializer"
        proto = "akka.serialization.ProtobufSerializer"
        myown = "akka.docs.serialization.MyOwnSerializer"
      }
      serialization-bindings {
        "java.lang.String" = java
        "akka.docs.serialization.Customer" = java
        "com.google.protobuf.Message" = proto
        "akka.docs.serialization.MyOwnSerializable" = myown
        "java.lang.Boolean" = myown
      }
    }
  }
""")
```

注意：
Akka 缺省提供使用 java.io.Serializable 和 protobuf com.google.protobuf.GeneratedMessage 的序列化工具（后者仅当定义了对 akka-remote 模块的依赖时才有），所以通常你不需要添加这两种配置。
由于 com.google.protobuf.GeneratedMessage 实现了 java.io.Serializable，在不特别指定的情况下，protobuf 消息将总是用 protobuf 协议来做序列化。
要禁止缺省的序列化工具，将其对应的类型设为 none：

```
akka.actor.serialization-bindings {
  "java.io.Serializable" = none
}
```

二、通过代码使用序列化

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

val serialization = SerializationExtension(system)          // 获取序列化扩展工具

val original = "woohoo"    // 定义一些要序列化的东西

val serializer = serialization.findSerializerFor(original)    // 为它找到一个 Serializer

val bytes = serializer.toBinary(original)                    // 转化为字节

val back = serializer.fromBinary(bytes, manifest = None)    // 转化回对象

back must equal(original)      // 就绪!
```

三、自定义 Serializer

首先，自定义 Serializer 的实现类：

```
class MyOwnSerializer extends Serializer {
  // 指定 "fromBinary" 是否需要一个 "clazz"
  def includeManifest: Boolean = false

  // 为你的 Serializer 选择一个唯一标识。基本上所有的整数都可以用，但 0 - 16 是 Akka 自己保留的
  def identifier = 1234567

  // 将对象序列化为字节数组
  def toBinary(obj: AnyRef): Array[Byte] = {
    // 将序列化的代码写在这儿
  }

  // 对字节数组进行反序列化。使用类型提示 (如果有的话, 见上文的 "includeManifest" )，使用可能提供的 classLoader.
  def fromBinary(bytes: Array[Byte], clazz: Option[Class[_]]): AnyRef = {
    // 将反序列化的代码写在这儿
  }
}
```

然后，在配置文件中将它绑定到一个名称，然后列出需要用它来做序列化的类即可。

有限状态机 FSM

一个 FSM 可以描述成一组具有如下形式的关系：

State(S) x Event(E) -> Actions (A), State(S’)

这些关系的意思可以这样理解：如果我们当前处于状态 S，发生了 E 事件，我们应执行操作 A，然后将状态转换为 S’.

FSM trait 只能被混入到 Actor 子类中.

FSM trait 有两个类型参数：

- 所有**状态名称**的父类型，通常是一个 sealed trait，状态名称作为 case object 来继承它
- **状态数据**的类型，由 FSM 模块自己跟踪.

注意：

状态数据与状态名称一起描述了状态机的内部状态;

1、定义初始状态

每个 FSM 都需要一个起点，用 startWith(state, data[, timeout])来声明。

可选的超时参数将覆盖所有为期望的初始状态指定的值. 想要取消缺省的超时，使用 Duration.Inf.

2、定义状态

状态的定义是通过一次或多次调用:

when(<name>[, stateTimeout = <timeout>])(stateFunction)

name 参数：给定的名称对象必须与为 FSM trait 指定的第一个参数类型相匹配.

这个对象将被用作一个 hash 表的键，所以你必须确保它正确地实现了 equals hashCode 方法;

特别是它不能是可变量. 满足这些条件的最简单的就是 case objects.

stateTimeout 参数：如果给定了 stateTimeout 参数，那么所有到这个状态的转换，包括停留，缺省都会收到这个超时.

发起状态转换时可以显式指定一个超时用来覆盖这个缺省行为.

在操作执行的过程中可以通过 setStateTimeout(state, duration)来修改任何状态的超时时间. 这使得运行时配置（e.g 通过外部消息）成为可能.

stateFunction 参数：是一个 PartialFunction[Event, State]。

Event(msg: Any, data: D) case class 以 FSM 所持有的数据类型为参数，以便进行模式匹配.

3、未处理事件

如果一个状态未能处理一个收到的事件，可以交给 whenUnhandled 方法。

这种情况下如果你想做点其它的事，你可以指定 whenUnhandled(stateFunction)

例如：

```
whenUnhandled {
  case Event(x : X, data) =>
    log.info(this, "Received unhandled event: " + x)
    stay
  case Event(msg, _) =>
    log.warn(this, "Received unknown event: " + x)
    goto(Error)
}
```

4、发起状态转换

任何 stateFunction 的结果都必须是新状态的定义
状态定义可以是当前状态，由 **stay** 指令描述；
或由 **goto(state)** 指定的另一个状态.

产生的新的 State 实例可以通过下面列出的修饰器作进一步限制:
forMax(duration)
这个修饰器为新状态指定状态超时. 这意味着将启动一个定时器，当新状态过期时将向 FSM 发送一个 StateTimeout 消息.
如果在超时时间之内接收到任何其它消息时定时器将被取消; 也就是说 StateTimeout 消息不会在任何一个中间消息之后被处理.

using(data)
这个修饰器用给定的新数据取代旧的状态数据。这是内部状态数据被修改的唯一位置。

replying(msg)
这个修饰器为当前处理的消息发送一个应答，不同的是它不会改变状态转换.
reply 机制可以用来在状态转换前向消息的发送者回复任何信息。

所有的修饰器都可以链式调用来获得优美简洁的表达方式:

```
when(State) {
   case Event(msg, _) =>
      goto(Processing) using (msg) forMax (5 seconds) replying (WillDo)
}
```

5、监控状态转换

内部监控
onTransition(handler)　　// 将操作与状态转换联系在一起，而不是将操作与状态或事件联系在一起。
参数 handler 是一个偏函数，它以一对状态作为输入；不需要结果状态因为不可能改变正在进行的状态转换.

需要注意的是：用这个方法注册的处理器是迭加的，也就是说，所有的处理器对每一次状态转换都会被调用，而不是最先匹配的那个.

示例：

```
onTransition {
   case Idle -> Active => setTimer("timeout")
   case Active -> _ => cancelTimer("timeout")
   case x -> Idle => log.info("entering Idle from "+x)
}
```

extractor -> 用来以清晰的形式解开状态对并表达了状态转换的方向.

6、定时器

- 可以用 setTimer(name, msg, interval, repeat) 设置定时器。
  - 其中 name 是定时器的标识
  - 其中 msg 是经过 interval 时间以后发送的消息. 如果 repeat 设成 true，定时器将以 interval 指定的时间段重复规划.
- 用 cancelTimer(name)来取消定时器
- 查询定时器的状态可以用 timerActive_?(name)

7、终止 FSM

(1) 从内部终止 FSM

将结果状态设置为 stop([reason[, data]]) 将终止 FSM

- reason 参数必须是 Normal (which is the default), Shutdown 或 Failure(reason) 之一；
- 第二个参数用来改变状态数据，在终止处理器中可以使用该数据。

注意：必须注意 stop 并不会停止当前的操作，不会立即停止 FSM。 stop 操作必须象状态转换一样从事件处理器中返回。

可以用 onTermination(handler) 来指定当 FSM 停止时要运行的代码.

其中的 handler 是一个以 StopEvent(reason, stateName, stateData) 为参数的偏函数:

```
onTermination {
  case StopEvent(Normal, s, d)          => ...
  case StopEvent(Shutdown, _, _)        => ...
  case StopEvent(Failure(cause), s, d)  => ...
}
```

(2) 从外部终止

当 FSM 关联的 ActorRef 被 stop 方法停止后, 它的 postStop hook 将被执行.

在 FSM trait 中的缺省实现是执行 onTermination 处理器（如果有的话）来处理 StopEvent(Shutdown, ...)事件.

注意：

如果你重写了 postStop 而希望你的 onTermination 处理器被调用, 别忘了调用 super.postStop.

8、FSM 源码分析　　【重要】

1. State 类、Event 类源码

```
case class State[S, D](stateName: S, stateData: D, timeout: Option[FiniteDuration] = None, stopReason: Option[Reason] = None, replies: List[Any] = Nil) {

    def copy(stateName: S = stateName, stateData: D = stateData, timeout: Option[FiniteDuration] = timeout, stopReason: Option[Reason] = stopReason, replies: List[Any] = replies): State[S, D] = {
      new State(stateName, stateData, timeout, stopReason, replies)
    }

    /**
     * 为下一个状态指定超时时间
     */
    def forMax(timeout: Duration): State[S, D] = timeout match {
      case f: FiniteDuration  ⇒  copy(timeout = Some(f))
      case Duration.Inf       ⇒  copy(timeout = SomeMaxFiniteDuration)
      case _                  ⇒  copy(timeout = None)
    }

    /**
     * 为当前消息的发送者发送响应
     * 此方法返回转换时的下一个状态
     */
    def replying(replyValue: Any): State[S, D] = {
      copy(replies = replyValue :: replies)
    }

    /**
     * 用新的状态数据替代旧的状态数据
     */
    def using(@deprecatedName('nextStateDate) nextStateData: D): State[S, D] = {
      copy(stateData = nextStateData)
    }
  }
```

```
// 当前的收到的消息和当前的状态数据会被包装成 Event 对象。通过模式匹配可以得到发生的事件以及当前的状态数据。
final case class Event[D](event: Any, stateData: D)
```

2. FSM 特质源码剖析

```
trait FSM[S, D] extends Actor with Listeners with ActorLogging {

  // 一旦在混入 FSM 特质时指定了 S 和 D 的类型，那么也就是状态名称和状态数据的类型确定了
  type State = FSM.State[S, D]
  type Event = FSM.Event[D]
  type StopEvent = FSM.StopEvent[S, D]
  type StateFunction = scala.PartialFunction[Event, State]
  type Timeout = Option[FiniteDuration]
  type TransitionHandler = PartialFunction[(S, S), Unit]


  // 调用 startWith 方法，得到的是一个 State case class 的实例
  final def startWith(stateName: S, stateData: D, timeout: Timeout = None): Unit =
    currentState = FSM.State(stateName, stateData, timeout)     // 创建一个 State 类实例，赋值给 currentState 变量


  // 调用 when 方法，将发生如下事情：
     将当前的状态名称与状态函数存在 Map 映射中；
     将当前的状态名称与状态超时时间存在 Map 映射中；(setStateTimeout 方法可以修改指定 State 的超时时间)
  final def when(stateName: S, stateTimeout: FiniteDuration = null)(stateFunction: StateFunction): Unit =
    register(stateName, stateFunction, Option(stateTimeout))

  private val stateFunctions = mutable.Map[S, StateFunction]()
  private val stateTimeouts = mutable.Map[S, Timeout]()
  private def register(name: S, function: StateFunction, timeout: Timeout): Unit = {
    if (stateFunctions contains name) {
      stateFunctions(name) = stateFunctions(name) orElse function
      stateTimeouts(name) = timeout orElse stateTimeouts(name)
    } else {
      stateFunctions(name) = function
      stateTimeouts(name) = timeout
    }
  }


  private val handleEventDefault: StateFunction = {
    case Event(value, stateData) ⇒
      log.warning("unhandled event " + value + " in state " + stateName)
      stay
  }
  private var handleEvent: StateFunction = handleEventDefault
  // 如果一个状态未能处理一个收到的事件，则交给 whenUnhandled 方法
  // 如果我们提供的 stateFunction 也不能处理，则交给默认的 handleEventDefault 处理
  final def whenUnhandled(stateFunction: StateFunction): Unit =
    handleEvent = stateFunction orElse handleEventDefault
```

```
// 调用 onTransition 方法：向一个不可变列表中添加 TransitionHandler 实例，也就是说监控状态转换的操作是不断叠加的
final def onTransition(transitionHandler: TransitionHandler): Unit = transitionEvent :+= transitionHandler

private var transitionEvent: List[TransitionHandler] = Nil   // 是一个不可变列表，用来存储 TransitionHandler
private def handleTransition(prev: S, next: S) {          // 循环 transitionEvent 中存储的所有 TransitionHandler，如果参数匹配则执行
  val tuple = (prev, next)
  for (te ← transitionEvent) { if (te.isDefinedAt(tuple)) te(tuple) }
}


// 调用 setStateTimeout 方法：修改 Map 映射 stateTimeouts 中 key 为 state 参数值的 value 值为指定的 timeout
final def setStateTimeout(state: S, timeout: Timeout): Unit = stateTimeouts(state) = timeout

// 调用 goto 方法：以当前的状态数据和下个状态名称为参数重新创建一个 State 类实例
final def goto(nextStateName: S): State = FSM.State(nextStateName, currentState.stateData)

// 调用 stay 方法：重新创建一个 State 类实例，但是状态名称和状态数据不变
final def stay(): State = goto(currentState.stateName).withNotification(false)


// 调用 stop 方法时不指定参数，默认创建一个新的 State 类实例：包含当前状态名称、当前状态数据、终止理由为 Normal
// 之所以 stop 能终止 FSM，因为新的 State 的 stopReason 参数有值，在 recevie 方法的处理流程中，如果判断到新 State 的 stopReason 参数有值，就会终止 FSM
final def stop(): State = stop(Normal)
final def stop(reason: Reason): State = stop(reason, currentState.stateData)
final def stop(reason: Reason, stateData: D): State = stay using stateData withStopReason (reason)


// 调用 terminationHandler 方法：指定 FSM 终终止时要运行的代码。要运行的代码是个偏函数，以 StopEvent 为参数类型，结果类型为 Unit
private var terminateEvent: PartialFunction[StopEvent, Unit] = NullFunction
final def onTermination(terminationHandler: PartialFunction[StopEvent, Unit]): Unit =
  terminateEvent = terminationHandler
```

3. receive 方法的内部处理逻辑

收到正常消息：

1. 由**当前的收到的消息**和**当前的状态数据**组装成 Event 类示例
2. 从 Map 映射中取出当前状态名称对应的偏函数 stateFunc
3. 判断 Event 实例是否符合 stateFunc 的参数要求

   执行 when 方法中的偏函数 stateFunc

   或

   执行 whenUnhandled 方法中的偏函数

   产生一个新的 State 实例

4. 判断新的 State 实例的 stopReason 参数有没有值？

   4.1 若新 State 实例的 stopReason 参数没有值

   ①如果新 State 实例的 replies 列表中有值，则将新 State 实例的 replies 列表中存储的值逐一发送给 sender

   ②如果状态名称确实发生了改变，则调用 handleTransition 方法，以旧状态和新状态为参数值，去执行符合偏函数参数要求的 onTransition

   ③currentState = nextState

   ④不管是通过 forMax(duration)为新状态指定了超时时间，

   还是 stateTimeout Map 映射已经存储了新状态名称对应的超时时间(调用 when 方法干的)

   都会调用 System 的定时器 scheduler，在状态的超时时间之后，向自身发送 TimeoutMarker 对象消息

   4.2 若新 State 实例的 stopReason 参数有值

   ①和 4.1 的①一样处理

   ②取消所有的定时器 Timer，并清空定时器集合 timers

   ③取消状态的超时处理。也就是在超时时间到达后，不会向自身发送 TimeroutMarker 对象消息

   ④由终止原因 Reason、新状态名称、新状态数据为参数，创建 StopEvent 实例，

   如果 StopEvent 实例符合 terminateEvent 的参数要求，则执行 terminateEvent 偏函数(由 onTermination 方法指定)

   ⑤调用 context.stop(self)终止 FSM actor

Akka 扩展

AKKA 扩展由两部分组成：Extension 和 ExtensionId.

需要注意的：由于**扩展是在整个 ActorSystem 中共享的实例**，需要保证我们定义的扩展是线程安全的。

1、定义一个扩展：

```
class CountExtensionImpl extends Extension {
  private val counter = new AtomicLong(0)
  //扩展所提供的功能
  def increment() = counter.incrementAndGet()
}
```

2、为扩展指定一个 ExtensionId 这样我们可以获取它的实例.

```
object CountExtension extends ExtensionId[CountExtensionImpl] with ExtensionIdProvider {
  // ExtensionIdProvider 需要有 lookup 方法,
  // 返回自己, 这样就可以在 ActorSystem 启动时加载扩展
  override def lookup = CountExtension

  //Akka 将会调用此方法来创建扩展实例
  override def createExtension(system: ExtendedActorSystem) = new CountExtensionImpl
}
```

3、使用我们定义的扩展：

```
CountExtension(system).increment
```

或者

在 Akka Actor 中使用:

```
class MyActor extends Actor {
  def receive = {
    case someMessage ⇒
      CountExtension(context.system).increment()
  }
}
```

Extensions 在每个 ActorSystem 中只会加载一次, 并被 Akka 所管理。

你可以选择在需要的时候加载你的 Extension 或是在 ActorSystem 创建时通过 Akka 配置来加载。

为了能够从 Akka 配置中加载扩展，你需要在为 ActorSystem 提供的配置文件的 akka.extensions 部分加上 ExtensionId 或 ExtensionIdProvider 实现类的完整路径。

```
akka {
  extensions = ["akka.docs.extension.CountExtension$"]
}
```

注意在这里 CountExtension 是一个 object 所以类名以 $ 结尾.

从源码了解扩展创建流程：

ExtensionId 实现类在调用 apply(system: ActorSystem)时，会调用 system 的 registerExtension 方法：

```
def apply(system: ActorSystem): T = system.registerExtension(this)
```

在 registerExtension 方法内部，就会调用 ExtensionId 的 createExtension 方法创建扩展

因此，apply 方法的返回值就是我们定义的 Extension 实现类

测试 Actor 系统

Akka 有一个专门的 akka-testkit 模块来支持不同层次上的测试，很明显共有两个类别：

- 测试独立的不包括 actor 模型的代码，即没有多线程的内容；在事件发生的次序方面有完全确定性的行为，没有任何并发考虑，这在下文中称为单元测试（Unit Testing）。
- 测试包装过的 actor，包括多线程调度；事件的次序没有确定性但由于使用了 actor 模型，不需要考虑并发，这在下文中被称为集成测试（Integration Testing）。

单元测试通常是白盒测试，而集成测试是对完整的 actor 网络进行的功能测试。其中重要的区别是并发的考虑是否是测试的一部分。

一、用 TestActorRef 做单元测试

获取 TestActorRef

```
val actorRef = TestActorRef[MyActor]
val actor = actorRef.underlyingActor
```

预料中的异常

测试处理消息时抛出的预料中的异常可以使用基于 TestActorRef receive 的调用 ：

```
val actorRef = TestActorRef(new Actor {
  def receive = {
    case boom ⇒ throw new IllegalArgumentException("boom")
  }
})
intercept[IllegalArgumentException] { actorRef.receive("hello") }
```

测试有限状态机

如果你要测试的 actor 是一个 FSM，你可以使用专门的 TestFSMRef，它拥有普通 TestActorRef 的所有功能，并且能够访问其内部状态:

```
val fsm = TestFSMRef(new Actor with FSM[Int, String] {
  startWith(1, "")
  when(1) {
    case Event("go", _) ⇒ goto(2) using "go"
  }
  when(2) {
    case Event("back", _) ⇒ goto(1) using "back"
  }
})

assert(fsm.stateName == 1)
assert(fsm.stateData == "")
fsm ! "go" // being a TestActorRef, this runs also on the CallingThreadDispatcher
assert(fsm.stateName == 2)
assert(fsm.stateData == "go")
```

```
fsm.setState(stateName = 1)
assert(fsm.stateName == 1)
```

测试 Actor 的行为

TestActorRef 支持所有正常的 ActoRef 中的操作。发往 actor 的消息在当前线程中同步处理，应答像正常一样回送。
TestActorRef 是 LocalActorRef 的子类，只不过多加了一些特殊功能，所以像监管和重启也能正常工作。

TestActorRef 使用的消息派发器默认是 CallingThreadDispatcher，这个派发器被隐式地用于所有实例化为 TestActorRef 的 actor。
使用了 CallingThreadDispatcher 这个派发器，那么 actor 所有的执行过程都是严格同步的，它会将任何消息直接运行在当前线程中。一旦引入了异步操作，就离开了单元测试的范畴。

eg：

```
val actorRef = TestActorRef(new MyActor)
// 假设消息计算出结果 '42'
val result = Await.result((actorRef ? Say42), 5 seconds).asInstanceOf[Int]
result must be(42)
```

二、用 TestKit 进行集成测试

TestKit 中有一个名为 testActor 的 actor 作为被不同的 expectMsg...断言检查的消息的入口。
当混入了 ImplicitSender 特质后，testActor 在从测试过程中派发消息时将被隐式地用作发送引用 sender。

对接受消息的行为进行设置

- **receiveOne**(d: Duration): AnyRef

尝试等待给定的时间来等待收到一个消息，如果失败则返回 null . 如果给定的 Duration 是 0，这一调用是非阻塞的(轮询模式).

- **receiveWhile**[T](max: Duration, idle: Duration, messages: Int)(pf: PartialFunction[Any, T]): Seq[T]

只要满足下面 4 个条件之一就收集消息。返回收集到的所有消息。

消息与偏函数匹配
指定的时间还没用完
在空闲的时间内收到了下一条消息
消息数量还没有到上限

时间上限缺省值是最深层的 within 块中剩余的时间；
空闲时间缺省为无限 (也就是禁止空闲超时功能)；
期望的消息数量缺省值为 Int.MaxValue, 也就是不作这个限制.

- **awaitCond**(p: => Boolean, max: Duration, interval: Duration)

每经过 interval 时间就检查一下给定的条件，直到它返回 true 或者 max 时间用完了。
时间间隔缺省为 100 ms，而最大值缺省为最深层的 within 块中的剩余时间。

对收到的消息进行断言

上面提到的 expectMsg 并不是唯一的对收到的消息进行断言的方法. 以下是完整的列表：

- **expectMsg**[T](d: Duration, msg: T): T

给定的消息必须在指定的时间内到达；返回此消息.

- **expectMsgPF**[T](d: Duration)(pf: PartialFunction[Any, T]): T

在给定的时间内，必须有消息到达，必须为这类消息定义了偏函数；返回偏函数应用到收到的消息的结果。可以不指定时间段（这时需要一对空的括号）， 这时使用最深层的 within 块中的期限.

- **expectMsgClass**[T](d: Duration, c: Class[T]): T

在指定的时间内必须接收到 Class 类型的对象；返回收到的对象。 注意它的类型匹配是子类兼容的；如果需要类型是相等的，参考使用单个 class 参数的 expectMsgAllClassOf 。

- **expectMsgType**[T: Manifest](d: Duration)

在指定的时间内必须收到指定类型 (擦除后)的对象; 返回收到的对象。这个方法基本上与 expectMsgClass(manifest[T].erasure)等价.

- **expectMsgAnyOf**[T](d: Duration, obj: T*): T

在指定的时间内必须收到一个对象，而且此对象必须与传入的对象引用中的一个相等(用 == 进行比较); 返回收到的对象.

- **expectMsgAnyClassOf**[T](d: Duration, obj: Class[_ <: T]*): T

在指定的时间内必须收到一个对象，它必须至少是指定的某 Class 对象的实例; 返回收到的对象.

- **expectMsgAllOf**[T](d: Duration, obj: T*): Seq[T]

在指定时间内必须收到与指定的数组中相等数量的对象, 对每个收到的对象，必须至少有一个数组中的对象与它相等(用==进行比较) . 返回收到的整个对象集合.

- **expectMsgAllClassOf**[T](d: Duration, c: Class[_ <: T]*): Seq[T]

在指定时间内必须收到与指定的 Class 数组中相等数量的对象，对数组中的每一个 Class， 必须至少有一个对象的 Class 与它相等(用==进行比较) (这 不是 子类兼容的类型检查). 返回收到的整个对象集合.

- **expectMsgAllConformingOf**[T](d: Duration, c: Class[_ <: T]*): Seq[T]

在指定时间内必须收到与指定的 Class 数组中相等数量的对象，对数组中的每个 Class 必须至少有一个对象是这个 Class 的实例。返回收到的整个对象集合。

- **expectNoMsg**(d: Duration)

在指定时间内不能收到消息。如果在这个方法被调用之前已经收到了消息，并且没有用其它的方法将这些消息从队列中删除，这个断言也会失败。

- **receiveN**(n: Int, d: Duration): Seq[AnyRef]

指定的时间内必须收到 n 条消息; 返回收到的消息.

- **fishForMessage**(max: Duration, hint: String)(pf: PartialFunction[Any, Boolean]): Any

只要时间没有用完，并且偏函数匹配消息并返回 false 就一直接收消息. 返回使偏函数返回 true 的消息或抛出异常, 异常中会提供一些提示以供 debug 使用。

预料的异常

由于集成测试无法进入参与测试的 actor 的内部处理流程，无法直接确认预料中的异常。

为了做这件事，只能使用日志系统：将普通的事件处理器替换成 TestEventListener，然后使用 EventFilter 可以对日志信息、包括由于异常产生的日志，做断言。

eg：

```
implicit val system = ActorSystem("testsystem", ConfigFactory.parseString("""
  akka.event-handlers = ["akka.testkit.TestEventListener"]
"""))
try {
  val actor = system.actorOf(Props.empty)
  EventFilter[ActorKilledException](occurrences = 1) intercept {
    actor ! Kill
```

```
    }
} finally {
    system.shutdown()
}
```

对定时的断言

功能测试的另一个重要部分与定时器有关：有些事件不能立即发生（如定时器），另外一些需要在时间期限内发生。
有一个新的工具来管理时间期限：

```
within([min, ]max) {
    ...
}
```

within 所带的代码块必须在一个介于 min 和 max 之间的 Duration 之前完成。
其中 min 默认值为 0。 如果你没有指定 max 值，它会从最深层的 within 块继承这个值。

如果代码块的最后一条接收消息断言是 expectNoMsg 或 receiveWhile，对 within 的最终检查将被跳过，以避免由于唤醒延迟导致的错误的 true 值。
这意味着虽然其中每一个独立的断言仍然使用时间上限，整个代码块在这种情况下会有长度随机的延迟。
eg：

```
val worker = system.actorOf(Props[Worker])
within(200 millis) {
    worker ! "some work"
    expectMsg("some result")
    expectNoMsg       // 在剩下的 200ms 中会阻塞
    Thread.sleep(300) // 不会使当前代码块失败
}
```

使用探针 Actor

如果待测的 actor 会发送多个消息到不同的目标，这时候测试就可以使用探针 Actor。它的功能可以用下面的例子说明：

```
class MyDoubleEcho extends Actor {
      var dest1: ActorRef = _
      var dest2: ActorRef = _
      def receive = {
        case (d1: ActorRef, d2: ActorRef) ⇒
          dest1 = d1
          dest2 = d2
        case x ⇒
          dest1 ! x
          dest2 ! x
      }
}
val probe1 = TestProbe()
val probe2 = TestProbe()
val actor = system.actorOf(Props[MyDoubleEcho])
actor ! (probe1.ref, probe2.ref)
```

```
actor ! "hello"
probe1.expectMsg(500 millis, "hello")
probe2.expectMsg(500 millis, "hello")
```

为探针 actor 配备自定义的断言：

```
case class Update(id: Int, value: String)
val probe = new TestProbe(system) {
  def expectUpdate(x: Int) = {
    expectMsgPF() {
      case Update(id, _) if id == x ⇒ true
    }
    sender ! "ACK"
  }
}
```

对探针 actor 收到的消息进行应答：

```
val probe = TestProbe()
val future = probe.ref ? "hello"
probe.expectMsg(0 millis, "hello")       // TestActor 运行在 CallingThreadDispatcher 上
probe.sender ! "world"
assert(future.isCompleted && future.value == Some(Right("world")))
```

对探针 actor 收到的消息进行转发：

```
class Source(target: ActorRef) extends Actor {
  def receive = {
    case "start" ⇒ target ! "work"
  }
}
class Destination extends Actor {
  def receive = {
    case x ⇒      // Do something..
  }
}
val probe = TestProbe()
val source = system.actorOf(Props(new Source(probe.ref)))
val dest = system.actorOf(Props[Destination])
source ! "start"
probe.expectMsg("work")
probe.forward(dest)
```

三、跟踪 Actor 调用

当测试失败时，通常是由程序员来查找原因，进行修改并进行下一轮测试。这个过程既有 debugger 支持，又有日志支持，Akka 工具箱提供以下日志选项：

- 对 Actor 实例中抛出的异常记录日志
  相比其它的日志机制，这一条是永远打开的；它的日志级别是 ERROR.
- 对某些 actor 的消息记录日志
  这是通过在配置文件里添加设置项来打开，设置项为 akka.actor.debug.receive，它使得 loggable 语句作用于 actor 的 receive 函数：

```
def receive = LoggingReceive {
  case msg ⇒ // Do something...
}
```

.

- 对特殊的消息记录日志
  Actor 会自动处理某些特殊消息, e.g. Kill, PoisonPill, 等等. 打开对这些消息的跟踪只需要设置 akka.actor.debug.autoreceive, 这对所有 actor 都有效.
- 对 actor 生命周期记录日志
  Actor 的创建、启动、重启、开始监控、停止监控和终止可以通过打开 akka.actor.debug.lifecycle 来跟踪;
  这也是对所有 actor 都有效的.

所有这些日志消息都记录在 DEBUG 级别. 总结一下, 你可以用以下配置打开对 actor 活动的完整日志:

```
akka {
  loglevel = DEBUG
  actor {
    debug {
      receive = on
      autoreceive = on
      lifecycle = on
    }
  }
}
```

Akka Actor 最佳实践 【重要】

1、SHOULD evolve the state of actors only in response to messages received from the outside

下面的代码是非常错误的：

```
class SomeActor extends Actor {
  private var counter = 0
  private val scheduler = context.system.scheduler
    .schedule(3.seconds, 3.seconds, self, Tick)

  def receive = {
    case Tick =>
      counter += 1
  }
}
```

不要在 actor 内部更新它的状态。因为这会使 actor 的行为变得不确定性，并且不可能正确地测试。

如果真的需要定时的在 actor 中做一些事情，一定不能在 actor 内部初始化定时器，请把它移到外部。

2、SHOULD do back-pressure

对于生产者-消费者场景中，一般会需要考虑几个问题：

1. if the queue of messages is unbounded, with slow consumers that queue can blow up
2. distribution can be inefficient, as a worker could end up with multiple pending items whereas another worker could be standing still

A correct, worry-free design does this:

1. workers must signal demand (i.e. when they are ready for processing more items)
2. the producer must produce items only when there is demand from workers

```
// 用来通知上游可以发送数据项
case object Continue

// Message used by the producer for continuously polling the data-source, while in the polling state.
case object PollTick

/**
 * State machine with 2 states:
 *
 *  - Standby, which means there probably is a pending queue of items waiting to
 *    be sent downstream, but the actor is waiting for demand to be signaled
 *
 *  - Polling, which means that there is demand from downstream, but the actor is waiting for items to happen
 *
```

```
  * IMPORTANT: as a matter of protocol, this actor must not receive multiple
  *            Continue events - downstream Router should wait for an item
  *            to be delivered before sending the next Continue event to this
  *            actor.
  */
class Producer(source: DataSource, router: ActorRef) extends Actor {
  import Producer.PollTick

  override def preStart(): Unit = {
    super.preStart()
    // this is ignoring another rule I care about (actors should evolve
    // only in response to external messages), but we'll let that be for didactical purposes
    context.system.scheduler.schedule(1.second, 1.second, self, PollTick)
  }

  // actor starts in standby state
  def receive = standby

  def standby: Receive = {
    case PollTick =>
      // ignore
    case Continue =>
      // demand signaled, so try to send the next item
      source.next() match {
        case None =>
          // no items available, go in polling mode
          context.become(polling)
        case Some(item) =>
          // item available, send it downstream, and stay in standby state
          router ! item
      }
  }

  def polling: Receive = {
    case PollTick =>
      source.next() match {
        case None =>
          ()            // ignore - stays in polling
        case Some(item) =>
          // item available, demand available
          router ! item
          // go in standby
          context.become(standby)
      }
  }
}
```

```
/**
 * The Router is the middleman between the upstream Producer and the Workers, keeping track of demand (to keep the producer simpler).
 * NOTE: the protocol of Producer needs to be respected - so
 *           we are signaling a Continue to the upstream Producer
 *           after and only after a item has been sent downstream
 *           for processing to a worker.
 */
class Router(producer: ActorRef) extends Actor {
  var upstreamQueue = Queue.empty[Item]
  var downstreamQueue = Queue.empty[ActorRef]

  override def preStart(): Unit = {
    super.preStart()
    // signals initial demand to upstream
    producer ! Continue
  }

  def receive = {
    case Continue =>
      // demand signaled from downstream, if we have items to send then send, otherwise enqueue the downstream consumer
      if (upstreamQueue.isEmpty) {
        downstreamQueue = downstreamQueue.enqueue(sender)
      } else {
        val (item, newQueue) = upstreamQueue.dequeue
        upstreamQueue = newQueue
        sender ! item

        // signal demand upstream for another item
        producer ! Continue
      }

    case item: Item =>
      // item signaled from upstream, if we have queued consumers then signal it downstream, otherwise enqueue it
      if (downstreamQueue.isEmpty) {
        upstreamQueue = upstreamQueue.enqueue(item)
      } else {
        val (consumer, newQueue) = downstreamQueue.dequeue
        downstreamQueue = newQueue
        consumer ! item

        // signal demand upstream for another item
        producer ! Continue
      }
  }
}
```

```
class Worker(router: ActorRef) extends Actor {
  override def preStart(): Unit = {
    super.preStart()
    // signals initial demand to upstream
    router ! Continue
  }

  def receive = {
    case item: Item =>
      process(item)
      router ! Continue
  }
}
```

参考：

https://github.com/alexandru/scala-best-practices/blob/master/sections/5-actors.md